Advanced Computer Architecture

Mahsa Rezaei Firuzkuhi

Problem Description:

The project has three goals. First, we need to identify the most time-consuming functions in the given code in project and four sections of interest (function) is specified. The given source code is for analyzing the logging data of a communication library (files minmax.c and minmax.h). In order to achieve this goal, we need to perform a performance analysis of the code using the "gprof" tool to quantify the time spent in each function, and to identify the most time-consuming functions. Second goal of the project is performing the branch instruction performance analysis for different branch instruction and different datasets. We need to measure the number of branch instructions (PAPI_BR_INS) and the number of mispredictions (PAPI_BR_MSP) for three specific branches instruction which are in three different functions. In order to achieve this goal, I reported several branch instruction performance parameters using PAPI library present in C. The third goal of the project is using the gcc extensions "likely()" and "unlikely()" in order to reduce the number of branch mispredictions for the same three branches analyzed in part two of project. In order to achieve this branch optimization, we need to rewrite the three branches (if statements) including these gcc extensions (in the branch condition section) and re-running the code. we need to perform the measurement of the same PAPI events (captured in part two) in the presence of the two gcc macros "Likely()" and "Unlikely()" and compare these parameters with the same parameters extracted in part two and as a result, observe the effect of this gcc macros on branch performance and potentially to find the relationship between the performance analysis of the code using the "gprof" (identify the most time-consuming functions and sorting them based on this parameter) and the effectiveness of the "Likely()" and "Unlikely()" macros on branch performance. In other word, to observe that for each specific branch instruction located in one of the functions (with different time consumption) if the branch performance improvement using the gcc macros are effective. Another goal is to observe the effect of the data set size (input size of the code) on the branch performance parameters in part two and performance improvements in part three. So, we need to compare the branch performance parameters for the two data sets (input sizes 64 and 512) of the project for each branch instruction.

Solusion Strategy:

Part 1:

First, we need to identify the most time-consuming functions in the given code in project and four sections of interest (function) is specified. The given source code is for analyzing the logging data of a communication library (files minmax.c and minmax.h). In order to achieve this goal, we need to perform a performance analysis of the code using the "gprof" tool to quantify the time spent in each function, and to identify the most time-consuming functions.

Flat Profile:

Gprof can produce several different output styles, the most important of which is the Flat Profile shows the total amount of time the program spent executing each function. The table 1 shows a sample of a flat profile for the project program with data set size of 64. Here is what the fields in each line mean:

Column of flat profile	description
% time	This is the percentage of the total execution time your program spent in this function. These should all add up to 100%.
cumulative seconds	This is the cumulative total number of seconds the computer spent executing this function, plus the time spent in all the functions above this one in this table.
self seconds	This is the number of seconds accounted for by this function alone. The flat profile listing is sorted first by this number
Calls	This is the total number of times the function was called.
self ms/call	his represents the average number of milliseconds spent in this function per call, if this function is profiled.
total ms/call	This represents the average number of milliseconds spent in this function and its descendants per call, if this function is profiled.
Name	This is the name of the function. The flat profile is sorted by this field alphabetically after the self seconds and calls fields are sorted.

minmax.c | Analysis_64.txt

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 44.32      0.97     0.97      999   971.57   971.57  minmax_read_input
 20.56      1.42     0.45      999   450.73   490.79  minmax_calc_statistics
 16.45      1.78     0.36                             tcompare
 12.79      2.06     0.28      999   280.45   280.45  minmax_filter_timings
  1.83      2.10     0.04      999    40.06    40.06  minmax_clear_poison_field
  1.37      2.13     0.03      999    30.05    30.05  minmax_finalize
  1.37      2.16     0.03      999    30.05    30.05  minmax_init
  0.91      2.18     0.02      999    20.03    20.03  minmax_calc_decision
  0.46      2.19     0.01                             minmax_calc_per_iteration
```

Table 1. Sample Flat profile extracted from gprof for the program with the data sets size of 64.

The Call Graph

The call graph is the other important part of the report using gropf tool shows how much time was spent in each function and its children. From this information, we can find functions that, while they themselves may not have used much time, called other functions that did use unusual amounts of time. We need to quantify the time spent in each function and identify the most time-consuming functions. The required information about time spend in each function and the most time-consuming function can be extracted from the sample call graphs in the gprof report. Here is a sample call graph from the program with the data set size of 64 in table 2.

minmax.c | Analysis_64.txt | Analysis_512.txt

```
53
54  index % time    self  children    called     name
55                                                <spontaneous>
56  [1]     83.1    0.00    1.82                 main [1]
57                  0.97    0.00     999/999         minmax_read_input [2]
58                  0.45    0.04     999/999         minmax_calc_statistics [3]
59                  0.28    0.00     999/999         minmax_filter_timings [5]
60                  0.03    0.00     999/999         minmax_init [8]
61                  0.03    0.00     999/999         minmax_finalize [7]
62                  0.02    0.00     999/999         minmax_calc_decision [9]
63  -----------------------------------------------
64                  0.97    0.00     999/999         main [1]
65  [2]     44.3    0.97    0.00     999         minmax_read_input [2]
66  -----------------------------------------------
67                  0.45    0.04     999/999         main [1]
68  [3]     22.4    0.45    0.04     999         minmax_calc_statistics [3]
69                  0.04    0.00     999/999         minmax_clear_poison_field [6]
70  -----------------------------------------------
71                                                <spontaneous>
72  [4]     16.4    0.36    0.00                 tcompare [4]
73  -----------------------------------------------
74                  0.28    0.00     999/999         main [1]
75  [5]     12.8    0.28    0.00     999         minmax_filter_timings [5]
76  -----------------------------------------------
77                  0.04    0.00     999/999         minmax_calc_statistics [3]
78  [6]      1.8    0.04    0.00     999         minmax_clear_poison_field [6]
79  -----------------------------------------------
80                  0.03    0.00     999/999         main [1]
81  [7]      1.4    0.03    0.00     999         minmax_finalize [7]
82  -----------------------------------------------
83                  0.03    0.00     999/999         main [1]
84  [8]      1.4    0.03    0.00     999         minmax_init [8]
85  -----------------------------------------------
86                  0.02    0.00     999/999         main [1]
87  [9]      0.9    0.02    0.00     999         minmax_calc_decision [9]
88  -----------------------------------------------
89                                                <spontaneous>
90  [10]     0.5    0.01    0.00                 minmax_calc_per_iteration [10]
91  -----------------------------------------------
92
```

Table 2. Sample Call Graph extracted from gprof for the program with the data sets size of 64.

The lines full of dashes divide this table into entries, one for each function. Each entry has one or more lines. In each entry, the primary line is the one that starts with an index number in square brackets. The end of this line says which function the entry is for. The preceding lines in the entry describe the callers of this function and the following lines describe its subroutines (also called children when we speak of the call graph). The entries are sorted by time spent in the function and its subroutines.

Part 2:

Second goal of the project is performing the branch instruction performance analysis for different branch instruction and different datasets. We need to measure the number of branch instructions (PAPI_BR_INS) and the number of mispredictions (PAPI_BR_MSP) for three specific branches instruction which are in three different functions. In order to achieve this goal, I reported several branch instruction performance parameters using PAPI library present in C. In order to Instrument the provided code to use PAPI hardware performance counters to determine the behavior of each branch instruction separately, I have been using the PAPI - Performance API that is provided with certain number of hardware counters offered by the cluster used to compute operations like Branch Mispredictions and various other performance related measures. This information regarding the available counters and PAPI events that could be used to accomplish the task could be viewed by "PAPI_avail" command. I need to understand the behavior of different branch instructions when running with different data sets. This will help in analyzing the behavior of branch instructions when applying the Likely and unlikely macros. The PAPI counters used to measure the attributes are listed in table 3.

Metric	Description
PAPI_BR_UCN	Unconditional branch instructions completed
PAPI_BR_CN	Conditional branch instructions completed
PAPI_BR_TKN	Conditional branch instructions taken
PAPI_BR_NTK	Conditional branch instructions not taken
PAPI_BR_MSP	Conditional branch instructions mispredicted
PAPI_BR_INS	Total branch instructions completed

Table 3. Measured metrics used in the implementation.

I am running each branch instructions analysis (for each of the three branch instructions in functions) by reporting the following parameters in the program.

1- Total Branch instructions
2- Total Conditional Branch instructions
3- Total Conditional Branch instructions Mispredicted
4- Total Branch instructions Taken
5- Total Branch instructions Not Taken
6- Branch Misprediction Rate

As a result, I had defined one event sets containing all the PAPI events mentioned above, at the beginning of each of the functions that contains those three specific branch instructions at lines 181, 224 and 319 and added the counters to the event set using the "PAPI_add_event" function.

The if/else statements, switch/case statements, while loops and for loops all conditionally execute code depending on some test. To sequentially execute instructions, the program counter increments by 4 after each instruction. Branch instructions modify the program counter to skip over sections of code or to repeat previous code. Conditional branch instructions perform a test and branch only if the test is TRUE. Unconditional branch instructions, called jumps, always branch. a short IF-THEN-ELSE program sequence that normally requires one conditional branch and an unconditional branch. the value of the parameter "Total branch instructions completed" is exactly twice the value of the parameter "Conditional branch instructions completed".

In part two of project, we need to measure the number of branch instructions (PAPI_BR_INS) and the number of mispredictions (PAPI_BR_MSP) for different data sets. Ratios derived from a combination of hardware events can provide more useful information than raw metrics. For example, in order to be able to compare the branch performance results of each branch for the two data sets, we need to compare the branch misprediction rates. Misprediction rates can be calculated by dividing the number of mispredicted branch instructions by the total number of the branch instruction. Since in the project, we are required to report the number of branch instructions (PAPI_BR_INS) and the number of mispredictions (PAPI_BR_MSP) while the PAPI event (PAPI_BR_MSP) is the "Conditional branch instructions mispredicted", as described in table 1, I prefer to divide this parameter by the "Conditional branch instructions completed" in order to get the branch misprediction rate. As a result, I can easily deduce the conditional branch misprediction rate from the above information.

$$\text{conditional branch misprediction rate} = \frac{\text{Conditional branch instructions mispredicted}}{\text{Conditional branch instructions completed}}$$

Part 3:

The third goal of the project is using the gcc extensions "likely()" and "unlikely()" in order to reduce the number of branch mispredictions for the same three branches analyzed in part two of project. In order to achieve this branch optimization, we need to rewrite the three branches (if statements) including these gcc extensions (in the branch condition section) and re-running the code. we need to perform the measurement of the same PAPI events (captured in part two) in the

presence of the two gcc macros “Likely()” and “Unlikely()” and compare these parameters with the same parameters extracted in part two and as a result, observe the effect of this gcc macros on branch performance

Letting an instruction move from the instruction decode stage of the pipeline into the execution stage is called instruction issue. An instruction is completed once all logically previous instructions have completed, and only then is its result added to the visible state of the CPU. Because of speculative execution, a mispredicted branch can cause instructions that have been executed but not completed to be discarded. Resource contention can cause instructions to be issued more than once before being completed. Normally branch mispredictions and reissues are rare. A high number of mispredicted branches (PAPI_BR_MSP) indicates that something is wrong with the compiler options or that something is unusual about the algorithm. In Linux kernel code, we often find calls to likely() and unlikely(), in conditions, like :

```
if (likely(!condition)) {
  do something!;
else{
  do something else!
}
```

In fact, these functions are hints for the compiler that allows it to correctly optimize the branch, by knowing which is the likeliest one. In other word, they are hint to the compiler to emit instructions that will cause branch prediction to favour the "likely" side of a jump instruction. This can be a big win, if the prediction is correct it means that the jump instruction is basically free and will take zero cycles. The definitions of these macros are the following :

```
#define likely(x)       __builtin_expect(!!(x), 1)
#define unlikely(x)     __builtin_expect(!!(x), 0)
```

One of the most used optimization techniques in the Linux kernel is “__builtin_expect”. When working with conditional code (if-else statements), we often know which branch is true and which is not. If compiler knows this information in advance, it can generate most optimized code. In the above example, we are marking branch as likely true. We should use it only in cases when the likeliest branch is most likely, or when the unlikeliest branch is most unlikely. For above example, we have marked “if” condition as “likely()” true, so compiler will put true code immediately after branch, and false code within the branch instruction. In this way compiler can achieve optimization. We should not use “likely()” and “unlikely()” macros blindly. If prediction is correct, it means there is zero cycle of jump instruction, but if prediction is wrong, then it will take several cycles, because processor needs to flush it’s pipeline which is worse than no prediction.

Results

Description of Resources:

We are using the slurm cluster for conducting this experiment. The hardware information that was used when the experiment was conducted are as follows:

PAPI version: 5.7.0.0 Operating system: Linux 4.4.138-59-default Vendor string and code: Genuine Intel (1, 0x1) Model string and code: Intel(R) Xeon(R) CPU E5-2665 0 @ 2.40GHz (45, 0x2d) CPU revision: 7.000000 CPUID: Family/Model/Stepping 6/45/7, 0x06/0x2d/0x07 CPU Max MHz: 2400 CPU Min MHz: 1200 Total cores: 32 SMT threads per core: 2 Cores per socket: 8	Sockets: 2 Cores per NUMA region: 16 NUMA regions: 2 Running in a VM: no Number Hardware Counters: 10 Max Multiplex Counters: 384 Virtualization: VT-x L1d cache: 32K L1i cache: 32K L2 cache: 256K L3 cache: 20480K NUMA node0 CPU(s): 0-7,16-23 NUMA node1 CPU(s): 8-15,24-31

Table 4. Hardware Configuration

Measurements

For all the statistic measurements in this project, I have run the program three times and have reported the average of those information in all the tables and plots.

Part 1:

minmax.c | Analysis_64.txt

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 44.32      0.97     0.97      999   971.57   971.57  minmax_read_input
 20.56      1.42     0.45      999   450.73   490.79  minmax_calc_statistics
 16.45      1.78     0.36                             tcompare
 12.79      2.06     0.28      999   280.45   280.45  minmax_filter_timings
  1.83      2.10     0.04      999    40.06    40.06  minmax_clear_poison_field
  1.37      2.13     0.03      999    30.05    30.05  minmax_finalize
  1.37      2.16     0.03      999    30.05    30.05  minmax_init
  0.91      2.18     0.02      999    20.03    20.03  minmax_calc_decision
  0.46      2.19     0.01                             minmax_calc_per_iteration
```

minmax.c | Analysis_64.txt | Analysis_512.txt

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 50.25      9.14     9.14      999     9.14     9.14  minmax_read_input
 18.16     12.44     3.30      999     3.31     3.31  minmax_filter_timings
 14.31     15.04     2.60      999     2.60     3.13  minmax_calc_statistics
 12.27     17.27     2.23                             tcompare
  2.86     17.79     0.52      999     0.52     0.52  minmax_clear_poison_field
  1.05     17.98     0.19                             minmax_calc_per_iteration
  0.50     18.07     0.09      999     0.09     0.09  minmax_init
  0.39     18.14     0.07      999     0.07     0.07  minmax_finalize
  0.28     18.19     0.05      999     0.05     0.05  minmax_calc_decision
```

Table 5. Sample Flat profile extracted from gprof for the program with the data sets size of 64 and 512 respectively.

The code contains four sections (functions) of interest:

– minmax_read_input: read the input files

– minmax_filter_timings: look for outliers in the performance data

– minmax_calc_decision: determine the fastest communication method

– minmax_calc_statistics: calculate some internal statistics on the data

function	Time spend in function(sec)_64	%time_64	Time spend in function(sec)_ 512	%time_512
minmax_read_input	0.97	44.3	9.14	50.2
minmax_filter_timings	0.28	12.8	3.30	18.2
minmax_calc_decision	0.02	0.9	0.05	0.3
minmax_calc_statistics	0.45	22.4	2.60	17.2

Table 3. The time spent in each function and the most time-consuming functions from call graph for the four functions we are interested in.

The call graph in the report of the gprof tool shows how much time was spent in each function and its children. From this information, we can find functions that, while they themselves may not have used much time, called other functions that did use unusual amounts of time. We need to quantify the time spent in each function and identify the most time-consuming functions. The required information about time spend in each function and the most time-consuming function extracted from the sample call graphs from program with two data sets size 64 and 512 in table 2. As a result, function “minmax_read_input” is the most time-consuming function in both cases of data set 64 and 512 with the percentage of the execution time of 44.3% and 50.2% respectively. In the second order, we have the function “minmax_calc_statistics” for the 64 data set as the second most time-consuming and the function “minmax_filter_timings” for the 512 data set as the second most time-consuming.

Part 2 and Part 3:

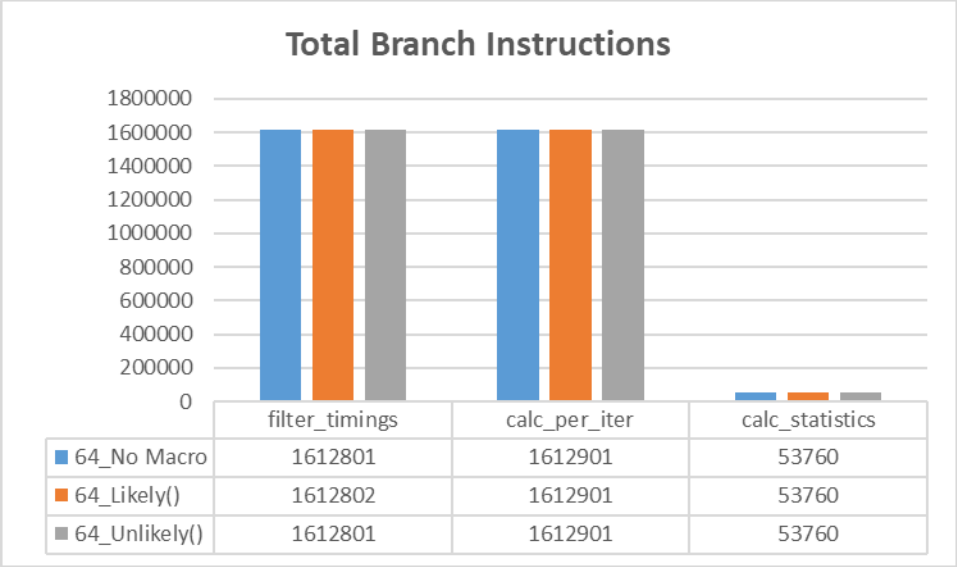

Figure. 1. Averaged total number of branch instructions comparison for the three branch instructions in the functions (minmax_filter_timings, minmax_calc_per_iteration and minmax_calc_statistics) for the actual code, the code optimized with Likely() and the code optimized with unlikely() and 64 dataset.

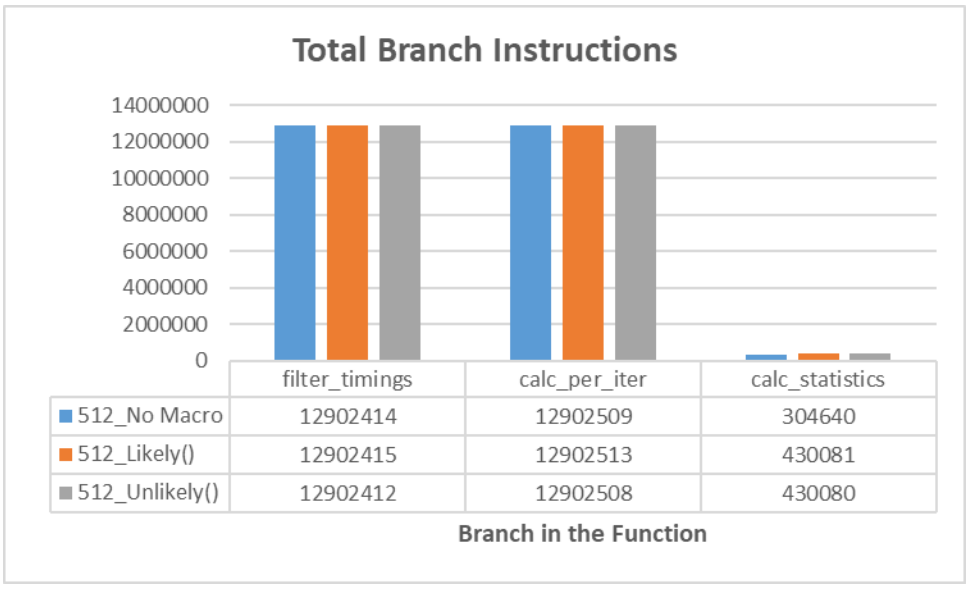

Figure. 2. Averaged total number of branch instructions comparison for the three branch instructions in the functions (minmax_filter_timings, minmax_calc_per_iteration and minmax_calc_statistics) for the actual code, the code optimized with Likely() and the code optimized with unlikely() and 512 dataset.

Total Branch instructions:

We compare the total number of branch instructions comparison for the three branch instructions in the functions (minmax_filter_timings, minmax_calc_per_iteration and minmax_calc_statistics) for the actual code, the code optimized with Likely() and the code optimized with unlikely() and 64 dataset. I see almost similar number of branch instructions counted for each of the branches in three function. Even adding the Likely() or Likely() macros did not affect the equal number of total branch instruction for the 64 data set (figure 1). We observe a very different situation for the total branch instructions in case of the 512 data set in figure 2. In this figure we see that, with using the Likely/Unlikely macros the total number of branch instruction will increase for the branch in function "minmax_calc_statistics". If we check the last column of the table in this figure 2, we observe that the total branches increase by the value of (125441) for the both case of adding Likely() and Unlikely.

Conditional Branch Mispredictions:

Takeaways:

1- Dataset 64

We compare the total number of branch mispredictions for the three branch instructions in the functions (minmax_filter_timings, minmax_calc_per_iteration and minmax_calc_statistics) for the actual code, the code optimized with Likely() and the code optimized with unlikely() and 64 dataset in figure 3.

1- For the branch instruction in function filter_timing (first column of table in figure 3) in both case of adding likely and unlikely, the number of mispredictions decreased but the reduction in the case of adding the UnLikely() macro is much more (about 0.25% reduction).
2- For the branch instruction in function calc_per_iter (second column of table in figure 3) in case of adding likely we see no change and adding unlikely had increased the number of mispredictions.
3- For the branch instruction in function calc_statistics (third column of table in figure 3) in case of adding likely and unlikely we see increase in the number of mispredictions.

2- Dataset 512

We compare the total number of branch mispredictions for the three branch instructions in the functions (minmax_filter_timings, minmax_calc_per_iteration and minmax_calc_statistics) for the actual code, the code optimized with Likely() and the code optimized with unlikely() and 512 dataset in figure 4.

1- For the branch instruction in function filter_timing (first column of table in figure 4) in both case of adding likely and unlikely, the number of mispredictions decreased but the

reduction in the case of adding the UnLikely() macro is much more. So has a similar pattern with the data set size 64.

2- For the branch instruction in function calc_per_iter (second column of table in figure 4) in case of adding likely we see increase and in case of adding unlikely we see decrease in the number of mispredictions. In order to be able to compare the first two branch instructions better, I plotted the information of the only these two branches in a separate figure 5.

3- For the branch instruction in function calc_statistics (third column of table in figure 4) in case of adding likely and unlikely we see increase in the number of mispredictions (better to be observed from figure 6).

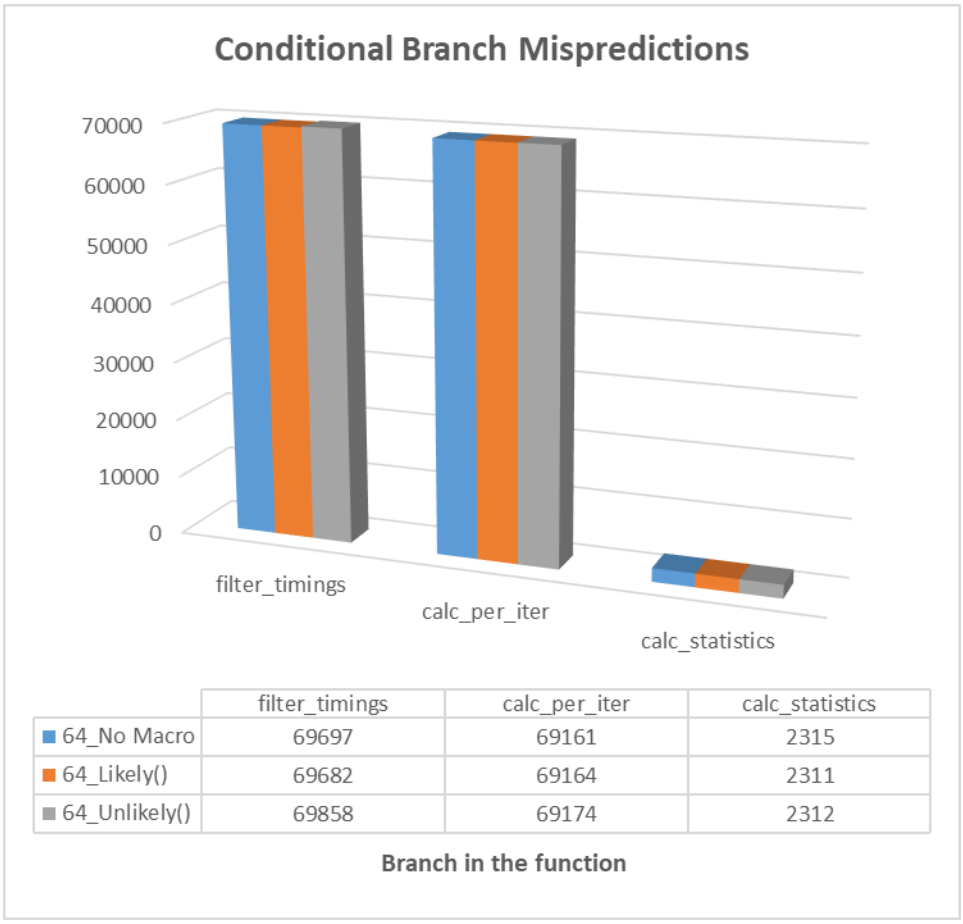

Figure. 3. Averaged total number of branch misprediction comparison for the three branch instructions in the functions (minmax_filter_timings, minmax_calc_per_iteration and minmax_calc_statistics) for the actual code, the code optimized with Likely() and the code optimized with unlikely() and 64 dataset.

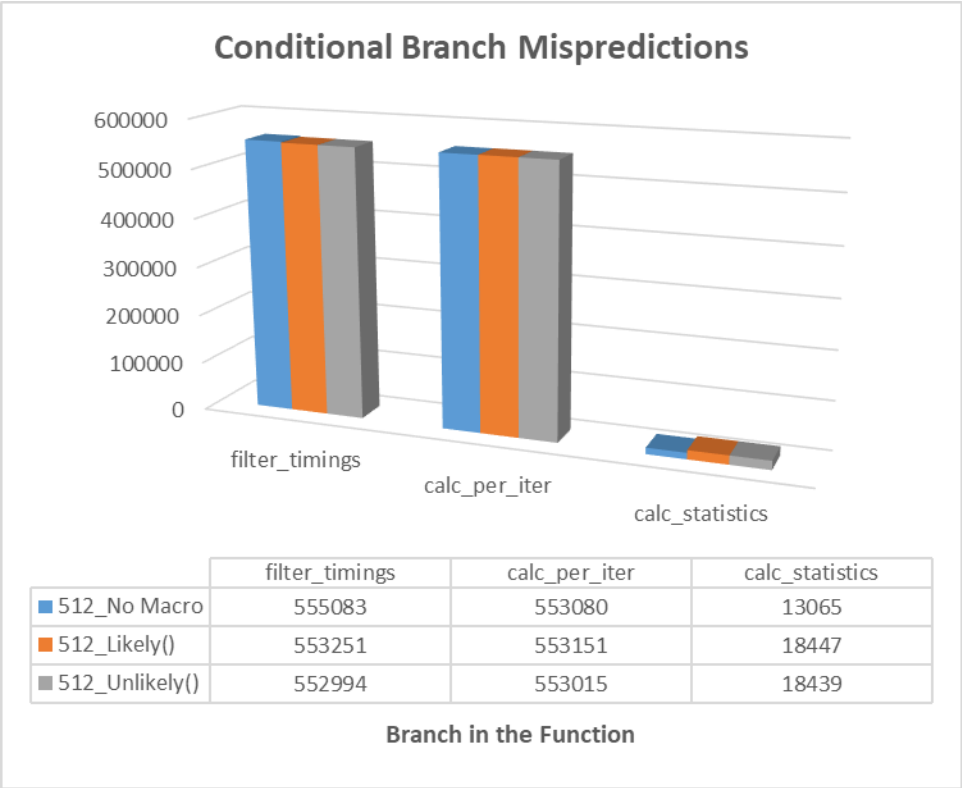

Figure. 4. Averaged total number of branch misprediction comparison for the three branch instructions in the functions minmax_filter_timings, minmax_calc_per_iteration and minmax_calc_statistics) for the actual code, the code optimized with Likely() and the code optimized with unlikely() and 512 dataset.

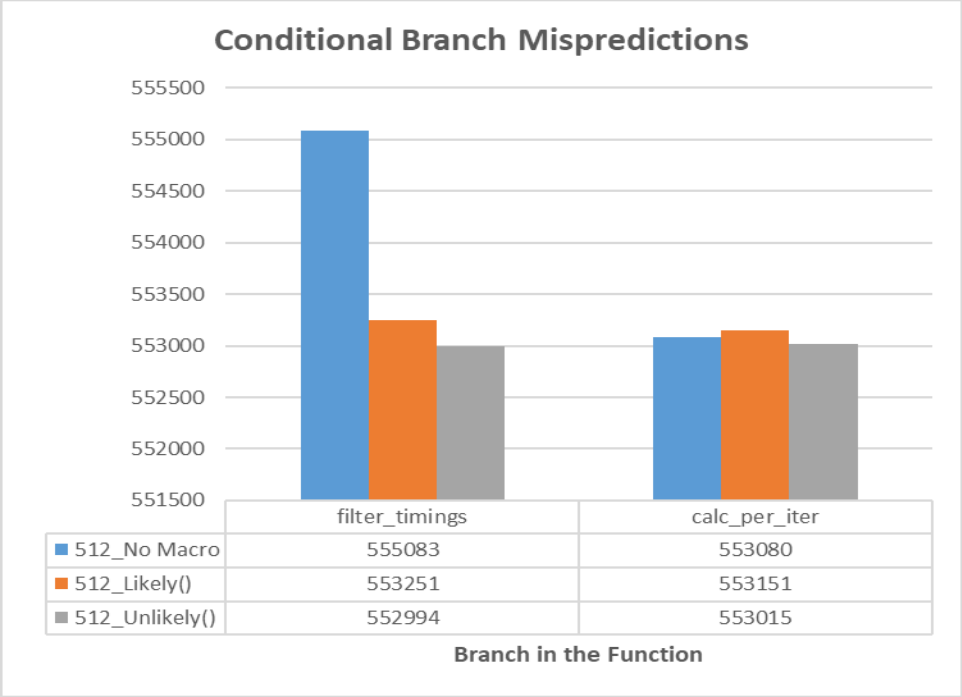

Figure. 5. Averaged total number of branch misprediction comparison for ONLY the two branch instructions in the functions minmax_filter_timiings and calc_per_iteration for the actual code, the code optimized with Likely() and the code optimized with unlikely() and 512 dataset.

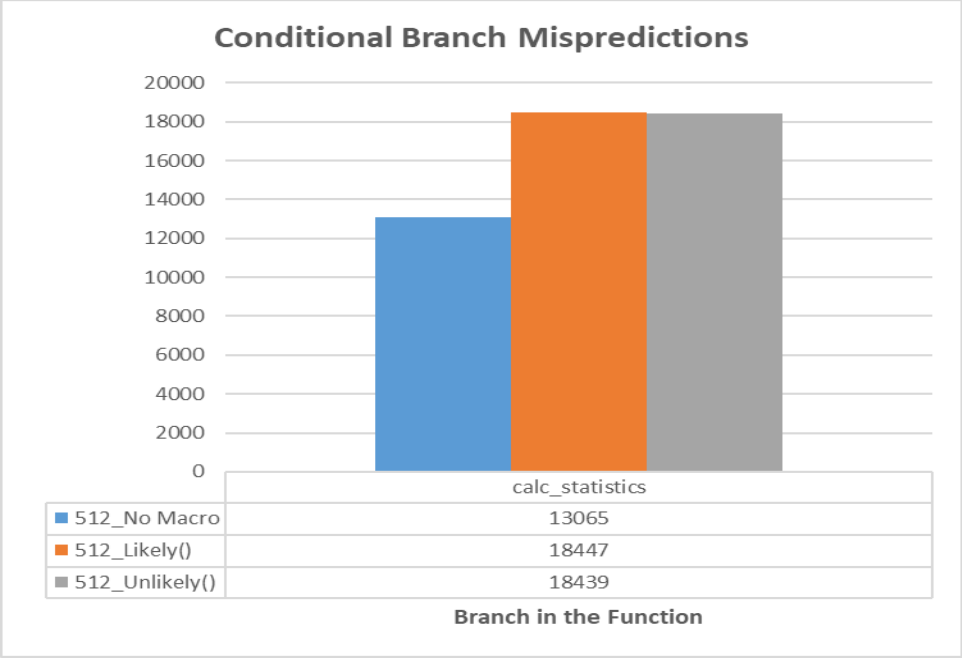

Figure. 6. Averaged total number of branch misprediction comparison for branch instructions in the function minmax_calc_statistics for the actual code, the code optimized with Likely() and the code optimized with unlikely() and 512 dataset.

Branch Miss Prediction Rates:

Takeaways:

1- **Data set 64**

If we look at the figure 7, we compared the misprediction rates of the three branches in the presence of macro for the 64 data set. We see that for the branch instruction in the "filter_timing" function adding unlikely had increased the misprediction rate by 0.02% while adding likely decreased this rate by very small amount. we see the very similar pattern in the branch instruction in function "calc_per_iter" (decrease by likely and increase by unlikely). We see reduction in the misprediction rate for the branch in function "calc_statistics" for both likely and Unlikely while the winner is the likely() macro. Overall, looking at the figure 7, we can conclude that in case of data set 64, for all three branch instruction in functions, using the likely() macro was the best option in decreasing the misprediction rates (red bars in bar chart)

2- **Data set 512**

If we look at the figure 8, we compared the misprediction rates of the three branches in the presence of macro for the 512 data set. We see that for the branch instruction in the "filter_timing" function adding unlikely had decreased the misprediction rate by maximum 0.03% while adding likely decreased this rate by 0.02%. we see the very similar reduction of 0.01% and 0.02 in the branch instruction in function "calc_statistics". We see almost zero change in the misprediction rate for the branch in function "calc_per_iter" (the likely() macro had increased it a very small amount). Overall, looking at the figure 8 we can conclude that in case of data set 512, for all three branch instruction in functions, using the Unlikely() macro was the best option in decreasing the misprediction rates (gray bars in bar chart)

3- **Both data sets**

If we focus on each of the branch instruction, for example the first one which is filter_timing, and compare the charts 7 and 8 at the same time, we see that in the baseline code (no macro added) increasing the size of data set from 64 to 512 has decreased the misprediction rate.

For the branches in the two other function we see the reduction as well. So for the baseline code, increasing the input size, improved the branch prediction performance.

If we compare the case of the adding Likely() macro, we again observe the same pattern of reduction in misprediction by increasing the data set size. If we compare the case of the adding UnLikely() macro, we again observe the same pattern of reduction in misprediction by increasing the data set size.

Figure. 7. Averaged branch misprediction rate comparison for branch instructions in the three function for the actual code, the code optimized with Likely() and the code optimized with unlikely() and 64 dataset.

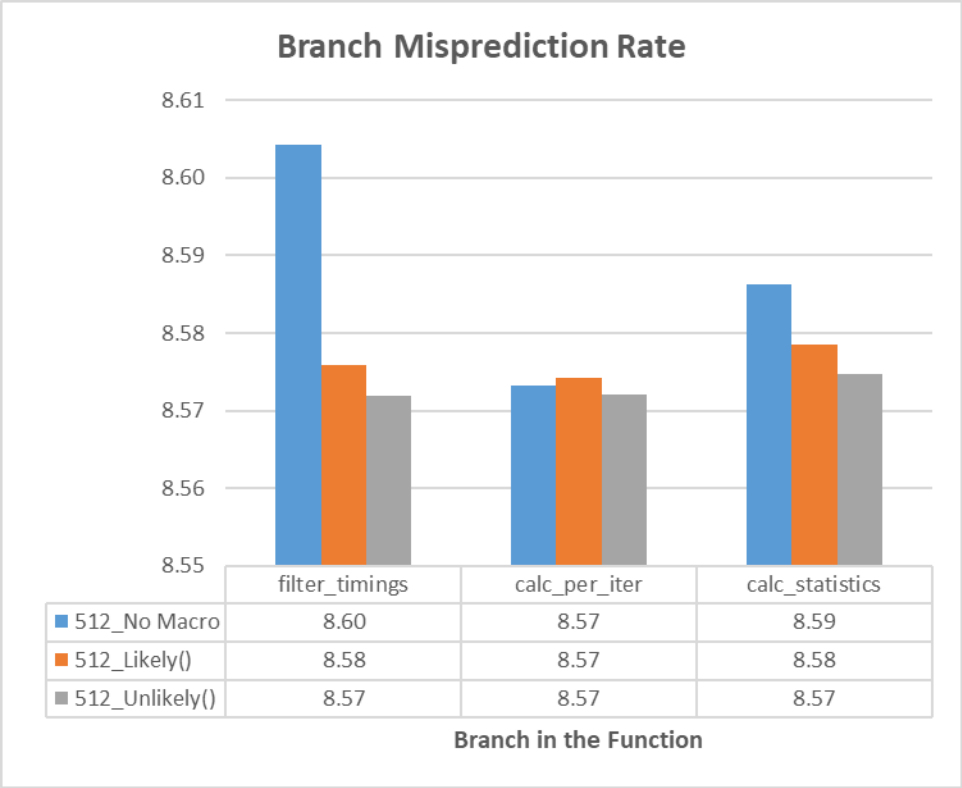

Figure. 8. Averaged branch misprediction rate comparison for branch instructions in the three function for the actual code, the code optimized with Likely() and the code optimized with unlikely() and 512 dataset.

Percentage of the Taken and NotTaken Branch instruction:

For additional observation in this project I decided to report the percentage of the taken and Not Taken branch instructions. Ratios derived from a combination of hardware events can provide more useful information than raw metrics. For example the ratios of the two PAPI metrics conditional branch instructions taken and the conditional branch instructions Not taken can help us to decide to apply the likely() or unlikely() macros in order to optimize the branch performance (reducing the branch misprediction rate). If the branch condition is frequently false and branch is "frequently Not Taken", then in that case using the Unlikely() macro should help more. For this reason, I have reported the percentage of the branch instruction that have been taken or not taken in the three function for the both datasets in a cumulative bar chart in figures 9-11. We observe that in all three figures the percentage of taken and not taken branch instruction are remaining the same and there is 40% of branches that had been taken and 60% of them had been not taken. So we should not use the "likely()" and "unlikely()" macros blindly. If prediction is correct, it means there is zero cycle of jump instruction, but if prediction is wrong, then it will take several cycles, because processor needs to flush it's pipeline which is worse than no prediction. We should use it only in cases when the likeliest branch is most likely, or when the unlikeliest branch is most unlikely. In our case, since the ratio of 40% is not very low (in other word, the 60% is not a very high percentage), adding the macros had minimum effect on branch prediction performance (misprediction rates)in all the cases and dataset sizes.

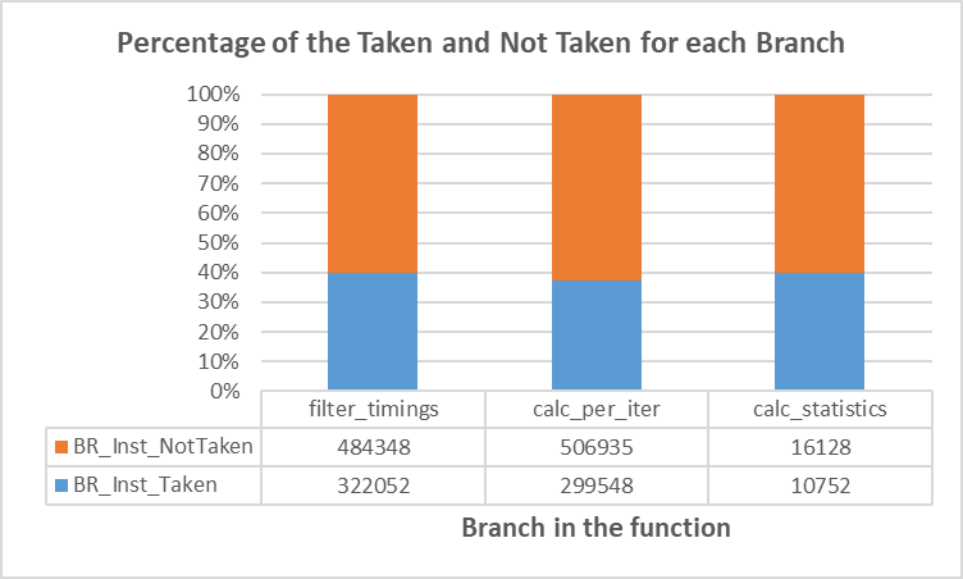

Figure. 9. Averaged percentage of the branch instruction that have been taken or not taken in the three function for the 64 dataset.

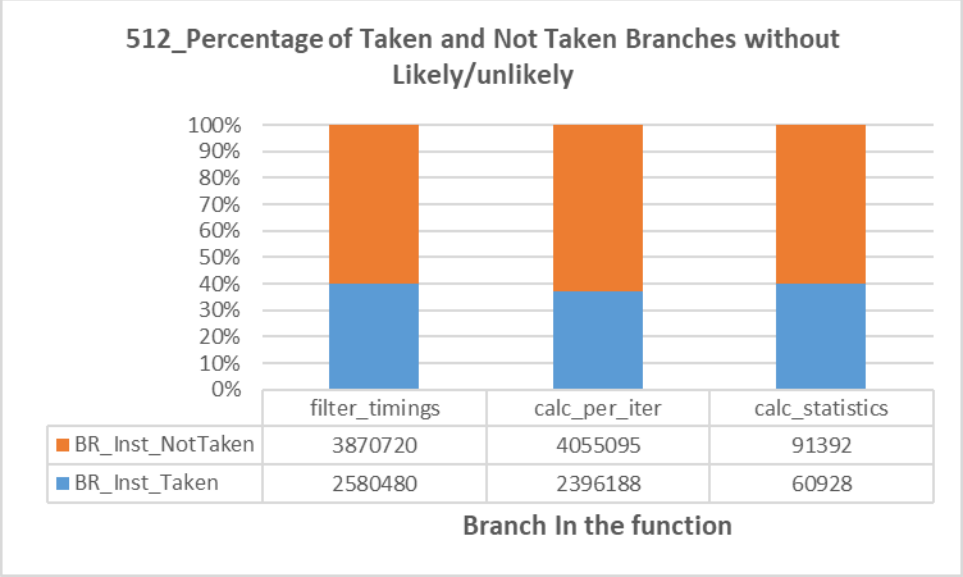

Figure. 10. Averaged percentage of the branch instruction that have been taken or not taken in the three function for the 512 dataset without adding any likely macro.

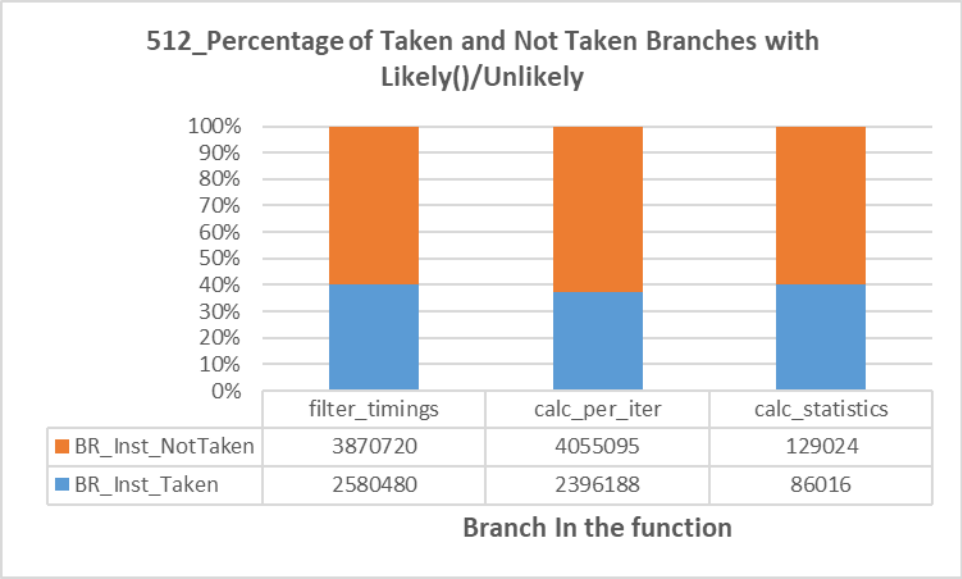

Figure. 11. Averaged percentage of the branch instruction that have been taken or not taken in the three function for the 512 dataset with likely macros