CPSC 501 - Assignment 2

Ahad Hamirani - UCID: 30063218

Instructions on how access my gitlab repository:

I have already invited my TA Navid Alipour, as a reporter to my assignment 2 repository.

Here is the link for my repository:
https://gitlab.cpsc.ucalgary.ca/ahad.hamirani/cpsc_501_a2.git

How to compile and run the code:

There are two different folders in my repository named “base_program” and “FFT_program”, the code for the base program along with its output and gprof file are located in the base_program folder. Similarly, the code for the FFT program along with all the different outputs for each optimization and all the gprofs for each optimization are located in the “FFT_program” folder. I have made all optimizations on the FFT.cpp file itself so in order to see unoptimized and optimize progression you will have to look at the commit history. For both (base.cpp and FFT.cpp) programs compilation is identical. Here is an example:

g++ base.cpp
./a.out guitar_dry.wav big_hall_IR_mono.wav output.wav

g++ FFT.cpp
./a.out guitar_dry.wav big_hall_IR_mono.wav output.wav

Just a note that if you don’t provide enough inputs or inputs in incorrect order the compilation and final result will be impacted. So please make sure to provide the correct number of arguments and in the correct order as shown above.

Time Table containing times for all stages of the code:

	Base Program **(seconds)**	FFT optimization **(seconds)**	Compiler optimization **(seconds)**	Minimize work optimization **(seconds)**	Partial unrolling optimization **(seconds)**	Strength Reduction optimization **(seconds)**	Code Jamming optimization **(seconds)**
guitar_dry	471.94	1.72	0.96	0.91	0.86	0.81	0.82
guitar_trim	10.63	0.20	0.11	0.09	0.06	0.05	0.04

Note: The times after base program time (FFT optimization inclusive and onwards) are all additional optimizations. To elaborate, the Minimize work Optimization time is calculated by the FFT optimization, Compiler optimization and Minimize work optimization all working in

conjunction. Due to this we can see a constant decrease in time as we add more and more optimizations.

Base Solution:

My base solution for this assignment is just a simple time-domain convolution. It uses two for loops and follows the algorithm shown in class. Below is a screenshot of my code displaying the time-domain convolution algorithm:

```
int i;
int j;
// covulution
for(i = 0; i < impulseArraySize; i++){
    for(j = 0; j < inputArraySize; j++){
        outputArray[i+j] += impulseArray[i] * inputArray[j];
    }
}
```

Optimizations:

There are a total of six optimizations that I have done. The first and the most important one is using the Fast Fourier Transform to speed up the convolution process. The rest are Minimize work (labelled as opt1 in repository), Partial unrolling (labelled as opt2 in repository), Strength Reduction (labelled as opt3 in repository) and finally Jamming (labelled as opt4 in repository). Now I will elaborate on each optimization individually.

Optimization 1: Fast Fourier Transform:

This is by far the most important optimization in this assignment. In this optimization we take the time-domain entries from the input and impulse files, convert them to frequency domain arrays, and apply the Fast Fourier Transform algorithm to each of these frequency arrays. Next we entrywise multiply these transformed arrays and finally we convert them back into a single output time-domain array. I have used an online resource (linked above) to implement the Fast Fourier Transform.

Code snippets:

Lots of changes were made and too many to screenshot so please just check the repository to see the changes from the base solution

Regression testing:

I do a regression test in two different ways. Firstly, I use the “cmp” function to compare the new output sound file from the optimized version and the original output sound file from the base program. Moreover, I also listen to the new output sound file and make sure that sounds exactly like the original sound output file. Below is the code snippet of me using the compare function:

```
ahad.hamirani@ms176-3eb:~/CS501$ cmp output.wav FFT_output.wav
ahad.hamirani@ms176-3eb:~/CS501$
```

```
ahad.hamirani@ms176-3eb:~/CS501$ cmp trim_base_output.wav trim_FFT_output.wav
ahad.hamirani@ms176-3eb:~/CS501$
```

Here we can see that no output displaying the differences between two files was shown meaning that the two files are exactly alike and we have successfully optimized using the Fast Fourier Transform while preserving correctness.

GPROF timings:

Guitar_dry.wav time

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 94.67      1.63     1.63        3   542.77   542.77  fft(double*, int, int)
  3.20      1.68     0.06                             main
  1.17      1.70     0.02        1    20.04    20.04  multiply(double const*, double const*, double*, int)
  0.58      1.71     0.01        2     5.01     5.01  readWavFile(int*, int*, char*)
  0.58      1.72     0.01        1    10.02    10.02  writeWavFile(double*, int, int, char*)
  0.00      1.72     0.00       10     0.00     0.00  freadIntLSB(_IO_FILE*)
  0.00      1.72     0.00        8     0.00     0.00  freadShortLSB(_IO_FILE*)
  0.00      1.72     0.00        5     0.00     0.00  fwriteIntLSB(int, _IO_FILE*)
  0.00      1.72     0.00        4     0.00     0.00  fwriteShortLSB(short, _IO_FILE*)
  0.00      1.72     0.00        2     0.00     0.00  readWavFileHeader(int*, int*, _IO_FILE*)
  0.00      1.72     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z15convertToStereoPdPii
  0.00      1.72     0.00        1     0.00     0.00  writeWavFileHeader(int, int, double, _IO_FILE*)
  0.00      1.72     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      1.72     0.00        1     0.00     0.00  __gnu_cxx::__promote_2<int, double, __gnu_cxx::__promote<int,
  0.00      1.72     0.00        1     0.00     0.00  __gnu_cxx::__enable_if<std::__is_integer<int>::__value, double
```

Guitar_trim.wav time

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 97.70      0.20     0.20        3    65.13    65.13  fft(double*, int, int)
  0.00      0.20     0.00   150806     0.00     0.00  std::abs(double)
  0.00      0.20     0.00       10     0.00     0.00  freadIntLSB(_IO_FILE*)
  0.00      0.20     0.00        8     0.00     0.00  freadShortLSB(_IO_FILE*)
  0.00      0.20     0.00        5     0.00     0.00  fwriteIntLSB(int, _IO_FILE*)
  0.00      0.20     0.00        4     0.00     0.00  fwriteShortLSB(short, _IO_FILE*)
  0.00      0.20     0.00        2     0.00     0.00  readWavFile(int*, int*, char*)
  0.00      0.20     0.00        2     0.00     0.00  readWavFileHeader(int*, int*, _IO_FILE*)
  0.00      0.20     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z15convertToStereoPdPii
  0.00      0.20     0.00        1     0.00     0.00  writeWavFile(double*, int, int, char*)
  0.00      0.20     0.00        1     0.00     0.00  writeWavFileHeader(int, int, double, _IO_FILE*)
  0.00      0.20     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      0.20     0.00        1     0.00     0.00  multiply(double const*, double const*, double*, int)
  0.00      0.20     0.00        1     0.00     0.00  __gnu_cxx::__promote_2<int, double, __gnu_cxx::__promote<int,
  0.00      0.20     0.00        1     0.00     0.00  __gnu_cxx::__enable_if<std::__is_integer<int>::__value, double
```

Here we can clearly see that the overall time of the code in both instances has drastically reduced.

Optimization 2: Compiler optimization:

This optimization is fairly straightforward as it requires little effort. All I have done in this optimization is use the -O3 flag when compiling the code.

Code snippets:

Below is the example command:

g++ -O3 FTT.cpp
./a.out guitar_dry.wav big_hall_IR_mono.wav output.wav

Regression testing:

I do a regression test in two different ways. Firstly, I use the “cmp” function to compare the new output sound file from the optimized version and the original output sound file from the base program. Moreover, I also listen to the new output sound file and make sure that sounds exactly like the original sound output file. Below is the code snippet of me using the compare function:

```
ahad.hamirani@ms176-3eb:~/CS501$ cmp output.wav opt1_output.wav
ahad.hamirani@ms176-3eb:~/CS501$ cmp trim_base_output.wav opt1_trim_output.wav
ahad.hamirani@ms176-3eb:~/CS501$
```

Here you can see that in both instances no differences were outputted by the cmp function. The files are identical and we have successfully done the compiler optimization while maintaining correctness.

GPROF timings:

Guitar_dry.wav time

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 98.16      0.94     0.94                             fft(double*, int, int)
  1.04      0.95     0.01                             writeWavFile(double*, int, int, char*)
  1.04      0.96     0.01                             multiply(double const*, double const*, double*, int)
  0.00      0.96     0.00        2     0.00     0.00  readWavFileHeader(int*, int*, _IO_FILE*)
  0.00      0.96     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z15convertToStereoPdPii
  0.00      0.96     0.00        1     0.00     0.00  writeWavFileHeader(int, int, double, _IO_FILE*)
```

Guitar_trim.wav time

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.25      0.11     0.11                             fft(double*, int, int)
  0.00      0.11     0.00        2     0.00     0.00  readWavFileHeader(int*, int*, _IO_FILE*)
  0.00      0.11     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z15convertToStereoPdPii
  0.00      0.11     0.00        1     0.00     0.00  writeWavFileHeader(int, int, double, _IO_FILE*)
```

Here we can see that the time is improved from the compiler optimization.

Optimization 3: Minimize work:

This is the first code tuning optimization. In this optimization I take out a repeated calculation from a for loop. This way the calculation is only done once outside the for loop and that

constant is just referenced in the for loop. Since computations can be quite demanding by significantly reducing the amount of computations in this case we can save some time in the program's execution.

Code snippet:

```
      printf("Finished");
  @@ -446,10 +409,13 @@ double* fft(double *inputarr, int size, int dir)

      // roots of unity array
      double *w = new double[size/2];
-
      // For storing n complex nth roots of unity
+     // OPTIMIZATION 1: Minimize Work by creating a constant
+     double alpha_constant = -2 * M_PI / (size/2);
+
      for (int i = 0; i < (size/4); i++) {
-         double alpha = -2 * M_PI * i / (size/2);
+         // Changed to constant
+         double alpha = i * alpha_constant;
          w[2*i] = cos(alpha);
          w[2*i+1] = dir*sin(alpha);
      }
```

The following snippet shows the lines deleted and added in order to minimize work optimization. Here we can see that the -2 * M_PI / (size/2) calculation was being done many times in the for loop so by removing it outside the loop and declaring it a constant variable we have just saved a lot of computation time in execution.

Regression testing:

I do a regression test in two different ways. Firstly, I use the “cmp” function to compare the new output sound file from the optimized version and the original output sound file from the base program. Moreover, I also listen to the new output sound file and make sure that sounds exactly like the original sound output file. Below is the code snippet of me using the compare function:

```
ahad.hamirani@ms176-3eb:~/CS501$ cmp output.wav opt1_output.wav
ahad.hamirani@ms176-3eb:~/CS501$ cmp trim_base_output.wav opt1_trim_output.wav
ahad.hamirani@ms176-3eb:~/CS501$
```

Here we can see that the cmp function has not returned any differences so we have successfully optimized using Mininze work technique while preserving correctness.

GPROF timing:

Guitar_dry.wav time

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 98.04      0.89     0.89                             fft(double*, int, int)
  1.10      0.90     0.01                             writeWavFile(double*, int, int, char*)
  1.10      0.91     0.01                             multiply(double const*, double const*, double*, int)
  0.00      0.91     0.00        2     0.00     0.00  readWavFileHeader(int*, int*, _IO_FILE*)
  0.00      0.91     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z15convertToStereoPdPii
  0.00      0.91     0.00        1     0.00     0.00  writeWavFileHeader(int, int, double, _IO_FILE*)
```

Guitar_trim.wav time

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.25      0.09     0.09                             fft(double*, int, int)
  0.00      0.09     0.00        2     0.00     0.00  readWavFileHeader(int*, int*, _IO_FILE*)
  0.00      0.09     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z15convertToStereoPdPii
  0.00      0.09     0.00        1     0.00     0.00  writeWavFileHeader(int, int, double, _IO_FILE*)
```

Optimization 4: Partial unrolling:

This is also a code tuning optimization. In this optimization I take some for loops that are a very large number of times and alter them in such a way that instead of a single index of an array being updated at once two consecutive indices are updated at once. By doing this the for loops only have to run for half the number of times and therefore we can save some time during execution.

Code Snippets:

```
-     for(int i = 0; i < inputArraySize; i++){
-         inputFreq[i*2] = inputArray[i];
+     int counter;
+     // OPTIMIZATION 2: Partial unrolling
+     for(counter = 0; counter < inputArraySize-1; counter+=2){
+         inputFreq[counter*2] = inputArray[counter];
+         inputFreq[2*(counter+1)] = inputArray[counter+1];
      }
```

```
-     for(int i = 0; i < impulseArraySize; i++){
-         impulseFreq[i*2] = impulseArray[i];
+     if(counter == inputArraySize-1){
+         inputFreq[counter*2] = inputArray[counter];
+     }
+
+
+     // OPTIMIZATION 2: Partial unrolling
+     for(counter = 0; counter < impulseArraySize-1; counter+=2){
+         impulseFreq[counter*2] = impulseArray[counter];
+         impulseFreq[2*(counter+1)] = impulseArray[counter+1];
+     }
+
+     if(counter == impulseArraySize-1){
+         impulseFreq[counter*2] = impulseArray[counter];
      }
```

```
    double *inverse_fft = fft(outputFreqArr,freqSize,-1);

-   for(int i = 0; i < freqSize; i++){
+   // OPTIMIZATION 2: Partial unrolling
+   for(int i = 0; i < freqSize; i+=2){
        inverse_fft[i] = inverse_fft[i] / (freqSize/2);
+       inverse_fft[i+1] = inverse_fft[i+1] / (freqSize/2);
    }
```

These code snippets shows all the lines added and removed to use the partial unrolling technique. As we can see by updating two indices at once for all the for loops above we can halving the time spent in these loops. This is especially useful in this case as these are that being iterated over are very large. Here in the first 2 for loops we also have a check for arrays that aren't always even to make sure that we don't miss last elements in the odd length arrays. The last loop has an array that is always a power of 2 and therefore will always be even so we don't need to work about missing any elements here.

Regression testing:

I do a regression test in two different ways. Firstly, I use the "cmp" function to compare the new output sound file from the optimized version and the original output sound file from the base program. Moreover, I also listen to the new output sound file and make sure that sounds exactly like the original sound output file. Below is the code snippet of me using the compare function:

```
ahad.hamirani@ms176-3eb:~/CS501$ cmp output.wav opt2_output.wav
ahad.hamirani@ms176-3eb:~/CS501$ cmp trim_base_output.wav trim_opt2_output.wav
ahad.hamirani@ms176-3eb:~/CS501$
```

Here we can see that the cmp function has not returned any differences so we have successfully optimized using Partial unrolling technique while preserving correctness.

GPROF timing:

Guitar_dry.wav time

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 99.09      0.85     0.85                             fft(double*, int, int)
  1.17      0.86     0.01                             multiply(double const*, double const*, double*, int)
  0.00      0.86     0.00        2     0.00     0.00  readWavFileHeader(int*, int*, _IO_FILE*)
  0.00      0.86     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z11readWavFilePiS_Pc
  0.00      0.86     0.00        1     0.00     0.00  writeWavFileHeader(int, int, double, _IO_FILE*)
```

Guitar_trim.wav time

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.25      0.06     0.06                             fft(double*, int, int)
  0.00      0.06     0.00        2     0.00     0.00  readWavFileHeader(int*, int*, _IO_FILE*)
  0.00      0.06     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z11readWavFilePiS_Pc
  0.00      0.06     0.00        1     0.00     0.00  writeWavFileHeader(int, int, double, _IO_FILE*)
```

Here we can see that there is a slight time increase after optimization. Usually code tunings are very minor time savers so therefore these small time increases are expected.

Optimization 5: Strength Reduction:

This is also a code tuning. In this optimization we replace taxing mathematical operations with less taxing operations while preserving the output correctness in an attempt to reduce execution time. In this case when we are entrywise multiplying the two frequency arrays we do originally in for multiplications. But with this optimization I have used a different algorithm approach that reduces the multiplications from four to three. Since we are running this multiplication algorithm over very large array sizes we can save sometimes by even just reducing one taxing mathematical operation.

Code Snippet:

```
// Code from TA
void multiply(const double *array1, const double *array2, double *outputArray, int arraySize) {
-
+    double a, b, c, d;
+    int re, im;
    // iterate over the complex entries, up to n = arraySize/2
    for (int i = 0; i < arraySize; i+=2) {
-        int re = i;    // index of the real parts
-        int im = i+1;    // index of the imaginary parts
-
-        outputArray[re] = array1[re]*array2[re] - array1[im]*array2[im]; // compute the real part of
 the output entry
-        outputArray[im] = array1[re]*array2[im] + array1[im]*array2[re]; // Compute the imaginary part
 of the output entry
+        re = i;    // index of the real parts
+        im = i+1;    // index of the imaginary parts
+        // OPTIMIZATION 3: Strength Reduction; change 4 step multiplication to 3 step
+        a = array1[re];
+        b = array1[im];
+
+        c = array2[re];
+        d = array2[im];

        // 1. (a + b) * (c + d)
+        double one = (a + b) * (c + d);
+

        // 2. a * c
+        double two = a * c;
+

        // 3. b * d
+        double three = b * d;
+

        // to get (ac - bd) -> step 2 - step 3
-        // to get i(bc + ad) -> step 1 - step 2
+        outputArray[re] = two - three;

+        // to get i(bc + ad) -> step 1 - step 2 - step 3
+        outputArray[im] = one - two  - three;
        // a = array1[re]
        // b = array1[im]
```

The following is the code snippet and it shows the lines added and deleted to change the entrywise multiplication from four multiplications to three. The algorithm is shown in the comments of the code snippet and is directly followed from Professor Janet's explanation in class.

Regression testing:

I do a regression test in two different ways. Firstly, I use the "cmp" function to compare the new output sound file from the optimized version and the original output sound file from the base program. Moreover, I also listen to the new output sound file and make sure that sounds exactly like the original sound output file. Below is the code snippet of me using the compare function:

```
ahad.hamirani@ms176-3eb:~/CS501$ cmp output.wav opt3_dry_output.wav
ahad.hamirani@ms176-3eb:~/CS501$ cmp trim_base_output.wav opt3_trim_output.wav
ahad.hamirani@ms176-3eb:~/CS501$
```

Here we can see that the cmp function has not returned any differences so we have successfully optimized using strength reduction technique while preserving correctness.

GPROF timing:

Guitar_dry.wav time

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 97.78      0.79     0.79                             fft(double*, int, int)
  1.24      0.80     0.01                             writeWavFile(double*, int, int, char*)
  1.24      0.81     0.01                             multiply(double const*, double const*, double*, int)
  0.00      0.81     0.00        2     0.00     0.00  readWavFileHeader(int*, int*, _IO_FILE*)
  0.00      0.81     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z11readWavFilePiS_Pc
  0.00      0.81     0.00        1     0.00     0.00  writeWavFileHeader(int, int, double, _IO_FILE*)
```

Guitar_trim.wav time

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.25      0.05     0.05                             fft(double*, int, int)
  0.00      0.05     0.00        2     0.00     0.00  readWavFileHeader(int*, int*, _IO_FILE*)
  0.00      0.05     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z11readWavFilePiS_Pc
  0.00      0.05     0.00        1     0.00     0.00  writeWavFileHeader(int, int, double, _IO_FILE*)
```

Here we can see that there is a slight improvement in times from the previous optimization. Once again code tunings are very slight optimizations and have a slight effect on overall execution time especially when the program is already so efficient.

Optimization 6: Jamming:

This is the fourth and the final code tuning optimization. This is a basic optimization in which we have two for loops that are both iterating different arrays for the same amount of times and doing similar things to the arrays inside the loops. Here we can use the jamming technique and put both of those calls inside one for loop. Since the times of iteration were

the same in both the for loops the outputs of these arrays once jammed in one for loop are gonna preserve correctness.

Code snippet:

```
+     // OPTIMIZATION 4: Code Jamming
      for(int i = 0; i < freqSize; i++){
          inputFreq[i] = 0.0;
-     }
-
-     for(int i = 0; i < freqSize; i++){
          impulseFreq[i] = 0.0;
      }
+
+     // for(int i = 0; i < freqSize; i++){
+     //     impulseFreq[i] = 0.0;
+     // }
```

This is the code snippet showing the implementation of jamming in our case. It is very simple and self explanatory. We just move impulseFreq[i] = 0.0; into the upper for loop.

Regression testing:

I do a regression test in two different ways. Firstly, I use the “cmp” function to compare the new output sound file from the optimized version and the original output sound file from the base program. Moreover, I also listen to the new output sound file and make sure that sounds exactly like the original sound output file. Below is the code snippet of me using the compare function:

```
ahad.hamirani@ms176-3eb:~/CS501$ cmp output.wav opt4_dry_output.wav
ahad.hamirani@ms176-3eb:~/CS501$ cmp trim_base_output.wav opt4_trim_output.wav
ahad.hamirani@ms176-3eb:~/CS501$
```

Here we can see that the cmp function has not returned any differences so we have successfully optimized using Jamming technique while preserving correctness.

GPROF timing:

Guitar_dry.wav time

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 97.20      0.80     0.80                             fft(double*, int, int)
  1.22      0.81     0.01                             writeWavFile(double*, int, int, char*)
  1.22      0.82     0.01                             multiply(double const*, double const*, double*, int)
  0.61      0.82     0.01                             freadShortLSB(_IO_FILE*)
  0.00      0.82     0.00        2     0.00     0.00  readWavFileHeader(int*, int*, _IO_FILE*)
  0.00      0.82     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z11readWavFilePiS_Pc
  0.00      0.82     0.00        1     0.00     0.00  writeWavFileHeader(int, int, double, _IO_FILE*)
```

Guitar_trim.wav time

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.25      0.04     0.04                             fft(double*, int, int)
  0.00      0.04     0.00        2     0.00     0.00  readWavFileHeader(int*, int*, _IO_FILE*)
  0.00      0.04     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z11readWavFilePiS_Pc
  0.00      0.04     0.00        1     0.00     0.00  writeWavFileHeader(int, int, double, _IO_FILE*)
```

Here we see that the time barely changes and in one case is even a little higher than the previous optimization time. This is because this code tuning is very basic and does not improve the overall time by much. And therefore the randomness of the output time is greater than the time difference made by this optimization and therefore the time is the same or slightly higher than the previous one.