

International Conference on Computational Science, ICCS 2013

Automatic Tuning of Compiler Optimizations and Analysis of their Impact

Dmitry Plotnikov^a, Dmitry Melnik^{a,*}, Mamikon Vardanyan^a, Ruben Buchatskiy^a,
Roman Zhuykov^a, Je-Hyung Lee^b

^aISP RAS, 25 Alexander Solzhenitsyn st., Moscow 109004, Russia

^bSamsung Electronics Co., Ltd., Next-Generation Computing Lab., Suwon, Korea

Abstract

Modern compilers can work on many platforms and implement a lot of optimizations, which are not always tuned well for every target platform. In the paper we present the Tool for Automatic Compiler Tuning (TACT), which helps to identify such underperforming compiler optimizations. Using GCC for ARM, we show how this tool can be used to improve performance of several popular applications, and how the results can be further analyzed to find places for improvement in the GCC compiler itself.

© 2013 The Authors. Published by Elsevier B.V. Open access under [CC BY-NC-ND license](#).

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science

Keywords: Compiler Optimization; Automatic Performance Tuning; GCC

1. Introduction

The initial motivation [1] for developing the TACT tool was the task of improving the GCC compiler for ARMv7 architecture. Given the complexity of GCC, its long development history, the number of supported target platforms and optimizations, usually its performance can be improved by tuning the compiler optimization parameters for the specific target platform. However, it isn't obvious which compiler optimizations out of dozens provided by the compiler (most of those are platform-independent) may benefit from such tuning.

The traditional approach to compiler performance tuning implies choosing some test applications, finding their hot spots, analyzing the generated assembly code, and then tracing portions of suboptimal code back to optimizations that have generated it (or those that we believe should have worked, but didn't), and then improving those optimizations. This work requires much effort and experience, but still some optimization opportunities could be overlooked, because the code produced by an optimizing compiler is difficult to analyze.

*Corresponding author. Tel.: +7-495-912-0754

Email addresses: leitz@ispras.ru (Dmitry Plotnikov), dm@ispras.ru (Dmitry Melnik), mamikon@ispras.ru (Mamikon Vardanyan), ruben@ispras.ru (Ruben Buchatskiy), zhroma@ispras.ru (Roman Zhuykov), jehyung.lee@samsung.com (Je-Hyung Lee)

It could be helpful to compare performance with another compiler, which could be tuned better for the target platform, so to estimate how much the original compiler could be improved from the current level, and then compare generated code and fix the deficiencies found. In case of ARM, there are at least LLVM and ARMCC compilers available for this architecture, but comparing the code generated by two optimizing compilers could be even more complicated than analyzing the output code from one of them. Also, as we found later, GCC's performance is already substantially better than ARMCC's¹, and also though LLVM may outperform GCC on some tests, is still far from being a reference compiler for ARM.

This way, we came to the idea of using automatic tuning for improving the compiler: it can provide an estimate of the achievable performance level for a given application and give some hints on which optimizations can be improved. The information from the automatic tuning is especially valuable if it shows that the test case improves from disabling an optimization that was enabled in the default compiler configuration, or if it finds a new value for an optimization's parameter. As GCC includes in the `-O2` optimization level only conservative enough optimizations, the speedup coming from disabling such optimization could point to an architecture-specific issue (or test-specific issue, but the former case is more valuable). The subsequent improvement of the cases found with an automatic parameter tuning comes at much less effort, because there is already a code path in the compiler that yields better code, and, in this case, we just need to analyze why the default optimization parameters perform worse, and what does it take to generate better code with the default optimization level. Also, finding suboptimal pieces of code by comparing assembly dumps from runs of the same compiler which differ only in a single optimization is a much easier task than finding that code by manual analysis.

Initially, we started automatic performance tuning with ACOVEA [2]. This is a simple open-source tool that uses a genetic algorithm to search the compiler optimization parameters space. It was included in Gentoo Linux repository to provide its users with a tool for determining optimal compilation flags for building their system. We improved ACOVEA with support for cross-compilation, and it has helped us to find [1] first performance issues in GCC for ARM. However, we came across few drawbacks that made this tool inconvenient to use for the task.

In order to use an automatic tuning tool effectively to find places for improvement in the compiler, we believe that the following features are important. First, it should provide tools for tuning results analysis, so to determine what compiler options from the resulting long obscure parameter string contribute most to the performance improvement found with tuning. Second, it needs support for error checking to filter out compiler options that cause incorrect program behavior or build error. Third, it needs support for parallel compilation and execution on several devices to speedup the tuning. Fourth, the tool should not assume anything about the compiler internals, e.g. options that are included in default optimization levels, or should not require custom patches for the compiler, because most compiler optimization work for open-source compilers such as GCC or LLVM happen on nightly repository snapshots, which are changing fast.

To meet the above requirements, we developed the Tool for Automatic Compiler Tuning (TACT). In this paper, we give an overview of its features and discuss how it can be used in work on compiler improvement, as well as regular application tuning.

2. Related Work

A lot of work is dedicated to the automatic search for optimal compiler settings. In this section, we discuss the available tools for automatic tuning, and give an overview of known approaches to the problem.

One of the well known approaches is iterative compilation. In works [3, 4, 5, 6, 7] the authors are considering different strategies for searching optimization space as well as different optimization goals.

Bodin et al. [3] describe a method for solving the three numerical parameters optimization problem. The algorithm measures the performance for points in optimization space located at equal intervals. The points lying between the current minimum and average value of performance are added to a queue. Then these points are sequentially removed from the queue and their neighborhood is again split into spaced intervals and similarly

¹On SPEC2000 INT, GCC is on average 5% better than ARMCC at `-O2` in ARM mode, and 50% better (geomean) for *GNU Go*, *CxImage* and *C-Ray* applications in Thumb-2 mode, though in the latter case ARMCC parameters were tuned with TACT to rule out the difference in set of `-O2` optimizations

investigated. The process is repeated until the specific number of points has been evaluated, and the point with the minimum execution time is reported. In their work, the authors show that it is possible to find a solution close to optimal (within 0.3%), while considering less than 0.25% of the possible combinations of options and finding the minimum after studying less than 1% of combinations. However, it appears difficult to determine the size of each step, as it depends on the program, on the size of the input data and on the target architecture. Reducing the size of the step can help to achieve a better result, but it also greatly increases the run time; in addition, the algorithm can get stuck in a local minimum. Also, it's not clear how well this algorithm scales to the large number of dimensions, if used to optimize compiler with hundreds of exported optimization parameters (in case of GCC, it's around 200, around half of them having numerical values).

In COLE [4], the authors investigated the standard levels of compiler optimization, and searched for Pareto-optimal levels for performance and compile time. They show that using the genetic algorithm it's possible to find the set of compiler options that are more Pareto-effective (i.e. better performance for the same compile time or vice versa) than the standard optimization levels, which once were set up manually and are not reviewed too often. The authors used SPEC2000 INT, which is a popular benchmark suite for evaluating the compiler performance. However, it doesn't guarantee that the parameters selected for SPEC still will be optimal for other applications.

Bashkansky et al. [8] introduced interesting modifications to the genetic algorithm. They shaped population size as a function of the generation number. In the paper they considered random number of entities, constant number on each population, linear decrease, L-shaped and exponential decrease. Experimental evaluations have shown that extending the random coverage in the beginning and focusing on convergence in the end improves the cost/performance efficiency. Maximum performance gain was achieved by L-shaped and Exponential policies.

Pekhimenko et al. [9] focused on decreasing the compile time for the static commercial compiler TPO (Toronto Portable Optimizer), while preserving the execution time. They extended the compiler with a class to collect static program features such as total operations, float operations, levels of nested loops and others. It is shown that using logistic regression machine learning technique to predict set of transformations and values for their heuristic parameters can decrease the average compile time by at least a factor of two with a negligible increase in the execution time (1%).

Likely the most related work to ours is the MILEPOST project [10]. To our knowledge, only the MILEPOST open-source project is currently freely available for public use (besides ACOVEA [2] that is no longer being developed). It uses machine learning methods to find set of the best compiler optimization parameters for a program based on its features. A variety of static and dynamic parameters are evaluated for each function of the program, which form their feature vectors. During the search of optimization space for a program, these vectors are saved along with the corresponding set of optimizations and the measured result. Then, MILEPOST can provide the best known set of optimizations for previously searched programs with similar features. Using MILEPOST GCC, one can even control the internal order of optimization passes in the GCC compiler, which is otherwise not controllable by user. Authors also suggest the idea of a collective program optimization, where anyone can share their experiences with the community [11]. Such teamwork gives the ability to quickly get sufficiently larger amount of tuning data for various programs. However, MILEPOST doesn't fit too well into our requirements. First, it's focused on using machine learning techniques, which implies finding a solution based on the previous knowledge. However, GCC evolves fast, and the data obtained for the previous compiler revision may quickly become outdated, so we would have to run training phase very often. Second, the GCC plugin it uses is GCC version specific, so updating compiler versions is complicated.

3. Overview of the Tool for Automatic Compiler Tuning

In the following subsections we describe the basic concepts of TACT and outline its main features.

3.1. Genetic Algorithm Core

The TACT evolutionary search core supports multiple optimization objectives, so it can tune either for a single optimization parameter, or for two selected parameters simultaneously, for example, for performance and code size (or compile time).

The implemented method is based on SPEA2 [12], which provides an effective way for finding Pareto-optimal front of compile option sets. First, it reads the configuration file with the compiler options to be tuned, and

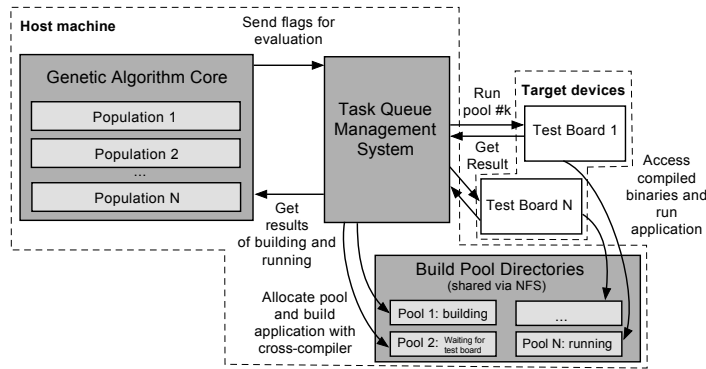


Fig. 1. TACT Components and Operational Scheme

generates a pool of strings ("chromosomes") consisting of random² set of compiler options. This pool of string represents one *population*, and there can be several of them. Then, each chromosome from every population is evaluated: the target application is compiled with a given option string, it executes on a target test board, and the measured performance is returned to TACT, along with application's binary size. In case of a single-parameter optimization, the compile strings corresponding to the best-performing (or smallest size) configurations, are added to the *archive* – a pool that stores predefined number of best configurations among all generations. If the archive is full, then the newly added configurations will push the inferior ones out of the archive. In case of the multi-objective tuning, the archive holds the best Pareto-optimal set, and to limit the number of points in the archive the clustering technique is used. Then, the compiler option configurations from the archive built on the previous generation are used to produce configurations for the next generation: two "parent" configurations are chosen randomly from the archive (in case of a single optimization parameter the configurations with the better parameter values have greater chance to be picked for crossbreeding), and the new option set is constructed by choosing at random option values either from the first or the second parent. At this stage, a limited number of chromosomes can migrate from one population to another. Then, the mutation pass runs on all chromosomes, which can randomly change a single option to the opposite value or shift a parameter value by a random number. After that, the process is repeated from the evaluation step until the fixed number of generations passes or a specific condition on best achieved result is satisfied. Figures 2 and 3 show how the tuning converges over generations for the multi- and the single-parameter cases respectively.

3.2. Operational Scheme

As TACT was designed primarily for automatic tuning on embedded systems running Linux, the system for tuning typically consists of one multi-core x86 host for cross-compilation and one or more target devices (in our case ARM CPU boards). There are two requirements: that all devices can access NFS mount from host, and that SSH server is installed on targets (the latter is used for running applications). Bare-metal targets without NFS/SSH could be also supported by extending the task manager with a routine to send binaries to and request to run them using the available communications. Also, the task management module can reside on a separate machine accessible from other compile hosts through SSH – this way available targets can be shared among tuning sessions of several users.

The overview of TACT operation is shown on Fig 1. The genetic algorithm core handles all operations that involve evolution of compiler options and interacts with other components only by sending requests for evaluation of compile strings to the task queue management system. This component is responsible for building application, for sending the request to run the application binary on a free target test board, and for synchronization of these

²To reduce the tuning time, numerical parameter values (in case of GCC `--param`) can be initialized with random values normally distributed around their defaults values, or evenly distributed within parameter range to increase diversity in the first population. This setting is controlled by configuration option.

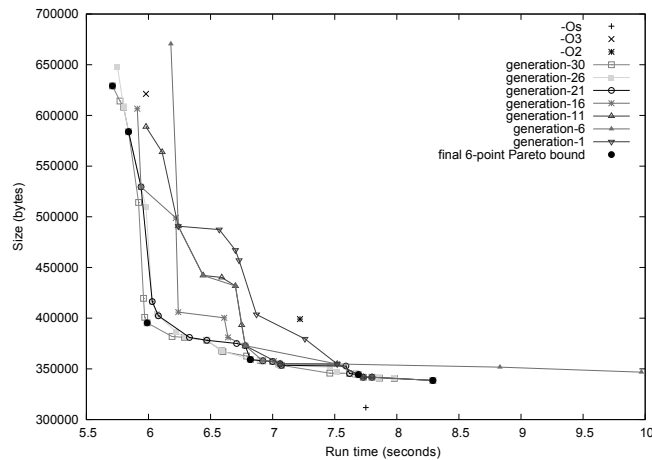


Fig. 2. Evolution of Pareto graphs for tuning *x264* by performance and code size simultaneously

processes. The build of an application is performed in one of the build pool directories. A build pool directory is shared across the host and the target devices via NFS and holds all the data that belongs to a single evaluation run: the program build tree, the installation tree, the temporary run logs and the profile data (if compiling with profiling support). After the application is built in the allocated pool, the system waits for the available target device to run application from the specified pool. If all devices are busy, the task waits until one becomes free, keeping the pool locked. After a device frees up, the run order is accepted and upon completion the performance value is reported back. The task queue manager frees the pool and sends the result back to genetic algorithm core.

3.3. Unified Structure for Application Deployment

TACT provides a benchmarking framework resembling that of SPEC CPU, which includes a directory structure for deploying application sources, shared libraries and resources, a common specification for scripts that build and run applications, verify correctness of run results, a common data format specification for exchanging run results and generating reports. In order to add a new application for tuning, the user just needs to copy the directory structure from a template application, to deploy the application source, and to adjust configuration files and build/run scripts for the specific application.

3.4. Parallel Build and Execution

Parallel compilation and execution support greatly speeds up the tuning process. The parallelism is exploited on two levels. First, we allow building application at the same time as the tool awaits for execution result of the previously compiled application. Second, we allow using several test boards in the tuning process to run tuned applications simultaneously on all of them.

Note that the test boards used for tuning should not be necessarily of the very same model or the same CPU speed. This is because each test board is assigned its own population, so results are compared only with those obtained on the same test board, so evolution branches progress independently. However, migrations between populations are allowed so the best GCC flag combinations are spread through other populations and continue their competition with the "native species" of those populations. This competition is fair since after migration the compile string will be evaluated again on the new test board.

3.5. Error Handling

TACT handles the following types of errors: compile-time errors (internal compiler errors, incompatible flag combinations, etc.), runtime errors (segmentation faults due to miscompilations), execution timeout and output hash mismatch. In all these cases the failing flag combination is eliminated from further evolution. The output

hash can be calculated either by the target application itself (e.g. for the *libevas* benchmark we have added the evaluation of the CRC32 checksum of the frame buffer image after each test), or calculated by the user script. The timeout value for the application is specified in its configuration file.

Interestingly, some miscompilations may cause program to execute faster (e.g. once we encountered a bug in the loop optimization that incorrectly reduced the number of iterations), and as at the time TACT didn't have output hash verification the tuning always converged to the flag combination causing such "profitable" miscompilation.

3.6. Multiple-objective Tuning

TACT supports tuning for multiple optimization objectives simultaneously. In this case, on each generation it evolves the Pareto-optimal set of configurations for the given criteria. Out of the box, it can tune simultaneously for code size and performance, while other user-specified optimization criterion can be added with scripting.

Multiple-objective tuning not only allows to achieve ultimate performance or code size values, but allows finding an acceptable tradeoff between them. This tuning type allows for a fixed threshold of one parameter to find the optimal value of the second one.

Fig 2 shows evolution of Pareto frontiers for tuning of the *x264* application³. Though the effect of the GCC's -Os option (optimize for size) can't be reproduced with a combination of parameters plus -O2, for the performance or code size of other baselines (-O2, -O3) the tool was able to find solutions with better value of the second parameter. For example, for the performance level of -O3 it has found a solution in which code size is 1/3 less.

3.7. Support for Profile-Guided Compilation

TACT can also tune applications with profile-guided optimizations. The tool's task queue manager allows interleaving two execution stages (profile collection and final evaluation) of different build pool directories to minimize the idle time of the test boards.

4. Tuning Speed and Convergence

The number of runs required to get a sufficient speedup depends on the application itself, the compiler, the set of flags chosen for tuning, the evolution parameters (the mutation and migration rates, the number of generations and populations, and the number of flag configurations in one population).

Upper and lower charts on Fig 3, show the maximum and the average speedup for each generation, respectively. Vertical axis measures the performance gain compared to the base optimization level (-O2) in fractions of the maximum speedup achieved during tuning. The value of 1.0 on both graphs corresponds to 16.62% performance improvement for *C-Ray*, 18.19% for *libevas* and 13.76% for *x264*, while zero corresponds to the baseline performance (-O2). These charts do not include the best configurations found on previous generations that are stored in the archive, nor runs failed due to a compilation error or a miscompile.

While tuning starts with the negative average performance gain (and may stay lower than the baseline for a substantial number of iterations), still for all three applications the solution better than the baseline was found already at the first generation. The configuration that gives at least 50% of the maximum performance gain (found in 30 iterations) for *x264* was achieved on the 1st generation, for *libevas* on the 2nd, and for *C-Ray* on the 5th. The 80% level was achieved at 9th, 14th, and 10th generations respectively.

Though the convergence of the solution is not required to achieve substantial performance improvements, from these graphs we can anticipate a little more improvement if we would continue tuning past 30 generations, since best values kept improving in the last two generations.

Our tuning setup involved 200 GCC flags, 80 flag configurations in a population, 2 populations being run in parallel each on its own testboard, and used 4 compilation pools. The mutation rate was set to 3%. Assuming that the average application takes less than a minute for a single run, and that the compile host has enough parallelism (and there are enough compile pools) to finish a compilation before an execution finishes, one generation takes just 80 minutes to complete. Thus, large tuning sessions providing two-digit speedup values can be completed in 1-2 days. In TACT, the number of generations to run can be set either statically or dynamically. In the latter case the tuning stops when the best value does not improve for the specified number of generations.

³Please note that in order to show the relevant part of the graph in better details the axis are shown at the offset and are not originating at zero.

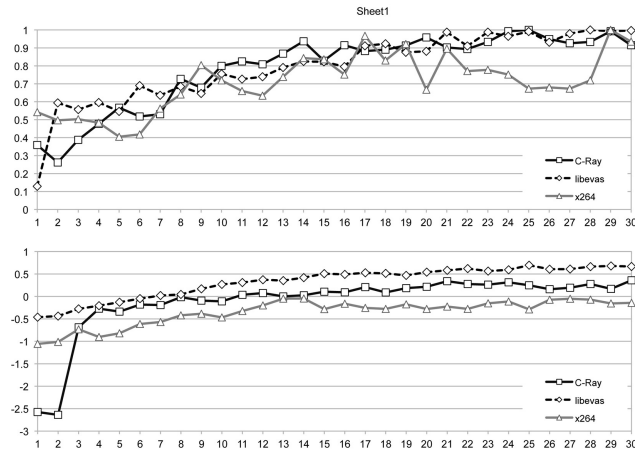


Fig. 3. Best (top) and average (bottom) performance improvement for each generation, 0 corresponds to the -O2 optimization level, 1 corresponds to the maximum performance achieved with tuning

5. Analyzing Tuning Results

Typically, the user of a performance tuning tool is only interested in reproducing the best tuning result by passing the exact compilation string found by the tool as CFLAGS. For the compiler developer the greatest interest is where the speedup comes from, e.g. which compiler optimizations are involved in the observed speedup, which parameters make them work better than with the default ones, which default optimizations were disabled to obtain top performance, etc. In this section we describe the after-tuning analysis tools provided by TACT, which can be useful for a compiler developer.

5.1. Normalizing the Flags Set

The results of automatic performance tuning are hard to analyze, since tuning tools usually provide a long and obscure string that may include hundreds of compiler flags. In TACT, we chose to explicitly include every parameter or option being tuned, no matter if it's already included or not in a baseline level. The other option was to make the tuning tool aware of the default set of optimizations and parameters for chosen base level of the specific version of the compiler. As for results interpretation, we are mostly interested in the difference between the optimal compile string compared to the -O2 default set of optimizations, the resulting compile string first should be normalized, excluding the flags corresponding to optimizations already enabled in the -O2. These also include the parameters that are specific only to the optimizations that are disabled in the resulting set.

The filtering is based on comparing MD5 hashes of application binary compiled with different sets of compiler flags. If the hashes match, then flags that constitute the difference can be omitted, i.e. normalized flag set satisfies the following⁴:

$$\forall \text{flag} \in \text{FlagSet} : \text{hash}(\text{FlagSet} \setminus \text{flag}) \neq \text{hash}(\text{FlagSet}).$$

TACT can filter flags based on the above condition. To optimize the number of required compilations, we are trying to throw out several options at a time, using binary search. Also, we automatically maintain historical database of *significance* for every option. This characteristic represents the probability of whether omitting the option from the compile string will yield the binary hash change. This database is updated after each filtering run for the same application. We sort the option set according to flag significance, so that insignificant flags are more likely to be grouped together and filtered out earlier. The described normalization procedure typically reduces the number of flags by the factor of 3 to 6.

⁴This doesn't cover flags that can be excluded only together with each other (e.g. mutually exclusive flags), but GCC doesn't have many such flags anyway, and they can be filtered with other methods (see Section 5.2).

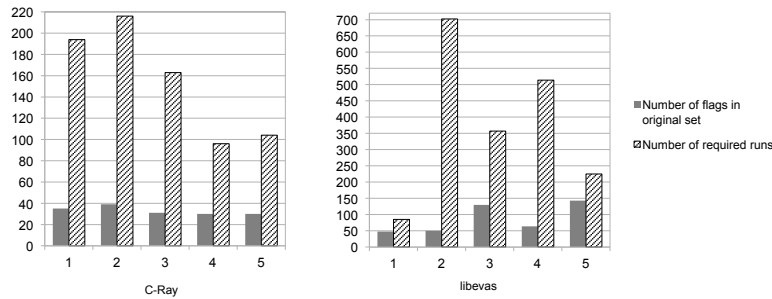


Fig. 4. Number of runs required to reduce configurations consisting of the given number of flags

5.2. Reducing the Resulting Flag Set Based on Performance

Though after the normalization the resulting flag set contains only those flags that affect the compiled binary, many of them do not significantly affect the application's performance. To identify flags that affect performance the most, we do further performance-based reducing of the resulting configurations.

The procedure starts with the original normalized flag set, and then on each step it tries to drop a single flag, so that its exclusion improves⁵ the performance. If such flag can not be found, then the flag with the least negative impact is excluded. This procedure repeats until only the base flags are left (usually `-O2`).

For the flag set consisting of N flags it requires $N - 1$ steps, and on each step the tool performs as many runs as there are currently flags left in the set, looking for the least significant. So the total number of runs required by a straightforward implementation of the reducing algorithm does not exceed $(N - 1)(N - 2)/2$, and usually is less, since we continue traversing the flag set from the flag dropped on the last step if the impact of such operation was positive. Also, to speedup the process, we're first trying to remove flags in large groups, which are formed based on the performance impact the flag has shown when reducing the previous configuration. This way, flags with historically least negative effect (for this tuning session) have priority. If removing the whole group decreases performance, then we again consider the worse half of the group. We continue the dichotomy until a better set is found or only one option is left.

After that, we proceed with a straightforward algorithm described above. Such a method helps to sufficiently reduce number of runs. Fig 4 shows how the number of runs required to reduce configurations of comparable size changes when sequentially reducing five best configurations. For *C-Ray*, the minimum number of iterations required to reduce all five configurations without reusing of previous knowledge or grouping would be 2671, but in our experiments it was 773. For *libevas*, the first configuration was lucky to drop most of its options at the first reducing step, so we didn't get many recommendations recorded for the second configuration. However, both graphs show a tendency of decreasing number of runs for the subsequent configurations.

Fig 5 shows how much the resulting compiler flag set can be reduced for chosen applications. Out of the original 200 compilation flags, after initial flag normalization by binary hash only 33-62 significant options were left. Then, the resulting set is further reduced to 12-27 flags by the method described in this section (first bar). Out of those, 4-14 options provide 80% of achieved improvement (second bar), so after manual inspection some of those may be included in the application's Makefile. And 4-17 options in results were actually optimizations turned off from the configuration specified by the `-O2` optimization level, or parameters that differ from their default values (third bar). Flags in the latter category are the most interesting for compiler developer, because they may point at potential problems in the compiler default optimizations or to their suboptimal tuning for the application or the target architecture.

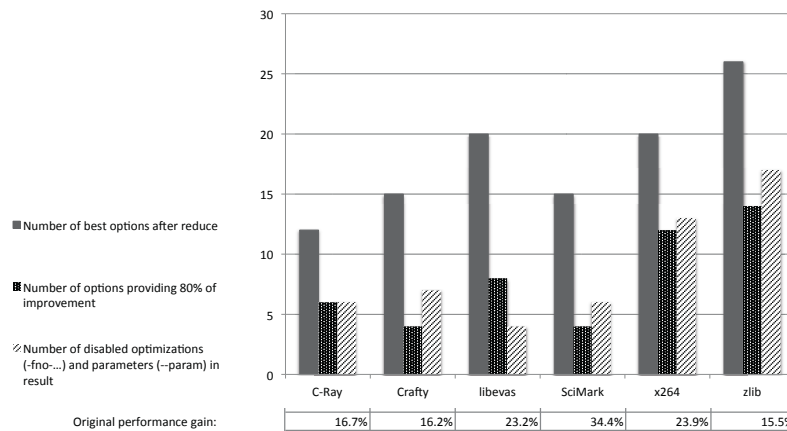


Fig. 5. Number of flags in the resulting set after reducing by performance, the number of most significant flags, and the number of most interesting flags for analysis

Score	%Prev	%Base	%Best	Flags diff	Reduction cost
25.319	0.00%	0.00%	-20.06%	-02	
25.627	1.22%	1.22%	-19.09%	-DEVAS_UNROLL_FACTOR=0	-27.10%
25.689	0.24%	1.46%	-18.89%	-mvectorize-with-neon-quad	-21.10%
25.622	-0.26%	1.20%	-19.11%	-mfpu=neon	-19.27%
26.839	4.75%	6.00%	-15.26%	-ftree-vectorize	-19.03%
28.431	5.93%	12.29%	-10.24%	-fvectorize-misaligned	-13.53%
29.561	3.98%	16.76%	-6.67%	-fprefetch-loop-arrays	-9.25%
30.639	3.64%	21.01%	-3.27%	--param l1-cache-line-size=64	-6.31%
...					
31.515	0.32%	24.47%	-0.50%	-fno-if-conversion2	-0.27%
31.637	0.38%	24.95%	-0.12%	-funwind-tables	-0.45%
31.638	0.00%	24.96%	-0.11%	-fno-thread-jumps	-0.35%
*31.674	0.11%	25.10%	-0.00%	-fno-expensive-optimizations	-0.11%
31.565	-0.34%	24.67%	-0.34%	-fno-tree-ter	

Fig. 6. Example of performance-based flag reducing with *libeas*

5.3. Analyzing the Resulting Reduced Flags

The reducing procedure described in the previous section allows one to estimate the relative importance of compiler flags for the performance. Given that our sequential exclusion algorithm saves flags with most impact for last steps, the earlier the flag was dropped, the less importance it has.

The example of relative importance based on reducing results by performance for *libeas* tuning is shown on Fig 6. The lines on the figure correspond to reducing procedure steps in the reverse order, i.e. the reducing procedure starts with a set of approximately 50 compiler flags (this state corresponds to the last line), and on its first step drops the worst performing option `-fno-tree-ter`, winning 0.34% with that. This state (with one excluded option) corresponds to the previous line (as it's the best result for reducing, it's marked with asterisk). We can also interpret this figure another way: starting with `-02`, we are walking down the list and adding up to compile string one flag per line. Then, the meaning of each line would be the gain from adding a new flag to the previous line configuration. The gain values shown in three columns are evaluated relative to the previous line, base, and the best result respectively.

The *reduction cost* in the last column is the value of slowdown measured with the best configuration excluding the correspondent option. All reduction costs are calculated on the step marked with an asterisk and are shown just as additional characteristic of flag importance.

⁵The presence of such "harmful" flags in the final tuning result can be attributed to measuring inaccuracy or insufficient number of tuning runs. The typical positive gain from the performance-based reducing is 0.2-2%.

From this table it's clear that manual loop unrolling in the application source (controlled by macro `-DEVAS_UNROLL_FACTOR`) was turned off by the tuning as a prerequisite in order for GCC loop optimizations to work; that an experimental patch for misaligned access support for ARM NEON (guarded by the custom flag `-fvectorize-misaligned`) immediately adds almost 6% to standard GCC vectorization; the proper setting of the cache line size according to ARMv7 specification (`--param l1-cache-line-size=64`) in prefetching contributes to the performance almost as much as enabling the prefetching itself. Note that the correct value for cache line size, as well as values for other binary options, were evolved during the tuning, and only later we have found that 64 is indeed the correct value for this parameter in ARMv7 architecture.

6. Conclusion

In this paper we have described a method for fast searching of possible compiler deficiencies for a given platform. We have proposed the requirements for an automatic tool to aid compiler analysis and optimization. We have presented our Tool for Automatic Compiler Tuning (TACT), which has the following main features: it has the tools for analysis of tuning results that allow to identify compiler optimizations that contribute the most to the improvement, supports parallel compilation and execution on several devices to speedup the tuning, and is capable of performing multi-objective optimization to evolve Pareto-front of optimal configurations.

Using TACT tool, we have tuned few popular open-source applications *C-Ray*, *Crafty Chess*, *libevas* (part of Enlightenment Foundation Libraries), *SciMark*, *x264* and *zlib*. All applications were tuned on ARM Cortex-A9 boards, using all of approximately 200 options and parameters available in GCC 4.8. The tuning has resulted in 15-34% speedup of these applications, while 80% of this improvement can be achieved with 4-17 options. Such results can be obtained in a period of a several hours to a few days, depending on the test application, target hardware and tuning parameters.

We used our results for improvement of the GCC compiler. We developed three patches that were accepted into GCC mainline. Also, steady appearance of the flag that disabled Global Common Subexpression Elimination in results for many applications and subsequent analysis of this problem draw our attention to an old patch in Linaro GCC, that yield improvement of SPEC 2000 INT by 4% for GCC mainline on ARM.

We are now continuing development of TACT, and we plan to release it as open source software later this year.

References

- [1] D. Melnik, A. Belevantsev, D. Plotnikov, S. Lee, A case study: optimizing gcc on arm for performance of libevas rasterization library, in: Proceedings of International Workshop on GCC Research Opportunities (GROW-2010), Pisa, Italy, 2010.
URL <http://ctuning.org/dissemination/grow10-03.pdf>
- [2] Acovea project.
URL <http://freecode.com/projects/acovea>
- [3] F. Bodin, T. Kisuki, P. M. W. Knijnenburg, M. O'Boyle, E. Rohou, Iterative compilation in a non-linear optimisation space (1998).
- [4] K. Hoste, L. Eeckhout, Cole: compiler optimization level exploration, in: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization, CGO '08, ACM, New York, NY, USA, 2008, pp. 165–174.
- [5] J. Cavazos, M. F. P. O'Boyle, Automatic tuning of inlining heuristics, in: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, SC '05, IEEE Computer Society, Washington, DC, USA, 2005, p. 14.
- [6] E. Park, S. Kulkarni, J. Cavazos, An evaluation of different modeling techniques for iterative compilation, in: Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems, CASES '11, ACM, New York, NY, USA, 2011, pp. 65–74.
- [7] G. G. Fursin, M. F. P. O'Boyle, P. M. W. Knijnenburg, Evaluating iterative compilation, in: Proceedings of the 15th international conference on Languages and Compilers for Parallel Computing, LCPC'02, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 362–376.
- [8] G. Bashkansky, Y. Yaari, Black box approach for selecting optimization options using budget-limited genetic algorithms, SMART'07 (2007) pp. 1–16.
- [9] G. Pekhimenko, A. D. Brown, Efficient program compilation through machine learning techniques, in: Proceedings of the The Fourth International Workshop on Automatic Performance Tuning (iWAPT), Tokyo, Japan, 2009.
- [10] G. Fursin, Y. Kashnikov, A. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. Williams, M. O'Boyle, Milepost gcc: Machine learning enabled self-tuning compiler, International Journal of Parallel Programming 39 (2011) 296–327.
- [11] G. Fursin, O. Temam, Collective optimization: A practical collaborative approach, ACM Transactions on Architecture and Code Optimization 7 (2010) 20:1–20:29.
- [12] E. Zitzler, M. Laumanns, L. Thiele, Spea2: Improving the strength pareto evolutionary algorithm.