COLE: Compiler Optimization Level Exploration

Kenneth Hoste Lieven Eeckhout
ELIS Department, Ghent University
Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium
{kehoste,leeckhou}@elis.UGent.be

ABSTRACT

Modern compilers implement a large number of optimizations which all interact in complex ways, and which all have a different impact on code quality, compilation time, code size, energy consumption, etc. For this reason, compilers typically provide a limited number of standard optimization levels, such as `-O1`, `-O2`, `-O3` and `-Os`, that combine various optimizations providing a number of trade-offs between multiple objective functions (such as code quality, compilation time and code size). The construction of these optimization levels, i.e., choosing which optimizations to activate at each level, is a manual process typically done using high-level heuristics based on the compiler developer's experience.

This paper proposes COLE, Compiler Optimization Level Exploration, a framework for automatically finding Pareto optimal optimization levels through multi-objective evolutionary searching. Our experimental results using GCC and the SPEC CPU benchmarks show that the automatic construction of optimization levels is feasible in practice, and in addition, yields better optimization levels than GCC's manually derived (`-Os`, `-O1`, `-O2` and `-O3`) optimization levels, as well as the optimization levels obtained through random sampling. We also demonstrate that COLE can be used to gain insight into the effectiveness of compiler optimizations as well as to better understand a benchmark's sensitivity to compiler optimizations.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation, compilers*

General Terms

Algorithms, Experimentation, Performance

Keywords

compiler optimization, multi-objective search

1. INTRODUCTION

Modern compilers provide a broad collection of optimizations. Anticipating the efficacy of these optimizations is not trivial though. The effect of a compiler optimization is highly dependent on the code being compiled. In addition, compiler optimization interactions are complex, and in many cases counter-intuitive, or at least hard to reason about. To make things even worse, objective functions, such as performance, code size, energy consumption and compilation time, may be affected conflictingly by different compiler optimizations.

This is a well recognized problem and to facilitate the end user in determining appropriate compiler optimizations, compiler developers typically provide their compilers with a set of standard optimization levels, such as `-O1`, `-O2`, `-O3`, and `-Os`. These optimization levels combine various compiler optimizations and provide different trade-offs in terms of code quality, compilation time and code size. For users willing to trade compilation time for high code quality, the highest level of optimization (e.g., `-O3`) may be worth trying. If on the other hand, compilation time is a concern, more so than code quality, `-O1` may be the optimization level of choice. Finally, if code size is of primary importance, for example for embedded applications, then `-Os` is a suitable optimization level.

Besides static compilers, also dynamic compilers provide multiple optimization levels. For example, the Just-In-Time compilers/optimizers in various Java Virtual Machines [3, 4, 19, 21] can optimize code to a number of optimization levels. In dynamic compilers, the problem of making an appropriate trade-off between code quality and compilation time is even more pressing than for static compilers, because the time spent compiling is part of the total execution time.

The construction of optimization levels however is troublesome. The search space is huge given the large number of compiler optimizations. For example, in the GNU Compiler Collection (GCC) compiler (which we use in this paper) 60 different optimizations are used in the various optimization levels (`-O1`, `-O2`, `-O3`, `-Os`); this results in a huge space with 2^{60} (in the order of $\approx 10^{18}$) possible optimization levels. An exhaustive search in such a huge space is obviously infeasible, because of the time required to compile and run the benchmarks for each combination of optimizations.

Therefore, compiler developers heavily rely on their experience and heuristics in their definition of optimization levels. These heuristics typically look like: optimizations that do not increase compilation time and are likely to produce good code, should be activated at `-O1`; optimizations that tend to increase code size but will likely result in better code quality, should be activated at -O2 and disabled at -Os; and optimizations that typically require a lot of compilation time, and might lead to even better code, should be activated at -O3. Ishizaki et al. [17] describe such a manual optimization level selection process for a dynamic Just-In-Time compiler.

In this paper, we propose Compiler Optimization Level Exploration (COLE), a method to automatically construct optimization levels that represent optimal trade-offs between multiple objective functions, such as performance, compilation time, code size, etc. COLE employs multi-objective evolutionary searching for identifying Pareto optimal optimization levels — a Pareto optimal solution is a solution that can not be beaten by another solution along all objective functions simultaneously. COLE does not only relieve the compiler developer from the tedious and time-consuming task of manually building optimization levels, it also (most likely) yields better performing optimization levels.

In this paper, we make the following contributions:

- To the best of our knowledge, we are the first to propose automated multi-objective compiler optimization level searching, of which the COLE framework is a prototype example. In contrast, today's current practice is to manually build optimization levels based on heuristics and compiler developers' experience and intuition. Similar to COLE, iterative compilation [1, 2, 5, 6, 10, 12, 13, 18, 20, 22, 23, 24] also uses automated searching for identifying the optimal compiler options for a given application of interest, however COLE performs multi-objective searching whereas iterative compilation to date is limited to single-objective searching.

- We experimentally demonstrate that the automatic construction of Pareto optimal compiler optimization levels is feasible in practice. In addition, using the GCC compiler and the SPEC CPU2000 benchmarks, we show that the multi-objective evolutionary search algorithm proposed in COLE outperforms random searching, as well as GCC's (manually derived) standard optimization levels.

- We analyze the Pareto optimal optimization levels obtained from searching a code quality (execution time) versus compilation time multi-objective space in order to gain insight into the importance of the various compiler optimizations. For example, we find that only 25% of the optimizations used in GCC's standard optimization levels appear in one of the Pareto optimal optimization levels, and only a handful optimizations appear in all Pareto optimal optimization levels.

- We use the COLE framework to characterize per-benchmark sensitivities with respect to compiler optimizations. This enables identifying groups of benchmarks exhibiting similar compiler optimization level sensitivity, which provides compiler developers and researchers with a way of picking representative benchmarks for their research or development project.

The paper is organized as follows. We first describe prior work in Section 2. In Section 3 we propose the COLE framework. Section 4 then describes our experimental setup. We evaluate the COLE framework in Section 5, and compare it against random searching and the standard GCC optimization levels. Subsequently, Section 6 analyzes the Pareto optimal optimization levels in terms of the compiler optimizations they enable. We employ the COLE framework to study benchmark optimization level sensitivity in Section 7. And finally, we conclude and discuss future work in Section 8.

2. PRIOR WORK

2.1 Iterative compilation

The work closest related to our work is iterative compilation. The basic idea of iterative compilation is to explore the compiler optimization space by iteratively compiling and measuring the effectiveness of optimization sequences. Driven by a search algorithm, iterative compilation explores the optimization space, and upon termination of the search algorithm, the best performing optimization sequence is reported. A large body of work has been done on iterative compilation over the past few years, and many researchers have reported impressive results showing significant performance, energy or code size improvements over standard optimization sequences, see for example [1, 2, 5, 6, 9, 10, 12, 13, 15, 18, 20, 22, 23, 24].

An important concern though with iterative compilation is that searching the optimization space is very time consuming. By consequence, the vast majority of the work on iterative compilation focuses on reducing the search time; there are basically two ways for doing so. One approach is to speedup the search process by either pruning the search space [22], or intelligently navigating through the search space using heuristic search algorithms, such as genetic algorithms [10, 18] or combination elimination [20]. Another approach is to reduce the time spent evaluating a design point during this search. Some researchers propose analytical modeling for estimating the effect of compiler optimizations on performance [22, 24]. Others build empirical models using predictive modeling built from static code features [1, 8, 23] or dynamic code features [6]. Yet others exploit the phase behavior observed during application execution, and evaluate different optimization sequences in subsequent occurrences of the same phase [12].

What all of this prior work on iterative compilation has in common is that it focuses on a single objective function to be optimized. For example, researchers typically focus on a single optimization criterion such as performance [1, 5, 6, 8, 10, 14, 12, 20, 22, 23, 24], or energy consumption [13], or code size [10]. And some researchers focus on optimizing a single objective function that combines multiple optimization criteria such as code quality and compilation time [7, 8], or code quality and code size [18]. All of these approaches optimize a single metric. And this is where the key difference lies between the prior work on iterative compilation and the COLE approach described in this paper. COLE aims at exploring a multi-objective compiler optimization space, whereas prior work is limited to single-objective optimization. Or, in other words, COLE yields multiple Pareto optimal points whereas prior work yields a single optimal point. An additional difference between existing iterative compilation work and COLE is that iterative compilation focuses on optimizing the performance of a single application, whereas COLE allows finding compiler optimization levels that improve the average performance for a collection of applications.

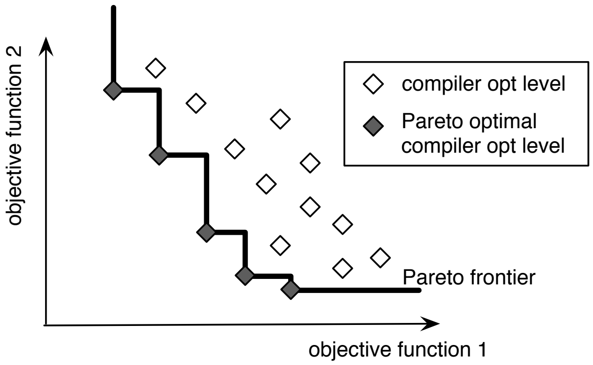

Figure 1: The Pareto frontier in a multi-objective design space.

2.2 Dynamic optimization

Next to static, standalone compilers for programming languages such as C, C++, Fortran, etc., also dynamic compilers which compile and optimize code at run time and which have multiple levels of optimization (as is the case in many Java Virtual Machines [3, 4, 19, 21]), can benefit from multi-objective compiler optimization exploration. The optimizing compiler in the Jikes Research VM for example, has three optimization levels (`O0`, `O1`, and `O2`), next to its baseline compilation level (which compiles methods upon their first invocation). Jikes RVM uses timer-based sampling to identify frequently executed methods which are passed to the controller. The controller uses a cost-benefit model to decide whether to optimize a candidate method, and, if so, to which level the method should be optimized. The cost-benefit model takes into account the estimated compilation time, as well as the expected future execution time once the method is optimized. Since compilation time is an integral part of the total execution time in a dynamic compiler, it is of utmost importance to make a good trade-off between compilation time and code quality when proposing optimization levels in a optimizing dynamic compiler. Ishizaki et al. [17] describe how optimization levels can be determined manually. The COLE framework on the other hand, enables making such a multi-objective trade-off in an automated manner, which is likely going to result in even better optimization levels.

3. MULTI-OBJECTIVE COMPILER OPTIMIZATION LEVEL EXPLORATION

The goal of the multi-objective optimization is to identify a set of compiler optimization levels that provide a Pareto optimal trade-off with respect to a number of objective functions. The objective functions are metrics of interest such as total execution time, compilation time, code size, energy consumption, etc.

3.1 Pareto optimization

In order to explain what Pareto optimality means, we need to introduce some terminology. A given compiler optimization level is called *Pareto dominant* with respect to another optimization level if the given optimization level achieves a better score for at least one objective funtion while achieving the same (or better) score along the other objective functions. A *Pareto optimal* compiler optimization level is an optimization level for which there exist no other optimization levels that achieve a better score for all objective functions. Multiple Pareto optimal compiler optimization levels can co-exist to form a so called *Pareto frontier* or *Pareto set*, see Figure 1. In other words, the Pareto set collects all the Pareto optimal compiler optimization levels. Once the Pareto frontier is identified through multi-objective exploration, the compiler developer or end user can then select a compiler optimization level that trades off the various objective functions according to his or her needs.

3.2 Multi-objective exploration

The COLE multi-objective search algorithm proposed in this paper is based on SPEA2 [25], which is an improved version of the well-established Strength Pareto Evolutionary Algorithm (SPEA) [26]. The SPEA2 algorithm is an elitist evolutionary algorithm that is inspired by genetic algorithms. Our multi-objective algorithm starts from a number of *populations* of randomly generated *entities*, collectively called a *generation*, see Figure 2. In our case, an entity is a set of compiler optimizations, which corresponds to a candidate optimization level. From each population, a number of entities are selected for inclusion in a so called *archive*. The archive entity selection is done based on the entities' Pareto optimality, i.e., the best set of entities from the current population and the previous archive are retained in the current archive. The next generation population is computed based on the current archive through mutation, crossover and migration. This is done in two steps. First, *mating pools* are created by selecting (possibly duplicate) entities from the archive through binary tournament selection, or through *migration* (with probability $p_{migration}$) which allows entities to switch populations. In the second step, mutation and crossover are applied on the mating pools. *Mutation* randomly changes a single entity with a probability $p_{mutation}$; and *crossover* generates new entities by mixing two existing entities within a mating pool with a probability $p_{crossover}$. This process of mutation, crossover and migration based on the archives, creates the next generation. The algorithm is repeated, i.e., new generations are constructed, until no more improvement is observed in the overall Pareto frontier. The end result of the multi-objective search algorithm is a number of Pareto frontiers — there is a Pareto frontier per population. The overall Pareto frontier is then determined based on these individual Pareto frontiers, i.e., only the Pareto optimal compiler optimization levels are selected for the overall Pareto frontier.

The multi-objective algorithm considered in this work is an extension and generalization of the SPEA2 algorithm in that we consider multiple populations, whereas SPEA2 only considers a single population. During our work, we found that different populations to start from results in different Pareto frontiers. In other words, it seems that, at least for our design space, SPEA2 may yield a Pareto frontier that is a local optimum. For this reason, we consider multiple populations on which to apply the SPEA2 algorithm in conjunction with migration which allows entities to switch populations.

Within COLE, we give higher priority to optimization levels that include fewer optimizations. For example, if two optimization levels achieve the same compilation time and execution time numbers — within the range of the run time

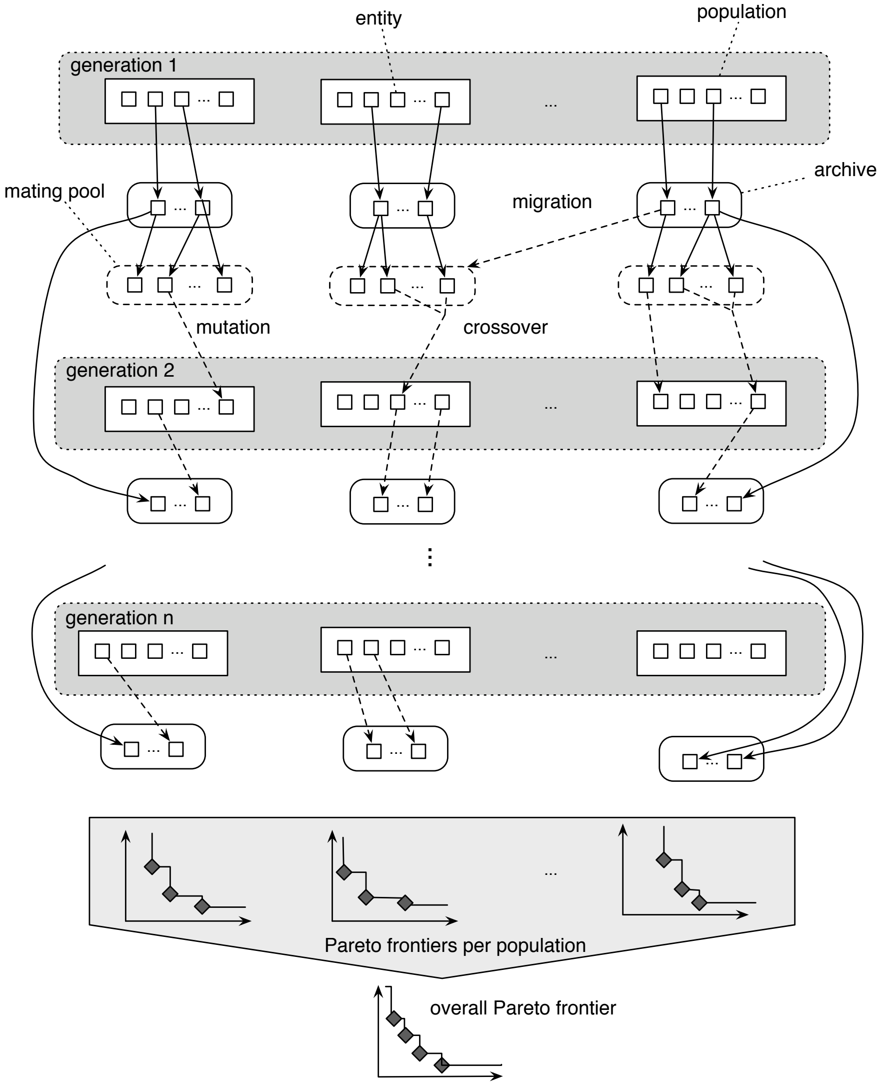

Figure 2: The COLE multi-objective optimization framework.

variability observed from real hardware executions — we will retain the optimization level with the fewest optimizations included. Next to providing concise optimization levels, this also helps the search algorithm in creating better quality populations, i.e., optimizations that are effective are given a bigger chance to be used along the evolutionary search process.

3.3 Exploration speed

Exploring the optimization level design space using the above multi-objective search algorithm requires that each entity is evaluated in order to understand its Pareto optimality. Evaluating an entity requires quantifying the entity in terms of the objective functions of interest. In our work, this requires compiling the benchmarks using the set of compiler optimizations that the entity represents. If performance and/or energy consumption are of interest as objective functions, the compiled benchmarks need to be run, and execution time and energy consumption need to be measured, either through simulation or real hardware execution. In case code size is of interest, the code size needs to be computed.

Evaluating an individual entity may be time-consuming. However, evaluating a generation as a whole is embarrassingly parallel. All entities in a generation — there are 3 × 20 = 60 entities in total in our setup — can be evaluated in parallel. Subsequent generations need to be run sequentially though because the next generation is computed based on the previous generation.

3.4 Compiler optimization level design space

The compiler optimization level design space can be huge, which is the motivation in the first place for employing an automated multi-objective evolutionary search algorithm. In our experimental setup, we use the GNU Compiler Collection (GCC) 4.1.2 compiler, and consider all the individual compiler optimizations appearing in the standard `-O1`, `-O2`, `-O3` and `-Os` compiler optimization levels that can be turned on and off individually through command-line compiler switches. There are 60 compiler flags in total, see Ta-

	compiler optimization		compiler optimization
	`-fearly-inlining`	-O2	`-falign-functions=0`
	`-ffunction-cse`		`-falign-jumps=0`
	`-fkeep-static-consts`		`-falign-labels=0`
	`-fpeephole`		`-falign-loops=0`
	`-fsplit-ivn-in-unroller`		`-fcaller-saves`
	`-ftree-vect-loop-version`		`-fcrossjumping`
-O1	`-fcprop-registers`		`-fcse-follow-jumps`
	`-fdefer-pop`		`-fcse-skip-blocks`
	`-fguess-branch-probability`		`-fdelete-null-pointer-checks`
	`-fif-conversion`		`-fexpensive-optimizations`
	`-fif-conversion2`		`-fgcse`
	`-fipa-pure-const`		`-fipa-type-escape`
	`-fipa-reference`		`-foptimize-sibling-calls`
	`-floop-optimize`		`-fpeephole2`
	`-fmerge-constants`		`-fregmove`
	`-ftree-ccp`		`-freorder-blocks`
	`-ftree-ch`		`-freorder-functions`
	`-ftree-copy-prop`		`-frerun-cse-after-loop`
	`-ftree-copyrename`		`-frerun-loop-opt`
	`-ftree-dce`		`-fschedule-insns2`
	`-ftree-dominator-opts`		`-fstrength-reduce`
	`-ftree-dse`		`-fstrict-aliasing`
	`-ftree-fre`		`-fthread-jumps`
	`-ftree-lrs`		`-ftree-pre`
	`-ftree-salias`		`-ftree-store-ccp`
	`-ftree-sink`		`-ftree-store-copy-prop`
	`-ftree-sra`		`-ftree-vrp`
	`-ftree-ter`	-O3	`-fgcse-after-reload`
	`-funit-at-a-time`		`-finline-functions`
			`-funswitch-loops`
			`-fno-reorder-blocks-and-partition`

Table 1: Compiler optimization flags considered in this paper.

ble 1. These flags are ordered in Table 1 according to their inclusion in the standard `-O1`, `-O2` and `-O3` compiler optimization levels. For example, the `-O1` optimization level includes all the optimizations listed in the left column of Table 1; `-O2` includes all optimizations in the left column plus all the optimizations in the right column from the top down to and including `-ftree-vrp`; `-O3` includes all optimizations in Table 1. Next to these 60 compiler optimization flags, we also consider a base optimization level with three possible options, `-Os-stripped`, `-O1-stripped`, and `-O2-stripped` (`-O3-stripped` is identical to `-O2-stripped`). These 'stripped' base optimization levels correspond to their respective standard compiler optimization levels `-O1`, `-O2`, `-O3` and `-Os` with all optimizations disabled that are controllable through compiler switches; disabling a standard optimization `-f<flag>` can be done using the `-fno-<flag>` compiler switch. The reason for including these stripped base optimization levels is that GCC includes some default optimizations that cannot be controlled through command-line compiler switches. Put together, the entire compiler optimization level design space includes $3 \times 2^{60} \approx 3.4 \times 10^{18}$ possible combinations of optimization flags.

4. SETUP AND METHODOLOGY

4.1 Benchmarks, compiler and hardware setup

We use the SPEC CPU2000 benchmark suite [16] in our setup. We consider the training inputs in this paper, because of time constraints. Compiling all the SPEC CPU2000 benchmarks and running them using the training inputs takes about 20 minutes in our setup. Since we had a single machine to our disposal for doing this research, evaluating a generation (which contains 60 candidate optimization levels) takes about 1,200 minutes, or 20 hours. Running the multi-objective algorithm until convergence (which required 70 generations) took about 50 days in our setup. As mentioned before, this can be trivially parallelized by evaluating all entities in a given generation in parallel. In our case, this would yield a 60× speedup, or it would take less than a day to run the multi-objective optimization using the training inputs. Extrapolating these results to the reference inputs of the SPEC CPU2000 benchmark suite, the whole multi-objective optimization level searching would take approximately 5 to 6 days, which we believe is acceptable for practical use — the definition of optimization levels can be postponed towards the end of the compiler building process and needs to be done only once. In addition, the above calculations assume starting the search process 'from scratch'. In practice though, the search process for determining the Pareto optimal optimization levels for the next generation compiler can start from the Pareto frontier obtained for the previous generation compiler, significantly reducing the search time. Nevertheless, speeding up the search process may be an interesting avenue for future research.

As mentioned before, we use the GCC 4.1.2 compiler in our experimental setup, because it provides a wide range of command-line optimizations that can be turned on and off, which is often not the case for commercial compilers (at least for the ones we considered for our research). We ran all of our experiments on an Intel Pentium 4 Prescott 3.0 GHz processor based machine running Fedora Linux (Core 4).

4.2 COLE setup

In the evaluation reported in this paper of the COLE framework, we consider two objective functions, namely code quality (i.e., performance, measured as the benchmark execution time) and compilation time. This does not affect the generality of the COLE framework though, because COLE can be (trivially) used for other objective functions of interest, such as code size, energy consumption, power con-

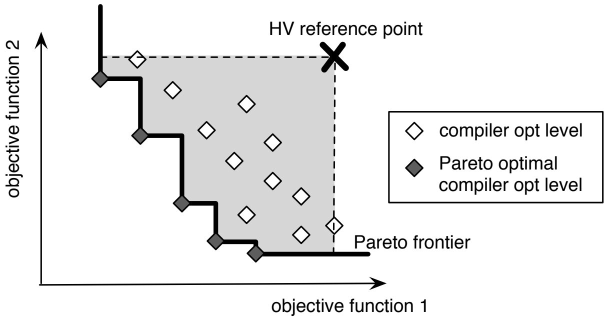

Figure 3: Example illustrating how the HV metric is computed.

sumption, etc. The code quality objective function in our experiments equals the total sum of the individual execution times for all the SPEC CPU benchmarks; the compilation time objective function equals the total sum of the individual compilation times for all the benchmarks. These objective functions give a higher weight to benchmarks that execute and compile for a longer time. If equal weight to all benchmarks is needed, the objective functions need to be computed differently, for example by taking an unweighted average over speedup numbers. Again, this is a design choice up to the COLE user.

In the initial generation for the COLE optimization process, we include the standard `-O1`, `-O2`, `-O3` and `-Os` optimization levels, next to a number of randomly generated optimizations, in order to give the multi-objective algorithm a head start in its exploration.

We consider three populations, 20 entities per population, and 10 entities per archive in our experiments. Recall that the archive selects the Pareto optimal entities from the population and the prior archive. In case there are less than 10 Pareto optimal entities, the archive is filled up with near Pareto optimal entities. In case there are more than 10 Pareto optimal entities, the archive selects the entities that cover the objective function ranges as widely as possible. The mutation probability is set to $p_{mutation} = 0.10$; the probability for crossover is set to $p_{crossover} = 0.90$; and the probability for migration is set to $p_{migration} = 0.10$. These settings are fairly standard settings for genetic algorithms and we found these settings to work well in our setup.

4.3 HV metric

Next to visually assessing the quality of the obtained Pareto frontiers, we also quantify their quality using the hypervolume (HV) metric discussed in [11]. Figure 3 illustrates how the HV metric is computed. First, the HV reference point needs to be determined. The HV reference point is the point obtained by taking the maximum value observed by any of the points along each objective function. Second, once the HV reference point is determined, the HV metric can be computed as the area between the Pareto frontier and the HV reference point. The HV metric thus is a higher-is-better metric. The higher the HV metric, the more the Pareto frontier reduces the scores for all objective functions.

5. EVALUATION

We now evaluate the quality of the obtained Pareto optimal optimization levels, both qualitatively by visual inspection of the Pareto frontiers, and quantitatively by using the HV metric.

5.1 Pareto frontiers

Figure 4 shows the Pareto frontiers as a function of code quality (execution time) and compilation time for four scenarios of optimization levels:

- The standard GCC optimization levels (`-O1`, `-O2`, `-O3` and `-Os`); the optimization level `-Os` is not a Pareto optimal optimization level (in terms of the code quality and compilation time objective functions).
- The Pareto frontier which is obtained from randomly sampling the optimization space: 600 randomly chosen optimization levels were evaluated and the Pareto optimal optimization levels are retained.
- The Pareto frontier obtained through COLE after 10 generations which corresponds to 600 optimization levels being evaluated — this is the same search time as under random sampling.
- The Pareto frontier obtained through COLE after convergence at 70 generations. This corresponds to 4200 optimization levels being evaluated during the exploration.

The results shown in Figure 4 clearly illustrate that the Pareto frontier obtained through COLE is substantially better than the standard GCC optimization levels. This means that for the same compilation time, shorter run times are obtained using the COLE Pareto frontier; or reverse, the same code quality can be obtained with much shorter compilation times, see Figure 5 for a summary result. In particular, the Pareto optimal optimization level which yields the best performance achieves 3.1% better performance on average than `-O3` with a 37.6% shorter compilation time. This same Pareto optimal optimization level even achieves 4.5% better performance than `-O1` for a slightly smaller compilation time (1.5%).

Comparing COLE versus random sampling — for doing so we consider COLE with 10 generations (and thus 600 entities) versus random sampling of 600 entities — shows that the evolutionary search algorithm in COLE is more effective than random searching. For the same exploration time, COLE achieves optimization levels that provide better trade-offs in code quality versus compilation time.

5.2 Evaluation using the HV metric

We now evaluate the quality of the Pareto frontiers obtained through COLE using the HV metric. Figure 6 shows the HV metric for COLE, random sampling and the standard compiler optimization levels (`-O1`, `-O2`, `-O3`). This graph restates our earlier finding more quantitatively, namely COLE outperforms random sampling by 10% with the same search budget, and significantly outperforms the standard (manually derived) optimization levels by 53%.

5.3 Cross-validation

So far, we evaluated the COLE framework using the SPEC CPU2000 benchmarks, which is also the set of benchmarks

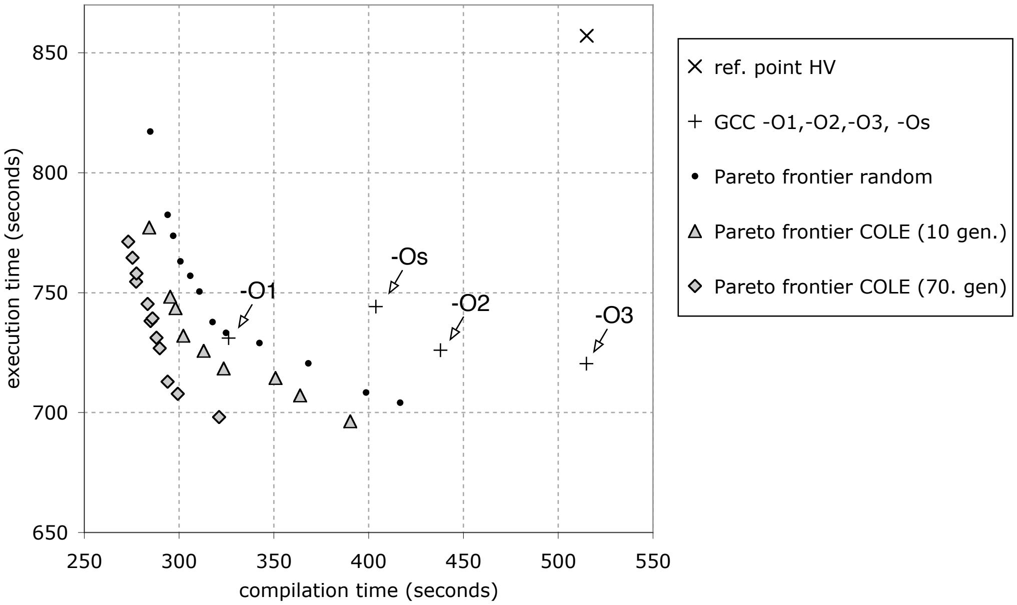

Figure 4: Evaluating the Pareto frontiers obtained through COLE after 70 generations (after convergence is reached) and 10 generations against the standard compiler optimization levels (-O1, -O2, -O3 and -Os) and random sampling (with the same exploration time budget as COLE with 10 generations).

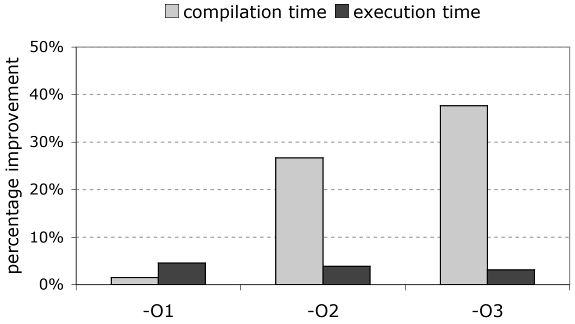

Figure 5: Comparison of the Pareto optimal optimization level that yields the best performance, i.e., the rightmost point on the Pareto frontier in Figure 4, against the standard GCC optimization levels -O1, -O2 and -O3. The vertical axis shows the percentage improvement in compilation time and execution time.

used to determine the Pareto optimal optimization settings. We now consider a cross-validation experiment in which we evaluate the effectiveness of the Pareto optimal optimization settings found through COLE using the SPEC CPU2000 benchmark suite on a previously unseen set of benchmarks, namely the SPEC CPU2006 benchmark suite with training inputs. The HV score for the COLE optimization levels (obtained using SPEC CPU2000) is 37% higher than the HV score for the standard optimization levels. This result shows that the Pareto-optimal optimization settings found with COLE using the SPEC CPU2000 benchmarks are good optimization levels for SPEC CPU2006 as well.

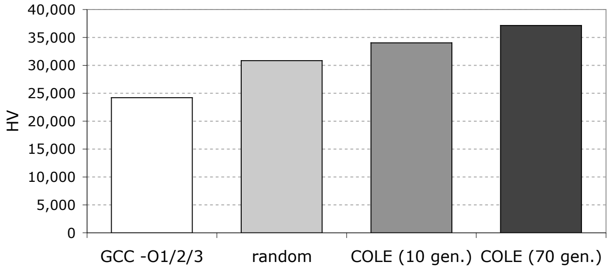

Figure 6: Evaluation of the Pareto frontiers obtained through COLE, random sampling and the standard compiler optimization levels (-O1, -O2, -O3) using the HV metric.

6. ANALYSIS

Now that we have obtained the Pareto optimal optimization levels, we can analyze what these optimization levels look like in terms of their constituent compiler optimizations. There are 12 Pareto optimal optimization levels in total, and only 15 out of the 60 compiler optimizations are included in at least one optimization level. In other words, 75% of the compiler optimizations do not contribute to any of the Pareto optimal optimization levels. Of course, this result is tied to the benchmarks as well as the hardware platform considered in the setup, and only reflects average performance. Nevertheless, this result may provide insight to compiler developers and researchers about the usefulness of the various compiler optimizations for the average user.

Table 2 shows the 12 Pareto optimal optimization levels in terms of these 15 compiler optimizations. The optimization levels are ordered from short compilation time and modest code quality (optimization level 0) towards longer compila-

	GCC	Pareto optimal optimization level											
	level	0	1	2	3	4	5	6	7	8	9	10	11
`-fno-keep-static-consts`	default			x									
`-fdefer-pop`	-O1	x	x	x	x								
`-fguess-branch-probability`	-O1						x		x		x	x	x
`-floop-optimize`	-O1											x	x
`-ftree-ccp`	-O1					x	x	x	x	x	x	x	x
`-ftree-dce`	-O1	x	x	x	x	x	x	x	x	x	x	x	x
`-ftree-fre`	-O1	x	x	x	x	x	x	x	x	x	x	x	x
`-ftree-lrs`	-O1	x	x	x	x	x	x	x	x	x			
`-ftree-sra`	-O1	x	x	x	x	x	x	x	x	x	x	x	x
`-ftree-ter`	-O1	x	x	x	x	x	x	x	x	x	x	x	x
`-funit-at-a-time`	-O1			x	x	x	x	x	x		x	x	x
`-fstrength-reduce`	-O2		x		x								
`-fstrict-aliasing`	-O2		x	x	x					x	x	x	x
`-ftree-store-copy-prop`	-O2	x						x	x	x	x	x	x
`-finline-functions`	-O3												x
`-Os-stripped`		x	x	x	x								
`-O1-stripped`						x	x	x	x	x	x	x	x
`-O2-stripped`													

Table 2: Marking the compiler optimizations and (stripped) base optimization levels included in the various Pareto optimal optimization levels, ordered from short compilation time and long run time (optimization level 0) towards long compilation time and short run time (optimization level 11).

tion time and better code quality (optimization level 11). An 'x' reflects that a given compiler optimization is included in the given optimization level. There are several interesting observations to be made from Table 2. For example, several of the language and platform-independent Static Single Assignment (SSA) tree optimizations (with the `-ftree` prefix) seem to be very effective. In particular, four tree SSA optimizations appear in all 12 Pareto optimal optimization levels. These optimizations are beneficial irrespective of the multi-objective trade-off in terms of compilation time versus code quality. On the contrary, some optimizations are only beneficial for a specific optimization trade-off. For example, the function inlining optimization (`-finline-functions`) and loop optimization (`-floop-optimize`) are only helpful when code quality is the primary concern: these flags result in significant code quality improvement at the cost of additional compilation time.

Many of the optimizations under the standard compiler optimization levels (`-O1`, `-O2`, `-O3` and `-Os`) do not appear in any of these optimization levels. For example, of the 27 optimizations appearing under the standard `-O2` optimization level, see Table 1, only three get selected in at least one of the Pareto optimal optimization levels; also, only one of the three `-O3` optimizations is being used. The `-O1` optimizations are more succesful in this respect: 10 out of the 23 optimizations are selected at least once.

Another interesting observation is that the Pareto optimal optimization levels 0 through 3 use the stripped `-Os-stripped` base optimization level; the Pareto optimal optimization levels 4 through 11 use the stripped `-O1-stripped` base optimization level. The reason is that the `-Os-stripped` base optimization level introduces less optimization opportunities than the `-O1-stripped` base optimization level, so that the optimizations applied on top of the `-Os-stripped` base optimization level introduce less compilation load. The `-O2-stripped` base optimization level is not selected for any of the Pareto optimal optimization levels: it seems like this base optimization level adds too much compilation load compared to the performance gain it provides.

7. BENCHMARK OPTIMIZATION LEVEL SENSITIVITY

Another interesting application of the COLE framework, next to the construction of Pareto optimal optimization levels as just decribed, is to study a benchmark's sensitivity with respect to the compiler optimization level. This will enable us to classify benchmarks according to their optimization sensitivity. This can be useful for compiler builders and researchers in order to pick representative benchmarks. Benchmarks that are sensitive to compiler optimizations may be good candidate benchmarks; benchmarks that are rather insensitive may be of less interest. Or, by looking into what compiler optimizations affect a benchmark insight can be gained into the internal behavior of the benchmark.

For doing so, we use COLE to find the Pareto frontier per benchmark, and then compare this per-benchmark Pareto frontier against the 'average' Pareto frontier obtained by considering all benchmarks (as we did before). The results are shown in Figure 7. The HV metric for the standard `-O1`, `-O2` and `-O3` optimization levels as well as for the average Pareto frontier are compared against the per-benchmark Pareto frontier — these HV metrics are normalized to the HV metric for the per-benchmark Pareto frontier. Since the Pareto frontiers for the standard GCC optimization levels as well as for the 'average' Pareto frontier are always worse than the per-benchmark Pareto frontier, the relative HV metrics are all smaller than one. These results show that several benchmarks are rather insensitive to benchmark-specific compiler optimizations, i.e., the 'average' Pareto frontier achieves nearly the same code quality versus compilation time trade-off as the per-benchmark Pareto frontier. Example benchmarks are crafty, gzip, wupwise and several others with a relative HV metric for the 'average' Pareto frontier over 90%. Other benchmarks are very sensitive to benchmark-specific optimizations, see for example sixtrack and gap, which have relative 'average' Pareto frontier HV scores around 40% and 50%. Overall though, the 'average' Pareto frontier obtained through COLE yields a trade-off between code quality and compilation time that

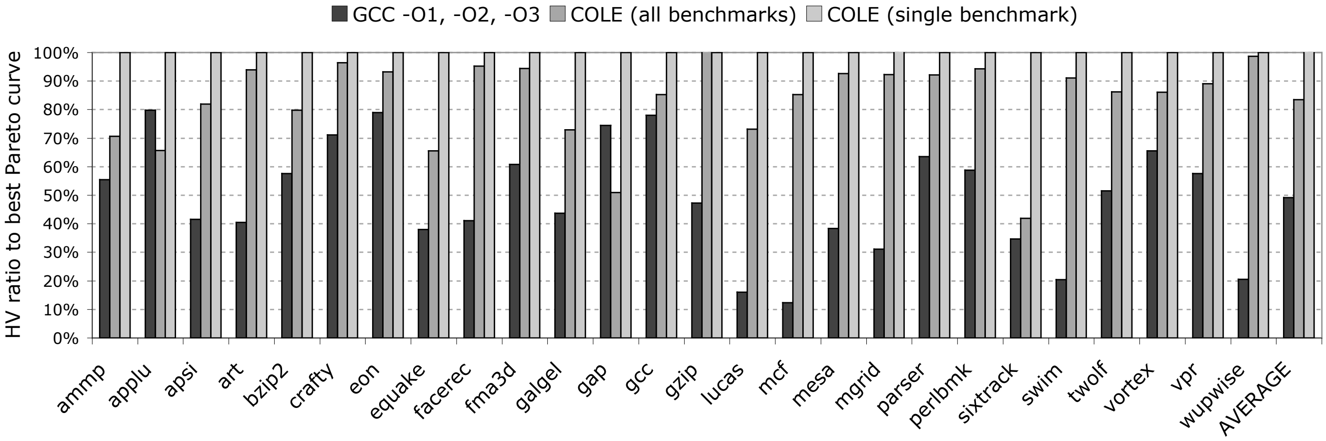

Figure 7: The HV metric per benchmark for the standard -O1, -O2 and -O3 optimization levels as well as the 'average' Pareto frontier relative to the per-benchmark Pareto frontier.

is fairly close to the per-benchmark optimal trade-off. The average HV score for the 'average' Pareto frontier equals 83%, which is substantially higher than the standard -O1, -O2 and -O3 optimization levels with an average HV score of 50%.

8. SUMMARY AND FUTURE WORK

Static compilers as well as dynamic compilers typically come with a number of optimization levels such as -O1, -O2, -O3 and -Os, which provide different trade-offs between code quality, compilation time and code size. Constructing these optimization levels typically is a manual process which is both tedious and time consuming. Identifying an appropriate set of optimization levels is particularly challenging because the search space is huge — for example, the design space in our setup counts in the order of 10^{18} candidate optimization levels.

This paper presented COLE, Compiler Optimization Level Exploration, which employs a multi-objective evolutionary algorithm to find Pareto optimal optimization levels. COLE is fully automated and is completely transparent to the compiler, the benchmarks, the hardware platform, as well as the objective functions. To the best of our knowledge, this paper is the first to study automated multi-objective compiler optimization exploration. COLE differs from iterative compilation in this respect because the work done so far in iterative compilation focused on single-objective optimization. Our experiments using GCC and the SPEC CPU benchmarks on an Intel Pentium 4 machine optimizing for run time (code quality) and compilation time show that the optimization levels obtained through COLE significantly outperform the standard (and manually derived) compiler optimization levels (-O1, -O2, -O3), and in addition, outperform the optimization levels obtained through random sampling.

We believe this work may trigger future research in the efficient construction of compiler optimization levels. For example, as mentioned in the paper, although the current framework is fast enough for practical use, we believe there is still room for improvement. There are various ways of speeding up the search process by either pruning the optimization space or by reducing the time spent evaluating an optimization level. Techniques proposed within the context of iterative compilation for single-objective optimization most likely need to be rethought within the context of multi-objective optimization. Another interesting opportunity for future research is to exploit the fact that optimization levels typically need to yield good *average* performance across a number of benchmarks, as opposed to iterative compilation which aims at optimizing the performance of a single application. One could take advantage of the optimization level sensitivity similarities that exist between benchmarks to further speedup the search process, i.e., evaluating the compiler optimizations on a representative set of benchmarks rather than the whole benchmark suite may yield significant speedups.

9. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their thoughtful comments and valuable suggestions. Kenneth Hoste is supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). Lieven Eeckhout is a Postdoctoral Fellow with the Fund for Scientific Research – Flanders (Belgium) (FWO Vlaanderen). This work is also supported in part by the FWO projects G.0160.02 and G.0255.08, and the HiPEAC Network of Excellence.

10. REFERENCES

[1] F. Agakov, E. Bonilla, J. Cavazos, B.Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 295–305, Mar. 2006.

[2] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Compilation order matters: Exploring the structure of the space of compilation sequences using randomized search algorithms. In *Proceedings of the ACM SIGPLAN Symposium on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 231–239, June 2004.

[3] M. Arnold, S. Finka, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 47–65, Oct. 2000.

[4] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization in Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 111–129, Oct. 2002.

[5] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proceedings of the Workshop on Profile and Feedback-Directed Compilation, in Conjunction with the Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 1998.

[6] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 185–197, Mar. 2007.

[7] J. Cavazos and M. O'Boyle. Automatic tuning of inlining heuristics. In *Proceedings of the Supercomputing Conference on High Performance Networking and Computing*, Nov. 2005.

[8] J. Cavazos and M. O'Boyle. Method-specific dynamic compilation using logistic regression. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 229–240, Oct. 2006.

[9] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. In *Proceedings of the Workshop on Feedback-Directed and Dynamic Optimization (FDDO)*, Nov. 1999.

[10] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–9, May 1999.

[11] K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms.* Wiley, 2001.

[12] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. Quick and practical run-time evaluation of multiple program optimizations. *Transactions on High Performance Embedded Architectures and Compilation Techniques (HiPEAC)*, 1(1):13–31, 2006.

[13] S. V. Gheorghita, H. Corporaal, and T. Basten. Iterative compilation for energy reduction. *Journal of Embedded Computing*, 1(4):509–520, July 2005.

[14] E. Granston and A. Holler. Automatic recommendation of compiler options. In *Proceedings of the Workshop on Feedback-Directed and Dynamic Optimization (FDDO)*, Dec. 2001.

[15] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Optimizing general purpose compiler optimization. In *Proceedings of the International Conference on Computing Frontiers*, pages 180–188, May 2005.

[16] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[17] K. Ishizaki, T. M., K. Kawachiya, T. Suganuma, O. Gohda, T. Inagaki, A. Koseki, K. Ogata, M. Kawahito, T. Yasue, T. Ogasawara, T. Onodera, H. Komatsu, and T. Nakatani. Effectiveness of cross-platform optimizations for a Java just-in-time compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 187–204, Oct. 2003.

[18] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 171–182, June 2004.

[19] D. Maier, P. Ramarao, M. Stoodley, and V. Sundaresan. Experiences with multithreading and dynamic class loading in a Java just-in-time compiler. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 87–97, Mar. 2006.

[20] Z. Pan and R. Eigenmann. Fast, automatic, procedure-level performance tuning. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 173–181, Sept. 2006.

[21] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. Design and evaluation of dynamic optimizations for a Java just-in-time compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):732–785, July 2005.

[22] S. Triantafyllis, M. Vachharajani, and D. I. August. Compiler optimization-space exploration. *Journal of Instruction-level Parallelism*, Jan. 2005. Accessible at `http://www.jilp.org/vol7`.

[23] H. Wu, E. Park, M. Kaplarevic, and Y. Zhang. Dynamic optimization option search in GCC. In *Proceedings of the GCC Developers Summit 2007*, June 2007.

[24] M. Zhao, B. R. Childers, and M. L. Soffa. Predicting the impact of optimizations for embedded systems. In *Proceedings of the ACM SIGPLAN Symposium on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–11, June 2003.

[25] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength pareto evolutionary algorithm. Technical Report TIK-Report 103, Swiss Federal Institute of Technology (ETH) Zurich, May 2001.

[26] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: A comparative case study and the strength perato approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, Nov. 1999.