Coursework of Compiler Optimization

Yi Zhen

UUN: s1563190

Lecturer: Dr. Hugh Leather

University of Edinburgh

February 2016

Abstract

Programmers always spend a large number of time to reorganize the code and tune the compiling parameters. The aim of that is hope the program can run faster. This report will focus on the computer-aided tuning the flags of GCC 4.7(on DICE) and analyzing such parameters. We will discuss if the we can always find the best sets of flags automatically under time constraint. And I do implement an improved random search to solve this problem. This report will also mention some problems when conducting with the testing and details should be pay attention to.

1. Introduction

1.1 What will we do in this assignment

This coursework should be based on GCC compiler. The major thing we should do is that we need to find the best sets of flags of each benchmark as well as a best one appropriate all benchmarks among 200 tests. The mean of the best flags equals to the shortest running time when we use those flags. Because different combination of flags may occur the different affects of the result and usually we do not know how to select them efficiently. The problem is that we do have a method to find the best one. However, it may cost too much time by doing complete search and we may not get results until our cosmos terminating. We cannot bare exponential exponential time complexity here.

1.2 Analysis of existing algorithms

Fortunately, we have some mature algorithms and methods to solve this problem. Exactly, the reason why we should to choose some specify sets of flags not all of them is that some flags dependent each other. For example, some flags cannot work together or they will make the compiling error or slowing down the speed of execution. That is the staff make us get stuck in selection. We all imagine that computer can find a pattern automatically by strong artificial intelligence. Thus, our work is only waiting for the result. But even though we can use the most advanced machine learning technologies. The number of 200 tests is not big enough to help the learning approach. On the other hand, what about genetic algorithm or simulated annealing? Both of them may work. However, according to Xu (2014) et al. ‘We can conclude that if the population is too small, the performance of the GA-based algorithm will be similar to the random algorithm.’. Therefore, why not improve the random search directly? I tried developing a mechanism of controlling

the performance of random search which inspired by Formula 1 racing. Every flag will get its own points and that will play an important role in the performance of the whole testing.

2. Evaluation methodology: Selection of flags

2.1 The kind of flags

This is an important part of this project. We may cost no less than 40% time to study the documents. From the official GCC document, we know that the optimizing flags have the uniform formalization that the prefix is always –f. For one thing, writing a function and parsing all the optimizing flags should be done perfect. Except the optimizing flags, a part of flags which can be set a parameter should also be considered. With the help of parameters, we can tune the behavior of compiling more accurately.

2.2 How to select flags

Generally, we should put all flags and parameters in our test project. However, some special conditions of combination of flags will be mentioned. Firstly, the flags which have the formalization –fprofile should be compiled more than one times. Secondly, some flags rely on other flags. If you want to use them, you should select all the dependent flags. The only way we could know which ones are dependent is studying the documents. Thirdly, every –param flags have its own domain of value. My work is that parsing all the parameters at first and divided them into two parts. The one is –f flags and another is –param flags. The set of dependent flags should be seemed as one flag.

We should implement many random value generators for all –param flags. Sounds boring and endless. Exactly, we can do discretization for the parameters. The concrete way is that we select 5(or 10 if you like) random values of each parameter. The value must be valid and different. That means we have the random parameters and the range of each of them is 5. And if the parameter is selected, it can only be assigned one value for sure. Following is the example of 3 parameter flags with their possible values:

PARAMETERS	1ST VALUE	2ND VALUE	3RD VALUE	4TH VALUE	5TH VALUE
SRA-MAX-STRUCTURE-SIZE	0	512	1024	2048	4096
MAX-CROSSJUMP-EDGES	100	200	300	400	500
MAX-GCSE-PASSES	1	2	3	4	5

Table 1 – 3 examples of parameter flag

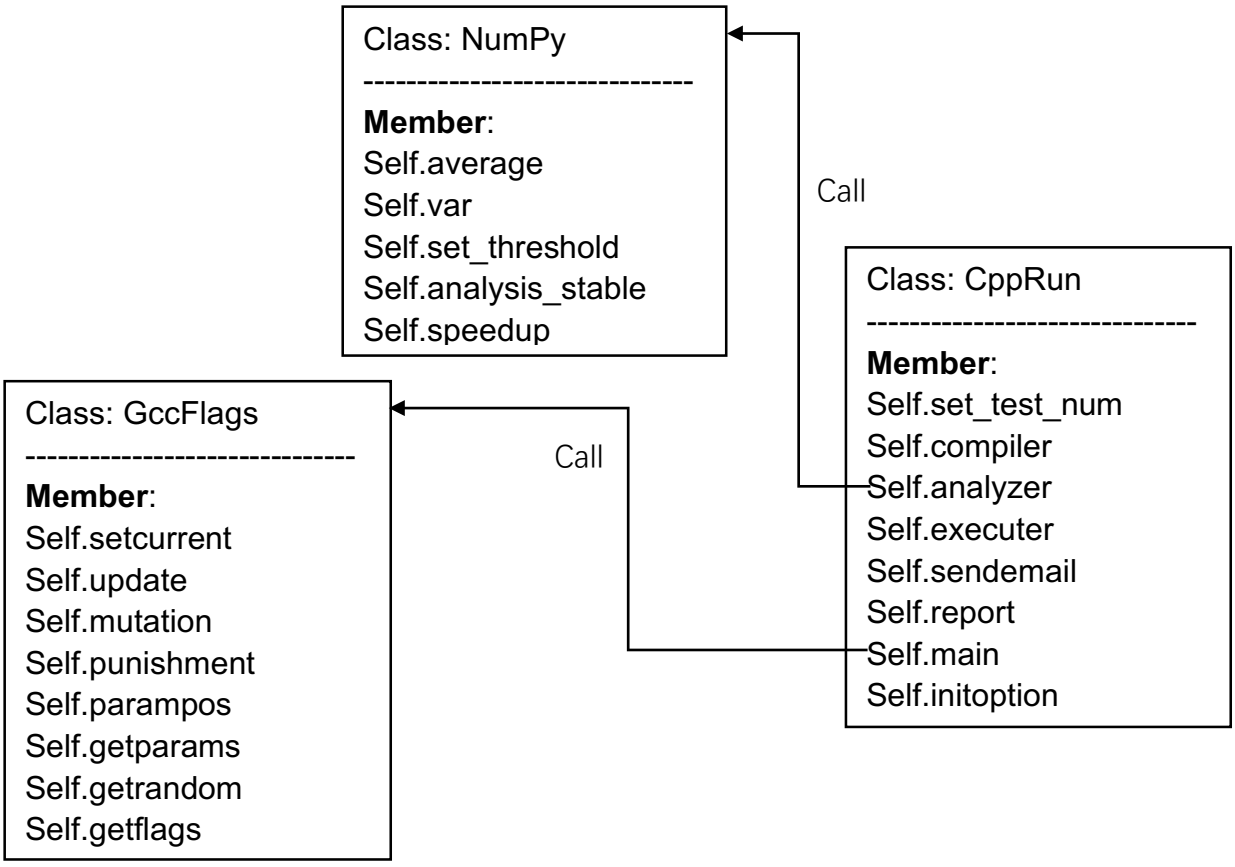
3. Experimental setup

3.1 The platform information: Please refer to appendix A

3.2 Design of program

3.2.1 The architecture of the program

I used a different order between compiling and running the benchmark in my program and the one which provided in the lecture to conduct each test. But it will not lead to different results (for more detail please refer to appendix B). Here is the brief architecture of the program:



The class CppRun is the entrance of the program. The work of it is same to

leader of a team to control everything. NumPy provides some basic tools of analysis and GccFlags is the core class which generates the optimization flags.

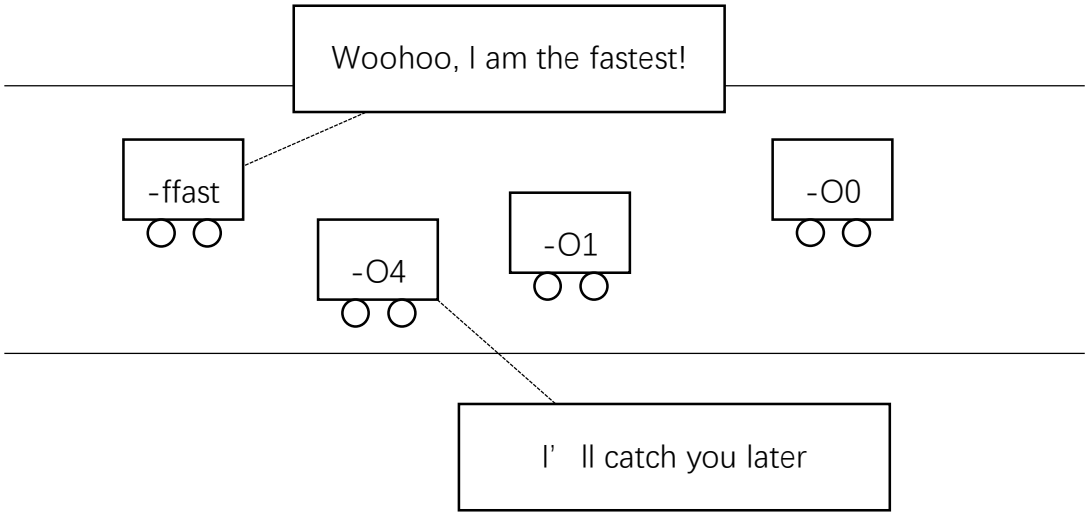
3.2.2 *The major idea of improved random search*

Each time, my program will yield a set of flags randomly. But not very random. It based on random search. However, I did more on it and hope it perform better while running. The naïve random selection method is that the random number generator yields a number from 0 to 1 for every flag. If the number bigger to a bound such as 0.5(initial probability of selection) then I will choose such flag and vice versa. This is the basic way which we may use.

-flag1	-flag2	- flag3	- flag4	- flag5	- flag6	…
0.1	0.9	0.8	0.4	0.3	0.7	…

Table 2 – the colored flags will be chosen

My idea is inspired from Grand Prix of Formula 1. If you have watched that racing, you may know that top 10 racer will get a relative point in one contest (If you do not familiar with that concept, please refer to wiki of F1). There are more than 10 contests of one racing season. The guy who accumulates most points will win that year's race.



Picture – Comic of racing of flags

That is a good idea. Imagine that We could design a mechanism and let all the flags compete with each other. First of all, we rank them by time once for every 10 successful sets of flags. The set of flags which ranks at the first place will get the highest score. Because we can think that the reason why this set of flags

performs well is that such flags have been selected. And then increase the probability of selection of each flag by referring to the total score. The top 1 flag will increase most for sure. Next time, that flag will have more probability to be chosen. Here is an example for first 3 sets of flags. We only show 3 sets this time, but actually we do 10 sets in program.

RANK	FLAGS	TIME	POINT
1	-f1 –f2 –f4 –f5	0.11111s	25
2	-f2 –f7 –f8	0.22222s	18
3	-f1 –f9 –f10	0.33333s	15

Table 3 – Top 3 sets of flags

Mutation: We rank once for every 10 runtimes. Each flag in the first place will gain 25 points and 18 points in the second place and so on (follow the F1 rule). According to table 3, we know that total points is (25 * 4 + 18 * 3 + 15 * 3) = 199. Among them, -f2 gains 43(25 + 18) points this time. It occupies 21.6% of total points. We compute all such percentage for other flags. And we will do mutation on those flags' probability of selection. Just increasing the probability of selection. But how much should be increased? Let us consider that if the initial probability of selection is 0.4(40%) for each flag and there are 100 flags. The total quantity of probability of flags is 40(0.4 * 100). The total space of increasing is 60 (100 * 1 – 40, the probability is between 0 and 1). If the program ranks 10 times, the maximum quantity of increasing each time is 6(60/10). Thus, in this example, the probability of selection of –f2 can increase 1.296(6 * 0.216). Finally, it will become 1.696. However, the upper bound is 1, thus we tune it to 1 or less than 1(such as 0.9, we can define the bounds) if the probability bigger than 1 after mutation (For more formal details, please refer to appendix B).

Punishment: The 'black hole' problem is that we choose a bad flag and increase its probability of selection. That might make the selection get worse and worse. Thus, we also need decrease the probability of selection. The opportunity of decreasing is that if we meet compiling error, we decrease the probability of selection of all flags. We can think all of them perform bad here (this is different from mutation).

3.2.3 Soundness of running

The program compute variance and mean of all runtimes and compute the coefficient of variance. That value indicates the dispersion of a frequency distribution. We can refer to that value and judge if it is reliable or not. We can

define a threshold like 20%. If the rate of dispersion bigger than 20% then it is incredible and vice versa.

3.3 Details in running

The program must be run in background because it will cost you more than 50 hours for running once. How could I monitor the states of it? A possible way is login the DICE and check it. But I implement a function which used SMTP protocol to send me emails for compiling errors, running errors and progress for proceeding 1% each time. You can find the details in my program.

4. Results and Analysis

The results of flags are too long, please refer to the appendix C.
Here just providing the general summary of the results:

4.1 Summary of baseline

The interesting result of this test is that for several benchmarks the best practice flags is not O3 but O2 even O0. However, most of the runtimes between O2 and O3 are very closed except few extreme conditions. We could always use O3 for the optimization or O2 for some benchmarks.

4.2 Summary of optimization flags

We beat O3! The mean of reciprocal of rate of speedup is less than 1.0 for the best found single set of flags across all programs. We did it by the 'Formula 1 racing' improved random search. Overall, the rates of speedup of all 200 tests are fluctuated, a part of them beat O3. Almost all the results are soundness. Some flags appear frequently such as -funroll-loops, -fgcse-las and so on. Here is the final probability of each flag. The lower bound is 0.5 and the initial probability is 0.75:

0.5 0.5 0.5 0.5 0.5 0.5 0.500443904956 0.500385372649 0.500346168339
0.5 0.5 0.500463661115 0.5 0.5 0.5 0.500430368818 0.5 0.5
0.500412790573 0.500006202353 0.5 0.5 0.500241259707 0.500439720143
0.50029623534 0.5 0.5 0.5 0.5 0.776320278073 0.500542412687 0.5 0.5 0.5
0.5 0.5 0.500345501167 0.5 0.5 0.500781127581 0.500483496204 0.5
0.500612733379 0.5 0.500010433861 0.5 0.500185536853 0.500268689075
0.5 0.500474149894 0.500462239739 0.5 0.5 0.500408789816 0.5 0.5 0.5
0.5 0.500380740573 0.500500529767 0.500156183169 0.5 0.500066227062
0.5 0.500717947565 0.5 0.5 0.500106826384 0.5 0.5 0.5 0.500565830515
0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.500229704493 0.5 0.5 0.500451678145

```
0.5 0.5 0.5 0.5 0.500228127203
```

We can see that many probabilities reach 0.5 the lower bound I defined. They are all the hot pot flags or the 'black hole' flags.

4.3 Average across all flags settings (expected random performance)

I use the mean of reciprocal rate of speedup to measure this. The value is 1.17233366271. A little bit slower than baseline.

5. Discussion of Results

5.1 Why using average rate of speedup not mean of runtimes

To judge if it faster than baseline or not, we just compute the overall rate of speedup of all benchmarks. We can do it by computing average of runtimes also. But it is not accurate enough to describe the result. Consider the average runtime of the first benchmark is 1s and 100s for the second on baseline. One optimization will make first benchmark get 0.1s and second get 50s. Another optimization will make the first benchmark get 0.9s and second get 40s. Which is better? Obviously, we cannot use arithmetic mean to deal with this question. Thus, we use the mean of rate of speedup here. It is more reasonable (rate of speedup is baseline runtime / optimization runtime).

5.2 How to deal with compiling error

In my program, I do not count compiling error. That means I will drop the flags if they make the compiling error occur. The general error is 'internal compiler error'. Maybe we should use a 64bits version compiler instead of 32bits one. But we could not recompile the compiler or to fix the bugs of it. My solution to improve the rate of success of compiling is that tune the parameters of random search. The parameter can be initial probability of selection. For example, if we make the selection of each flag get harder, we will get less total number of flags. That will reduce the occurrence of bugs.

6. Conclusion

In conclusion, I still cannot indicate which flag perform well directly although the result shows that we beat O3 in final. On the other hand, I think the results of optimization flags is not important. The really important thing is that I found a way to deal with the similar problems by doing this coursework. Maybe I can use machine learning approach to find the pattern automatically in the future. The major thought could be useful for further study in this area (I do not mention some technical details of implementation, but you can find them in program).

Reference

Xu, Yang, et al. "A Genetic Algorithm Based Multilevel Association Rules Mining for Big Datasets." Mathematical Problems in Engineering (2014).

Appendix A

Architecture	X86_64
CPU(s)	40
Thread(s) per core	2
Core(s) per socket	10
Core(s) per socket	3000.004
MemTotal	397013600kB

Appendix B - Pseudo code and Kernel

For each o **yields** in optimizations # it will be yielded online, not offline
  For each b in benchmarks
    Compile b with o
  End
  If (**all the benchmarks compiled successful**) then
    For each b in benchmarks
      Run b 10 times and record runtimes
      Analyze the runtimes: average, variance and CV
    End
End

Q: Why compile and run in different loops?

A: When I put my program on DICE to run. I found that the rate of success of compiling is only from 30% to 50%. I was sacred because I am afraid I cannot finish it before deadline. Thus, I prefer to drop the broken flags and yield a new one. If the flags cannot make all benchmarks compiled, I will not waste time to run them. Thus, I put the two thing in two loops.

Kernel functions:

```
    def mutation(self, end, step):
        """
```

```
        If you are a fan of Grands Prix of Formula 1. You must know the rule of
Scoring.
        1st place:  25
        2nd place:  18
        3rd place:  15
        4th place:  12
        5th place:  10
        6th place:  8
        7th place:  6
        8th place:  4
        9th place:  2
        10th place: 1
        :return:
        """
        decrate = len(self._flags) * 0.25 / 20.0
        point = dict()
        inc = 0
        total = 0

        out = dict()
        for i in range(end - step, end):
            out[self._current2[i][0]] = self._current2[i][1]

        for key in sorted(out.keys()):
            for k in out[key]:
                if k[1] != 'O'and k[1] != '-'\
                                    and k[0] == '-':
                    if not (k in point):
                        point[k] = self._points[inc]
                    else:
                        point[k] += self._points[inc]
                    total += self._points[inc]
            inc += 1

        for i in range(0, len(self._flags)):
            if self._flags[i] in point:
                dec = point[self._flags[i]] / total * decrate * random.random()
                if self._pos[i] - dec >= self._lower_bound:
```

```
                        self._pos[i] -= dec
                    else:
                        self._pos[i] = self._lower_bound

        return self._pos

    def punishment(self, compileflags):
        """
        If current set of flags make the compiling error, punish them. Increase the
_pos
        :return:
        """
        increate = len(self._flags) * 0.15 / 20.0
        for i in range(0, len(self._flags)):
            if self._flags[i] in compileflags:
                inc = 1.0 / len(compileflags) * increate * random.random()
                if self._pos[i] + inc <= self._upper_bound:
                    self._pos[i] += inc
                else:
                    self._pos[i] = self._upper_bound
        return self._pos
```

Appendix C

1. Coefficient of Variance:

$$V = \frac{\sigma}{E}$$

σ : standard deviation

E : mean

2. Baseline result:

Benchmark	Time	Best practice
462.libquantum	1.19975988865	CFLAGS="-O2"
h263enc	1.86563973427	CFLAGS="-O2"
mpeg2dec	0.238261008263	CFLAGS="-O0"
433.milc	15.142737484	CFLAGS="-O3"

429.mcf	2.28762185574	CFLAGS="-O3"
mpeg2enc	8.03720619678	CFLAGS="-O2"
456.hmmer	2.26343085766	CFLAGS="-O3"
h263dec	0.12042620182	CFLAGS="-O1"
464.h264ref	10.6004349947	CFLAGS="-O3"
401.bzip2	3.72388985157	CFLAGS="-O2"
470.lbm	2.46679661274	CFLAGS="-O3"
458.sjeng	3.3529822588	CFLAGS="-O3"
445.gobmk	15.3167249203	CFLAGS="-O2"

I choose the fastest flags as the baseline in my program. For complete results of baseline, please refer to the raw data.

3. Optimization results:

3.1 Best found single set of flags

CFLAGS="-O3 -fdata-sections -ftree-sink -funroll-loops -fstack-protector -fsignaling-nans -fbranch-target-load-optimize2 -fgcse-las -fconserve-stack -fivopts -ffast-math -fearly-inlining -fbtr-bb-exclusive -fpeel-loops -fzero-initialized-in-bss -freschedule-modulo-scheduled-loops -ftree-loop-ivcanon -ffunction-cse -ffinite-math-only -fmodulo-sched-allow-regmoves -fsched2-use-traces -fkeep-inline-functions -fsched2-use-superblocks -fsplit-ivs-in-unroller -ftree-copy-prop -ftree-loop-linear -fprefetch-loop-arrays -ftrapping-math -floop-block -fforward-propagate -fselective-scheduling -fassociative-math -ftree-loop-distribution -ftree-vect-loop-version -fsched-spec-load -finline-limit=50 -finline-limit=1000 -finline-limit=10000 -falign-functions=8 -falign-functions=16 -fira-region=one -fira-region=all --param max-gcse-passes=1 --param large-stack-frame-growth=800 --param omega-hash-table-size=225 --param max-sched-region-blocks=5"

3.2 Best found for individual programs:

Format: benchmark | time | flags

462.libquantum 1.01366391182 CFLAGS="-O3 -fdata-sections -ftree-sink -funroll-loops -fstack-protector -fsignaling-nans -fbranch-target-load-optimize2 -fgcse-las -fconserve-stack -fivopts -ffast-math -fearly-inlining -fbtr-bb-exclusive -fpeel-loops -fzero-initialized-in-bss -freschedule-modulo-scheduled-loops -ftree-loop-ivcanon -ffunction-cse -ffinite-math-only -fmodulo-sched-allow-regmoves -fsched2-use-traces -fkeep-inline-functions -fsched2-use-superblocks -fsplit-ivs-in-unroller -ftree-copy-prop -ftree-loop-linear -fprefetch-loop-arrays -ftrapping-math -floop-block -fforward-propagate -fselective-scheduling -fassociative-math -ftree-loop-distribution -ftree-vect-loop-version -fsched-spec-load -finline-limit=50 -finline-limit=1000 -finline-limit=10000 -falign-functions=8 -falign-functions=16 -fira-region=one -fira-region=all --param max-gcse-passes=1 --param large-stack-

frame-growth=800 --param omega-hash-table-size=225 --param max-sched-region-blocks=5"

h263enc 1.90230343342 CFLAGS="-O3 -ffunction-sections -funroll-loops -fstack-protector -funsafe-loop-optimizations -ftree-loop-optimize -fbranch-target-load-optimize2 -fmodulo-sched -fivopts -ffast-math -fsingle-precision-constant -fearly-inlining -fbtr-bb-exclusive -frename-registers -fmath-errno -floop-interchange -fvariable-expansion-in-unroller -ftree-loop-ivcanon -ffinite-math-only -fmodulo-sched-allow-regmoves -fkeep-inline-functions -fsched2-use-superblocks -fira-coalesce -funsafe-math-optimizations -ftree-copy-prop -fmove-loop-invariants -frounding-math -ftree-loop-linear -ftracer -fprefetch-loop-arrays -finline-functions-called-once -fmerge-all-constants -fforward-propagate -floop-strip-mine -fassociative-math -ftree-loop-distribution -fvect-cost-model -ftree-vect-loop-version -fsched-spec-load -finline-limit=50 -finline-limit=500 -finline-limit=100 -falign-functions=2 -falign-functions=16 -fira-algorithm=CB -fira-algorithm=priority -fira-region=one -fira-region=all -fira-region=mixed -fira-verbose=5 -fira-verbose=10 --param inline-call-cost=24 --param min-vect-loop-bound=4 --param max-sched-region-blocks=30 --param max-pipeline-region-blocks=25 --param integer-share-limit=128 --param min-virtual-mappings=400"

mpeg2dec 0.225188708305 CFLAGS="-O3 -fdata-sections -ffunction-sections -funroll-loops -funsafe-loop-optimizations -fsignaling-nans -fbranch-target-load-optimize2 -fgcse-las -freciprocal-math -fsigned-zeros -ftree-reassoc -fsel-sched-pipelining-outer-loops -fbranch-target-load-optimize -frename-registers -fmath-errno -ffunction-cse -ffinite-math-only -fmodulo-sched-allow-regmoves -fsched2-use-superblocks -fsplit-ivs-in-unroller -funsafe-math-optimizations -fprefetch-loop-arrays -finline-functions-called-once -fkeep-static-consts -fselective-scheduling -fvect-cost-model -fsched-spec-load -finline-limit=100 -falign-functions=8 -falign-functions=16 -fira-region=one --param large-stack-frame-growth=800 --param min-inline-recursive-probability=20 --param omega-max-eqs=512 --param omega-max-keys=250 --param max-cse-path-length=30 --param min-spec-prob=50 --param max-sched-insn-conflict-delay=9 --param sched-spec-prob-cutoff=60 --param selsched-max-sched-times=4"

433.milc 13.72305305 CFLAGS="-O3 -funroll-loops -fbranch-target-load-optimize2 -fwrapv -fgcse-las -fconserve-stack -ftree-reassoc -fivopts -fbtr-bb-exclusive -fvariable-expansion-in-unroller -ftree-loop-ivcanon -ffunction-cse -fmodulo-sched-allow-regmoves -fsched2-use-superblocks -fstack-protector-all -frounding-math -ftree-loop-linear -ftrapping-math -fmerge-all-constants -fcx-fortran-rules -fkeep-static-consts -funroll-all-loops -fselective-scheduling2 -fweb -finline-limit=50 -finline-limit=1000 -falign-functions=8 -falign-functions=32 -fira-algorithm=priority -fira-region=one -fira-region=mixed -fira-verbose=10 --param

max-gcse-passes=2 --param min-vect-loop-bound=4 --param omega-max-keys=1000 --param max-sched-region-blocks=5 --param min-spec-prob=50 --param max-sched-insn-conflict-delay=7 --param sched-spec-prob-cutoff=80 --param max-last-value-rtl=25000"

429.mcf 2.19984345436 CFLAGS="-O3 -ftree-sink -funroll-loops -freciprocal-math -ftree-reassoc -fvariable-expansion-in-unroller -ftree-loop-ivcanon -fsched2-use-traces -fsee -funsafe-math-optimizations -fmerge-all-constants -fforward-propagate -ftree-loop-distribution -fweb -finline-limit=50 -falign-functions=2 -fira-algorithm=CB -fira-algorithm=priority --param max-crossjump-edges=100 --param omega-max-geqs=128 --param max-sched-insn-conflict-delay=5 --param selsched-max-lookahead=80 --param selsched-max-sched-times=3 --param integer-share-limit=1024"

mpeg2enc 7.89865338802 CFLAGS="-O3 -fdata-sections -ffunction-sections -funsafe-loop-optimizations -ftree-loop-optimize -fbranch-target-load-optimize2 -fsel-sched-pipelining -fwrapv -fmodulo-sched -ffast-math -fbtr-bb-exclusive -fmath-errno -fpeel-loops -fvariable-expansion-in-unroller -fcx-limited-range -ftree-loop-ivcanon -ffunction-cse -ffinite-math-only -fkeep-inline-functions -fsched2-use-superblocks -fmove-loop-invariants -ftree-loop-linear -fprefetch-loop-arrays -fmerge-all-constants -ffloat-store -fcx-fortran-rules -fkeep-static-consts -fgcse-sm -ftree-loop-distribution -fweb -ftree-vect-loop-version -finline-limit=1000 -finline-limit=10000 -finline-limit=100 -falign-functions=4 -falign-functions=32 -fira-region=one -fira-verbose=5 --param max-crossjump-edges=200 --param max-gcse-memory=128 --param max-gcse-passes=2 --param omega-max-vars=512 --param max-cse-path-length=20 --param selsched-max-lookahead=40 --param virtual-mappings-ratio=2 --param sccvn-max-scc-size=1000"

456.hmmer 2.18498792648 CFLAGS="-O3 -ftree-sink -ffunction-sections -funsafe-loop-optimizations -ftree-loop-optimize -fsignaling-nans -fbranch-target-load-optimize2 -fgcse-las -fmodulo-sched -freciprocal-math -fivopts -ffast-math -fearly-inlining -fbranch-target-load-optimize -frename-registers -fmath-errno -fpeel-loops -floop-interchange -ftree-loop-ivcanon -ffunction-cse -fkeep-inline-functions -fsched2-use-superblocks -funsafe-math-optimizations -ftree-copy-prop -fmove-loop-invariants -ftree-loop-linear -ftree-loop-im -finline-functions-called-once -fkeep-static-consts -fforward-propagate -fassociative-math -funroll-all-loops -freorder-blocks-and-partition -ftree-vect-loop-version -fsched-spec-load -finline-limit=1000 -finline-limit=10000 -finline-limit=500 -falign-functions=2 -falign-functions=8 -falign-functions=16 -fira-algorithm=CB -fira-verbose=10 --param ipcp-unit-growth=10 --param min-inline-recursive-probability=10 --param max-sched-region-blocks=30 --param sched-mem-true-dep-cost=3 --param selsched-max-lookahead=80 --param sccvn-max-scc-size=1000"

h263dec 0.1146671772 CFLAGS="-O3 -fdata-sections -ffunction-sections -funroll-loops -funsafe-loop-optimizations -fsignaling-nans -fbranch-target-load-optimize2 -fgcse-las -freciprocal-math -fsigned-zeros -ftree-reassoc -fsel-sched-pipelining-outer-loops -fbranch-target-load-optimize -frename-registers -fmath-errno -ffunction-cse -ffinite-math-only -fmodulo-sched-allow-regmoves -fsched2-use-superblocks -fsplit-ivs-in-unroller -funsafe-math-optimizations -fprefetch-loop-arrays -finline-functions-called-once -fkeep-static-consts -fselective-scheduling -fvect-cost-model -fsched-spec-load -finline-limit=100 -falign-functions=8 -falign-functions=16 -fira-region=one --param large-stack-frame-growth=800 --param min-inline-recursive-probability=20 --param omega-max-eqs=512 --param omega-max-keys=250 --param max-cse-path-length=30 --param min-spec-prob=50 --param max-sched-insn-conflict-delay=9 --param sched-spec-prob-cutoff=60 --param selsched-max-sched-times=4"

464.h264ref 10.4053230286 CFLAGS="-O3 -fdata-sections -ftree-sink -ffunction-sections -fstack-protector -funsafe-loop-optimizations -fsigned-zeros -ftree-reassoc -fivopts -freschedule-modulo-scheduled-loops -fmodulo-sched-allow-regmoves -fkeep-inline-functions -fira-coalesce -ftree-loop-linear -ftracer -floop-block -ffloat-store -fkeep-static-consts -fselective-scheduling -floop-strip-mine -funroll-all-loops -fgcse-sm -freorder-blocks-and-partition -ftree-loop-distribution -fvect-cost-model -fweb -ftree-vect-loop-version -finline-limit=100 -falign-functions=32 --param omega-max-geqs=128 --param omega-hash-table-size=225 --param max-sched-region-blocks=5 --param selsched-max-lookahead=60 --param max-last-value-rtl=15000 --param min-virtual-mappings=300 --param virtual-mappings-ratio=7"

401.bzip2 3.42020332813 CFLAGS="-O3 -fdata-sections -ftree-sink -ffunction-sections -fstack-protector -funsafe-loop-optimizations -fsigned-zeros -ftree-reassoc -fivopts -freschedule-modulo-scheduled-loops -fmodulo-sched-allow-regmoves -fkeep-inline-functions -fira-coalesce -ftree-loop-linear -ftracer -floop-block -ffloat-store -fkeep-static-consts -fselective-scheduling -floop-strip-mine -funroll-all-loops -fgcse-sm -freorder-blocks-and-partition -ftree-loop-distribution -fvect-cost-model -fweb -ftree-vect-loop-version -finline-limit=100 -falign-functions=32 --param omega-max-geqs=128 --param omega-hash-table-size=225 --param max-sched-region-blocks=5 --param selsched-max-lookahead=60 --param max-last-value-rtl=15000 --param min-virtual-mappings=300 --param virtual-mappings-ratio=7"

470.lbm 2.2863366127 CFLAGS="-O3 -ffunction-sections -funsafe-loop-optimizations -fbranch-target-load-optimize2 -fgcse-las -fconserve-stack -ftree-reassoc -ffast-math -fearly-inlining -fmath-errno -fbranch-count-reg -fvariable-expansion-in-unroller -fcx-limited-range -freschedule-modulo-scheduled-loops -

ffunction-cse -ffinite-math-only -fmodulo-sched-allow-regmoves -ftree-copy-prop -ftree-loop-linear -fprefetch-loop-arrays -ftrapping-math -ftree-loop-im -fmerge-all-constants -fkeep-static-consts -fforward-propagate -floop-strip-mine -funroll-all-loops -fselective-scheduling2 -ftree-loop-distribution -ftree-vect-loop-version -fsched-spec-load -finline-limit=50 -finline-limit=1000 -finline-limit=500 -falign-functions=4 -falign-functions=8 -falign-functions=16 -falign-functions=32 -fira-algorithm=priority -fira-region=one -fira-region=mixed -fira-verbose=5 --param min-inline-recursive-probability=30 --param omega-max-geqs=128 --param omega-max-keys=250 --param sched-spec-prob-cutoff=80 --param sccvn-max-scc-size=1000"

458.sjeng 3.28923933506 CFLAGS="-O3 -ffunction-sections -fstack-protector -funsafe-loop-optimizations -ftree-loop-optimize -fsignaling-nans -fsel-sched-pipelining -fwrapv -fgcse-las -fconserve-stack -freciprocal-math -fsigned-zeros -ftree-reassoc -ffast-math -fearly-inlining -fzero-initialized-in-bss -ftree-loop-ivcanon -ffunction-cse -fmodulo-sched-allow-regmoves -fsched2-use-superblocks -fsplit-ivs-in-unroller -fira-coalesce -fmove-loop-invariants -frounding-math -ftracer -ftrapping-math -ftree-loop-im -floop-block -ffloat-store -fcx-fortran-rules -floop-strip-mine -fassociative-math -fgcse-sm -fvect-cost-model -finline-limit=50 -finline-limit=500 -finline-limit=100 -falign-functions=2 -falign-functions=8 -falign-functions=32 -fira-algorithm=CB -fira-region=mixed -fira-verbose=5 -fira-verbose=10 --param max-crossjump-edges=100 --param large-stack-frame-growth=1000 --param omega-max-wild-cards=16 --param max-cse-path-length=5 --param max-sched-insn-conflict-delay=5 --param sched-spec-prob-cutoff=60 --param selsched-max-lookahead=60 --param selsched-max-sched-times=3 --param integer-share-limit=2048"

445.gobmk 15.1843244314 CFLAGS="-O3 -ftree-sink -funsafe-loop-optimizations -fgcse-las -fconserve-stack -freciprocal-math -ftree-reassoc -fivopts -fmath-errno -fvariable-expansion-in-unroller -freschedule-modulo-scheduled-loops -ffunction-cse -fsched2-use-traces -fsplit-ivs-in-unroller -fsee -ftree-copy-prop -fmove-loop-invariants -ftracer -ftree-loop-im -fselective-scheduling -floop-strip-mine -fassociative-math -fgcse-sm -fweb -finline-limit=1000 -finline-limit=100 -falign-functions=32 --param sra-max-structure-size=0 --param max-crossjump-edges=100 --param omega-max-vars=512 --param sched-spec-prob-cutoff=80 --param max-last-value-rtl=25000"

3.3 *Average across all flags settings (expected random* performan)

I use the mean of reciprocal rate of speedup to measure this. The value is 1.17233366271. A little bit slower than baseline.

*I am sorry for that I have not considered how to use –fprofile correctly this time. Because my program cannot hold that kind of flags well.