Collective Optimization: A Practical Collaborative Approach

GRIGORI FURSIN and OLIVIER TEMAM
INRIA Saclay, France
HiPEAC members

Iterative optimization is a popular and efficient research approach to optimize programs using feedback-directed compilation. However, one of the key limitations that prevented widespread use in production compilers and day-to-day practice is the necessity to perform a large number of program runs with the same dataset and environment (architecture, OS, compiler) to test many different combinations of optimizations. In this article, we propose to overcome such a practical obstacle using *collective optimization*, where the task of optimizing a program or tuning default compiler optimization heuristic leverages the experience of many other users continuously, rather than being performed in isolation, and often redundantly, by each user. During this unobtrusive approach, performance information is sent to a central database after each run and statistically combined with the data from all users to suggest most profitable optimizations for a given program and an architecture, or to gradually improve default optimization level of a compiler for a given architecture.

In this article, we address two key challenges of collective optimization. We show that it is possible to simultaneously learn and improve performance while avoiding long training phases. We also demonstrate how to use our approach with static compilers to learn optimizations across multiple datasets and architectures without even a reference run normally needed to compute speedups over the baseline optimization by using static function cloning and dynamic adaptation. We present a novel probabilistic approach based on *competition* among pairs of optimizations *(program reaction to optimizations)* to enable optimization knowledge reuse and achieve nearly the best possible iterative optimization performance. We implemented our technique in GCC (widespread production open-source compiler that supports multiple architectures) and connected it to a public collective optimization database at cTuning.org to gather profile and optimization data continuously and transparently in realistic environments ranging from desktop PCs and mobile systems to supercomputers and data centers.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors-*Compilers; optimization*; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; C.4 [**Computer Systems Organization**]: Performance of Systems-*Measurement techniques; modeling techniques*

A preliminary version of this research is described in *Proceedings of the International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC'09).*
This work has been partially supported by the EU IST FP6 STREP MILEPOST Project (035307) [Link-MILEPOST] and Google Summer of Code Program'09.
Author's address: G.Fursin (corresponding author), O.Temam, INRIA Saclay, 3, rue Jean Rostand, Orsay, 91893, France; email:Grigori.Fursin@unidapt.org or Grigori.Fursin@inria.fr

General Terms: Design, Languages, Measurement, Experimentation, Performance
Additional Key Words and Phrases: Collective optimization, iterative compilation, continuous optimization, statistical optimization, adaptive compiler, self-tuning computing systems, collective optimization database, multiple datasets, program characterization, program reaction to optimization, function cloning, runtime adaptation

1. INTRODUCTION

Many recent research efforts have shown how iterative compilation can outperform static compiler optimizations and quickly adapt to complex processor architectures obtaining significant performance improvements [Whaley and Dongarra 1998; Matteo and Johnson 1998; Bodin et al. 1998; Cooper et al. 1999; Kisuki et al. 2000; Fursin et al. 2002; Cooper et al. 2002; Kulkarni et al. 2003; Triantafyllis et al. 2003; Singer and Veloso 2000; Pan and Eigenmann 2004; 2006; Hoste and Eeckhout 2008]. Over the years, the approach has been perfected with fast optimization space search techniques, sophisticated machine-learning algorithms, and continuous optimization [Voss and Eigenmann 2000; Monsifrot et al. 2002; Stephenson et al. 2003; Lu et al. 2004; Lattner and Adve 2004; Franke et al. 2005; Stephenson and Amarasinghe 2005; Zhao et al. 2005; Agakov et al. 2006; Qasem et al. 2006; Cavazos et al. 2007; Bailey et al. 2008; Fursin et al. 2008; Dubach et al. 2009]. Nevertheless, empirical iterative optimization is far from mainstream in production environments. Besides the usual inertia for adopting novel approaches, there are hard technical hurdles which hinder the adoption of iterative approaches.

One of the key challenges is that iterative techniques almost always rely on a large number of training runs (either from the target program or other training programs) to *learn* the best candidate optimizations. Moreover, all these runs must be performed with the *same programs*, generated with the *same compiler* on the *same architecture* with the *same datasets*, and repeated *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 can rarely afford execution of the same dataset multiple times, will change architectures every so often, and may eventually upgrade a 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 produced 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 executed in servers, data centers, and cloud computing systems or that we use daily on our mobiles and desktop PCs are also run by many other users, either globally if they are general tools or within one or a few institutions if they are more domain specific.

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 slowing down execution or compilation, should support statically compiled programs, and should avoid complicating compiler usage. The key research issue is that we must progressively improve overall program and compiler performance while, *at the same time*, we learn how programs *react to the various optimizations* using *production runs*. In this approach training phase and test/use phase occur simultaneously unlike traditional iterative compilation. Hence, we must understand *whether or not* it is possible and *how* to learn across datasets, programs or platforms at the same time. An associated research issue is to develop a knowledge representation scheme that is relevant across datasets, programs, and platforms. Finally, because a user will generally run a dataset only once, we must learn the impact of optimizations on program performance without even a *reference* run to decide whether selected optimizations improve or degrade performance compared to the baseline optimization.

In this article, we show that it is possible to continuously learn across datasets, programs, or platforms, relying solely on production runs, and progressively improve overall performance across runs, reaching close to the best possible iterative optimization performance, itself achieved under idealized (and nonrealistic) conditions. We show that extensively relying on statistical *competition* among pairs of optimizations provides a robust and efficient method for capturing the impact of optimizations on program performance, without requiring reference runs to calculate speedups and while remaining relevant across datasets, programs, and architectures. While most recent research studies are focused on learning across programs [Stephenson and Amarasinghe 2005; Agakov et al. 2006; Cavazos et al. 2007; Fursin et al. 2008; Dubach et al. 2009], we found that in the case of collective optimization, learning across datasets, 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 with collective optimization plugins which we connected to a central optimization repository through public Web services at cTuning.org [Link-repository ; Link-ICI ; Fursin 2009] to effectively distribute optimization process among many users, aggregate multiple profiling and optimization data, and perform program behavior characterization and continuous competitions between optimizations during runs.

2. EXPERIMENTAL SETUP

Benchmarks, datasets, and architectures used throughout the article are briefly introduced in this section.

Benchmarks and datasets. In order to perform a realistic evaluation of collective optimization, each benchmark has to come with several datasets in order to emulate truly distinct runs. To our knowledge, only the MiDataSets/cBench benchmark and dataset suite [Fursin et al. 2007; Link-MiDatasets ; Fursin 2009] based on the MiBench [Guthaus et al. 2001] currently provides more than 20 datasets for each of the 26 benchmarks.

Compiler and profiler. All programs are optimized using the GCC 4.2.0 compiler; several benchmarks (`qsort`, `dijkstra`, `patricia`, `stringsearch`) had to be modified in order to successfully compile with GCC. The collective optimization approach and framework are compatible with other compilers, but GCC is now becoming a competitive optimizing compiler with a large number of program transformation techniques and support for more than 30 different families of architectures.

We currently use the standard `gprof` tool to profile programs at function level. This tool may introduce some overhead which we briefly analyze in Section 5 and suggest possible solutions to reduce it. We use the Interactive Compilation Interface (ICI) [Link-ICI ; Fursin et al. 2008; Huang et al. 2010] to perform function cloning, select combinations of optimizations per function and instrument programs. ICI is a plugin system that acts as a "middleware" interface between production compilers such as GCC and user-definable research plugins. The ICI framework provides a high-level compiler-independent and a low-level compiler-dependent API that open up and reuse the available functionality of production compilers in order to transform them into stable, portable, and modular compiler research infrastructure and enable interactive control of all internal decisions.

Optimizations. We selected 88 program transformations such as inlining, unrolling, scheduling, register allocation, constant propagation among many others and which are known to influence performance, and 8 parameters for each parametric optimization. One should bear in mind that GCC has not been originally designed for research so it is not possible to explore the whole optimization space by simply combining multiple compiler optimization flags, since some of them are initiated only with a given global GCC optimization level (-Os,-O1,-O2,-O3). We overcome this issue by selecting a global optimization level -O1 .. -O3 first and then either turning on a particular optimization through a corresponding flag `-f<optimization name>` or turning it off using `-fno-<optimization name>` flag.

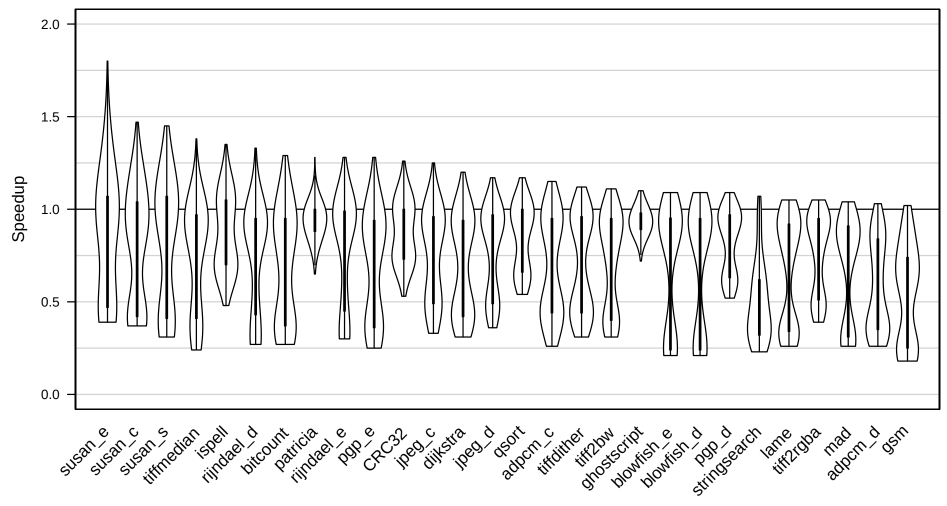
For our study, we selected 200 combinations of optimizations using a random search strategy with 50% probability to select each flag and either turn it on or off. We use this strategy to allow uniform unbiased exploration of unknown optimization search spaces. In order to validate the resulting diversity of program transformations, we have checked that no two combinations of optimizations generated the same binary for any of the benchmarks using the MD5 checksum of the assembler code obtained through `objdump -d` command. Occasionally, random selection of flags in GCC may result in an invalid code. In order to avoid such situations, we validated all generated combinations of optimizations by comparing the outputs of all benchmarks used in our study with the recorded outputs during reference runs when compiled with -O3 global optimization level.

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

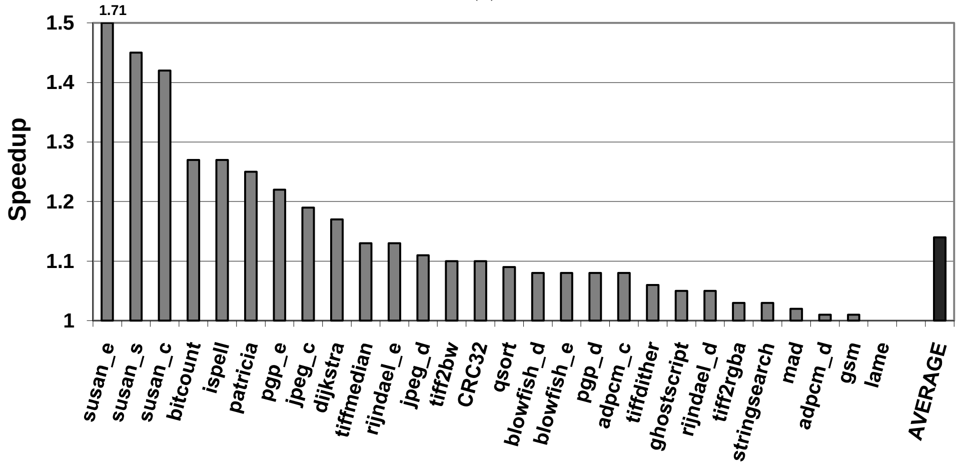
Collective Optimization Database. We gradually make all our experimental data publicly available through the online collective optimization database at cTuning.org repository [Link-repository] to help end-users improve their computing systems and help researchers reproduce the experimental results.

3. MOTIVATION

The performance upper bound of iterative optimization. Because the experimental methodology of research in iterative optimization consists of running many times the same program on the same dataset and on the same platform, it can be interpreted as an idealized case of collective optimization, where the experi-



(a)



(b)

Fig. 1. *(a) Distribution of speedups for all benchmarks and datasets. (b) Performance upper bound of collective optimization averaged across datasets (AMD Athlon 64 3700+; GCC 4.2.0; 88 program transformations applied globally; 200 iterations with random search strategy; 50% probability to select each transformation; speedups computed over the highest GCC optimization level -O3).*

ence of others (program, dataset, platform) would always perfectly match the target run, in other words, a case where no experimental noise would be introduced by differences in datasets, programs, or platforms. Consequently, iterative optimization can be considered as a performance upper bound of collective optimization.

-O1 -falign-loops=10 -fpeephole2 -fschedule-insns -fschedule-insns2 -fno-tree-ccp -fno-tree-dominator-opts -funroll-loops
-O1 -fpeephole2 -fno-rename-registers -ftracer -fno-tree-dominator-opts -fno-tree-loop-optimize -funroll-all-loops
-O2 -finline-functions -fno-tree-dce -fno-tree-loop-im -funroll-all-loops
-O2 -fno-guess-branch-probability -fprefetch-loop-arrays -finline-functions -fno-tree-ter
-O2 -fno-tree-lrs
-O2 -fpeephole -fno-peephole2 -fno-regmove -fno-unswitch-loops
-O3 -finline-limit=1481 -falign-functions=64 -fno-crossjumping -fno-ivopts -fno-tree-dominator-opts -funroll-loops
-O3 -finline-limit=64
-O3 -fno-tree-dominator-opts -funroll-loops
-O3 -frename-registers
-O3 -fsched-stalled-insns=19 -fschedule-insns -funroll-all-loops
-O3 -fschedule-insns -fno-tree-loop-optimize -fno-tree-lrs -fno-tree-ter -funroll-loops
-O3 -funroll-all-loops
-O3 -funroll-loops

Table I. *Some of the "Best" combinations of GCC flags for MiBench across all datasets on AMD64 (these are pruned combinations of optimizations where flags that do not influence performance have been removed)*

The violin graphs in Figure 1(a) show the distribution of the speedups for all benchmarks and datasets for AMD64 when applying 200 random combinations of optimizations over the highest GCC optimization level (-O3). Figure 1(b) shows the best speedup achieved for each benchmark averaged over 20 distinct datasets. The combinations of optimizations corresponding to the best speedups across all programs and datasets are presented in Table I or can be found online at cTuning.org repository [Link-repository]. [1] The diversity of compiler optimizations involved demonstrates that the compiler optimization space is not trivial; the compiler best optimization heuristic (-O3) is far from optimal and half of the benchmarks can achieve more than 20% speedup after iterative compilation. Interestingly, some individual transformations have to be turned off in order to achieve higher speedup that can also help to reduce compilation time. The benchmarks that have sharp peaks of speedup density usually require the longest time to obtain the best speedup. Overall, these experiments implicitly show that collective optimization has the potential to yield high speedups if it is possible to learn from the experience of others.

It seems possible but not straightforward to learn from the experience of others. Let us illustrate and quantify the difficulty of learning from the experience of others by using a simple strategy, comparable to what users would do: selecting the best optimization for a given context (dataset, program, architecture) and applying it to another context. Next, we consider learning across datasets, programs, and architectures.

We consider learning across datasets first. In Figure 2, for each program, we select the dataset (among 20) which exhibits the best speedup, and apply the corresponding combination of optimizations to all other datasets. We then report the % difference between this performance and the best performance obtained for each

[1] The flags that do not influence performance have been iteratively removed from the original combination of random optimizations to simplify the analysis of the results.

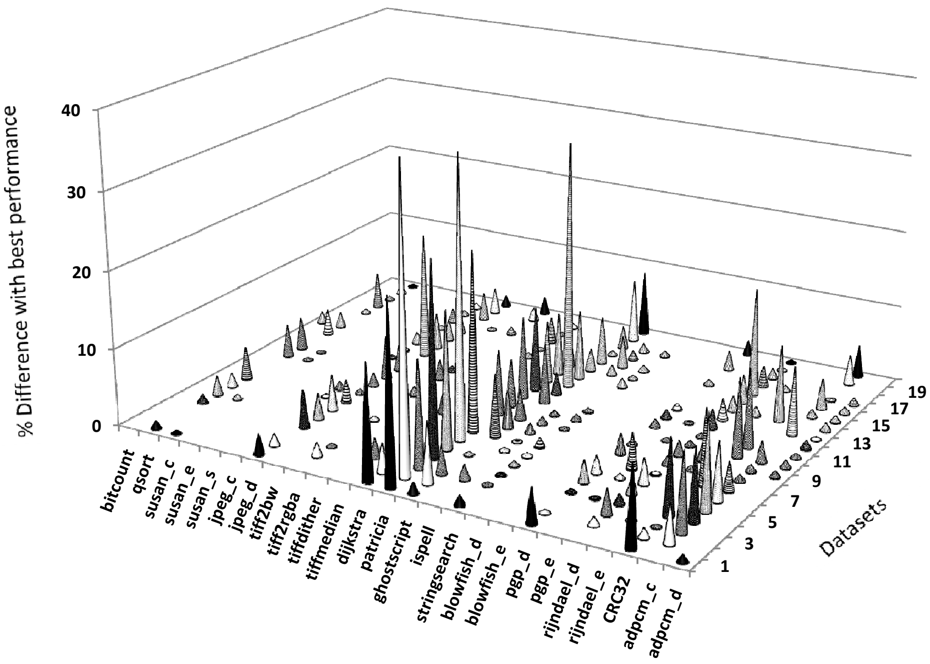


Fig. 2. *Learning across datasets.*

dataset. While programs like `susan_e` can almost perfectly learn across datasets, for other programs like `dijkstra,patricia,adpcm_c`, the best optimizations for a given dataset can yield much lower than optimal performance for other datasets. This observation further confirms other similar experiments using the commercial compiler PathScale [Fursin et al. 2007].

We performed similar experiments for programs. We selected the combination of optimizations which yields the best performance averaged over the 20 datasets of each program; see `reference` in Figure 3 (these combinations are listed in Table I). Then, we apply all the "best" combinations to all other programs (see `applied to`) and report the performance difference (again averaged over the 20 datasets). A program like `susan_c`, which is fairly stable across data sets seems to behave poorly using the best combinations of other programs. Conversely, the best combination for `patricia`, a program which does not easily learn across datasets, provides a good trade-off for many programs.

Finally, we performed similar experiments again for the three aforementioned architectures (AMD64, AMD32, and IA32). Performance is again averaged over all datasets, and we first select the best combination of optimizations for a given program on one of the architectures: AMD64. Then, we apply this combination to the other two architectures, and report the performance difference in Figure 4. Not surprisingly, AMD32 exhibits the closest performance behavior to AMD64, while the discrepancy between IA32 and AMD64 performance is likely due to their architectural differences. Again, these simple observations suggest that experience gathered on an architecture is potentially useful but cannot always straightforwardly apply

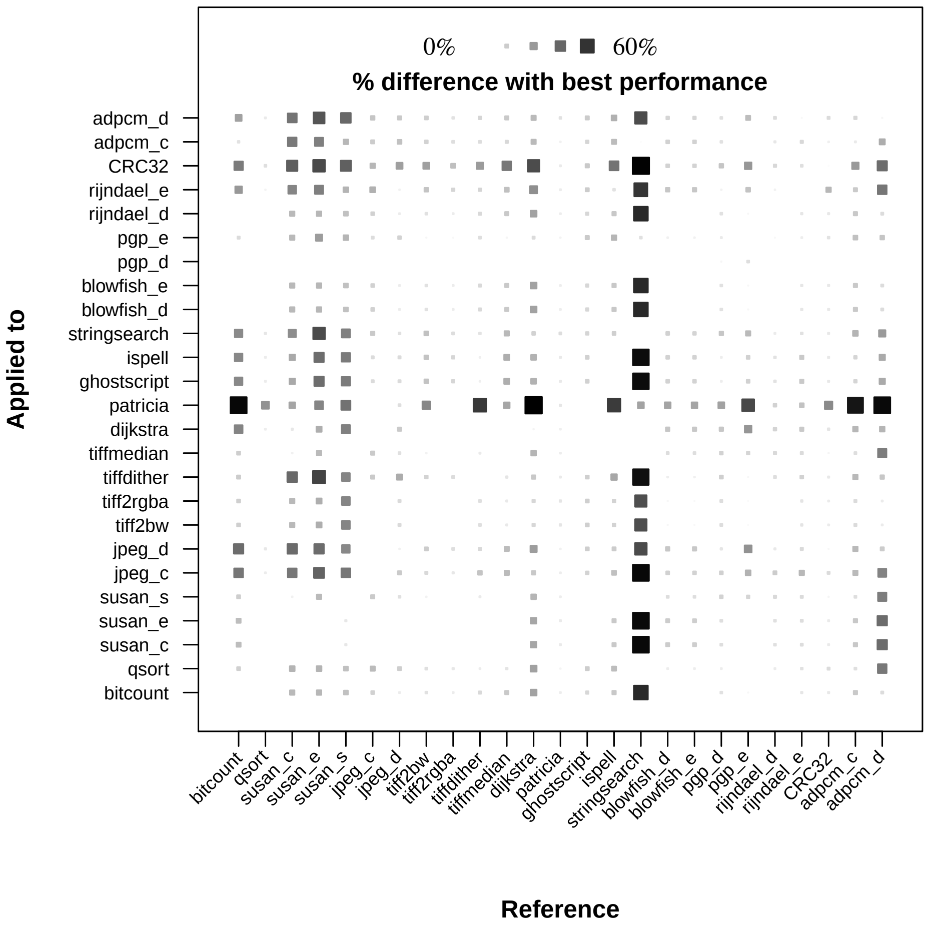


Fig. 3. *Learning across programs.*

to another architecture, even if the compiler platform is identical, as is the case here.

4. COLLECTIVE OPTIMIZATION 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 collective optimization database; and, after each run, based on all the knowledge gathered so far, a new combination of optimizations is selected and the program is recompiled accordingly. The key issue is which combination of optimizations 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

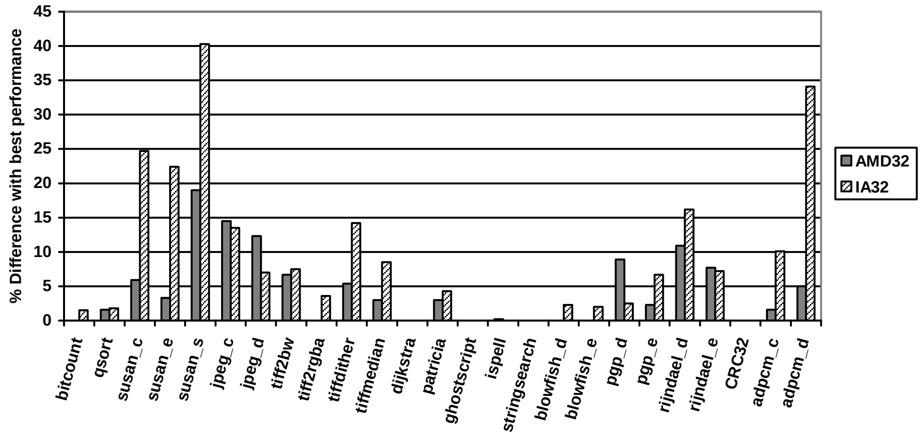


Fig. 4. *Attempting to learn across architectures: % difference of the best speedup achieved for a given program on AMD32 and IA32 (averaged across all datasets) with the speedup obtained when applying "best" found combination of optimizations for a program on AMD64.*

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 this 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 combinations of optimizations for itself. This knowledge takes the form of a program-specific probability distribution called d_3. Stage 3 corresponds to learning across datasets.

Stage 2: Program known, a few runs only. At this 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 combinations of optimizations 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 this 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 and allows us to automate and simplify the tuning of the default compiler optimization heuristic using realistic programs, datasets and collective knowledge from multiple users instead of ad hoc, repetitive training with some dedicated and not always representative benchmarks.

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 and then 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 combinations of optimizations 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 datasets) is by far the most useful stage. Stage 1 and Stage 2 are respectively useful in the first ten, and the first hundreds of 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 6. Using that meta-distribution, the distribution with the best score is favored.

5. COLLECTIVE OPTIMIZATION FRAMEWORK

In Figure 5, we show the collective optimization framework and two key components to enable continuous aggregation and reuse of optimization knowledge: collective compiler and collective optimization database (repository hosted at cTuning.org [Link-repository]).

Collective compiler. Our collective compiler is based on GCC with the Interactive Compilation Interface (ICI) [Link-ICI ; Fursin et al. 2008; Huang et al. 2010]. ICI abstracts the optimization process from a particular production compiler and helps us reuse the same collective optimization framework with other compilers. The extended compiler functionality and ICI collective optimization plugins allow to clone functions, instrument programs to select original functions or clones at runtime, control internal decisions, apply optimizations on a function level, and intercept `main` and `exit` routines to collect profiling statistics and send it to the optimization repository or obtain a new combination of optimizations to improve a program based on collective knowledge.

Usage scenarios. The collective optimization framework is compatible with the original unmodified GCC and invokes collective optimization plugins only when a user decides to participate in the collective learning by setting an environment variable `CTUNING` to nonzero value. Currently, the collective compiler supports two optimization scenarios: `training mode` with a reference run and `production mode` with transparent runtime evaluation of optimizations using function cloning. In both cases, the compiler adds `gprof` profiling routines to a compiled program, produces clones of all or only the most time consuming functions (if this statistic is available after several executions) and applies some combinations of optimizations

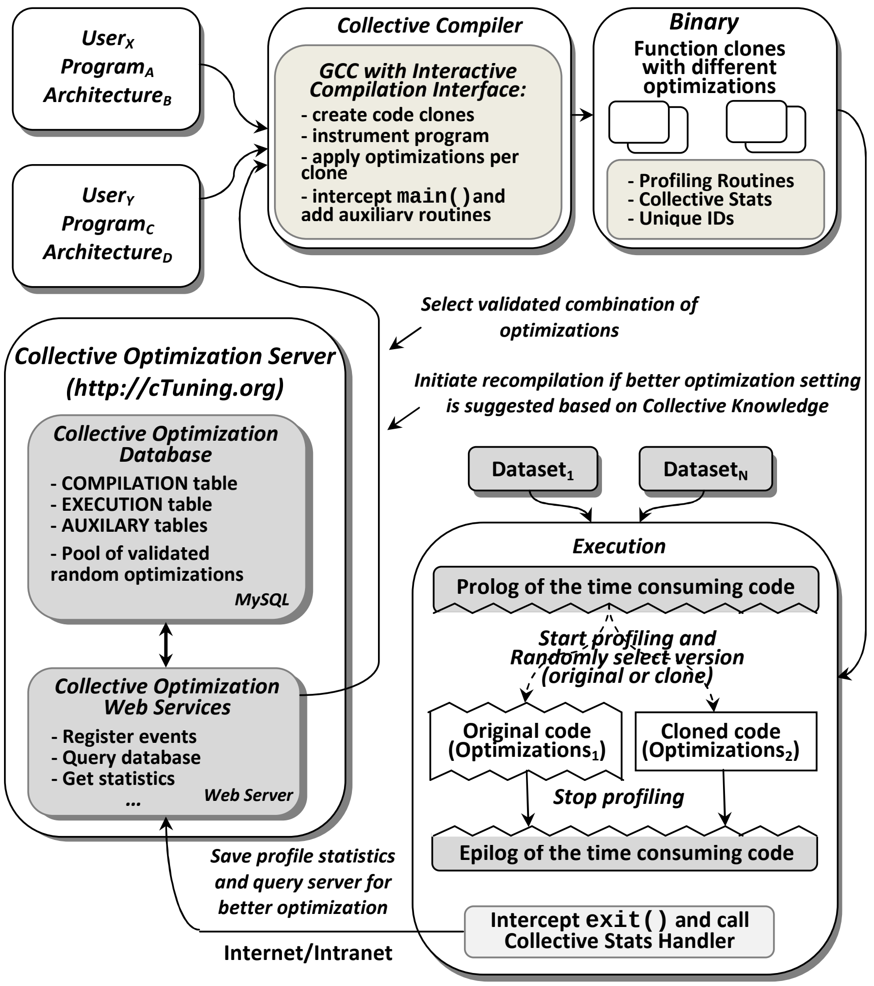

User$_X$
Program$_A$
Architecture$_B$
User$_Y$
Program$_C$
Architecture$_D$
Collective Compiler
GCC with Interactive Compilation Interface:
- create code clones
- instrument program
- apply optimizations per clone
- intercept main() and add auxiliary routines
Binary
Function clones with different optimizations
- Profiling Routines
- Collective Stats
- Unique IDs
Select validated combination of optimizations
Initiate recompilation if better optimization setting is suggested based on Collective Knowledge
Collective Optimization Server (http://cTuning.org)
Collective Optimization Database
- COMPILATION table
- EXECUTION table
- AUXILARY tables
- Pool of validated random optimizations
MySQL
Collective Optimization Web Services
- Register events
- Query database
- Get statistics
...
Web Server
Dataset$_1$
Dataset$_N$
Execution
Prolog of the time consuming code
Start profiling and Randomly select version (original or clone)
Original code (Optimizations$_1$)
Cloned code (Optimizations$_2$)
Stop profiling
Epilog of the time consuming code
Intercept exit() and call Collective Stats Handler
Save profile statistics and query server for better optimization
Internet/Intranet

Fig. 5. *Collective iptimization framework.*

to both original and cloned functions through ICI.

During `training mode` invoked by setting `CTUNING` variable to 1, a program is executed at least twice with the same dataset (similar to traditional iterative compilation approaches [Fursin et al. 2008]) to be able to correctly evaluate the effect of optimizations on the execution time. It is used for automatic offline search for profitable combinations of optimizations for a given architecture using standard benchmarks, programs, and datasets. This approach can be used along with collective optimization to speed up learning. Moreover, since the compiler sometimes generates an invalid random combination of optimizations, our `training mode` can reduce such risks and help to validate the correctness of a given combination by comparing the original and new outputs of a program on a number of datasets. A similar validation technique is commonly used in current compilers including GCC. On the collective optimization server, we gradually build a pool of validated optimizations favoring the best performing ones as shown in Table I, for example.

During `production mode`, invoked by setting `CTUNING` variable to 2, a program is compiled with random combinations of optimizations validated during the `training mode` and applied to both original and cloned functions. During continuous production runs of this program at data centers, PCs, mobiles, or other computing systems, random combinations of optimizations are statistically evaluated at runtime as described in Section 6. Furthermore, this and other profiling statistics are sent to the collective optimization server to both help other users improve their systems and return better optimization suggestions based on collective knowledge. A user can turn off this mode at any time by unregistering the environment variable `CTUNING` when reasonable performance improvement is achieved or when executing confidential programs. In this case, a user can either recompile the original program with the best found combination of optimizations while removing clones and profiling routines, or use the latest modified binary that will automatically invoke clones with the best optimizations and skip profiling and communication with the collective optimization server. In the latter scenario, the user may later easily resume the collective learning process.

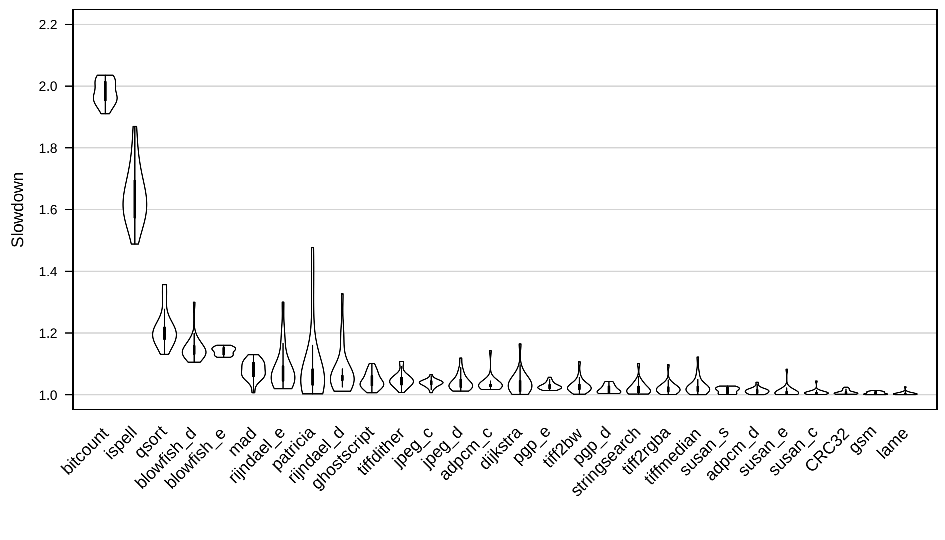
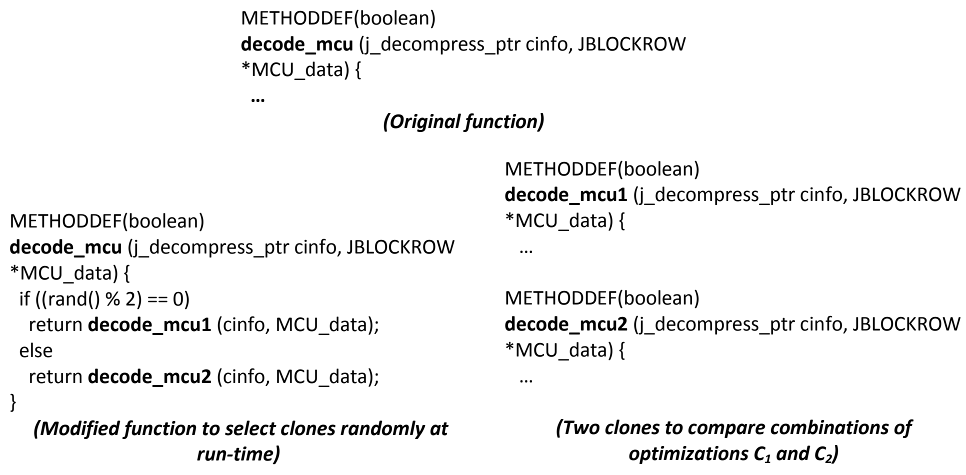


Fig. 6. *Distribution of slowdown for all benchmarks and datasets due to* `gprof` *function-level profiling overheads (AMD64).*

Profiling. We decided to use standard `gprof` profiling routines for execution time per function and number of calls. Though this tool introduces some overhead as shown in Figure 6, for half of the used benchmarks it is negligible and it is used only for the proof of the concept and to simplify the implementation. In the future, we plan to minimize this overhead by using our own optimized routines to obtain cycle-accurate profiling through performance counters or using tools such as oprofile with nonintrusive profiling through sampling [Link-OProfile]. Also, as mentioned earlier, a user can turn off collective optimization mode with profiling at any time and leave the best found combination of optimizations.

During profiling, we select the most time-consuming program routines (in this work we select up to 3 functions that cover 75% or more of the execution time in our benchmarks). The definition of the top routines can change across runs. Therefore, we progressively build an average ranking of the program routines, possibly learning new routines as they are exercised by different datasets.



```
METHODDEF(boolean)
decode_mcu (j_decompress_ptr cinfo, JBLOCKROW
*MCU_data) {
 ...
```

```
METHODDEF(boolean)
decode_mcu (j_decompress_ptr cinfo, JBLOCKROW
*MCU_data) {
 if ((rand() % 2) == 0)
   return decode_mcu1 (cinfo, MCU_data);
 else
   return decode_mcu2 (cinfo, MCU_data);
}
```

```
METHODDEF(boolean)
decode_mcu1 (j_decompress_ptr cinfo, JBLOCKROW
*MCU_data) {
  ...

METHODDEF(boolean)
decode_mcu2 (j_decompress_ptr cinfo, JBLOCKROW
*MCU_data) {
  ...
```

(Two clones to compare combinations of optimizations C_1 and C_2)

Fig. 7. *Function cloning to evaluate combinations of optimizations transparently or to enable dynamic adaptation for statically compiled programs.*

Cloning. We modified GCC to implement function cloning. That required changes in the core of the compiler since we had to implement full replication of parts of a program AST. This functionality is controlled by the interactive compilation interface. When collective optimization plugins trigger GCC to clone 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 as shown in Figure 7 for function *decode_mcu* of jpeg_d. This enables continuous transparent evaluation of combinations of optimizations at runtime for statically compiled programs as described in detail in Section 6.

Transparently collecting runtime information and reoptimizing. In order to unobtrusively collect information on a program run, and reoptimize the program, we modified GCC to intercept the compilation of the main() function, and insert another interceptor on the exit() function to call a termination routine. Whenever the program execution finishes, this routine is invoked and it in turn checks whether the *Collective Stats Handler* exists, invokes it to send compilation and execution information to the collective optimization server and database

[Fursin 2009].

COMPILE_ID	17053973767718039
PLATFORM_ID	2111574609159278179
ENVIRONMENT_ID	2781195477254972989
COMPILER_ID	7548127843267843
PROGRAM_ID	37097459005644868
DATE	2007-07-09
TIME	18:32:43
OPT_FLAGS	decode_mcu1={-O3 } decode_mcu2={-O3 -fno-inline-functions -fno-tree-ccp -fmove-loop-invariants}
COMPILE_TIME	3.4
BIN_SIZE	356655
OBJ_MD5CRC	a420280c164310a47e2d8655028d1a66

(compilation stage)

RUN_ID	5572268172923323
COMPILE_ID	17053973767718039
PLATFORM_ID	2111574609159278179
ENVIRONMENT_ID	2781195477254972989
PROGRAM_ID	37097459005644868
DATE	2007-07-09
TIME	18:32:52
OUTPUT_CORRECT	1
RUN_TIME	10.32
RUN_TIME_USER	8.94
RUN_PROFILE	{decode_mcu1=3.16, calls=139223} {decode_mcu2=3.21, calls=142948}

(execution stage)

Fig. 8. *Example of information packets sent to the Collective Optimization Server/Database (hosted at cTuning.org) for the program `jpeg_d`.*

Figure 8 shows compilation and execution information packets sent to the database for the program `jpeg_d`. Each computer system is assigned several unique identifiers (generated by the UUID tool unless such system already exists in the collective optimization database) describing architecture, environment, and compiler utilized. Compilation and execution information packets as well as a program are also assigned unique IDs to allow easy distributed sharing and referencing of optimization cases between multiple users. One should note that the `OUTPUT_CORRECT` field is used only during the `training mode` when a program can be executed with the same dataset more than once and program outputs can be compared to minimize the risk of invalid combinations of optimizations as described earlier. Only validated combinations of optimizations can be later used for transparent optimizations without a reference run.

When starting the optimization process, a user can either provide a program ID if such a program has already been registered in the collective optimization database, or use a tool we provide to define the program ID as the MD5 checksum of compiled source files. Eventually, after a few runs, we characterize programs using a portable method based on dynamic features, as described in details in Section 6.2.

The termination routine queries a Web service on the collective optimization server in order to obtain potentially better combination of optimizations for future runs. If such combinations exist, a recompilation takes place periodically (period set by the user) in the background, between several runs.[2] At any time, the user can opt in or out of collective optimization by setting or resetting an environment variable `CTUNING`.

Security. The concept of collective optimization raises new issues, especially security. Therefore, we never send any source code to the collective optimization

[2]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.

server (only MD5 or dynamic features vectors). If a user would like to submit optimization data from some privacy-critical applications to the common repository, we provide an option to obfuscate program and function names. Furthermore, the collective optimization server supports both public and private databases (for companies). Note that companies can then get the best of both worlds: leverage/read information accessible from the common database, while recording information about their runs solely to their private database.

6. COLLECTIVE LEARNING

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

6.1 Building the Program Distribution d_3 Using Statistical Comparison of Combinations of Optimizations

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 combinations of optimizations 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 datasets. Therefore, if two runs with respective execution times T_1 and T_2, and where combinations of optimizations 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 runtime comparison of *two* combinations of optimizations using *cloned* functions. C_1 and C_2 are respectively applied to the clones f_1 and f_2 of a function f. At runtime, 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 pseudorandom number generation technique emulating uniform distribution. We have shown in [Fursin et al. 2005] the possibility to evaluate optimizations for statically compiled programs with stable behavior using function cloning and runtime low-overhead phase detection. [Stephenson 2006] and [Lau et al. 2006] 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 transparent runtime performance evaluation for statically compiled programs with any behavior here. Even if the workload of the routine varies upon each call, we observed that in many cases, if the routine is executed a large number of times (> 100), the average workload performed by each randomly selected clone can be similar during one execution. As a result, the nonoptimized 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, that is, $C_1 > C_2$. This statistical comparison of combinations of optimizations requires no reference, test or training run, and the overhead is negligible.

Naturally, it is not always possible to use the preceding technique to evaluate two combinations of optimizations if there are only a few invocations of the function during the whole program execution, if the workload varies considerably across invocations, or there is too much noise during profiling. Therefore, we continuously monitor the speedups across runs until the speedup converges to some constant value. Otherwise, a user has to resort to other evaluation methods, such as an additional reference run with the same dataset, as for the `training mode`.

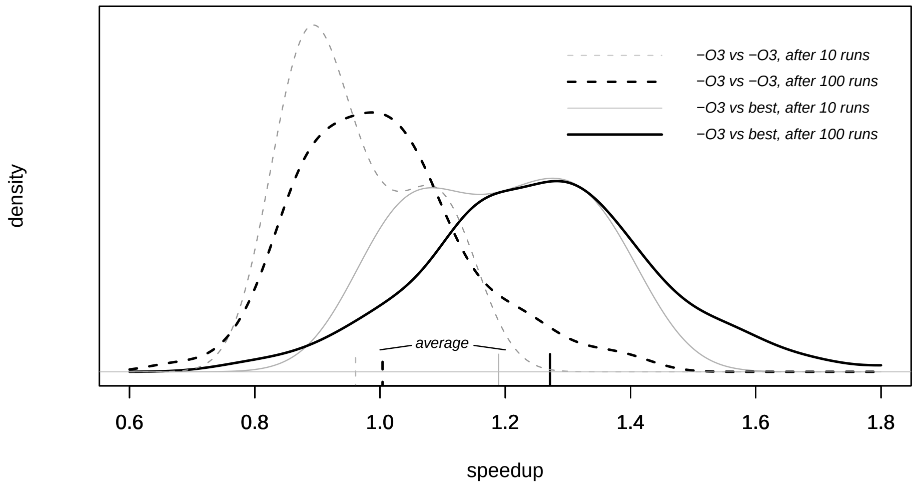


Fig. 9. *Speedup convergence during runtime comparison of combinations of optimizations across multiple runs of `jpeg_d` with random datasets on IA32.*

For example, Figure 9 demonstrates how the speedups for the hot function `decode_mcu` of `jpeg_d` evolve during runtime comparison of optimizations across multiple program executions with random datasets on IA32 using two scenarios. In the first scenario, `decode_mcu` is compiled with the baseline optimization level (-O3) while its clone is compiled with the top performing combination of optimizations. In the second scenario, both original function and its clone are compiled with -O3. The solid and dashed line show the evolution of the density of speedups after 10 and 100 program runs for the first and second scenario, respectively. This experiment demonstrates that in both scenarios the average speedup gradually converges to some constant value and in the second scenario the average speedup converges to 1.0 thus confirming the possibility to use our runtime approach to compare optimizations for this program and function.

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 combinations of optimizations, and the winning combination scores 1, the other 0, as shown in Figure 10. 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. Note that we also experimented with scoring using $\frac{Time_{C_1}}{Time_{C_2}} - 1$ instead of 0/1 depending on $C_1 > C_2$, which should provide a potentially richer, program, dataset, and platform-dependent information, but found no significant performance benefit. As a result, we decided to use more the independent, and thus potentially more robust information.

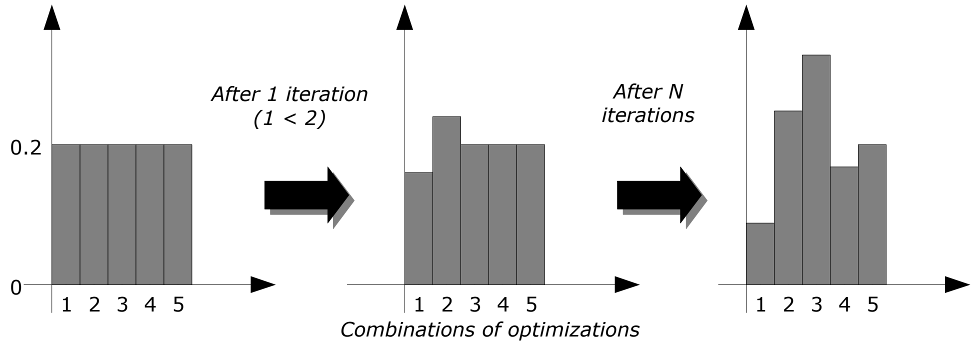


Fig. 10. *Computing the probability distribution used for selecting a combination of optimizations.*

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.

6.2 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 combinations of optimizations 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 combinations of optimizations a program favors.

As for d_3, we use the metric-independent comparison between two combinations of optimizations C_1 and C_2. For example, $C_1 > C_2$ 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 it programs. The more pairs (reactions to optimizations) are compared, the more accurate and reliable the program matching.

Still, we observed that beyond 100 characterizing pairs of combinations of optimizations (out of $C^2_{100} = \frac{200}{199}/2 = 19900$ possible pairs of combinations), 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 pairs of combinations, which are collected within a rolling window (FIFO). However, the window only contains distinct pairs of combinations of optimizations. 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. 2006] 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 $3/d_3$.

6.3 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 combinations of optimizations perform best in general. For example, it can be already used to systematically and continuously improve default optimization level of a given compiler on a given architecture by distributing the tuning process among many users and taking into account real applications and datasets thus avoiding specialized and often limited benchmarks. So even users not relying on collective optimization could benefit in a simple way from the collective knowledge gathered by others. Furthermore, it is also possible to compose more restricted aggregate distributions, such as per architecture, per compiler, per programs, or dataset subsets. We leave this for the future work.

6.4 Scoring Distributions

As mentioned in Section 4, a meta-distribution is used to select which stage distribution is used to generate the next combination of optimizations. For each run, two distributions d and d' are selected using two draws from the meta-distribution (they can be the same distributions). Then, a combination of optimizations 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 datasets 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.

7. PERFORMANCE EVALUATION

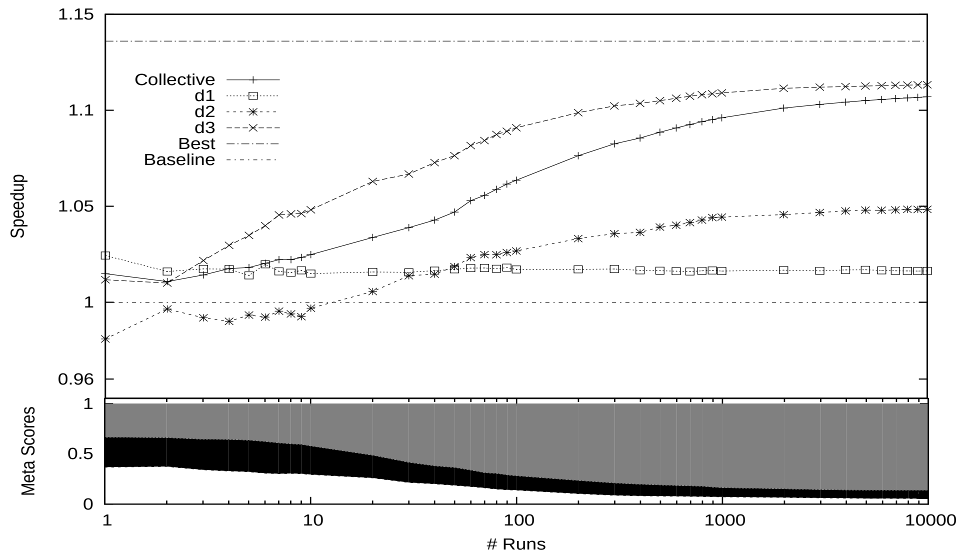


Fig. 11. *Average performance of collective optimization and individual distributions averaged across all programs and datasets for AMD64 (bottom: metascores of individual distributions; gray is d_3, black is d_2, white is d_1).*

In Figure 11, *Collective* corresponds to the full process described in earlier sections across all programs and data sets for AMD64, 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 datasets and can apply 200 different combinations of optimizations, for a total of 4000 distinct runs per program. The main approximation of our evaluation lays in the number of datasets; upon each run, we (uniformly) randomly select one among 20 datasets. While 20 datasets is higher than in most other studies to the best of our knowledge, it is still much fewer than the number of runs in our experiments. However, several studies have shown that datasets are often clustered within a few groups breeding similar behavior [Eeckhout et al. 2003], so that 20 datasets exhibiting sufficiently distinct behavior, as suggested by the experiments of Section 3, may be considered a nonperfect but reasonable emulation of varying program behavior across datasets. In order to further assess the impact of using a restricted number of datasets, we have evaluated the extreme case where a single dataset is used. These results are reported in Figure 14 (see *Single dataset*), where a single dataset 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 datasets. Using a single dataset improves convergence speed though only moderately, suggesting *Collective* could be a slightly optimistic but reasonable approximation of a real case where all datasets 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-dataset experiments, see (Section 3). The other graphs (*d1, d2, d3*) report the evolution of the average performance of combinations of optimizations drawn from each distribution. At the bottom of the figure, the gray filled curve corresponds to the metascore of d_3, the black one to d_2, and the white one to d_1.

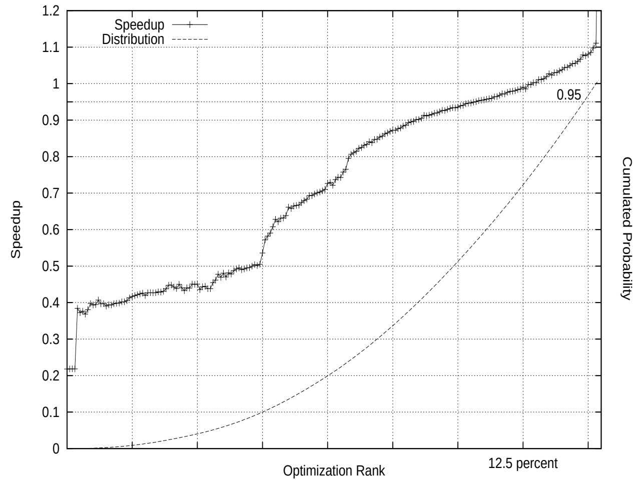


Fig. 12. *Average speedups and d_3 distributions profile (across all programs and datasets for AMD64).*

Convergence versus performance. d_3 converges fairly rapidly as it starts getting significant performance improvements after about 10 runs. The shape of distributions for combinations explains this rapid convergence. Figure 12 shows the average *profile* of cumulated program distributions d_3, as well as the average program speedup profile; such profiles are computed for each program, with optimizations ranked on the x-axis according to the distribution, and then these profiles are averaged over all programs. On average, 12.5% of combinations bring a speedup greater than 1, that is, 1 in 8; so, on average, drawing 8 combinations should yield one combination improving performance. This is why after 10 runs, d_3 distributions have usually identified one good candidate combination; performance keeps improving because new, better candidates are later discovered. In other words, there are relatively many "good" combinations.

Still, the majority ($\simeq$ 87.5%) of combinations are "poor", degrading performance. And even a distribution skewed toward good combinations, like d_3, will draw poor combinations more often than a user might tolerate. In order to filter these poor combinations out and to achieve consistently good performance, the d_3 distributions are not polled over the whole probability interval $[0, 1]$, but only the interval $[0.95, 1]$; in other words, only the 5% best combinations suggested by d_3 are used. These combinations yield about 10% performance improvement or more on average, as shown in Figure 12. This same restricted polling principle is applied to all d_i distributions, but not to the meta-distribution. Collective optimization performed with individual distributions polling in interval $[0, 1]$ yield poor performance for several reasons: individual distributions more frequently draw poor combinations; as a consequence, their score frequently decreases, and the meta-distribution has difficulties identifying a winning distribution.

The downside of polling in interval $[0.95, 1]$ is that a distribution may be stuck drawing almost always the same combination(s), thus being not reactive. That is where the competitions and the meta-distribution come into play. Even if distributions are strongly skewed toward one or a few combinations, once in a while these top combinations will be challenged against other combinations (drawn from one or the other distributions). If it turns out the choices recommended by d_3 have become poor, after a few such challenges, both the scores of the previously top combinations and the metascore of d_3 will plummet, allowing new combinations to emerge.

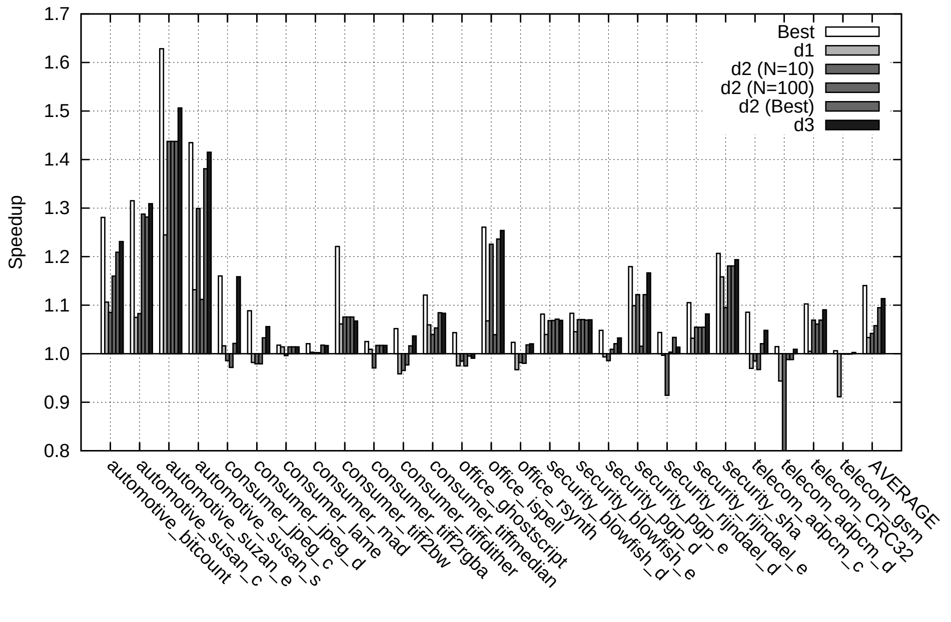


Fig. 13. *Distribution of performance per program using maximum available knowledge (speedups averaged across all datasets on AMD64).*

Learning across programs. While the behavior of d_2 in Figure 11 suggests that learning across programs yields modest performance improvements, this experiment is partly misleading. d_3 rapidly becomes a dominant distribution, and as explained earlier, quickly converges to one or a few top combinations due to restricted interval polling. d_2 performance will improve as more characterizing pairs of combinations of optimizations fill up the rolling window.

This is further outlined in Figure 13 that shows the steady-state performance for each distribution and program, where only a single distribution is used, that is, no meta-distribution, and distributions are built using all available data. For instance, for each program, the d_1 distribution is built using all 104000 collected runs for programs other than the target program (20 datasets, 200 combinations of optimizations for 25 out of 26 programs); the reported performance is the average speedup of 100 runs using 100 draws from that distribution. For the d_3 distribution of one program, 20 d_3 distributions are in fact built for each of the 20 datasets, using all programs runs except for that dataset (19 other datasets, 200 combinations per dataset, for a total of 3800 runs), the performance for one dataset is the average speedup of 100 runs using 100 draws from these dataset-specific distributions, and the reported performance is itself averaged over all datasets. For the d_2 distributions, the N pairs of characteristic combinations of optimizations are drawn using d_1 (recall d_1 is itself built excluding all runs from the target program), and the dataset is (uniformly) randomly selected among 20. These N runs are then used to score all other programs, or more exactly their own d_3 distributions as explained in Section 6.2, and the program with the best score is considered the "matching" program; d_2 is then equal to the d_3 of that program, and the performance is evaluated by averaging over 100 runs using 100 draws from that distribution.

Even though the average performance of d_2 is still significantly lower than that of d_3 and the variability is high due to occasional poor matching, the performance is close to d_3 for some codes, such as `susan_c`. These results confirm previous studies which have shown that learning across programs can yield good performance [Cavazos et al. 2006; Agakov et al. 2006; Fursin et al. 2008]. More sophisticated techniques for selecting characterizing combinations of optimizations, as proposed in [Cavazos et al. 2006] for instance, might improve performance. However, in practice, this effort is not critical since, even if matching were optimal (see *d2 (Best)*), the average d_2 performance would still be lower than d_3 ("best" matching is obtained by selecting the matching program which yields the highest speedups for the target program, averaged over 100 random draws from the d_3 candidate matching program distribution). But more importantly, the comparison of Figure 11 and Figure 13 highlights that the few runs necessary to characterize the behavior of a new program, with the intent of later matching it to fellow programs, usually gathers enough knowledge on the program itself, so that program matching then becomes useless. In other words, self-learning converges so fast that learning across programs is in fact not necessary. As a result, even though d_2 performance keeps increasing to a reasonable level in Figure 11, its metascore keeps decreasing because, in the meantime, d_3 contains even more relevant knowledge. Note that when training is purely static and requires no target program training runs [Stephenson and Amarasinghe 2005], program matching may still be useful at start-up.

Collective *versus* d_3. While the better performance of d_3 over *Collective*, shown in Figure 11, suggests this distribution should solely be used, one can note that its

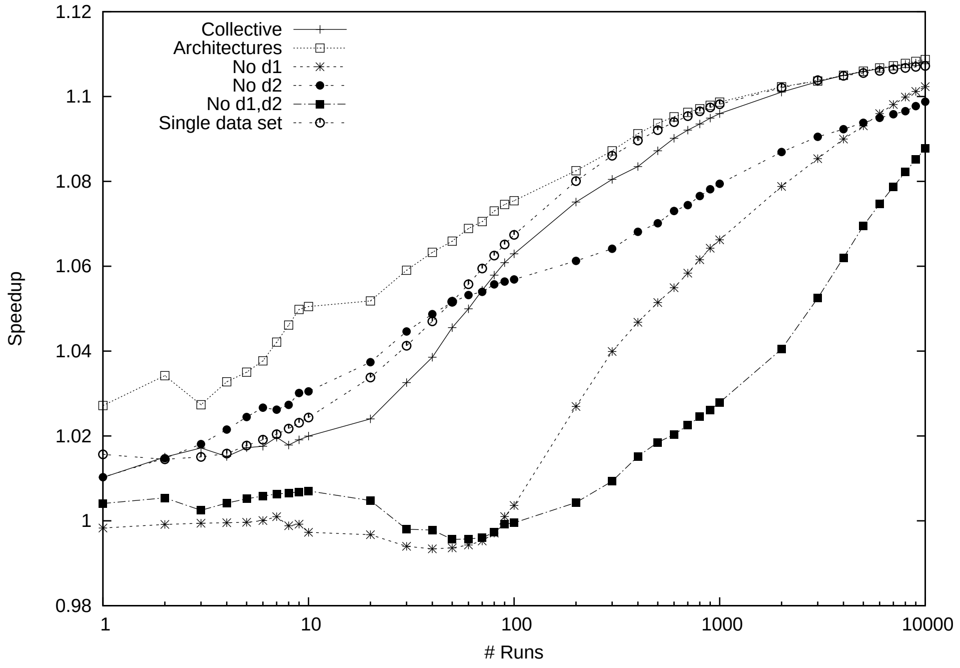


Fig. 14. *Several collective optimization variants (learning with d1 removed, d2 removed, d1 and d2 removed, using only one dataset, learning across multiple architectures including AMD64,AMD32,IA32).*

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 14, 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;[3] the uniform distribution is not used when only d_1 or d_2 are removed. As shown in Figure 14, 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. When contrasting Figure 13, where the performance of d_2 is modest even with $N = 100$ characterizing runs, and Figure 14, one can notice

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

that d_2 does in fact significantly help improve the performance of *Collective* after 100 runs, but essentially by helping discover good new optimizations, later adopted by d_3, rather than due to the intrinsic average performance of the combinations of optimizations suggested by d_2.

Learning across architectures. Besides learning across datasets 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 reported before 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 datasets, 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 14. 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. [Whaley and Dongarra 1998; Cooper et al. 1999; Bodin et al. 1998; Matteo and Johnson 1998; Cooper et al. 2002; Fursin et al. 2002; Kulkarni et al. 2003; Triantafyllis et al. 2003; Singer and Veloso 2000; Pan and Eigenmann 2004; 2006; Qasem et al. 2006; Bailey et al. 2008; Hoste and Eeckhout 2008] demonstrated that optimization 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.

Recently, machine-learning and statistical techniques have been used [Monsifrot et al. 2002; Stephenson et al. 2003; Stephenson and Amarasinghe 2005; Zhao et al. 2005] to select or tune program transformations based on program features. [Agakov et al. 2006], [Cavazos et al. 2007] and [Dubach et al. 2009] use machine-learning to focus iterative compilation and architectural design space exploration using either syntactic program features or dynamic hardware counters and multiple program transformations. In [Fursin et al. 2008] we demonstrated real machine-learning enabled research compiler based on GCC with ICI and plugins for static feature extraction and prediction of optimizations. Most of these works still require a large number of training runs. [Stephenson and Amarasinghe 2005] show more complementarity approach with collective optimization as program matching is solely based on static features.

Several frameworks have been proposed for continuous program optimization [Anderson et al. 1997; Voss and Eigenmann 2000; Lu et al. 2004; Lattner and Adve 2004]. Such frameworks tune programs either during execution or off-line, trying different program transformations. Such recent frameworks like [Lattner and Adve 2004] and [Lu et al. 2004] 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 [Fursin et al. 2005; Lau et al. 2006; Stephenson 2006] suggest to use procedure cloning to search for best optimizations at runtime or create adaptive applications that can react to runtime behavior. In this article we combine and extend techniques from [Fursin et al. 2005] that are compatible with regular scientific programs and use low-overhead runtime phase detection, and methods from [Stephenson 2006; Lau et al. 2006] 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 datasets [Fursin et al. 2007] and suggests this task is tractable though not necessarily straightforward.

The approaches closest to ours are presented in [Arnold et al. 2005] and [Stephenson 2006]. The system in [Arnold et al. 2005] 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 does not 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 are 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 datasets from multiple users and the optimization selection robustness (through competitions and meta-distribution).

Interestingly, in [Mytkowicz et al. 2009] the authors raise an important issue of the measurement bias demonstrating that the obtained speedups can differ due to different experimental setup settings. The authors suggest to use more diverse benchmarks and experimental setup randomization among other solutions to minimize this bias. The collective optimization approach already inherently takes into account these effects and ensures optimization portability by obtaining experimental results from multiple users with diverse setups, architectures, programs, and datasets.

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 *competi-*

tion (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 datasets 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 information on runs for statically compiled programs and for reoptimizing programs across runs, based on optimizations suggestions.

Furthermore, we started disseminating all the prototypes of our open collective optimization research platform based on GCC (and other compilers, including LLVM, ROSE, ICC, Open64, are also partially supported), connected to a public collective optimization database [Fursin 2009; Link-repository ; Link-ICI]. We initiated a long-term collaborative community-driven effort at cTuning.org to extend and deploy our framework in current cloud computing environments, data centers, mobile systems (based on Moblin or Android), or traditional personal computers to automate program and whole system optimization as well as compiler tuning.

The convergence time of collective optimization, while perfectly acceptable for largely used programs, remains long for programs which will be used tens or hundreds of times only. For that purpose, we intend to combine static and dynamic program characterization techniques from [Agakov et al. 2006; Stephenson and Amarasinghe 2005; Cavazos et al. 2007; Fursin et al. 2008] with our statistical collective optimization approach and program characterization by optimization reactions, to define initial d_3 distributions for the first runs.

While we have evaluated learning across datasets, programs and architectures, we have not yet evaluated learning across compilers. In the present article, all optimizations are defined as combinations of individual compiler transformations. As a result, they cannot be directly translated to other compilers. For that reason, we are in the process of expressing optimizations as combinations of standard compiler optimization algorithms (e.g., loop unrolling, induction variable elimination, etc.), a definition which would span all compiler platforms. Moreover, using the Interactive Compilation Interface and plugin system for GCC and other compilers [Fursin et al. 2008; Link-ICI], we can directly invoke transformations, change their parameters, and their order at the function or loop level, and add plugins with new transformations.

The collective optimization approach opens up many other research directions. For instance, thanks to the transparent collection of a large amount of optimization data, we can perform statistically rigorous evaluation of iterative optimization. Analyzing multiple performance anomalies we can detect weaknesses of current architectures and suggest possible improvements. Since different program versions may most likely benefit from runs of past versions, we can speed up the optimization of new software versions. Optimizations could be further refined at the dataset level by clustering datasets and using our cloning and runtime comparison mechanism to select the most appropriate combinations of optimizations or even reconfigure the processor at runtime. By combining our approach with program instrumentation of loads and stores, we might be able to distribute the burden of analysis of runtime data dependencies of multiple programs and datasets to automate and simplify program parallelization. And, after sufficient knowledge has been accumulated, the collective database may become a useful tool for defining truly representative benchmarks. Finally, we believe that the framework and the public optimization repository might also help improve research methodology by allowing to faithfully repeat and share experimental results.

ACKNOWLEDGMENTS

We would like to thank our colleagues from INRIA, University of Edinburgh, Institute of Computing Technology of Chinese Academy of Science and users from GCC, cTuning, and HiPEAC communities for their useful feedback. We would also like to thank anonymous reviewers for their insightful comments to improve this article.

REFERENCES

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

Anderson, J., Berc, L., Dean, J., Ghemawat, S., Henzinger, M., Leung, S., Sites, D., Vandevoorde, M., Waldspurger, C., and Weihl, W. 1997. Continuous profiling: Where have all the cycles gone. In *Technical Report 1997-016. Digital Equipment Corporation Systems Research Center, Palo Alto, CA.*

Arnold, M., Welc, A., and V.T.Rajan. 2005. 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)*.

Bailey, D. H., Chame, J., Chen, C., Dongarra, J., Hall, M., Hollingsworth, J. K., Hovland, P., Moore, S., Seymour, K., Shin, J., Tiwari, A., Williams, S., and You, H. 2008. Peri auto-tuning. *Journal of Physics: Conference Series (SciDAC 2008) 125.*

Bodin, F., Kisuki, T., Knijnenburg, P., O'Boyle, M., and Rohou, E. 1998. Iterative compilation in a non-linear optimisation space. In *Proceedings of the Workshop on Profile and Feedback Directed Compilation.*

Cavazos, J., Dubach, C., Agakov, F., Bonilla, E., O'Boyle, M., Fursin, G., and Temam, O. 2006. 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).*

Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'Boyle, M., and Temam, O. 2007. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the 5th Annual International Symposium on Code Generation and Optimization (CGO).*

Cooper, K., Schielke, P., and Subramanian, D. 1999. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. 1–9.

Cooper, K., Subramanian, D., and Torczon, L. 2002. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing 23,* 1, 7–22.

Dubach, C., Jones, T. M., Bonilla, E. V., Fursin, G., and O'Boyle, M. F. 2009. Portable compiler optimization across embedded programs and microarchitectures using machine learning. In *Proceedings of the 42nd International Symposium on Microarchitecture (MICRO).*

Eeckhout, L., Vandierendonck, H., and Bosschere, K. D. 2003. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism 5*, 1–33.

Franke, B., O'Boyle, M., Thomson, J., and Fursin, G. 2005. Probabilistic source-level optimisation of embedded programs. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*.

Fursin, G. 2009. Collective tuning initiative: automating and accelerating development and optimization of computing systems. In *Proceedings of the GCC Developers' Summit*.

Fursin, G., Cavazos, J., O'Boyle, M., and Temam, O. 2007. 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)*.

Fursin, G., Cohen, A., O'Boyle, M., and Temam, O. 2005. 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. Springer Verlag, 29–46.

Fursin, G., Miranda, C., Temam, O., Namolaru, M., Yom-Tov, E., Zaks, A., Mendelson, B., Barnard, P., Ashton, E., Courtois, E., Bodin, F., Bonilla, E., Thomson, J., Leather, H., Williams, C., and O'Boyle, M. 2008. Milepost gcc: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*.

Fursin, G., O'Boyle, M., and Knijnenburg, P. 2002. Evaluating iterative compilation. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computers (LCPC)*. 305–315.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*. Austin, TX.

Hoste, K. and Eeckhout, L. 2008. Cole: Compiler optimization level exploration. In *Proceedings of International Symposium on Code Generation and Optimization (CGO)*.

Huang, Y., Peng, L., Wu, C., Kashnikov, Y., Renneke, J., and Fursin, G. 2010. Transforming gcc into a research-friendly environment: plugins for optimization tuning and reordering, function cloning and program instrumentation. In *2nd International Workshop on GCC Research Opportunities (GROW), colocated with HiPEAC'10 conference*.

Kisuki, T., Knijnenburg, P., and O'Boyle, M. 2000. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 237–246.

Kulkarni, P., Zhao, W., Moon, H., Cho, K., Whalley, D., Davidson, J., Bailey, M., Paek, Y., and Gallivan, K. 2003. Finding effective optimization phase sequences. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. 12–23.

Lattner, C. and Adve, V. 2004. 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.

Lau, J., Arnold, M., Hind, M., and Calder, B. 2006. Online performance auditing: Using hot optimizations without getting burned. In *Proceedings of the ACM SIGPLAN Conference on Programming Languaged Design and Implementation (PLDI)*.

Link-ICI. ICI: Interactive Compilation Interface: unified plugin system to convert black-box production compilers into interactive research toolsets. `http://cTuning.org/ici`.

Link-MiDatasets. MiDataSets: multiple datasets for MiBench benchmark to enable realistic research on iterative compilation and adaptive optimization. http://cTuning.org/cbench.

Link-MILEPOST. MILEPOST project archive (MachIne Learning for Embedded PrOgramS opTimization). `http://cTuning.org/project-milepost`.

Link-OProfile. OProfile: system-wide profiler for Linux systems, capable of profiling all running code at low overhead. `http://oprofile.sourceforge.net`.

Link-repository. Public collaborative repository and tools for program and architecture characterization and optimization. `http://cTuning.org/cdatabase`.

Lu, J., Chen, H., Yew, P.-C., and Hsu, W.-C. 2004. Design and implementation of a lightweight dynamic optimization system. In *Journal of Instruction-Level Parallelism*. Vol. 6.

Matteo, F. and Johnson, S. 1998. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*. Vol. 3. Seattle, WA, 1381–1384.

Monsifrot, A., Bodin, F., and Quiniou, R. 2002. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the International Conference on Artificial Intelligence: Methodology, Systems, Applications*. LNCS 2443. 41–50.

Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. 2009. Producing wrong datawithout doing anything obviously wrong! In *Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Pan, Z. and Eigenmann, R. 2004. Rating compiler optimizations for automatic performance tuning. In *Proceedings of the International Conference on Supercomputing*.

Pan, Z. and Eigenmann, R. 2006. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 319–332.

Qasem, A., Kennedy, K., and Mellor-Crummey, J. 2006. Automatic tuning of whole applications using direct search and a performance-based transformation system. *Journal of Supercomputing 36,* 2, 183–196.

Singer, B. and Veloso, M. 2000. Learning to predict performance from formula modeling and training data. In *Proceedings of the Conference on Machine Learning*.

Stephenson, M. and Amarasinghe, S. 2005. Predicting unroll factors using supervised classification. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

Stephenson, M., Martin, M., and O'Reilly, U. 2003. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 77–90.

Stephenson, M. W. 2006. Automating the construction of compiler heuristics using machine learning. Ph.D. thesis, MIT, USA.

Triantafyllis, S., Vachharajani, M., Vachharajani, N., and August, D. 2003. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 204–215.

Voss, M. and Eigenmann, R. 2000. Adapt: Automated de-coupled adaptive program transformation. In *Proceedings of the International Conference on Parallel Processing (ICPP)*.

Whaley, R. and Dongarra, J. 1998. Automatically tuned linear algebra software. In *Proceedings of the Conference on High Performance Networking and Computing*.

Zhao, M., Childers, B. R., and Soffa, M. L. 2005. A model-based framework: an approach for profit-driven optimization. In *Proceedings of the Interational Conference on Code Generation and Optimization (CGO)*. 317–327.

Received November 2008; revised February 2010; accepted June 2010