An Automatic Tool for Tuning Compiler Optimizations

Dmitry Plotnikov, Dmitry Melnik, Mamikon Vardanyan,
Ruben Buchatskiy, Roman Zhuykov

ISP RAS
Moscow, Russia

e-mail: {leitz, dm, mamikon, ruben, zhroma}@ispras.ru

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.

Keywords

Compiler Optimization, Automatic Performance Tuning, GCC.

1. INTRODUCTION

Modern compilers typically contain a number of optimization passes that are sequentially run on a program in intermediate representation form. Performance of these optimizations varies depending on target plat- form and properties of a source code. Also, there are many parameters controlling optimizations, such as costs model parameters and thresholds in heuristics. This gives an opportunity for tuning the compiler parame- ters for specific architecture or application.

The results of such tuning are not only interesting from the perspective of getting maximum performance for specific application. For compiler developer they may demonstrate the compiler's potential for improvement, provide test cases where optimizations are conflicting or underperforming. Also, automatic tuning may be used to tune cost and heuristic parameters.

The problem of selecting best compiler options for the given application is being solved by various methods ranging from iterative compilation to automatic tuning tools that use genetic algorithm or machine learning to find the best set of compiler options (e.g. COLE [1], ACOVEA [2], MILEPOST [3]). To our knowledge only MILEPOST open-source project is currently freely available for public use (besides ACOVEA that is no longer being developed). However, MILEPOST is focused on quickly finding a solution based on the previous knowledge by using machine learning techniques, and relies on a large database of applications (which in theory is shared among the project participants). However, for our needs of GCC performance analysis and development, which implies working always with the latest compiler version, we needed a tool which performs thorough optimization space search (rather than making a prediction based on data collected for the previous compiler versions). Another focus of our tool is facilitating the after-tuning analysis to identify the most important performance options.

In this paper, we present the new tool for automatic compiler tuning, the main features of which are genetic algorithm search, support for tuning in cross environment, support for tuning with profiling, integrated tools for result analysis and a convenient benchmarking and tuning framework that allows easy deployment of new applications. We also discuss how this tool can be used in a compiler development and show the automatic tuning results on several applications.

2. MAIN FEATURES OF THE TOOL FOR AUTOMATIC COMPILER TUNING

In the following subsections we describe the basic concepts of the Tool for Automatic Compiler Tuning (TACT) and outline its main features.

2.1 Genetic Algorithm Core

The TACT evolutional 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 [4], 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 generates a pool of strings ("chromosomes") consisting of random 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

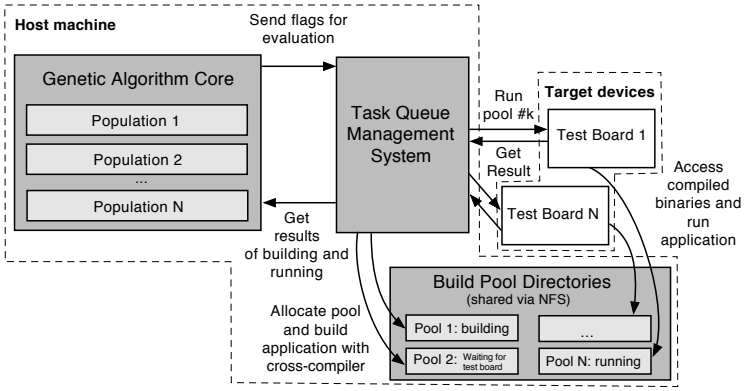

Figure 1. TACT Components and Operational Scheme

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. Figure 2 shows how the tuning converges over generations in multi-parameter tuning.

2.2 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 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.

2.3 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.

2.4 Parallel Build and Execution

Parallel compilation and execution support can greatly speed 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.

2.5 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

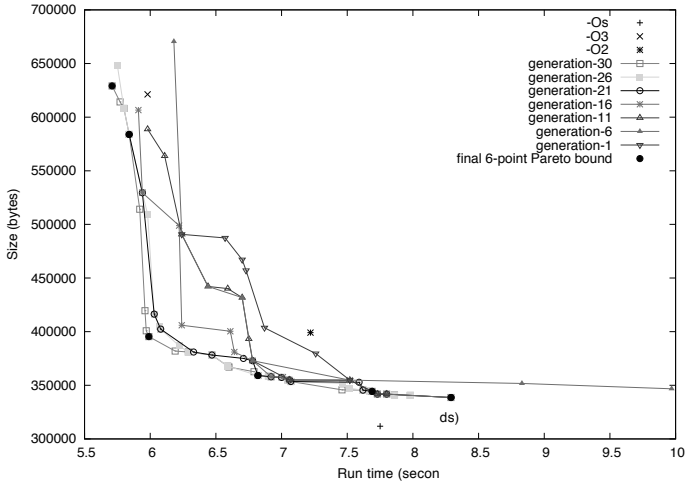

Figure 2. Evolution of Pareto graphs for tuning *x264* by performance and code size simultaneously

image after each test), or calculated by the user script. The timeout value for the application is specified in its configuration file.

2.6 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* application. 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.

2.7 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.

3. 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 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.

3.1 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. the normalized flag set satisfies the following:

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

TACT can filter flags based on the above condition. To reduce the number of required compilations, we are trying to throw out several options at a time, using binary search and based on historical *significance* of the option within the application.

The described normalization procedure typically reduces the number of flags by the factor of 3 to 6.

3.2 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.

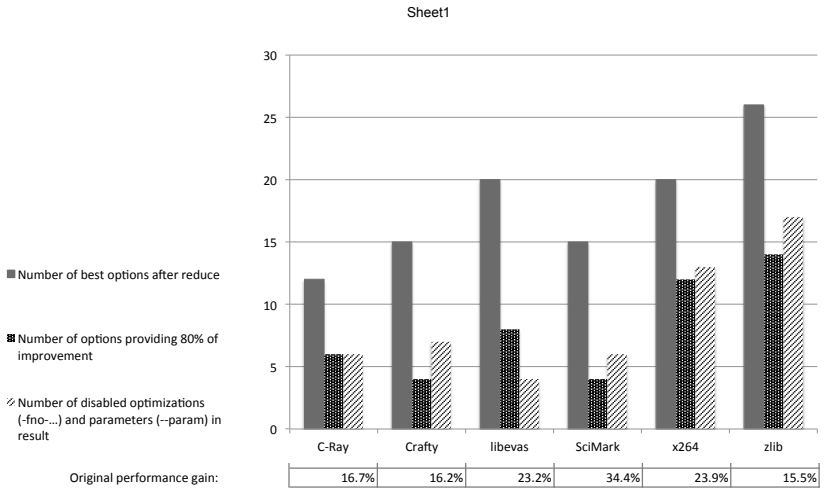

Figure 3. The number of flags in the resulting set after reducing by performance

The procedure starts with the original normalized flag set, and then on each step it tries to drop the single flag, so that its exclusion has the least negative impact on performance. This procedure repeats until only the base flags are left (usually `-O2`). To speedup the process, we're first trying to remove flags in large groups, formed from the flags which historically had the least negative effect for the application (this technique typically reduces the number of required runs by 3-5 times).

Fig. 3 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.

4. CONCLUSION

In this paper 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 cross-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 (Fig. 3), 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 planning to release it as open source software later this year.

REFERENCES

[1] K. Hoste, L. Eeckhout. COLE: Compiler Optimization Level Exploration. Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization, CGO 08, ACM, New York, NY, USA, 2008, pp. 165174. `http://boegel.kejo.be/ELIS//pub/hoste08cole_CGO08_paper.pdf`

[2] ACOVEA home page. `www.coyotegulch.com/products/acovea/index.html`

[3] 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) 296327.

[4] E. Zitzler and M. Laumanns and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm.