*Note: A previous knowledge in the following topics would be very helpful to appreciate/understand/comment/blame this article. C++ - Move semantics – std::vectors- micro benchmarking etc.*

I have written decent chunk of C/C++ codes as a software engineer. All through my experience I have seen that once in every 138 days someone comes up with a topic like, “which is better? C++ or C”. It is easy to get involved in such discussions. Making a statement during such discussions would be like Donald Trump talking about climate change. For a naive viewer it will make sense, but when a real climate scientist hears those arguments it would be like a horror show. Long back, I had such a discussion with my wife, she is a Java developer. I was like, how C/C++ is faster in processing than Java, how it has pointers and we could do magic with it etc. She stopped me interrupting, looking at my eyes, she says, “Being fast is not really a yardstick in every scenario. Sometimes slow and steady is better, if you know what I mean!! “. Of course, I understood exactly what she meant and from that night, I decided that … making generic statements like “C++ is better than Java” is really not worth it unless we understand the context and requirements.

Talking about making generic statements, I am writing this article to highlight one such incident. This happened during an interview process. The discussion was about the changes that C++11 brought in. We talked about "Move Semantics". I was told that due to the introduction of “move”, the performance of STL containers after C++11 are automatically faster. Here we were specifically talking about vectors.

Vectors are dynamically growing arrays. If you are new to vectors, learn more about them here, <https://en.cppreference.com/w/cpp/container/vector>. I hope you did not have to click that link since a basic knowledge of vector is, a sort of prerequisite to this discussion.

Anyway, we were talking about vectors. As you all know, vectors grow dynamically. Assume we create a vector. The cost of the push\_back to the vector is constant “amortized”. So basically once in a while your push\_back will reallocate a bigger chunk of memory, copy the current N(vect.size()) entries to this new memory, then push back the new entry to the N+1th location. To understand a bit more about "amortized cost" you could read it here, <http://www.drdobbs.com/c-made-easier-how-vectors-grow/184401375>.

Back to the discussion I had. He said, “Vectors became faster since they started supporting move”. This is because, when the push\_back does the reallocation, the performance of the move would be better than the copy, he declared. I replied that it really depends on few other factors like what we store in the vector, the actual number of entries, caching schemes etc. We both did not argue on the topic further. I don’t know why he didn’t but for me, I was suspecting myself whether the statement I made was right. I kept thinking about it and decided to test this out.

The main focus of this test is to compare the performance of a vector which stores an object which “copies” vs the ones which “moves”. We will mainly focus on the copy/move performed during the reallocation mechanism when we perform push\_back to the currently full vector.

Assume the following is the data to be stored in the vector.

*struct node {*

*int e[100];*

*int a;*

*float b;*

*double c;*

*char d;*

*};*

I have created two wrapper classes to store the structure **node**. One class has implemented move constructor/assignment and aptly named as **testMove** and the other class which only has the copy constructor is **testCopy**.

When we store these two classes in a vector, during reallocation of vector memory, it will perform copy/move of the appropriate classes to the newly allocated memory. We will test the performance of the vector during this process.

*class testCopy {*

*public:*

*blah blah blah!!*

*private:*

***node n1;***

*};*

*class testMove {*

*public:*

*blah blah blah !!*

*private:*

***node \*n1=nullptr;***

*};*

Look at the file *vectorTest.cpp* to see the complete implementation but I would like to highlight something from the above code snippet. You might have noticed that the **testCopy** class stores the **node** directly whereas the **testMove** stores the node in the **heap** (a pointer). These two looks basically different and that raises a question if it is fair to compare these two different classes. But the following is the rationale behind this. Let’s assume my product (from pre-C++11 times) has the class **testCopy** implemented. Now, with C++ 11, after move semantics got introduced I am adding a move constructor/assignment to my class. To properly leverage the advantage I gained with the “move”, I would change the classes private member “**node”** to a pointer. Only when the data is in the heap (a pointer), I can easily steal the guts of it and thereby make my “move” more sensible. If I still store it in the stack, my “move” should in turn perform a “copy” which kind of breaks the actual purpose/advantage we have with move.

Now objects of these two classes will be pushed back to vectors. The function *vectorWorkCopy* and *vectorWorkMove* does this for the respective classes. During the periodic resizing of the vector push\_back, the vector will have to allocate a bigger chunk of memory, copy the existing objects to the new memory and delete the objects in the old memory before freeing it. During this process, while copying the existing objects, the vector which stored the *testCopy* class would perform an actual copy(copy constructor) of node whereas the *testMove* class will perform a move(move constructor). We will micro benchmark these two functions *vectorWorkCopy* and *vectorWorkMove* and compare the results. I used google benchmark for this.

The tests were performed with two different inputs,

1) The struct **node** has the array e[100] commented out , i.e, it only has an int, float, double and char.

2) With the array e[100] included, Basically a bigger structure.

The following is the benchmarking result for these cases. *You could compile and execute the source code yourselves with the help of the instructions in Install.txt.*

Result for the testcase 1,

***struct node {***

***int a;***

***float b;***

***double c;***

***char d;***

***};***

***Running ./vectorFilt***

***Run on (4 X 3800 MHz CPU s)***

***CPU Caches:***

***L1 Data 32K (x4)***

***L1 Instruction 32K (x4)***

***L2 Unified 256K (x4)***

***L3 Unified 6144K (x1)***

***---------------------------------------------------------***

***Benchmark Time CPU Iterations***

***---------------------------------------------------------***

***BM\_VecCopy/1 209 ns 209 ns 3338822***

***BM\_VecCopy/8 982 ns 982 ns 687399***

***BM\_VecCopy/64 4894 ns 4894 ns 136651***

***BM\_VecCopy/512 31228 ns 31225 ns 21983***

***BM\_VecCopy/4096 255484 ns 255447 ns 2722***

***BM\_VecCopy/32768 2275073 ns 2274431 ns 309***

***BM\_VecCopy/100000 7909717 ns 7907723 ns 87***

***BM\_VecMove/1 247 ns 247 ns 2846989***

***BM\_VecMove/8 1337 ns 1336 ns 532322***

***BM\_VecMove/64 7387 ns 7386 ns 93674***

***BM\_VecMove/512 50187 ns 50181 ns 13153***

***BM\_VecMove/4096 388109 ns 388044 ns 1795***

***BM\_VecMove/32768 3082180 ns 3081843 ns 226***

***BM\_VecMove/100000 10442224 ns 10440024 ns 66***

***bash-4.2$***

What do we see here? When the struct “node “ is small with just an int, char, float and double, the performance of the copy is far better than move(see the “Time” column) even for 100000 entries added to the vector. Such use cases to store simple datatypes is very prevalent and it happens often. Contradicting the statement that, “Vectors became faster since they started supporting move”, we see that the copy is actually faster here than the move. Now let’s go to the next example,

***struct node {***

***int e[100];***

***int a;***

***float b;***

***double c;***

***char d;***

***};***

***Running ./vectorFilt***

***Run on (4 X 3800 MHz CPU s)***

***CPU Caches:***

***L1 Data 32K (x4)***

***L1 Instruction 32K (x4)***

***L2 Unified 256K (x4)***

***L3 Unified 6144K (x1)***

***---------------------------------------------------------***

***Benchmark Time CPU Iterations***

***---------------------------------------------------------***

***BM\_VecCopy/1 243 ns 243 ns 2869208***

***BM\_VecCopy/8 1256 ns 1256 ns 558660***

***BM\_VecCopy/64 7164 ns 7160 ns 96927***

***BM\_VecCopy/512 132352 ns 132329 ns 5395***

***BM\_VecCopy/4096 1359904 ns 1359654 ns 499***

***BM\_VecCopy/32768 13329690 ns 13326112 ns 49***

***BM\_VecCopy/100000 50061409 ns 50004230 ns 14***

***BM\_VecMove/1 315 ns 315 ns 2200089***

***BM\_VecMove/8 1596 ns 1596 ns 434969***

***BM\_VecMove/64 8571 ns 8570 ns 80687***

***BM\_VecMove/512 58544 ns 58538 ns 11579***

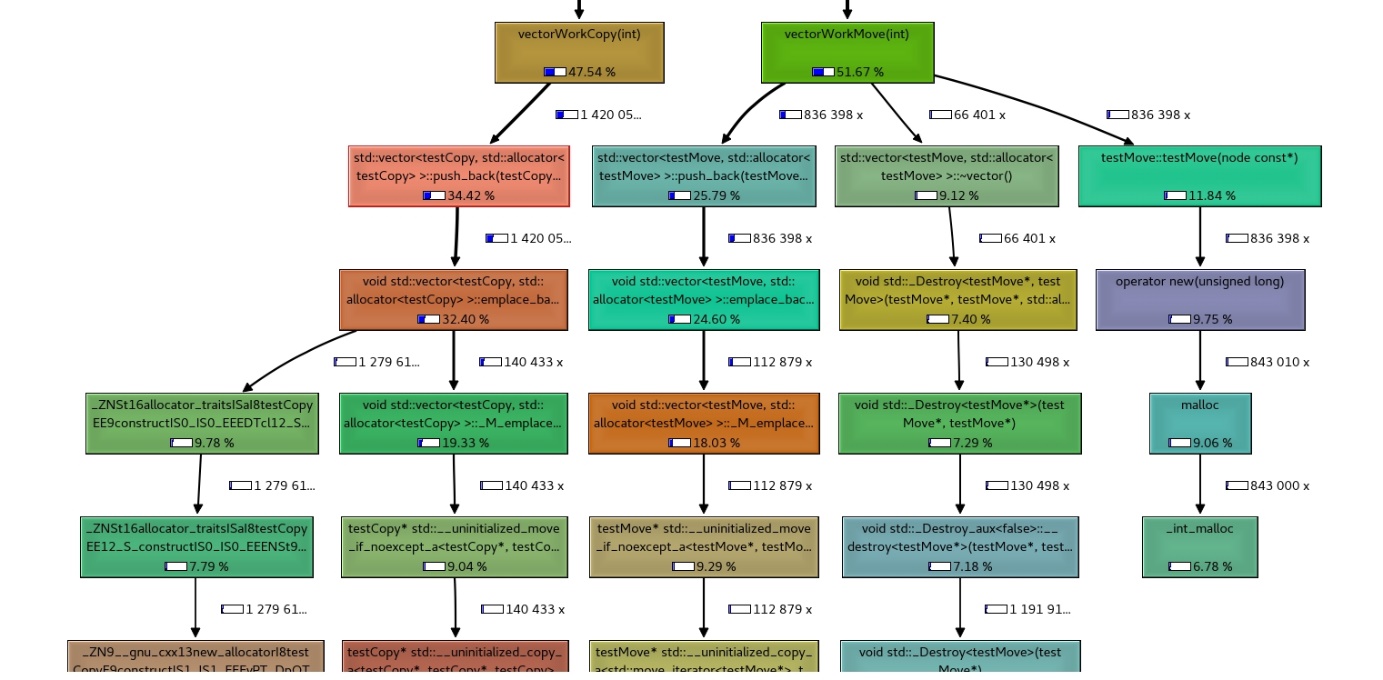
***BM\_VecMove/4096 461126 ns 461061 ns 1505***

***BM\_VecMove/32768 4664562 ns 4663553 ns 113***

***BM\_VecMove/100000 16424874 ns 16422625 ns 40***

Now when the size of the structure increases, by adding an array e[100] to the node, the copy gets costlier than move *eventually*. When the number of vector entries reaches somewhere around adding 512, we realize that the move is getting better than copy.

Let’s go a bit deeper to visualize the internals. I am using the tool kcachegrind to see what exactly is happening inside and which function call takes more time. The complete graph is attached to the repository. Below, I have pasted the relevant snapshot for the discussion,



This is the call graph of our scenario along with the percentage utilization of the CPU by each function. We see that the vectorWorkCopy function takes 47.54% of CPU time whereas the vectorWorkMove takes more, i.e 51.67 %. Our benchmark already highlighted this.

Though we see that the overall performance of vectorWorkCopy is better than the vectorWorkMove, when we see closer, the push\_back of the move was actually cheaper than the push\_back of the copy. But you don’t get a “*Candy from Willy*” for guessing why. This is pretty straight forward.Lets see what the move constructor does,

***testMove(testMove&& dat) noexcept : n1{dat.n1} {***

***dat.n1 = nullptr;***

***}***

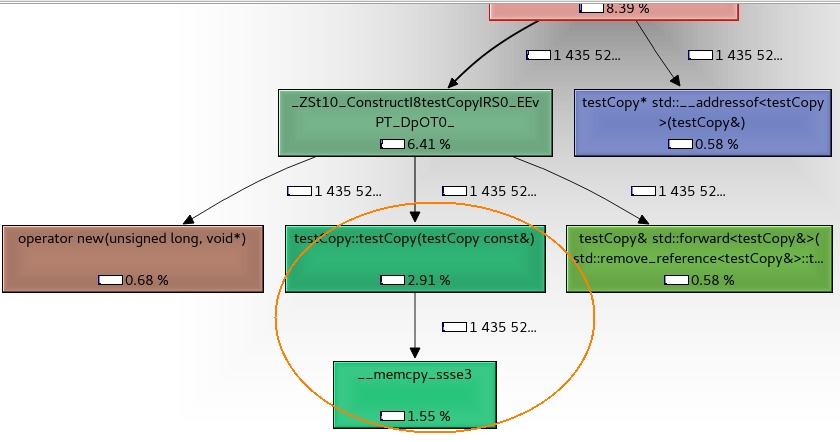
In comparison to the copy constructor which is something like,

***testCopy(const testCopy& dat){***

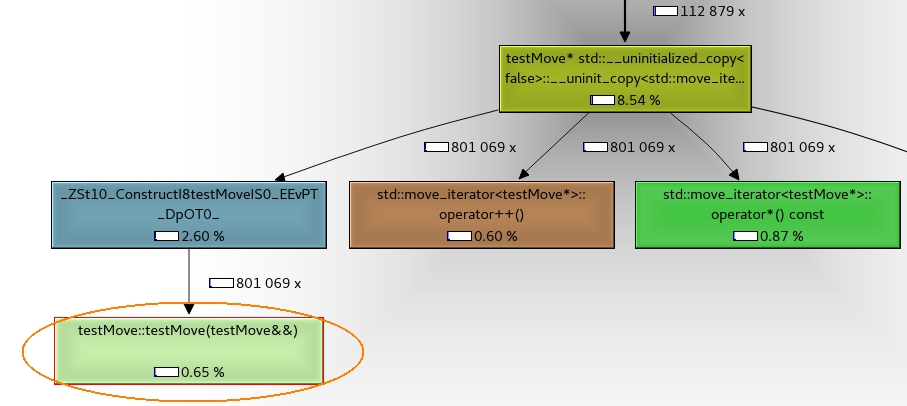
***memcpy(&n1, &dat.n1,sizeof(node));***

***}***

The actual performance numbers associated with these constructors are as follows. For copy, the percentage utilization is 2.91 from total activities.



For the move it is just 0.65 %.



What we have seen is that the “move” was indeed proving why he is worthier than the “copy”. But overall why is the “move” costlier than the “copy”?

This is evident in the details from the first graph. The flow of *vectorWorkMove* performs better in the actual “move” operation since it only moves the heap address whereas the “copy” copies the entire data. But along with that, it also performs extra memory allocation and deletion w.r.t the pointer. The destructor (~) of the *vectorWorkCopy* does nothing whereas the *vectorWorkMove* performs an actual *delete*. The advantage we gained by having a pointer and *moving* it instead of *copying* is negated by the fact that we are performing multiple new and delete w.r.t the pointer. Depending upon the size of the data and the number of operations, this behavior changes and the “move” becomes economical when the test involves bigger data to be copied which was evident from our testcase 2.

Apart from these there are few other factors which probably influenced these results. One for example is the caching scheme of the CPU I tested with. You could see from the logs that the system I used has an L1, L2 and L3 cache. During copy, there are probably lots happening behind w.r.t prefetching and other optimizations. In embedded world, our actual target processors usually do not have such caching possibilities. In such systems, the copy could probably be costlier than our current result and the move might perform better in all the cases. But that is what I would like us all to infer with these tests. Generally, it might look harmless when we make statements like, “STL containers became faster after they started supporting move semantics”. But I believe it is important for us to clearly understand the system, the type of data used, use cases etc., before arriving at a conclusion. If we are doubtful, we only need to benchmark to see what exactly we are dealing with. What do you say? Feel free to share your comments.