Collective Optimization

Grigori Fursin and Olivier Temam

ALCHEMY Group, INRIA Futurs, France
HiPEAC members
{grigori.fursin,olivier.temam}@inria.fr

Abstract. Iterative compilation is an efficient approach to optimize programs on rapidly evolving hardware, but it is still only scarcely used in practice due to a necessity to gather a large number of runs often with the same data set and on the same environment in order to test many different optimizations and to select the most appropriate ones. Naturally, in many cases, users cannot afford a training phase, will run each data set once, develop new programs which are not yet known, and may regularly change the environment the programs are run on.
In this article, we propose to overcome that practical obstacle using *Collective Optimization*, where the task of optimizing a program leverages the experience of many other users, rather than being performed in isolation, and often redundantly, by each user. Collective optimization is an unobtrusive approach, where performance information obtained after each run is sent back to a central database, which is then queried for optimizations suggestions, and the program is then recompiled accordingly. We show that it is possible to learn across data sets, programs and architectures in non-dynamic environments using static function cloning and run-time adaptation without even a reference run to compute speedups over the baseline optimization. We also show that it is possible to simultaneously learn and improve performance, since there are no longer two separate training and test phases, as in most studies. We demonstrate that extensively relying on *competition* among pairs of optimizations (*program reaction to optimizations*) provides a robust and efficient method for capturing the impact of optimizations, and for reusing this knowledge across data sets, programs and environments. We implemented our approach in GCC and will publicly disseminate it in the near future.

1 Introduction

Many recent research efforts have shown how iterative compilation can outperform static compiler optimizations and quickly adapt to complex processor architectures [33, 9, 6, 24, 16, 10, 20, 31, 27, 26, 18, 19]. Over the years, the approach has been perfected with fast optimization space search techniques, sophisticated machine-learning algorithms and continuous optimization [25, 29, 28, 34, 3, 8, 32, 23, 21]. And, even though these different research works have demonstrated significant performance improvements, the technique is far from mainstream in production environments. Besides the usual inertia for adopting new

approaches, there are hard technical hurdles which hinder the adoption of iterative approaches.

The most important hurdle is that iterative techniques almost all rely on a large number of training runs (either from the target program or other training programs) to *learn* the best candidate optimizations. And most of the aforementioned articles run the *same programs*, generated with the exact *same compiler* on the *same architecture* with the *same data sets*, and do this *a large number of times* (tens, hundreds or thousands of times) in order to deduce the shape of the optimization space. Naturally, in practice, a user is not going to run the same data set multiple times, will change architectures every so often, and will upgrade its compiler as well. We believe this practical issue of collecting a large number of training information, relying only on *production* runs (as opposed to training runs where results are not used) to achieve good performance is the crux of the slow adoption of iterative techniques in real environments.

We propose to address this issue with the notion of *Collective Optimization*. The principle is to consider that the task of optimizing a program is not an isolated task performed by each user separately, but a *collective* task where users can mutually benefit from the experience of others. Collective optimization makes sense because most of the programs we use daily are run by many other users, either globally if they are general tools, or within our or a few institutions if they are more domain-specific tools. Achieving collective optimization requires to solve both an *engineering* and a *research* issue.

The engineering issue is that users should be able to seamlessly share the outcome of their runs with other users, without impeding execution or compilation speed, or complicating compiler usage. The key research issue is that we must progressively improve overall program performance while, *at the same time*, we learn *how it reacts to the various optimizations*, all solely using *production runs*; in reality, there is no longer such a thing as a training phase followed by a test/use phase, both occur simultaneously. Moreover, we must understand *whether* it is possible and *how* to learn across data sets, programs or platforms. An associated research issue is to come up with a knowledge representation that is relevant across data sets, programs and platforms. Finally, because a user will generally run a data set only once, we must learn the impact of optimizations on program performance without even a *reference* run to decide whether selected optimizations improved or degraded performance compared to the baseline optimization.

In this article, we show that it is possible to continuously learn across data sets, programs or platforms, relying solely on production runs, and progressively improving overall performance across runs, reaching close to the best possible iterative optimization performance, itself achieved under idealized (and non-realistic) conditions. We show that extensively relying on *competition* among pairs of optimizations provides a robust and efficient method for capturing the impact of optimizations on program performance, without requiring reference runs and while remaining relevant across data sets, programs and architectures. While most recent research studies are focused on learning across programs [28,

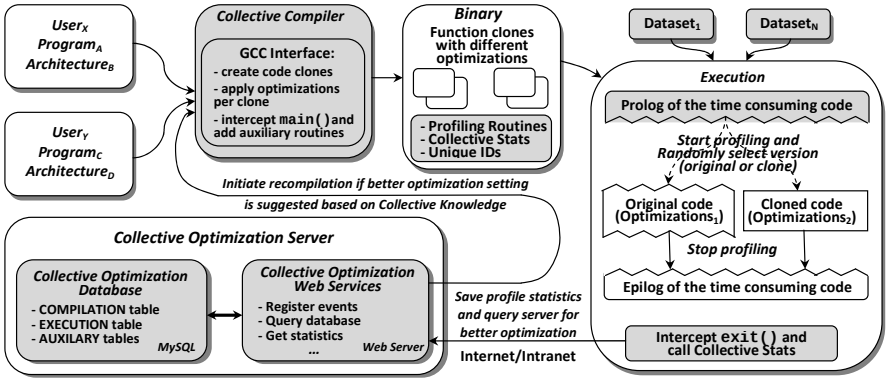


Fig. 1. *Collective Optimization Framework*

3, 7], we find that, in practice, learning across data sets, and to a lesser extent, across architectures, is significantly more important and useful. Finally, we present a solution to the engineering issue in the form of an extension to GCC which relies on a central database for transparently aggregating runs results from many users, and performing competitions between optimizations during runs.

2 Experimental Setup

In order to perform a realistic evaluation of collective optimization, each benchmark has to come with several data sets in order to emulate truly distinct runs. To our knowledge, only the MiDataSets [13] data set suite based on the MiBench [17] benchmark suite currently provides 20+ data sets for 26 benchmarks.

All programs are optimized using the GCC 4.2.0 compiler. The collective optimization approach and framework are compatible with other compilers, but GCC is now becoming a competitive optimization compiler with a large number of embedded program transformation techniques. We identified 88 program transformations, called through corresponding optimization flags, that are known to influence performance, and 8 parameters for each parametric optimization. These transformations are randomly selected to produce an optimization combination.

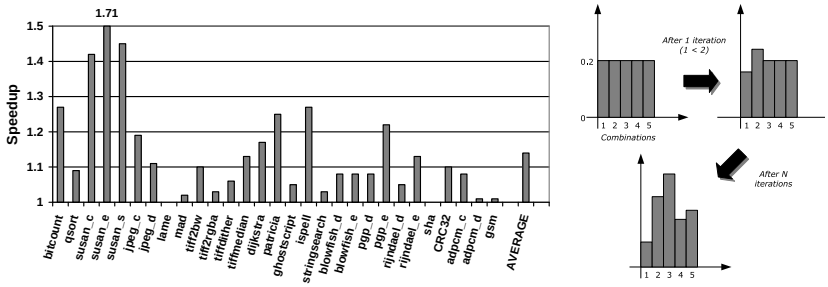
In order to unobtrusively collect information on a program run, and reoptimize the program, GCC is modified so as to add to each program a routine which is executed when the program terminates. This termination routine collects several information about the program (a program identifier, architecture and compiler identifiers, which optimizations were applied) and about the last run (performance measurements; currently, execution time and profiling information), and stores them into a remote database.

Then, it queries a server associated with the remote database in order to select the next optimizations combination. The recompilation takes place peri-

odically (set by user) in the background, between two runs.[1] No other modification takes place until the next run, where the process loops again, as shown in Figure 1.

The programs were compiled and run on three architectures - AMD Athlon XP 2800+ (AMD32) - 5 machines, AMD Athlon 64 3700+ (AMD64) - 16 machines and Intel Xeon 2.80GHz (IA32) - 2 machines.

3 Motivation



(a) (b)

Fig. 2. *(a) Performance upper-bound of Collective Optimization (AMD Athlon 64 3700+, GCC 4.2.0), (b) Computing the probability distribution to select an optimization combination based on continuous competition between combinations*

The experimental methodology of research in iterative optimization usually consists in running many times the same program on the same data set and on the same platform. Hence, it can be interpreted as an idealized case of collective optimization, where the experience of others (program, data set, platform) would always perfectly match the target run. In other words, it is a case where no experimental noise would be introduced by differences in data sets, programs or platforms. Consequently, iterative optimization can be considered as a performance upper-bound of collective optimization. Figure 2(a) shows the best speedup achieved for each benchmark and each data set (averaged over 20 distinct data sets) over the highest GCC optimization level (-O3) by selecting the best optimizations combination among 200 for each program and data set. This experiment implicitly shows that collective optimization has the potential to yield high speedups.

[1] Note that if the recompilation is not completed before another run starts, this latter run just uses the same optimizations as the previous run, and the evaluation of the new optimizations is just slightly delayed by one or a few runs.

4 Overview

This section provides an overview of the proposed approach for collective optimization. The general principle is that performance data about each run is transparently collected and sent to a database; and, after each run, based on all the knowledge gathered so far, a new optimizations combination is selected and the program is recompiled accordingly. The key issue is which optimizations combination to select for each new run, in order to both gather new knowledge and keep improving average program performance as we learn.
In collective optimization, several global and program-specific probability distributions capture the accumulated knowledge. Combinations are randomly selected from one of several probability distributions which are progressively built at the remote server.

The different "maturation" stages of a program. For each program, and depending on the amount of accumulated knowledge, we distinguish three scenarios: (1) the server may not know the program at all (new program), (2) only have information about a few runs (infrequently used or a recently developed program), or (3) have information about many runs.

Stage 3: Program well known, heavily used. At that maturation stage, enough runs have been collected for that program that it does not need the experience of other programs to select the most appropriate optimizations combinations for itself. This knowledge takes the form of a program-specific probability distribution called d_3. Stage 3 corresponds to learning across data sets.

Stage 2: Program known, a few runs only. At that maturation stage, there is still insufficient information (program runs) to correctly predict the best combinations by itself, but there is already enough information to start "characterizing" the program behavior. This characterization is based on the comparison of the impact of optimizations combinations tried so far on the program against their impact on other programs (program reaction to optimizations). If two programs behave alike for a subset of combinations, they may well behave alike for all combinations. Based on this intuition, it is possible to find the best matching program, after applying a few combinations to the target program. Then, the target program probability distribution d_2 is given by the distribution d_3 of the matching program. This matching can be revisited with each additional information (run) collected for the target program. Stage 2 corresponds to learning across programs.

Stage 1: Program unknown. At that stage, almost no run has been performed, so we leverage and apply optimizations suggested by the "general" experience collected over all well-known programs. The resulting d_1 probability distribution is the unweighted average of all d_3 distributions of programs which have reached Stage 3. Stage 1 is an elementary form of learning across programs.

Selecting stages. A program does not follow a monotonic process from Stage 1 to Stage 3, even though it should intuitively mature from Stage 1 to Stage 2 to Stage 3 in most cases. There is a permanent *competition* between the different stages distributions (d_1, d_2, d_3). At any time, a program may elect to draw optimizations combinations from any stage distribution, depending on

which one appears to perform best so far. In practice and on average, we find that Stage 3 (learning across data sets) is by far the most useful stage. Stage 1 and Stage 2 stages are respectively useful in the first ten, and the first hundreds runs of a program on average, but Stage 3 rapidly becomes dominant. The competition between stages is implemented through a "meta" distribution d_m, which reflects the current score of each stage distribution for a given program. Each new run is a two-step random process: first, the server randomly selects the distribution to be used, and then, it randomly selects the combination using that distribution. How scores are computed is explained in Section 5. Using that meta-distribution, the distribution with the best score is usually favored.

5 Collective Learning

In this section, we explain in more detail how to compute the aforementioned distributions to achieve collective learning.

5.1 Building the program distribution d_3 using statistical comparison of optimizations combinations

Comparing two combinations C_1, C_2. In order to build the aforementioned distributions, one must be able to compare the impact of any two optimizations combinations C_1, C_2 on program performance.

However, even the simple task of deciding whether $C_1 > C_2$ can become complex in a real context. Since the collective optimization process only relies on production runs, two runs usually correspond to two distinct data sets. Therefore, if two runs with respective execution times T_1 and T_2, and where optimizations combinations C_1 and C_2 have been respectively applied, are such that $T_1 < T_2$, it is not possible to deduce that $C_1 > C_2$.

To circumvent that issue, we perform run-time comparison of *two* optimizations combinations using *cloned* functions. C_1 and C_2 are respectively applied to the clones f_1 and f_2 of a function f. At run-time, for each call to f, either f_1 or f_2 is called; the clone called is randomly selected using an additional branch instruction and a simple low-overhead pseudo-random number generation technique emulating uniform distribution. Even if the routine workload varies upon each call, the random selection of the clone to be executed ensures that the average workload performed by each clone is similar. As a result, the non-optimized versions of f_1 and f_2 account for about the same fraction of the overall execution time of f. Therefore, if the average execution time of the clone optimized with C_1 is smaller than the average execution time of the clone optimized with C_2, it is often correct to deduce that C_1 is better than C_2, i.e., $C_1 > C_2$. This statistical comparison of optimizations combinations requires no reference, test or training run, and the overhead is negligible.

We have shown in [14] the possibility to detect the influence of optimizations for statically compiled programs with stable behavior using function cloning and run-time low-overhead phase detection. Stephenson et al. [30] and Lau et al. [22]

demonstrated how to evaluate different optimizations for programs with irregular behavior in dynamic environments using random function invocations and averaging collected time samples across a period of time. We combined these techniques to enable run-time transparent performance evaluation for statically-compiled programs with any behavior here.

On the first program run, profiling information is collected and sent to the database. All the most time-consuming routines accounting for 75% or more of the total execution time and with an average execution time per call greater than a platform-specific threshold are cloned. The purpose of this threshold is to ensure that the overhead of the additional branch instruction remains negligible (less than 0.1%) compared to the average execution time per function call. Since profiling information is periodically collected at random runs, more routines can be added during the next runs (for instance if different parts of the call graph are reached depending on the data sets), the target routines are not set in stone after the first run. More implementation details are provided in Section 6.

Computing d_3. When two combinations C_1 and C_2 are compared on a program using the aforementioned cloned routines, the only information recorded is whether $C_1 > C_2$ or $C_1 < C_2$. Implicitly, a run is a *competition* between two optimizations combinations, and the winning combination scores 1 and the losing is 0 as shown in Figure 2(b). These scores are cumulated for each combination and program. The scores are then normalized per combination, by the number of times the combination was tried (thus implicitly decreasing the average score of the losing combination). Then the overall distribution is normalized so that the sum of all combinations scores (probabilities) is 1.

Because this distribution only reflects the *relative* "merit" of each combination, and not the absolute performance (e.g., execution time or speedup), it is a fairly resilient metric, tolerant to variations in measurements.

5.2 Building the aggregate distribution d_1

d_1 is simply the average of all d_3 distributions of each program. d_1 reflects the most common cases: which optimizations combinations perform best in general. It is also possible to compose more restricted aggregate distributions, such as per architecture, per compiler, per programs subsets,..., though this is left for future work.

5.3 Building the matching distribution d_2

Stage 2 is based on the intuition that it is unlikely that all programs exhibit widely different behavior with respect to compiler optimizations, or conversely that, once the database is populated with a sufficient number of programs, it is likely that a new program may favor some of the same optimizations combinations as some of the programs already in the database. The main difficulty is then to identify which programs best correspond to the current target one. Therefore, we must somehow *characterize* programs, and this characterization should reflect which optimizations combinations a program favors.

As for d_3, we use the metric-independent comparison between two optimizations combinations C_1 and C_2. E.g., $C_1 > C_2$ is a *reaction to program optimizations* and is used as one *characterization* of the program. Let us assume that $C_1 > C_2$ for the target program P and $C_1 > C_2$ for a program P' and $C_1 < C_2$ for a program P'' compared against P. Then, P' gets a score of 1, and P'' a score of 0. The program with the best score is considered the *matching* program, and d_2 is set to the d_3 of that program. In other words, for d_2 we use a competition among *programs*. The more combinations pairs (reactions to optimizations) are compared, the more accurate and reliable the program matching.

Still, we observed that beyond 100 characterizing combinations pairs (out of $C^2_{100} = \frac{200\times199}{/}2 = 19900$ possible combinations pairs), performance barely improves. In addition, it would not be practical to recompute the matching upon each run based on an indefinitely growing number of characterizations. Therefore, we restrict the characterization to 100 combinations pairs, which are collected within a rolling window (FIFO). However, the window only contains distinct optimizations combinations pairs. The rolling property ensures that the characterization is permanently revisited and rapidly adapted if necessary. The matching is attempted as soon as one characterization is available in the window, and continuously revisited with each new modification of the rolling window.

Cavazos et al. [7] have shown that it is possible to improve similar program characterizations by identifying and then restricting to optimizations which carry the most information using the *mutual information* criterion. However, these optimizations do not necessarily perform best, they are the most *discriminatory* and one may not afford to "test" them in production runs. Moreover, we will later see that this approach could only yield marginal improvement in the start-up phase due to the rapid convergence of Stage $3/d_3$.

5.4 Scoring distributions

As mentioned in Section 4, a meta-distribution is used to select which stage distribution is used to generate the next optimizations combination. For each run, two distributions d and d' are selected using two draws from the meta-distribution (they can be the same distributions). Then, an optimizations combination is drawn from each distribution (C_1 using d and C_2 using d'), which will compete during the run. Scoring is performed upon the run completion; note that if C_1 and C_2 are the same combinations, no scoring takes place.

Let us assume, for instance, that for the run, $C_1 > C_2$. If, according to d, $C_1 > C_2$ also, then one can consider that d "predicted" the result right, and gets a score of 1. Conversely, it would get a score of 0. The server also keeps track of the number of times each distribution is drawn, and the distribution value in the meta-distribution is the ratio of the sum of all its scores so far and the number of times it was drawn. Implicitly, its score decreases when it gets a 0, increases when it gets a 1, as for individual distributions.

This scoring mechanism is robust. If a distribution has a high score, but starts to behave poorly because the typical behavior of the program has changed (e.g., a very different kind of data sets is used), then its score will plummet, and

the relative score of other distributions will comparatively increase, allowing to discover new strong combinations. Note that d_3 is updated upon every run (with distinct combinations), even if it was not drawn, ensuring that it converges as fast as possible.

6 Collective Compiler

Program identification. At the moment, a program is uniquely identified using a 32-byte MD5 checksum of all the files in its source directory. This identifier is sufficient to distinguish most programs, and it has the added benefit of not breaking confidentiality: no usable program information is sent to the database. In the future, we also plan to use the vector of program reactions to transformations as a simple and practical way to characterize programs based solely on execution time.

Termination routine. In order to transparently collect run information, we modified GCC to intercept the compilation of the `main()` function, and to insert another interceptor on the `exit()` function. Whenever the program execution finishes, our interceptor is invoked and it in turn checks whether the *Collective Stats Handler* exists, invokes it to send program and run information to the *Collective Optimization Database*. At any time, the user can opt in or out of collective optimization by setting or resetting an environment variable.

Cloning. As mentioned before, optimizations combinations are evaluated through cloned routines. These routines are the most time-consuming program routines (top 3 routines and/or 75% or more of the execution time). They are selected using the standard `gprof` utility. The program is profiled at the first and then random runs. Therefore, the definition of the top routines can change across runs. We progressively build an average ranking of the program routines, possibly learning new routines as they are exercised by different data sets. The speedups mentioned in the following performance evaluation section are provided with respect to a *non-instrumented* version of the program, i.e., they factor in the instrumentation, which usually has a negligible impact.

We modified GCC to enable function cloning and be able to apply different optimizations directly to clones and original functions. This required changes in the core of the compiler since we had to implement full replication of parts of a program AST, and to change the optimization pass manager to be able to select specific optimizations combinations on a function level. When GCC clones a function, it inserts profiling calls at the prolog and epilog of the function, replaces `static` variables and inserts additional instructions to randomly select either the original or the cloned version.

Security. The concept of collective optimization raises new issues, especially security. First, very little program information is in fact sent to the database. The profile routine names are locally hashed using MD5, and only run-time statistics (execution times) are sent. Second, while we intend to set up a global and openly accessible database, we do expect companies will want to set up their own internal collective optimization database. Note that they can then get

the best of both worlds: leverage/read information accessible from the global database, while solely writing their runs information to their private database. At the moment, our framework is designed for a single database, but this two-database system is a rather simple evolution.

7 Performance Evaluation

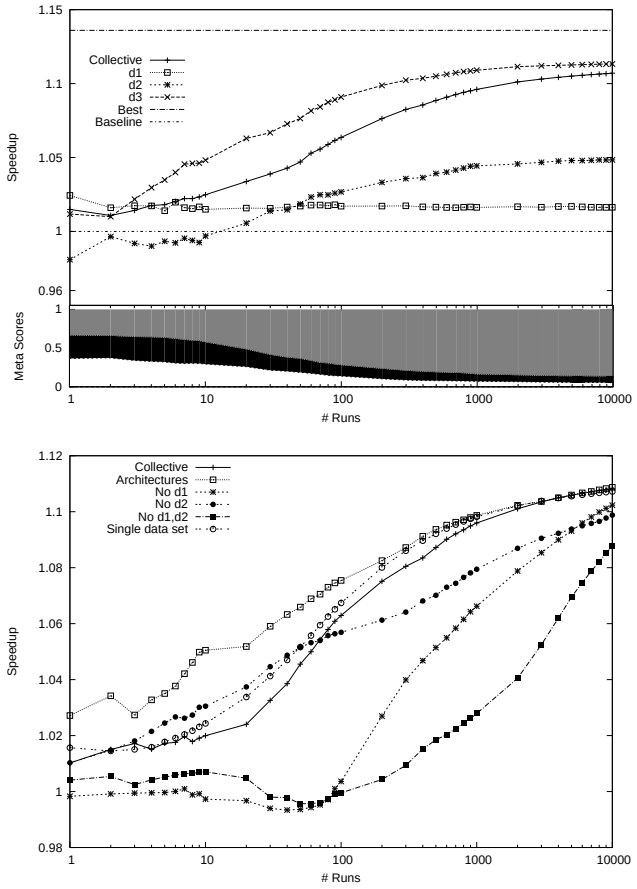


Fig. 3. *(a) Average performance of collective optimization and individual distributions (bottom: meta-scores of individual distributions; grey is d_3, black is d_2, white is d_1), (b) Several collective optimization variants.*

In Figure 3(a), *Collective* corresponds to the full process described in earlier sections, where the appropriate distribution is selected using the meta-distribution before every run; performance is averaged over all programs (for

instance, `Run=1` corresponds to performance averaged over 1 random run for each program). For each program, we have collected 20 data sets and can apply 200 different optimizations combinations, for a total of 4000 distinct runs per program. The main approximation of our evaluation lays in the number of data sets; upon each run, we (uniformly) randomly select one among 20 data sets. However, several studies have shown that data sets are often clustered within a few groups breeding similar behavior [11], so that 20 data sets exhibiting sufficiently distinct behavior, may be considered a non-perfect but reasonable emulation of varying program behavior across data sets. In order to further assess the impact of using a restricted number of data sets, we have evaluated the extreme case where a single data set is used. These results are reported in Figure 3(b), see *Single data set*, where a single data set is used per program in each experiment, and then, for each x-axis value (number of runs), performance is averaged over all programs and all data sets. Using a single data set improves convergence speed though only moderately, suggesting *Collective* could be a slightly optimistic but reasonable approximation of a real case where all data sets are distinct.

After 10000 runs per program, the average *Collective* speedup, 1.11, is fairly close to the *Best* possible speedup, 1.13, the asymptotic behavior of single-data set experiments. The other graphs (*d1, d2, d3*) report the evolution of the average performance of optimizations combinations drawn from each distribution. At the bottom of the figure, the grey filled curve corresponds to the meta score of d_3, the black one to d_2 and the white one to d_1.

Learning across programs. While the behavior of d_2 in Figure 3(a) suggests that learning across programs yields modest performance improvements, this experiment is partly misleading. d_3 rapidly becomes a dominant distribution, and as explained above, quickly converges to one or a few top combinations due to restricted interval polling. d_2 performance will improve as more characterizing optimizations combinations pairs fill up the rolling window. And Figure 3(b) shows that without d_2, the meta distribution does not converge as fast or to an as high asymptotic value.

Collective *versus* d_3. While the better performance of d_3 over *Collective*, in Figure 3(a), suggests this distribution should solely be used, one can note its performance is not necessarily the best in the first few runs, which is important for infrequently used codes. Moreover, the average *Collective* performance across runs becomes in fact very similar after d_3 has become the dominant distribution, since mostly d_3 combinations are then drawn. But a more compelling reason for privileging *Collective* over d_3 is the greater robustness of collective optimization thanks to its meta-distribution scheme.

In Figure 3(b), we have tested collective optimization without either d_1, d_2 or neither one. In the latter case, we use the uniform random distribution to discover new optimizations, and the meta-distribution arbitrates between d_3 and *uniform*; by setting the uniform distribution initial meta-score to a low value with respect to d_3, we can both quickly discover good optimizations without degrading average performance;[2] the uniform distribution is not used when only

[2] This is important since the average speedup of the uniform distribution alone is 0.7.

d_1 or d_2 are removed. As shown in Figure 3(b), collective optimization converges more slowly when either d_1, d_2 or both are not used. These distributions help in two ways. d_1 plays its main role at start-up, by bringing a modest average 2% improvement, and performance starts lower when it is not used. Conversely, d_2 is not useful at start-up, but provides a performance boost after about 50 to 100 runs when its window is filled and the matching is more accurate. d_2 does in fact significantly help improve the performance of *Collective* after 100 runs, but essentially by discovering good new optimizations, later adopted by d_3, rather than due to the intrinsic average performance of the optimizations combinations suggested by d_2.

Learning across architectures. Besides learning across data sets and programs, we have also experimented with learning across architectures. We have collected similar runs on an Athlon 32-bit (AMD32) architecture and an Intel 32-bit (IA32) architecture (recall all experiments above are performed on an Athlon 64-bit architecture), and we have built the d_3 distributions for each program. At start-up time, on the 64-bit architecture, we now use a d_4 distribution corresponding to the d_3 distribution for this program but other architectures (and 19 data sets, excluding the target data set); the importance of d_4 will again be determined by its meta-score. The rest of the process remains identical. The results are reported in curve *Architectures* on Figure 3(b). Start-up performance does benefit from the experience collected on the other architectures. However, this advantage fades away after about 2000 runs. We have also experimented with simply initializing d_3 with the aforementioned d_4 instead of using a separate d_4 distribution. However the results were poorer because the knowledge acquired from other architectures was slowing down the rate at which d_3 could learn the behavior of the program on the new architecture.

8 Background and Related Work

Iterative or adaptive compilation techniques usually attempt to find the best possible combinations and settings of optimizations by scanning the space of all possible optimizations. [33, 9, 6, 24, 10, 20, 31, 27, 26, 19] demonstrated that optimizations search techniques can effectively improve performance of statically compiled programs on rapidly evolving architectures, thereby outperforming state-of-the-art compilers, albeit at the cost of a large number of exploration runs.

Several research works have shown how machine-learning and statistical techniques [25, 29, 28, 34] can be used to select or tune program transformations based on program features. Agakov et al. [3] and Cavazos et al. [8] use machine-learning to focus iterative search using either syntactic program features or dynamic hardware counters and multiple program transformations. Most of these works also require a large number of training runs. Stephenson et al. [28] show more complementarity with collective optimization as program matching is solely based on static features.

Several frameworks have been proposed for continuous program optimization [4, 32, 23, 21]. Such frameworks tune programs either during execution or offline, trying different program transformations. Such recent frameworks like [21] and [23] pioneer lifelong program optimization, but they expose the concept rather than research practical knowledge management and selection strategies across runs, or unobtrusive optimization evaluation techniques. Several recent research efforts [14, 22, 30] suggest to use procedure cloning to search for best optimizations at run-time. In this article we combine and extend techniques from [14] that are compatible with regular scientific programs and use low-overhead runtime phase detection, and methods from [30, 22] that can be applied to programs with irregular behavior in dynamic environments by randomly executing code versions and using statistical analysis of the collected execution times with a confidence metric. Another recent research project investigates the potential of optimizing static programs across multiple data sets [13] and suggests this task is tractable though not necessarily straightforward.

The works closest to ours are by Arnold et al. [5] and Stephenson [30]. The system in [5] collects profile information across multiple runs of a program in IBM J9 Java VM to selectively apply optimizations and improve further invocations of a given program. However it doesn't enable optimization knowledge reuse from different users, programs and architectures. On the contrary, Stephenson tunes a Java JIT compiler across executions by multiple users. While several aspects of his approach is applicable to static compilers, much of his work focuses on Java specifics, such as canceling performance noise due to methods recompilation, or the impact of garbage collection. Another distinctive issue is that, in a JIT, the time to predict optimizations and to recompile must be factored in, while our framework tolerates well long lapses between recompilations, including several runs with the same optimizations. Finally, we focus more on the impact of data sets from multiple users and the optimization selection robustness (through competitions and meta-distribution).

9 Conclusions and Future Work

The first contribution of this article is to identify the true limitations of the adoption of iterative optimization in production environments, while most studies keep focusing on showing the performance potential of iterative optimization. We believe the key limitation is the large amount of knowledge (runs) that must be accumulated to efficiently guide the selection of compiler optimizations. The second contribution is to show that it is possible to simultaneously *learn* and *improve* performance across runs. The third contribution is to propose multi-level competition (among optimizations and their distributions which capture different program knowledge maturation stages, and among programs) to understand the impact of optimizations without even a reference run for computing speedups, while ensuring optimization robustness. The program reactions to transformations used to build such distributions provide a simple and practical way to characterize programs based solely on execution time. The fourth

contribution is to highlight that knowledge accumulated across data sets for a single program is more useful, in the real and practical context of collective optimization, than the knowledge accumulated across programs, while most iterative optimization studies focus on knowledge accumulated across programs; we also conclude that knowledge across architectures is useful at start-up but does not bring any particular advantage in steady-state performance. The fifth and final contribution is to address the engineering issue of unobtrusively collecting runs information for statically-compiled programs using function cloning and run-time adaptation mechanism.

The collective optimization approach opens up many possibilities which can be explored in the future. We plan to use it to automatically and continuously tune default GCC optimization heuristic or individual programs using recent plugin system for GCC [15, 2] that allows to invoke transformations directly, change their parameters, orders per function or even add plugins with new transformations to improve performance, code size, power, and so on. After sufficient knowledge has been accumulated, the central database may become a powerful tool for defining truly representative benchmarks. We can also refine optimizations at the data set level by clustering data sets and using our cloning and run-time adaptation techniques to select the most appropriate optimizations combinations or even reconfigure processor at run-time thus creating self-tuning intelligent ecosystem. We plan to publicly disseminate our collective optimization framework, the run-time adaptation routines for GCC based on [12, 8] as well as the associated central database at [1] in the near future.

10 Acknowledgments

This research was partially supported by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC), and the IST STREP project MILEPOST. The authors would also like to thank colleagues and anonymous reviewers for their feedback.

References

1. Continuous Collective Compilation Framework. `http://cccpf.sourceforge.net`.
2. GCC Interactive Compilation Interface. `http://gcc-ici.sourceforge.net`.
3. 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)*, 2006.
4. J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone. In *Technical Report 1997-016. Digital Equipment Corporation Systems Research Center, Palo Alto, CA*, 1997.
5. M. Arnold, A. Welc, and V.T.Rajan. Improving virtual machine performance using a cross-run profile repository. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*, 2005.

6. 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*, 1998.
7. J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. O'Boyle, G. Fursin, and O. Temam. Automatic performance model construction for the fast software exploration of new hardware designs. In *Proceedings of the International Conference on Compilers, Architecture, And Synthesis For Embedded Systems (CASES)*, October 2006.
8. J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the 5th Annual International Symposium on Code Generation and Optimization (CGO)*, March 2007.
9. K. Cooper, P. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–9, 1999.
10. K. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23(1), 2002.
11. L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 5:1–33, 2003.
12. B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded programs. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.
13. G. Fursin, J. Cavazos, M. O'Boyle, and O. Temam. Midatasets: Creating the conditions for a more realistic evaluation of iterative optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC)*, January 2007.
14. G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *Proceedings of the 1st International Conference on High Performance Embedded Architectures & Compilers (HiPEAC)*, number 3793 in LNCS, pages 29–46. Springer Verlag, November 2005.
15. G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, P. Barnard, E. Ashton, E. Courtois, F. Bodin, E. Bonilla, J. Thomson, H. Leather, C. Williams, and M. O'Boyle. Milepost gcc: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*, June 2008.
16. G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computers (LCPC)*, pages 305–315, 2002.
17. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.
18. K. Heydemann and F. Bodin. Iterative compilation for two antagonistic criteria: Application to code size and performance. In *Proceedings of the 4th Workshop on Optimizations for DSP and Embedded Systems, colocated with CGO*, 2006.
19. K. Hoste and L. Eeckhout. Cole: Compiler optimization level exploration. In *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, 2008.
20. P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In

Proc. Languages, Compilers, and Tools for Embedded Systems (LCTES), pages 12–23, 2003.
21. C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, Palo Alto, California, March 2004.
22. J. Lau, M. Arnold, M. Hind, and B. Calder. Online performance auditing: Using hot optimizations without getting burned. In *Proceedings of the ACM SIGPLAN Conference on Programming Languaged Design and Implementation (PLDI)*, 2006.
23. J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu. Design and implementation of a lightweight dynamic optimization system. In *Journal of Instruction-Level Parallelism*, volume 6, 2004.
24. F. Matteo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.
25. A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the International Conference on Artificial Intelligence: Methodology, Systems, Applications*, LNCS 2443, pages 41–50, 2002.
26. Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 319–332, 2006.
27. B. Singer and M. Veloso. Learning to predict performance from formula modeling and training data. In *Proceedings of the Conference on Machine Learning*, 2000.
28. M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 2005.
29. M. Stephenson, M. Martin, and U. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 77–90, 2003.
30. M. W. Stephenson. *Automating the Construction of Compiler Heuristics Using Machine Learning*. PhD thesis, MIT, USA, 2006.
31. S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 204–215, 2003.
32. M. Voss and R. Eigenmann. Adapt: Automated de-coupled adaptive program transformation. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, 2000.
33. R. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the Conference on High Performance Networking and Computing*, 1998.
34. M. Zhao, B. R. Childers, and M. L. Soffa. A model-based framework: an approach for profit-driven optimization. In *Proceedings of the Interational Conference on Code Generation and Optimization (CGO)*, pages 317–327, 2005.