I have written decent chunk of C and C++ codes as a software engineer. All trough my experience I have seen that once in every 13.5 months 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 naïve viewer it will make sense, but when a real climate scientist hears those arguments he would feel 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 made a decision that … making generic statements like “C++ is better than Java” is really not worth it unless we understand the complete context and requirements.

Talking about making generic statements, I wanted to write this article to highlight one such incident. This happened during a interview process. The topic was about the changes that C++11 brought in. We talked about "Move Semantics". I was told that because of the move semantics, the STL containers after C++11 are automatically faster. 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>. Anyway, if you needed to click that link to read about the vector (for the first time), then you are actually new. May be you should do some more programming with Vectors before proceeding further !!



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. Again, I assume this is not a new concept to you all since you did not click <https://en.cppreference.com/w/cpp/container/vector> earlier. Still to know more about this "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 what we actually store in the vector and the actual number of entries, caching schemes etc. We both did not take up 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.

This whole build-up I made until now is to lead us to this test. To see whether the vector automatically performs better since it could move or if it actually performs bad depending on what it stores.

Assume the following is the data to be stored in the vector, or this is the internal data which is encapsulated by a class that facilitates what we would like to see. The complete code with make files, pictures, results etc, are available in the repository.

*struct node {*

*int e[100];*

*int a;*

*float b;*

*double c;*

*char d;*

*};*

I need to create a wrapper class to store **node**. Two classes, one for copy and other for move are created,

*class testCopy {*

*public:*

*blablabla*

*private:*

***node n1;***

*};*

*class testMove {*

*public:*

*blablabla*

*private:*

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

*};*

You could look into the complete code in the repo to understand what I actually did but I would like to highlight something here. You might have noticed that the **testCopy** class has the **node** in the stack directly whereas the **testMove** stores the node in the **heap** (a pointer). These two looks basically different but I would assume this is how I would have designed the respective classes pre and post move times. The idea is that, the node is not too big, and it is fine for me to keep it in the stack. But when “move” got introduced in C++11, I would not keep the node n1 in the stack. 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” within, which kind of puts off the advantage of why “move” is used. So, these two classes testCopy and testMove stores the data node, but they are defined with the mindset of pre and post move times.

Assume these classes are to be stored in a vector. During the periodic reallocation of the vector push\_back, the testCopy class would perform an actual copy of n1(copy constructor) whereas the testMove class will perform a move (move constructor). I used google benchmark to micro benchmark the operations on these two classes.

The tests were performed in two ways,

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] present, Basically a big structure. The following is a sample result,

For the node,

***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$***

For the node

***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***

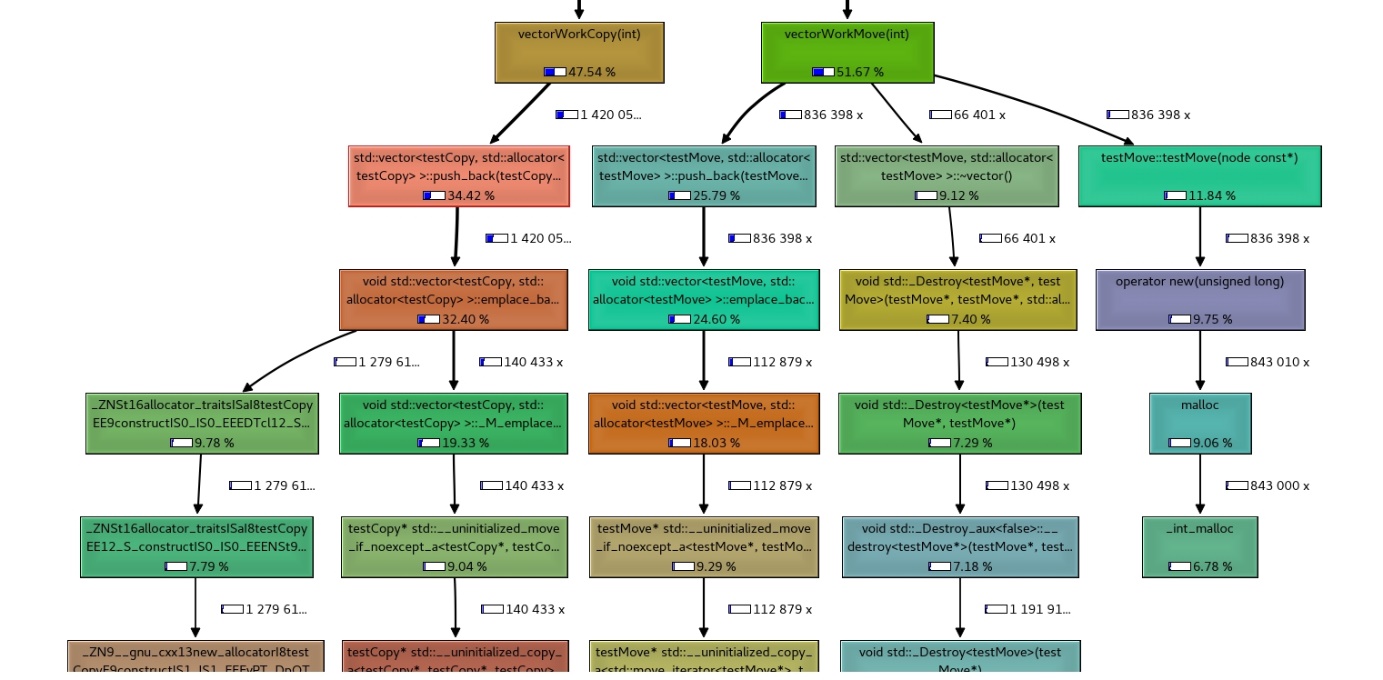
***BM\_VecMove/4096 461126 ns 461061 ns 1505***

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

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

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 even for 100000 entries added to the vector. Such use cases to store simple datatypes is very prevalent and it happens often. Now when the size of the structure increases, by adding an array e[100] to the node, the copy gets costlier *eventually*. When the number of entries reaches somewhere around adding 512 entries to the vector, we realize that the move is getting better than copy.

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



In general, we see that the vectorWorkCopy function takes 47.54% of time whereas the vectorWorkMove takes more, i.e 51.67 %. Our benchmark already highlighted this.

It seems like the push\_back itself for the copy and move is not the reason, in fact the push\_back of the move is cheaper than the one of copy. But that is not a surprise for us. The move constructor only does this,

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

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

***}***

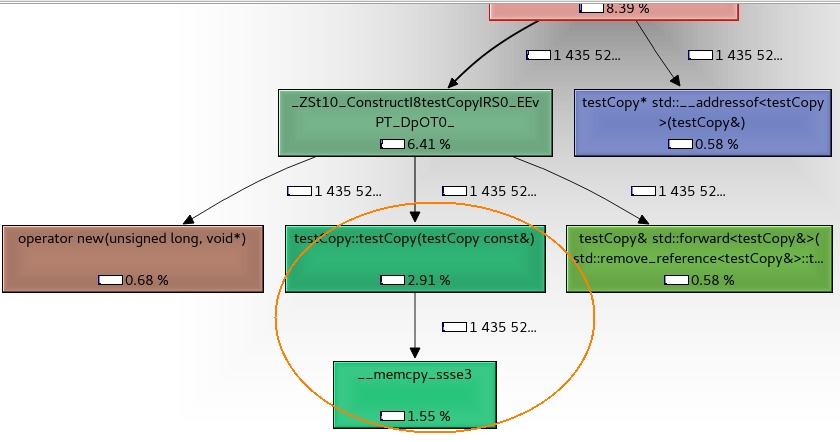
In comparision to the copy constructor which is something like,

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

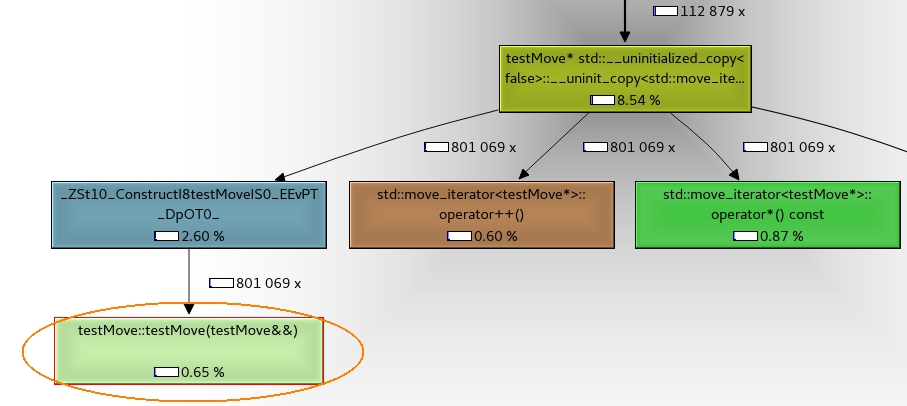
***memcpy(&n1, &dat.n1,sizeof(node));***

***}***

The actual numbers associated to the constructors are as follows. For copy the percentage utilization is 2.91 from total activities.



For the move it is just 0.65 %.



So, what really happens and why the move is costlier in total than the copy. If you have seen the first graph I pasted above, this was evident. The flow of vectorWorkMove takes more time for allocation and deletion of the pointer. Infact ,even during the push\_back reallocation, the destructor of the corresponding classes will be called .The destructor of the vectorWorkCopy does nothing whereas the vectorWorkMove performs a 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 amount of “copy” done gets increased.

Here there are many other factors too that are relevant. 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. So, in such systems, the copy could probably be costlier than this and the move might always be economical. **But that’s the point**. In a generic sense it might fine when we make a statement 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 making such a statement/conclusion. If we are doubtful, we only need to micro benchmark to see what exactly we are dealing with. What do you say? Feel free to share your comments.

P.S: The interview proceeded fine and the result was positive.