Compiler Optimisation using a Genetic Algorithm

B138813

***Abstract*—This paper presents, and evaluates a genetic algorithm for use in compiler optimisation, used on the GNU C compiler's front end. We attempted to find configurations of the GNU C compiler (gcc) which would outperform the Intel C compiler (icc) on a variety of benchmarks, originating from SPEC CPU2006 and MediaBench v2 suites. We were ultimately unsuccseful.**

***Index Terms*—Genetic Algorithm, Compiler Optimisation, Compiler Flags, gcc, icc.**

I. Introduction

A large amount of research has recently been carried out in examining the applicablilty of machine learning techniques to Compiler optimisation[4][2]. Much of this work has been quite sophisiticated and complex, interacting which most parts of the compilation process. Inspired by this work, this paper sets out to find a simplistic genetic algorithm (GA) to find an optimal set of flags for a given benchmark, and the best set of flags for all benchmarks, in an attempt to beat -O3 optimisation.

Whilst we are aware that work has already been done in using GAs for compiler optimisation[1], our work pits the GNU C compiler against the Intel C compiler. We made this choice as we found icc to outperform gcc when both were set to optimisation level -O3. The intel compiler is also paid software, where as gcc is free. We therefore thought it would be an a worthwhile challenge to ourselves to beat -O3 optimisation on icc, using gcc and our GA.

II. Evaluation Methodolgy

We first set about ensuring all our benchmarks would compile on the backend of our compute cluster through PBS scripts. Once we were satisfied, we began development of our Genetic Algorithm. We decided to use Python3, as it is a versitile scripting language with good support for running other programs, and receiving from the STDOUT stream. Although Python3 does lack somewhat in speed, it is inconsequential for our needs, as most of the program's execution time would be spent compiling and running the benchmarks.

We decided to not use flags which could be expressed in a binary on/off format, like `-fsched-critical-path-heuristic`

Our algorithm was ported from a simple JavaScript implementation[3], and then fine tuned to our needs. The purpose of the original GA was help rockets develop a path to a fixed point around an obstacle. In our GA, we attempted to find the quickest run time for our groupings of compiler flags.

Each flag group in our population would run a the same benchmark 10 times, through use of the Python `subprocess` module, allowing the benchmarks to run in parallel. These subprocesses were required to run in their own directories, which were duplicates of the original, to ensure that they would not interfere with each others access to data folders. We would verify that each line of the programs output was correct, so that we did not accidently create performant but incorrect solutions. We carried out this verification with a naive line matching algorithm, which required only 60% of the lines of a programs output to be identical. We made this choice of algorithm as it was quick to implement, and many of the benchmarks would simply print pre-defined strings.

Once a certain flag group had been determined to successfully complete, it's ten execution times were transformed into a mean. We used this mean time to calculate the fitness of each flag groups, in which those that had a faster execution time were given a higher fitness. Fitness was defined as an integer value between 100 and 1.

Flag groups were then placed into a mating pool, where the number of instances of a particular flag group in the mating pool was equal to that flag groups fitness value. We would then create a new population by randomly selecting two flag groups, and mixing their flag configurations. Flags configurations were represented by a dictionary of all possible flags, set to `True` or `False` depending if they were selected or not. Configurations were combined by selecting by giving the child configuration all of the flag settings from parent A up to a random point, and then giving the child all of the flag settings from parent B. The child flag group would then be placed into the population pool and we restart the process of evaluating the population.

III. Experimental Setup

This experiment was carried out on a back end node of Cirrus. Each node on Cirrus has two 2.1 GHz 18-core Intel Xeon E5-2695 processors[1].

For the duration of the experiment run time, we had exlusive access to the back end node. The total runtime of the experiment was 23 hours and 51 minutes (wall time) to find the best flags for each benchmark, and TODO: TIME HERE for the best flags on average across all benchmarks. We used version 17.0.2 of the Intel Compiler, and version 6.2.0 of the GNU C compiler. Benchmark execution time was measured using Python3's `time()` function around calls execution calls to the benchmark. We allowed for a complimation and execution time of ten minutes before terminating subprocesses. This choice was made to ensure our Genetic Algorithm did not break the compiler or compiled process in such a way that caused infite loops.

- Population size of 15

[1]https://cirrus.ac.uk/about/hardware.html

- Maximum number of iterations set to 25
- Our actors, represnted by the `Data_group.py`

IV. Results

Lots of tables here, if not figures. Will definitely need a good script for this.

A. Best Times for individual programs

The corresponding flags for these times can be found in appendix. Values below are given in milliseconds.

- **401.bzip2**: 4129
- **429.mcf**: 2499
- **433.milc**: 15519
- **445.gobmk**: 18188
- **456.hmmer**: 2241
- **458.sjeng**: 3523
- **462.libquantum**: 704
- **464.h264ref**: 12039
- **470.lbm**: 3224
- **h263dec**: 109
- **mpeg2dec**: 194
- **mpeg2enc**: 2071

V. Analysis

What do the results mean

VI. Conclusion

God this was exhausting. It's only 25%? That's mad.

Appendix
Detailed Results

- **401.bzip2**: 4129
-faggressive-loop-optimizations -fassociative-math -fbranch-target-load-optimize -fbtr-bb-exclusive -fcaller-saves -fcombine-stack-adjustments -fcprop-registers -fcse-follow-jumps -fcse-skip-blocks -fcx-fortran-rules -fdelayed-branch -fdelete-null-pointer-checks -fdevirtualize-at-ltrans -fearly-inlining -fipa-sra -ffloat-store -fforward-propagate -ffunction-sections -fgcse -fgcse-after-reload -fgraphite-identity -finline-functions-called-once -fipa-cp -fipa-cp-alignment -fipa-profile -fipa-pure-const -fipa-icf -fira-hoist-pressure -fira-loop-pressure -fno-ira-share-save-slots -fivopts -fkeep-static-consts -flive-range-shrinkage -floop-block -floop-interchange -floop-parallelize-all -flto -fmerge-all-constants -fmove-loop-invariants -fno-branch-count-reg -fno-inline -fno-peephole2 -fno-sched-spec -fno-signed-zeros -fno-trapping-math -fno-zero-initialized-in-bss -foptimize-sibling-calls -fprofile-correction -fprofile-values -fprofile-reorder-functions -freorder-blocks-and-partition -freorder-functions -freschedule-modulo-scheduled-loops -frounding-math -fsched-pressure -fsched-spec-insn-heuristic -fsched-dep-count-heuristic -fschedule-fusion -fselective-scheduling2 -fsel-sched-pipelining-outer-loops -fsingle-precision-constant -fsplit-ivs-in-unroller -fssa-phiopt -fstdarg-opt -fstrict-overflow -ftree-ccp -ftree-copy-prop -ftree-dominator-opts -ftree-dse -ftree-fre -ftree-loop-ivcanon -ftree-loop-linear -ftree-pta -ftree-sink -ftree-slsr -ftree-sra -ftree-ter -ftree-vectorize -ftree-vrp -funit-at-a-time -funroll-all-loops -funsafe-loop-optimizations -funsafe-math-optimizations -funswitch-loops -fipa-ra -fvpt -fuse-linker-plugin -O -O1 -O2 -O3
- **429.mcf**: 2499
- **433.milc**: 15519
- **445.gobmk**: 18188
- **456.hmmer**: 2241
- **458.sjeng**: 3523
- **462.libquantum**: 704
- **464.h264ref**: 12039
- **470.lbm**: 3224
- **h263dec**: 109
- **mpeg2dec**: 194
- **mpeg2enc**: 2071

References

[1] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael FP O'Boyle, John Thomson, Marc Toussaint, and Christopher KI Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305. IEEE Computer Society, 2006.

[2] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, et al. Milepost gcc: machine learning based research compiler. In *GCC summit*, 2008.

[3] Daniel Shiffman. Coding challenge 29: Smart rockets in p5.js. https://youtu.be/bGz7mv2vD6g, 2016.

[4] Zheng Wang and Michael O'Boyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, (99):1–23, 2018.