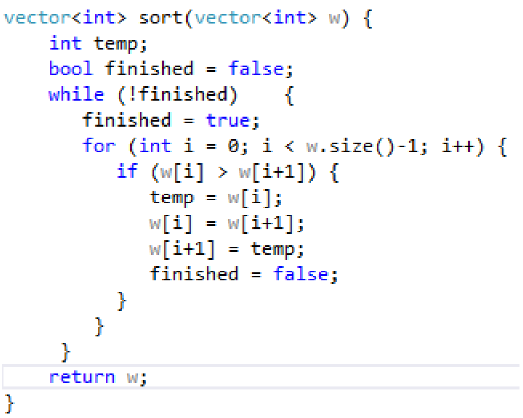
Run 1

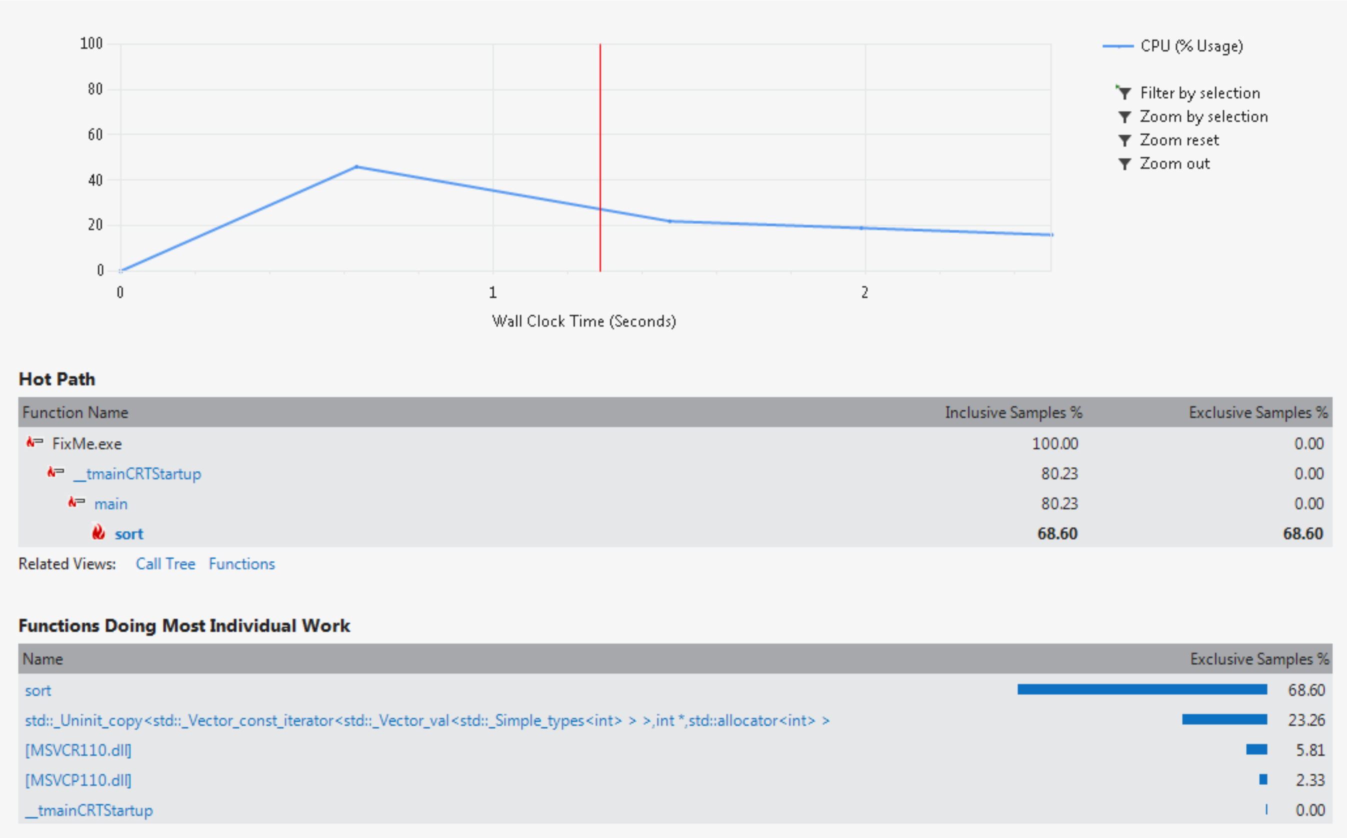
Before Optimization

Code



```
vector<int> sort(vector<int> w) {
    int temp;
    bool finished = false;
    while (!finished)    {
       finished = true;
       for (int i = 0; i < w.size()-1; i++) {
          if (w[i] > w[i+1]) {
             temp = w[i];
             w[i] = w[i+1];
             w[i+1] = temp;
             finished = false;
          }
        }
     }
    return w;
}
```

Results



Hot Path

Function Name	Inclusive Samples %	Exclusive Samples %
FixMe.exe	100.00	0.00
__tmainCRTStartup	80.23	0.00
main	80.23	0.00
sort	**68.60**	**68.60**

Related Views: Call Tree Functions

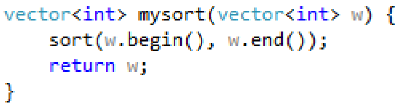
Functions Doing Most Individual Work

Name	Exclusive Samples %
sort	68.60
std::_Uninit_copy<std::_Vector_const_iterator<std::_Vector_val<std::_Simple_types<int> > >,int *,std::allocator<int> >	23.26
[MSVCR110.dll]	5.81
[MSVCP110.dll]	2.33
__tmainCRTStartup	0.00

As you can see, the sort function takes a vast majority of the time. This is because the sort is done using bubble sort, which has a wore case running time of n^2. This sort function can be upgraded to a different algorithm to decrease the amount of time spent sorting.

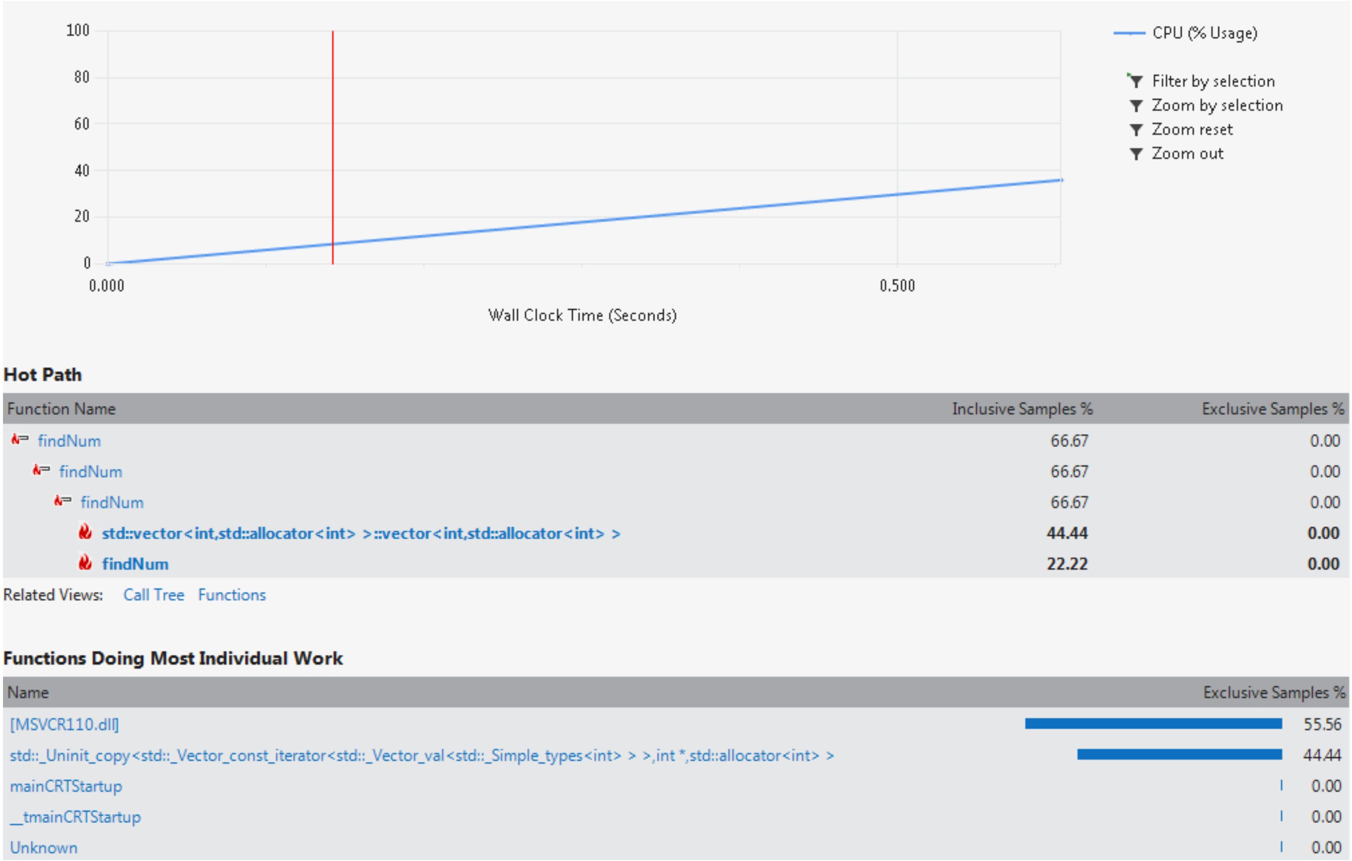
After Optimization

Fixed Code



```
vector<int> mysort(vector<int> w) {
    sort(w.begin(), w.end());
    return w;
}
```

Results



Hot Path

Function Name	Inclusive Samples %	Exclusive Samples %
findNum	66.67	0.00
findNum	66.67	0.00
findNum	66.67	0.00
std::vector<int,std::allocator<int> >::vector<int,std::allocator<int> >	**44.44**	**0.00**
findNum	**22.22**	**0.00**

Related Views: Call Tree Functions

Functions Doing Most Individual Work

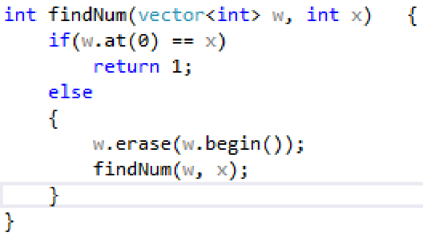
Name	Exclusive Samples %
[MSVCR110.dll]	55.56
std::_Uninit_copy<std::_Vector_const_iterator<std::_Vector_val<std::_Simple_types<int> > >,int *,std::allocator<int> >	44.44
mainCRTStartup	0.00
__tmainCRTStartup	0.00
Unknown	0.00

After changing the sort method to the stl's variation, the time spent sorting the numbers is mostly negligible.

Run 2

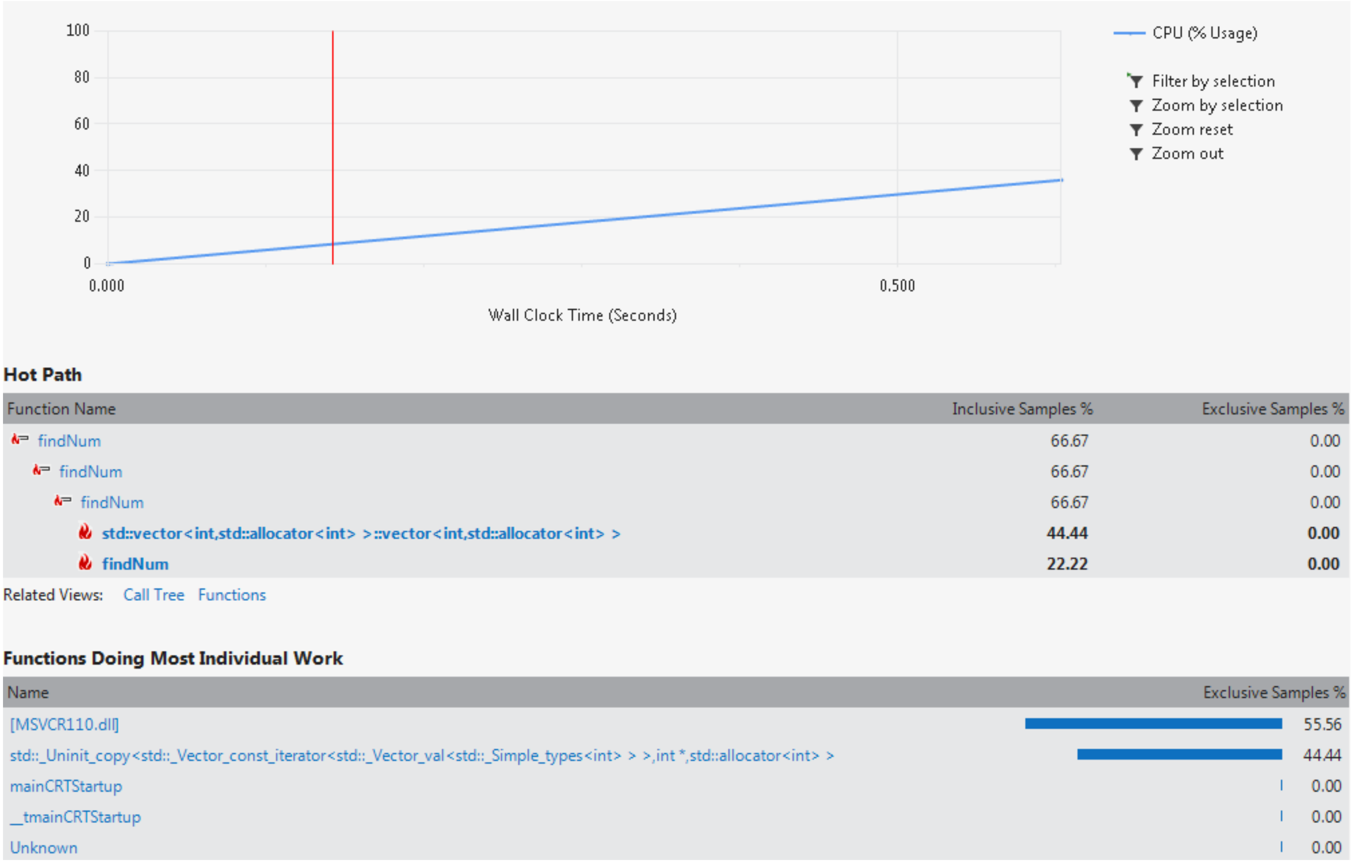
Before Optimization

Code



```
int findNum(vector<int> w, int x)   {
    if(w.at(0) == x)
        return 1;
    else
    {
        w.erase(w.begin());
        findNum(w, x);
    }
}
```

Results



Hot Path

Function Name	Inclusive Samples %	Exclusive Samples %
findNum	66.67	0.00
findNum	66.67	0.00
findNum	66.67	0.00
std::vector<int,std::allocator<int> >::vector<int,std::allocator<int> >	**44.44**	**0.00**
findNum	**22.22**	**0.00**

Related Views: Call Tree Functions

Functions Doing Most Individual Work

Name	Exclusive Samples %
[MSVCR110.dll]	55.56
std::_Uninit_copy<std::_Vector_const_iterator<std::_Vector_val<std::_Simple_types<int> > >,int *,std::allocator<int> >	44.44
mainCRTStartup	0.00
__tmainCRTStartup	0.00
Unknown	0.00

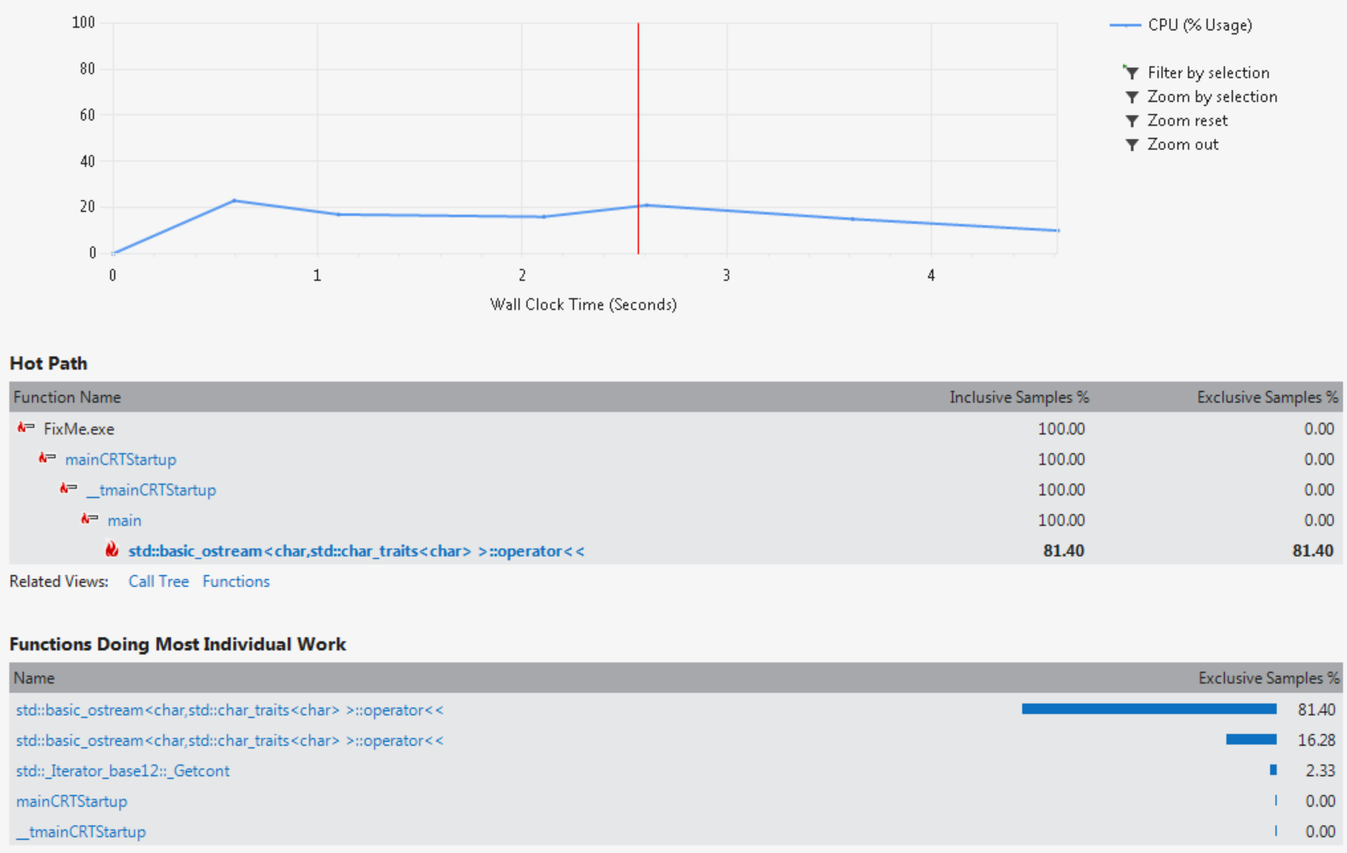
As you can see above, a lot of the time is now spent in the `findNum()` function. This is because, as the code above shows, the function is recursive. This means a new vector has be continually created as the function goes deeper and deeper. In the 'hot path' section we can see that a common path is for `findNum` to call itself. This can be greatly optimized by using a different searching algorithm.

After Optimization

Fixed Code

```
int findNum(vector<int> w, int x)   {
    for(int i = 0; i < w.size(); i++)   {
        if(w.at(0) == x)
            return 1;
    }
}
```

Results



Hot Path

Function Name	Inclusive Samples %	Exclusive Samples %
FixMe.exe	100.00	0.00
mainCRTStartup	100.00	0.00
__tmainCRTStartup	100.00	0.00
main	100.00	0.00
std::basic_ostream<char,std::char_traits<char> >::operator<<	**81.40**	**81.40**

Related Views: Call Tree Functions

Functions Doing Most Individual Work

Name	Exclusive Samples %
std::basic_ostream<char,std::char_traits<char> >::operator<<	81.40
std::basic_ostream<char,std::char_traits<char> >::operator<<	16.28
std::_Iterator_base12::_Getcont	2.33
mainCRTStartup	0.00
__tmainCRTStartup	0.00

After changing the algorithm to a linear search the program spent almost no time in the find method. The only thing taking up time really at that point is the debug printing.