Evaluating Iterative Optimization Across 1000 Data Sets

Yang Chen Yuanjie Huang
LCSA *, ICT, CAS, China
and Graduate School, CAS, China
{chenyang,huangyuanjie}@ict.ac.cn

Lieven Eeckhout
Ghent University, Belgium
Lieven.Eeckhout@elis.UGent.be

Grigori Fursin
INRIA, Saclay, France
grigori.fursin@inria.fr

Liang Peng
LCSA, ICT, CAS, China
and Graduate School, CAS, China
pengliang@ict.ac.cn

Olivier Temam
INRIA, Saclay, France
olivier.temam@inria.fr

Chengyong Wu
LCSA, ICT, CAS, China
cwu@ict.ac.cn

Abstract

While iterative optimization has become a popular compiler optimization approach, it is based on a premise which has never been truly evaluated: that it is possible to learn the best compiler optimizations across data sets. Up to now, most iterative optimization studies find the best optimizations through repeated runs on the same data set. Only a handful of studies have attempted to exercise iterative optimization on a few tens of data sets.

In this paper, we truly put iterative compilation to the test for the first time by evaluating its effectiveness across a large number of data sets. We therefore compose KDataSets, a data set suite with 1000 data sets for 32 programs, which we release to the public. We characterize the diversity of KDataSets, and subsequently use it to evaluate iterative optimization. We demonstrate that it is possible to derive a robust iterative optimization strategy across data sets: for all 32 programs, we find that there exists at least one combination of compiler optimizations that achieves 86% or more of the best possible speedup across *all* data sets using Intel's ICC (83% for GNU's GCC). This optimal combination is program-specific and yields speedups up to 1.71 on ICC and 2.23 on GCC over the highest optimization level (`-fast` and `-O3`, respectively). This finding makes the task of optimizing programs across data sets much easier than previously anticipated, and it paves the way for the practical and reliable usage of iterative optimization. Finally, we derive pre-shipping and post-shipping optimization strategies for software vendors.

Categories and Subject Descriptors D.3.4. [*Software: Programming Languages*]: Processors—Compilers, Optimization

General Terms Design, Experimentation, Measurement, Performance

* Key Laboratory of Computer System and Architecture

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.

Keywords Compiler optimization, Iterative optimization, Benchmarking

1. Introduction

Iterative optimization has become a popular optimization strategy [4, 10, 12, 22, 29] because of its simplicity while yielding substantial performance gains. It is essentially based on repeatedly trying out a large number of compiler optimizations until the best combination of compiler optimizations is found for a particular program (one 'iteration' is the evaluation of one combination of compiler optimizations). From the start though, there has been a nagging issue: how data set dependent is iterative optimization? More precisely, if one selects a combination of optimizations based on runs over one or a few data sets, will that combination still be the best for other data sets?

Answering this question is key in order to confidently use iterative optimization in practice. At the same time, this question is difficult to answer because there is no benchmark suite with a large enough number of data sets to emulate the two most likely application scenarios. Either a program is tuned through iterative optimization before shipping by the software vendor so that it performs well across a broad range of data sets; or a program is tuned after shipping, across a large number of production runs performed by users.

Several researchers have investigated how program behavior and performance varies across data sets. Zhong et al. [32] present two techniques to predict how program locality is affected across data sets; they consider 6 programs using up to 6 data sets each. Hsu et al. [20] compare the profiles generated using the 3 data sets of some SPECint2000 programs, and observe significant program behavior variability across data sets for some programs. Several studies focus on the impact of data sets on compiler optimization parameterization and selection. In particular, Berube et al. [6] collect 17 inputs per program on average for 7 SPECint2000 programs to study inlining. Haneda et al. [16] use the 3 SPECint2000 data sets to evaluate compiler optimizations. Mao et al. [24] collect between 8 and 976 inputs per program for 10 Java programs (150 inputs on average per program, 4 programs with more than 100 inputs) to investigate the impact of inputs on the selection of the best garbage collector. Fursin et al. [14] collect 20 data sets for each of the MiBench benchmarks — the same benchmark suite used in this paper — and also evaluate the data set sensitivity of compiler optimizations. These iterative optimization studies and others [4, 10, 12, 22, 29] underscore the fact that a significant number

of iterations (tens or hundreds) are required to find the best combination of compiler optimizations. However, to our knowledge, there is no benchmark suite available with multiple hundreds of distinct data sets per program. As a result, not only are researchers forced to evaluate iterative optimization using an unrealistic experimental setting, they are unable to answer the aforementioned fundamental question about whether iterative optimization is effective across a broad range of data sets.

In order to address this key question, we collect 1000 data sets for 32 programs, mostly derived from the MiBench benchmark suite [15]. Our results show that, for each benchmark, there is at least one combination of compiler optimizations that achieves 86% or more of the maximum speedup (i.e., the speedup obtained with the best possible combination per data set) across all data sets. This result has a significant implication: it means that, in practice, iterative optimization may often be largely data set insensitive. It also means that a program can be optimized by a software vendor on a collection of data sets before shipping, and will retain near-optimal performance for most other data sets. So the problem of finding the best optimization for a particular program may be significantly less complex than previously anticipated.

This paper is organized as follows. We present KDataSets in Section 2, and characterize its diversity and coverage in Section 3; we also compare KDataSets against the previously proposed MiDataSets, which comes with 20 data sets per benchmark, and show that a large population of data sets is indeed necessary to capture a sufficiently wide range of program behaviors. In Section 4, we then emulate realistic iterative optimization using the 1000 data sets in KDataSets, devise a strategy for selecting the best possible combination of optimizations, and evaluate the impact of an unrealistic experimental context on iterative optimization strategies. We also discuss the scope and the general applicability of the results obtained in this paper (Section 5). Finally, we summarize related work (Section 6) and conclude (Section 7).

2. KDataSets: A 1000-Data Set Suite

As mentioned in the introduction, we have collected 1000 data sets for each of our 32 benchmarks. Most of these benchmarks come from the MiBench benchmark suite. MiBench [15] is an embedded benchmark suite covering a broad spectrum of applications, ranging from simple embedded signal-processing tasks to smartphone and desktop tasks. It was developed with the idea that desktops and sophisticated embedded devices are on a collision course (for both applications and architectures, e.g., the x86 Intel Atom processor is increasingly used for embedded devices), calling for a broad enough benchmark suite. We use a modified version of the MiBench suite, which was evaluated across different data sets and architectures [2, 14]. We also added `bzip2` (both the compressor and the decompressor) to our set of benchmarks, and we plan to add and disseminate more programs over time. The benchmarks are listed in Table 1; the number of source lines ranges from 130 lines for kernels, e.g., `crc32`, to 99,869 lines for large programs, e.g., `ghostscript`.

Table 1 summarizes the various data sets in KDataSets; it describes the range in file size along with a description of how these data sets were obtained. The data sets vary from simple numbers and arrays, to text files, postscript files, pictures and audio files in different formats. Some data sets, such as the numbers for `bitcount` as well as the numbers in the array for `qsort`, are randomly generated. For other programs, such as `dijkstra` and `patricia`, we built data sets that exhibit distinct characteristics in terms of how the workload exercises different control flow paths and deals with different working set sizes — this was done based on studying the benchmarks' source code and our target domain knowledge.

Program (# source lines)	**Data set file size**	**Data set description**
bitcount (460)	-	Numbers: randomly generated integers
qsort1 (154)	32K-1.8M	3D coordinates: randomly generated integers
dijkstra (163)	0.06k-4.3M	Adjacency matrix: varied matrix size, content, percentage of disconnected vertices (random)
patricia (290)	0.6K-1.9M	IP and mask pairs: varied mask range to control insertion rate (random)
jpeg_d (13501)	3.6K-1.5M	JPEG image: varied size, scenery, compression ratio, color depth
jpeg_c (14014)	16k-137M	PPM image: output of jpeg_c (converted)
tiff_2bw (15477), _2rgba (15424), _dither (15399), _median (15870)	9K-137M	TIFF image: from JPEG images by ImageMagick converter (converted)
susan_c, _e, _s (each 1376)	12K-46M	PGM image: from jpeg images by ImageMagick converter (converted)
mad (2358)	28K-27M	MP3 audio: varied length, styles (ringtone, speech, music)
lame (14491), adpcm_c (210)	167K-36M	WAVE audio: output of mad (converted)
adpcm_d (211)	21K-8.8M	ADPCM audio: output of adpcm_c (converted)
gsm (3806)	83K-18M	Sun/NeXT audio: from MP3 audios by mad (converted)
ghostscript (99869)	11K-43M	Postscript file: varied page number, contents (slides, notes, papers, magazines, manuals, etc.)
ispell (6522), rsynth (4111), stringsearch1 (338)	0.1K-42M	Text file: varied size, contents (novel, prose, poem, technical writings, etc.)
blowfish_e (863)	0.6K-35M	Any file: a mix of text, image, audio, generated files
blowfish_d (863)	0.6K-35M	Encrypted file: output of blowfish_e
pgp_e (19575)	0.6K-35M	Any file: a mix of text, image, audio, generated files
pgp_d (19575)	0.4K-18M	Encrypted file: output of pgp_e
rijndael_e (952)	0.6K-35M	Any file: a mix of text, image, audio, generated files
rijndael_d (952)	0.7K-35M	Encrypted file: output of rijndael_d
sha (197)	0.6K-35M	Any file: a mix of text, image, audio, generated files
CRC32 (130)	0.6K-35M	Any file: a mix of text, image, audio, generated files
bzip2e (5125)	0.7K-57M	Any file: a mix of above text, image, audio, generated files, and other files like program binary, source code
bzip2d (5125)	0.2K-25M	Compressed file: output of bzip2e

Table 1. *KDataSets description.*

The text files are collected from the Gutenberg project (`gutenberg.org`) and `python.org`. Postscript files are collected from various web sites: `somethingconstructive.ne`, `oss.net`, `ocw.mit.edu`, `samandal.org`, `pythonpapers.org`, etc., and we converted some of the collected PDF files into PS format. Images are collected from `public-domain-image.com` and converted into the different required formats (TIFF, JPEG, PGM, PPM). Audio files are collected from `freesoundfiles.tintagel.net`, `jamendo.com`, `ejunto.com` and converted again into the appro-

priate formats (WAVE, ADPCM, Sun/NeXT). For programs with miscellaneous files as inputs (e.g., compression, encryption), we use a mix of the aforementioned files. And in some cases, the output of some programs are inputs to other programs, e.g., `mad`/`adpcm_c`. The entire data set suite corresponds to 27GB of data.

All data sets within KDataSets are free in the public domain or under the Creative Commons license, and will be publicly disseminated at `http://www.kdatasets.org`.

3. Data Set Characterization

In this section, we characterize KDataSets, analyze how distinct the data sets are with respect to each other, and how differently they react to compiler optimizations.

3.1 Experimental setup

Before doing so, we first briefly describe our experimental setup. We consider 7 identical machines with 3GHz Intel Xeon dual-core processors (E3110 family), 2GB RAM, 2×3MB L2 cache. We use the CentOS 5.3 Linux distribution based on Red Hat with kernel 2.6.18 patched to support hardware counters through the PAPI library [3].

Most of the results in the paper are obtained using the Intel ICC compiler v11.1. However, for completeness and in order to demonstrate that the results in this paper translate to other compilers, we have also conducted the same experiments using GNU GCC v4.4.1; these results are similar to the ICC results and are reported in Section 5.1. Both compilers feature a large number of optimizations. We selected 53 optimization flags for ICC and 132 flags for GCC that control inlining, unrolling, vectorization, scheduling, register allocation, constant propogation, among many others. We use a random optimization strategy for creating combinations of optimizations: a combination is based on the uniform random selection of flags; the number of flags is itself randomly selected. Random selection of compiler optimization combinations is a popular and effective approach for exploring the compiler optimization design space [8, 14, 27].

We consider 300 combinations of compiler optimizations for each program/data set pair throughout the paper — we study the results' sensitivity to the number of combinations in Section 5 by considering 8,000 combinations. For each combination and each data set, we measure the program's execution time. We also measure 9 performance characteristics using hardware performance counters through the PAPI interface, such as L1, L2 and TLB miss rates and branch prediction miss rates. And we also collect 66 architecture-independent characteristics using the MICA toolset [18], such as instruction mix, ILP, branch predictability, memory access patterns, and working set size.

Finally, Table 2 summarizes the terminology and definitions used throughout the text. We present this table here as a reference, and we will introduce the definitions in the text itself as need be. We use harmonic mean when reporting average speedup numbers.

3.2 Performance characterization

Figure 1 summarizes the performance characteristics for all programs and all data sets. The violin graphs show the distribution of the execution time (in cycles), IPC as well as other performance characteristics that relate to cache and branch behavior across all data sets per program. All graphs are shown on a logarithmic scale.

The goal of this characterization is to underscore the differences among the data sets — this illustrates the broad range of program behavior that these data sets generate. The execution time distribution (left top graph) shows that the amount of work varies greatly across data sets: while the standard deviation can be low, as for `tiffmedian`, it is fairly high for most other benchmarks.

data sets: $d \in D, \|D\| = 1000$.
optimization combinations: $o \in O, \|O\| = 300$.
speedup of o on d: s_d^o.
data set optimal speedup of d: $s_d^{optimal} = max\{s_d^o, o \in O\}$.
***fraction* of data set optimal speedup** of o on d: $f_d^o = \frac{s_d^o}{s_d^{optimal}}$.
program-optimal combination: o_{opt}:
o_{opt} **with highest mininum *fraction***: $o_{opt} = o/f_d^o = max_{o \in O} min\{f_d^o, d \in D\}$.
o_{opt} **with highest mininum *speedup***: $o_{opt} = o/s_d^o = max_{o \in O} min\{s_d^o, d \in D\}$.
o_{opt} **with highest average *fraction***: $o_{opt} = o/f_d^o = max_{o \in O} mean\{f_d^o, d \in D\}$.
o_{opt} **with highest average *speedup***: $o_{opt} = o/s_d^o = max_{o \in O} mean\{s_d^o, d \in D\}$.

Table 2. *Definitions and terminology.*

The other distributions (see the other graphs) show that not only does the amount of work vary wildly across data sets, but also the performance seen at the architecture level in terms of IPC and several other performance characteristics (L1 and L2 miss rates, and branch prediction rate).

3.3 How programs react to compiler optimizations across data sets

We now investigate how programs react to compiler optimizations across data sets. To this end, we compile each program with each of the 300 randomly generated combinations of compiler optimizations, and then run each of these 300 program versions with each of the 1000 data sets. We then record the best possible speedup for each data set, and we will refer to this speedup as the *data set optimal speedup*, see also Table 2 for a formal definition. The data set optimal speedup is the speedup relative to `-fast` and `-O3` for ICC and GCC, respectively. Because we have 1000 data sets per program (and hence 1000 data set optimal speedup numbers), we report these results as a distribution, see Figure 2 for the Intel ICC compiler; we report similar results for GNU's GCC in Section 5.1. The violin plots thus show the distribution of the data set optimal speedup for each benchmark. Figure 2(a) sorts the various benchmarks by average data set optimal speedup. Figure 2(b) shows the same data but sorts the benchmarks by the number of source code lines; this illustrates that the impact of compiler optimizations is uncorrelated with program size.

The results are contrasted. For some programs, the data set optimal speedups are within $\pm 1\%$ of the average for more than 98% of the data sets, i.e., there is hardly any discrepancy among the data sets. However, at the other end of the spectrum, ten programs exhibit significant variation in performance improvements, i.e., the deviation from their average performance ranges from 10% to 25%.

One can also note that, even for programs with a relatively small average performance improvement, there are some data sets for which the performance improvement is significant. Figure 3 illustrates this further. Here we have sorted all data sets and all programs according to the data set optimal speedup they achieve. For 14% of data sets, iterative optimization can achieve a speedup of 1.1 or higher, and for nearly half of the data sets, it can achieve a speedup of 1.05 or higher. The end conclusion is that iterative compilation yields substantial performance improvements across programs and data sets, and the significance of the improvement is sensitive to the data set.

We also find that the reactions to compiler optimizations are non-trivial, both across programs, and across data sets for the same program. In Figures 4 and 5, we plot the mean, standard deviation,

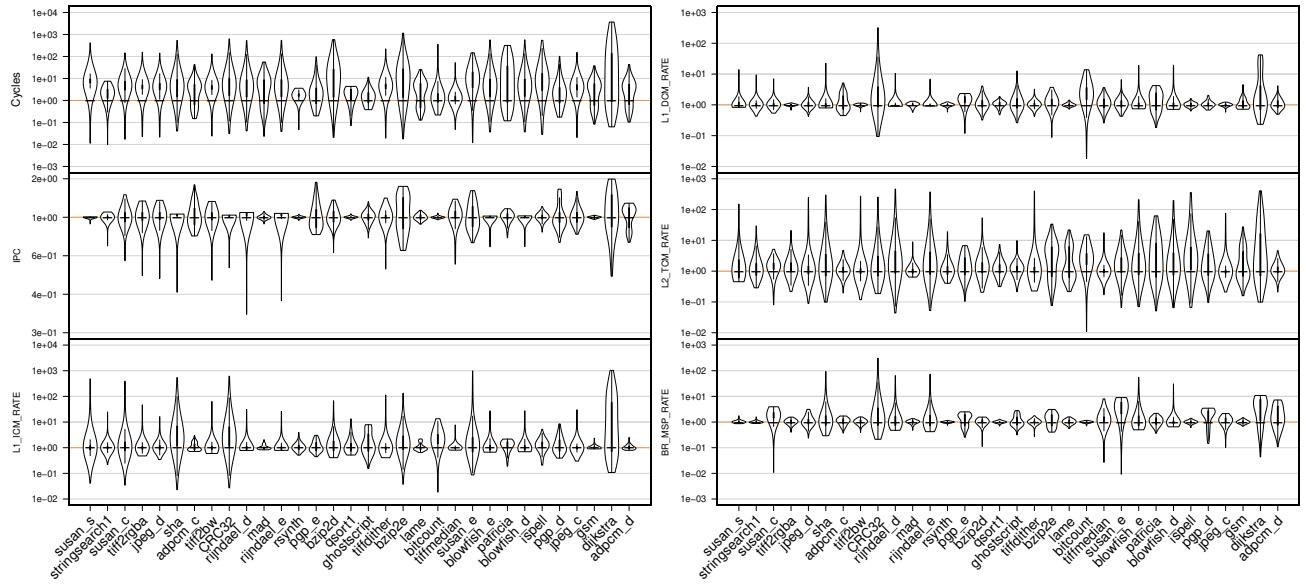
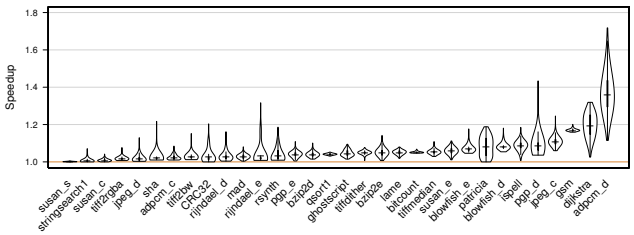
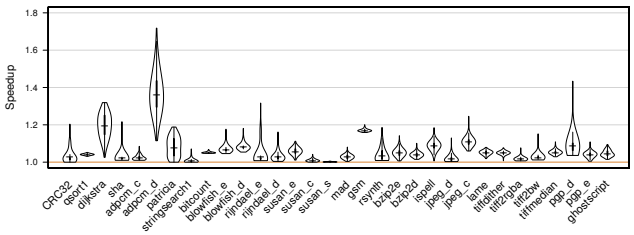


Figure 1. *Distribution of (normalized) performance characteristics across data sets (on a log scale): number of cycles, IPC, L1 instruction cache miss rate, L1 data cache miss rate, L2 cache miss rate, branch misprediction rate.*



(a) sorted by average data set optimal speedup



(b) sorted by number of source code lines

Figure 2. *Data set optimal speedups relative to* `-fast` *for KDataSets (the horizontal line shows the mean value).*

and maximum and minimum speedup for each compiler optimization, sorted by increasing mean speedup; this is done for the five programs with the highest average speedup across all data sets. A small arrow below the horizontal axis indicates the baseline combination (`-fast`). The large variations for the standard deviation show that the impact of each combination can vary strongly: for some programs, it is almost stable across all data sets, e.g., `gsm` (Figure 5(b)), or there are large variations across data sets (all other examples).

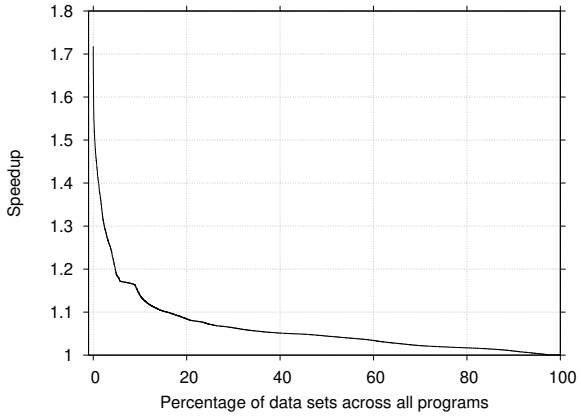


Figure 3. *Distribution of the data set optimal speedups across all data sets and all programs.*

3.4 KDataSets versus MiDataSets

We now compare KDataSets (1000 data sets) against MiDataSets (20 data sets) [14]. The reason for doing so is to investigate whether a 1000-data set suite is really needed compared to a 20-data set suite. The more fundamental question is whether the 20 data sets in MiDataSets cover most of the space built up by the 1000 data sets. In other words, do the 1000 data sets cover a much broader spectrum of the data set space to justify its usage over the 20 data sets that were previously collected? This is valid question, especially given that the data set optimal speedup are comparable for KDataSets (Figure 2) compared to MiDataSets (see Figure 6).

An in-depth analysis that involves a broad set of both micro-architecture-dependent and microarchitecture-independent characteristics reveals that the coverage of the program behavior space varies significantly across data sets. For that purpose, we have collected the 66 microarchitecture-independent features provided by

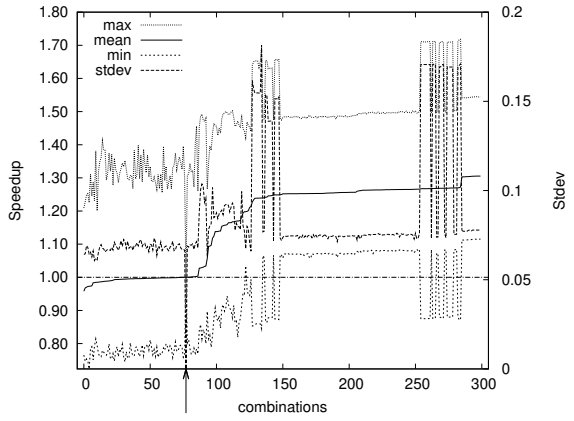
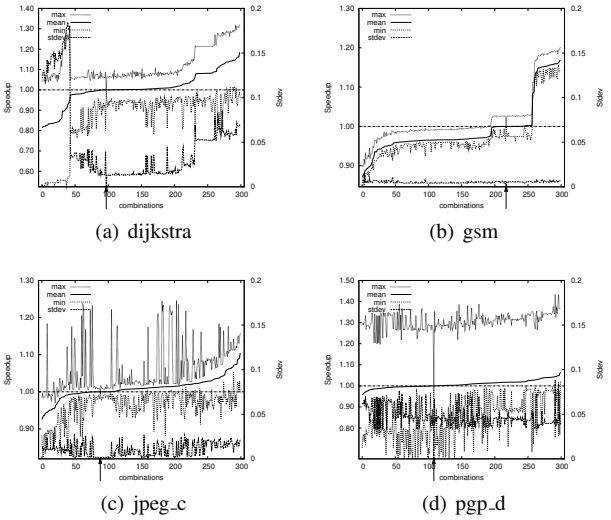


Figure 4. *Reactions to compiler optimizations for* `adpcm_d`.



(a) dijkstra (b) gsm

(c) jpeg_c (d) pgp_d

Figure 5. *Reactions to compiler optimizations for* `dijkstra`, `gsm`, `jpec_c`, `pgp_d`.

the MICA tool v0.22 [18] and 9 microarchitecture-dependent features using hardware performance counters as mentioned in Section 3.1. For each program, we apply principal component analysis (PCA) on the characteristics collected for all data sets, following the methodology by Eeckhout et al. [11]. We then plot the data sets along the two most significant principal components in Figure 7 — these principal components capture the most significant dimensions; we show plots for only a few representative programs due to paper length constraints. MiDataSets and KDataSets are shown using crosses and dots, respectively. KDataSets covers a larger part of the space than MiDataSets for 25 out of 32 programs, see for example `patricia`, `ghostscript`, `bitcount`, `pgp_d`. KDataSets substantially expands the space covered compared to MiDataSets for 4 out of 32 programs, e.g., `gsm`, `stringsearch1`. KDataSets and MiDataSets cover roughly the same space for three programs, such as `lame` and `adpcm_d`.

We have now made the case that KDataSets exhibits significantly different behavior compared to MiDataSets. However, this does not necessarily imply that KDataSets will react very differently to compiler optimizations. We therefore conduct yet another

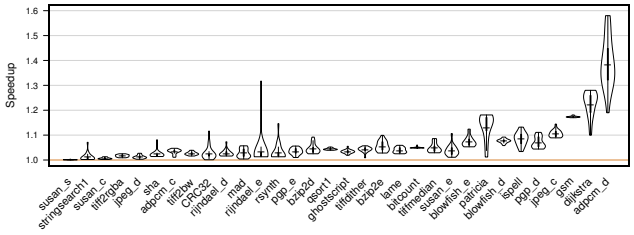


Figure 6. *Data set optimal speedups relative to* `-fast` *for MiDataSets.*

analysis using PCA with the features being the speedups obtained for each of the combinations, see Figure 8. The conclusion is essentially the same as before: the graphs clearly illustrate the greater diversity of reactions to compiler optimizations for KDataSets relative to MiDataSets.

4. Iterative Optimization in Practice

In this section, we investigate the stability of iterative optimization across data sets and revisit iterative optimization in a realistic experimental context for two application scenarios.

4.1 Program-optimal combinations

In order to investigate how sensitive the selection of combinations is to data sets, we first determine the data set optimal speedup, i.e., this is the highest speedup for each data set across all 300 combinations. Then, for each program, we retain the combination which yields the best average performance across all data sets. We term this combination the *program-optimal* combination, see Table 2 for a formal definition; we will clarify the exact selection process in the next section. In Figure 9, we report the performance for all data sets and for each program compiled with its program-optimal combination. The distribution of speedups (relative to `-fast`) across data sets is shown at the top, while the distribution of the *fraction of the data set optimal speedup* is shown at the bottom; the fraction of the data set optimal speedup is the ratio of the program-optimal combination speedup over the data set optimal speedup (see also Table 2) and it is always less than or equal to 1. The key observation is that, for each program, a single combination can achieve at least 86% of the data set optimal speedup for *all* data sets, with most programs standing at a minimum of 90% of the data set optimal speedup. As mentioned in the introduction, the consequences are significant. This result confirms that iterative optimization is robust across data sets: after learning over a sufficient number of data sets, the selected best tradeoff combination is likely to perform well on yet unseen data sets.

4.2 How to select the program-optimal combination

We now explain in more detail our strategy for selecting the program-optimal combination. In Table 3(a), we show an illustrative example with two data sets $D1$ and $D2$ and three combinations $comb1$, $comb2$ and $comb3$ plus the baseline (`-fast` for ICC). For each program, we evaluate every combination on every data set, deduce the data set optimal speedup, and, in the next two columns, compute the fraction of the data set optimal speedup achieved by every combination on every data set. We then want to select a combination that maximizes performance across all data sets. So we find the minimum fraction of the data set optimal speedup achieved by each combination (rightmost column), and pick the combination with the highest minimum fraction of the data set optimal speedup.

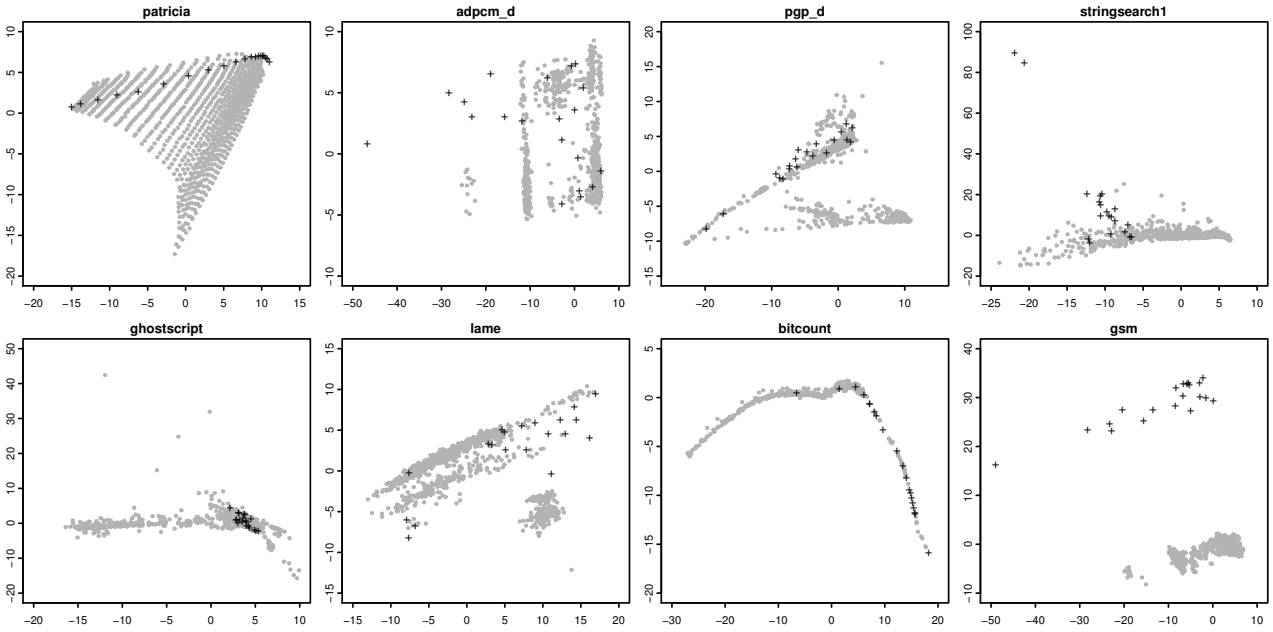


Figure 7. *Data set characterization using principal component analysis: the features are a set of microarchitecture-dependent and microarchitecture-independent characteristics; KDataSets is shown using gray dots, and MiDataSets is shown using black crosses.*

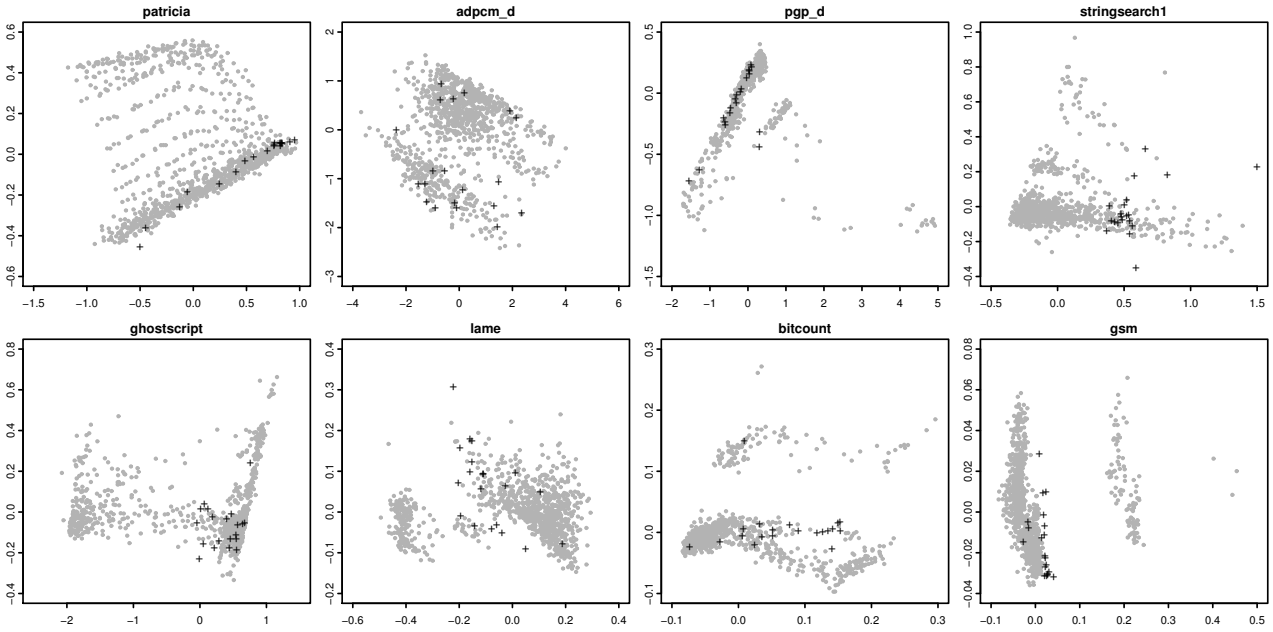


Figure 8. *Data set characterization using principal component analysis: the features are the speedup numbers across the 300 combinations of compiler optimizations; KDataSets is shown using gray dots, and MiDataSets is shown using black crosses.*

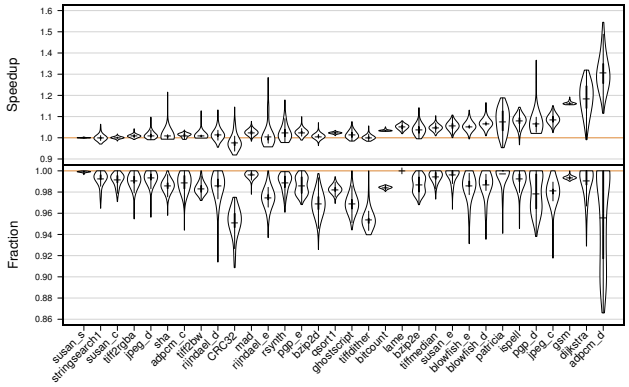


Figure 9. *Speedup (top graph) and fraction of the data set optimal speedup (bottom graph) for the program-optimal combinations.*

		Speedup			Fraction data set optimal speedup		
	Comb.	$D1$	$D2$	Avg.	$D1$	$D2$	Min
(a)	baseline	1.00	1.00	1.00	0.77	0.91	0.77
	comb1	1.20	1.10	1.15	0.92	1.00	0.92
	comb2	**1.30**	**1.07**	**1.19**	**1.00**	**0.97**	**0.97**
	comb3	1.25	1.05	1.15	0.96	0.95	0.95
	data set optimal	1.30	1.10	1.20	1.00	1.00	1.00

		Speedup			Fraction data set-optimal speedup		
	Comb.	$D1$	$D2$	Avg.	$D1$	$D2$	Min
(b)	baseline	1.00	1.00	1.00	0.77	0.99	0.77
	comb1	1.20	1.01	1.11	0.92	1.00	0.92
	comb2	**1.30**	**0.98**	**1.14**	**1.00**	**0.97**	**0.97**
	comb3	1.25	0.99	1.12	0.96	0.98	0.96
	data set optimal	1.30	1.01	1.16	1.00	1.00	1.00

Table 3. *Illustrative examples showing (a) how to select the program-optimal combination with the highest minimum fraction of the data set optimal speedup; (b) shows a case for which the program-optimal combination leads to slowdown relative to the baseline.*

Program-optimal combinations may sometimes, though rarely, induce slowdowns compared to the baseline. Let us define the data set optimal speedup for each data set as B; and let us define the fraction of the data set optimal speedup that the program-optimal speedup achieves as M. If the data set optimal speedup is small, it may be the case that $B \times M < 1$, i.e., a slowdown compared to the baseline. Because this only happens when B is small, the resulting slowdown is small as well. Consider the example in Table 3(b): because the data set optimal speedup for data set $D2$ is small, the fraction of the data set optimal speedup for $comb2$ and $comb3$ is high even though it induces a slight slowdown. As a result, $comb2$ gets selected as the combination with the highest minimum fraction of the data set optimal speedup; on average across $D1$ and $D2$, it does induce a significant average speedup, but there is a slight slowdown for $D2$. In Figure 9, we can see that this phenomenon happens (in top graph, violin part below speedup equal to one), though infrequently. For instance, programs such as `patricia` exhibit slowdowns for a few of their data sets. It is relatively more frequent for a couple programs such as `rijndael_e`; the most extreme example is `CRC32` for which the program-optimal combination yields a slowdown on average.

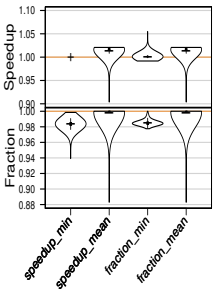


Figure 10. *Different ways for selecting program-optimal combinations for `CRC32`.*

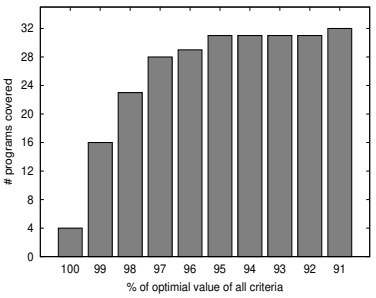


Figure 11. *Evaluation of the compromise selection strategy for determining the program-optimal combination.*

4.3 Alternative ways for selecting program-optimal combinations

There are different ways for how to determine the program-optimal combination. Depending on the user's objectives, it is possible to tune the selection of the program-optimal combination to minimize risk (maximize the minimum speedup), or to maximize average performance (maximize average speedup or maximize the average fraction of the data set optimal speedup). As an example, we show in Figure 10 the performance for program-optimal combinations of `CRC32` selected using these different criteria. While some exhibit an average slowdown, some result in an average speedup. Below, we study these different selection criteria and outline a general strategy for selecting program-optimal combinations.

We consider four different criteria for selecting the program-optimal combination: pick the combination that maximizes (i) the average speedup (across all data sets), or (ii) the minimum speedup, or (iii) the minimum fraction of the data set optimal speedup, or (iv) the average fraction of the best speedup. While the best criterion really depends on the user's objective, we propose the *compromise selection* strategy that performs well under all criteria. The strategy determines a set of program-optimal combinations that perform best under each criterion: each set for a criterion contains the combinations that achieve at least $P\%$ of the optimal value for that criterion. Then, we determine the intersection of these four sets of combinations. Each combination in this intersection is eligible to be a compromise program-optimal combination; we randomly pick a compromise program-optimal combination within that intersection. If the intersection is empty, we lower the value of P (from 100% downwards) and try again.

In Figure 11, we show the fraction of programs for which we can find at least one combination that achieves at least $P\%$ of the optimal value for all four criteria. For all programs, it is possible to find combinations that achieve 91% of the optimal value of all criteria; 95% of the optimal value is achieved for all but one program. As a result, the end conclusion is that, independently of the user's objective, there is almost always a combination that performs well for every criterion, at least among those tested above.

4.4 How many data sets are needed

One remaining issue for having a truly robust program optimization approach is to understand how many data sets are required in order to find a good program-optimal combination.

Program	Percentage of program-optimal performance			
	85%	90%	95%	99%
adpcm_d	2	3	5	11
dijkstra	2	2	2	5
gsm	2	2	2	7
jpeg_c	1	1	2	21
pgp_d	7	12	21	40
ispell	2	2	2	3
patricia	1	2	2	3
blowfish_d	1	2	6	154
blowfish_e	1	2	2	54
susan_e	122	145	179	259
tiffmedian	2	2	2	153
bzip2e	2	2	2	467
lame	2	2	2	2
bitcount	1	1	1	1
tiffdither	1	1	2	2
ghostscript	2	2	2	90
qsort1	1	2	16	34
bzip2d	1	32	74	340
pgp_e	2	2	2	294
rsynth	1	2	17	43
rijndael_e	2	20	130	623
mad	2	2	2	7
CRC32	2	2	138	326
rijndael_d	2	4	59	845
tiff2bw	1	1	2	615
adpcm_c	1	2	2	6
sha	8	21	41	231
jpeg_d	1	2	2	8
tiff2rgba	1	1	2	856
susan_c	1	1	2	7
stringsearch1	2	2	8	760
susan_s	2	2	60	119

Table 4. *Minimum number of data sets required to select a combination achieving X% of the program-optimal speedup; the benchmarks are sorted by their average data set optimal speedups.*

To do so, we set up the following experiment. We randomly sample N (out of 1000) data sets, select the program-optimal combination across these N data sets, evaluate its performance on all of the 1000 data sets, and deduce what percentage of the program-optimal combination performance is achieved. For each N, we repeat this process 1000 times and compute the average percentage. We then report the minimum number of data sets that are required to select a combination with a performance within 85%, 90%, 95% and 99% of the performance obtained when computing the program-optimal combination across all 1000 data sets, see Table 4.

The results vary across programs. For 14 out of the 32 programs, we achieve 85% of the program-optimal performance using only a single data set; only 1 program can achieve 95% of the program-optimal performance with a single data set. For 18 programs, at least 2 data sets are needed to achieve within 85% of the program-optimal speedup. The top five programs still need 6 data sets on average to achieve 95% of the program-optimal performance. Four programs require several tens, or even several hundreds, of data sets to achieve 90% of the program-optimal performance, and 14 programs require between 154 and 856 data sets to achieve 99% of the program-optimal performance.

In summary, about nine-tenths of the programs require from one to a few tens of data sets to achieve near-optimal performance, and the remaining tenth requires hundreds of data sets.

4.5 Evaluating iterative optimization within a realistic experimental context

The two most typical use cases for iterative optimization are: (i) the software vendor optimizes the software before shipping, and (ii) the end user is willing to tune the software she/he is using. There is one

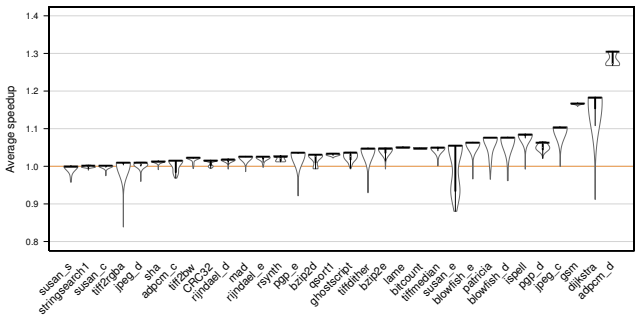


Figure 12. *Iterative optimization using a single data set versus using 1000 data sets; the horizontal line corresponds to the average speedup using the program-optimal combination selected over 1000 data sets.*

significant difference between these two cases for the iterative optimization learning process. In the first case, the software vendor can repeatedly run the same data sets, and thus easily compare the performance impact of combinations across runs. In the second case, the user is only doing production runs, so each run corresponds to a different data set. Most users are not willing to dedicate time, i.e., they do not want to perform repeated runs of the same data sets, in order to tune their compiler combinations; they instead want this process to happen transparently. As a result, the iterative optimization process must learn across different data sets, making it difficult to compare the performance impact of compiler combinations.

Until now, due to the lack of a large number of data sets, iterative optimization was evaluated in either scenario using unrealistic hypotheses. In the pre-shipping case, the evaluation was often conducted using a single data set. In the post-shipping case, the evaluation assumed that a user would be willing to run the same data sets multiple times. In the sections below, we use our large number of data sets to create realistic conditions for both scenarios, and to truly assess the potential performance benefits of iterative optimization in each scenario.

4.5.1 Pre-shipping scenario

In this scenario, we assess the performance benefit of tuning a software using a large number of data sets rather than a single one, see Figure 12. For the one-data set case, we compute the speedup as follows. For each program, we pick data sets one by one. For each data set, we run all possible combinations, select the combination with the maximum speedup, and apply it to all other 999 data sets; the average data set speedup is computed over the 999 data sets and the training data set. We report the corresponding distribution in Figure 12, as well as the average speedup obtained with the program-optimal combination selected using 1000 data sets (thick horizontal bar). The fact that the bulk of most violins is below the bar, and in most cases well below the bar, indicates that this training also often leads to sub-optimal performance. The fact that several violins are elongated also indicates that single-data set training is unstable/unreliable, i.e., performance will vary wildly depending on which data set is chosen.

4.6 Post-shipping scenario

In the post-shipping scenario, since we can only run each data set once, it is impossible to compute the speedup. This experimental context differs in two ways from the unrealistic context used in most iterative optimization studies. First, the relative ordering of combinations is imprecise: an experiment with a particular data set may conclude that one combination outperforms another, whereas

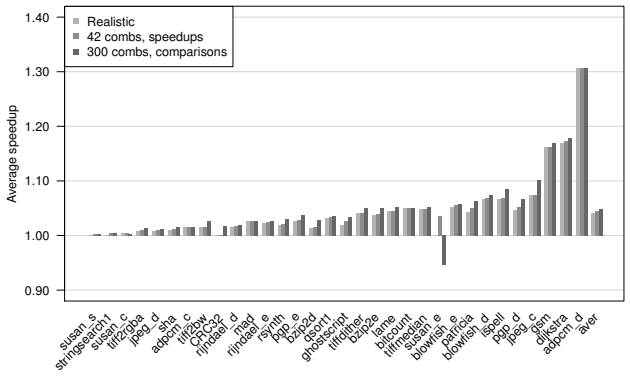


Figure 13. *Evaluation of a realistic versus unrealistic evaluation methodology in a post-shipping scenario.*

-fast -finline-limit=8 -no-vec
-fast -ansi-alias -finline -no-vec
-fast -no-vec
-O2
-fast -nolib-inline
-fast -fno-builtin -no-inline-factor
-fast -inline-max-total-size=3000 -no-inline-factor
-fast -inline-max-size=1024 -ipo3
-fast -no-inline-factor -unroll8
-fast -fno-fnalias -fno-omit-frame-pointer -no-vec
-fast -fno-alias -fpack-struct -no-vec
-fast -ansi-alias -finline-limit=512 -nolib-inline
-fast -finline-limit=512 -opt-jump-tables=large
-O3 -ipo1 -no-opt-jump-tables
-fast -no-inline-min-size
-fast -finline-limit=32 -fno-builtin -opt-malloc-options=2
-fast -ip-no-inlining -nolib-inline -opt-malloc-options=1
-fast -ipo2

Table 5. *The Intel ICC compiler optimizations that affect performance the most, obtained after pruning the program-optimal combinations.*

an experiment with another data set may conclude the opposite. Second, the number of combinations that can be evaluated is limited by the number of available data sets.

In order to compare combinations, we resort to the run-time approach proposed by Stephenson et al. [30] and Fursin et al. [13]. We use function cloning where each clone is compiled differently; this is akin to versioning: the code contains two versions of the same routines compiled using two different combinations. At runtime, either version of each method is randomly used upon each call, and the execution time is collected for each call, allowing to compare the sum of the execution times of the methods for each version. Even though this comparison is statistical, because of the discrepancies in numbers and durations of calls, previous research works have shown that it is reliable in practice and comes with low overhead [13, 30].

In order to realistically emulate iterative optimization in a post-shipping scenario, we must evaluate the combination selection strategies on unseen data sets. So we split the data sets into 900 training data sets and 100 test data sets. We randomly define one thousand such $900/100$ training/test sets, and conduct the evaluations below for all sets, and report the average results.

We now have to select the set of combinations to be evaluated. When optimizing one program, it is realistic to leverage the experience gathered from other programs. Therefore, for each program, we start with a set of high-potential combinations; these are the program-optimal combinations for the other 31 programs. In order to evaluate $N = 31$ combinations, we need $\frac{N\times(N-1)}{2} = 465$ comparisons and therefore need that many data sets. With 900 data sets at our disposal, we can increase the initial set of 31 combinations with N_r randomly selected combinations, in order to increase the probability of discovering new better suited combinations. The only constraint is that $\frac{(N+N_r)\times(N+N_r-1)}{2} \leq 900$; we find that the maximum value of N_r is 11, so we add 11 randomly selected combinations, for a total of 42 combinations.

We now compare the realistic case where 42 combinations are evaluated using $\frac{42\times41}{2} = 861$ runs versus the unrealistic case where the same set of combinations are evaluated on all data sets. The comparison between the realistic case, see the '`Realistic`' bars, and the unrealistic case, see the '`42 combs, speedups`' bars in Figure 13, helps discriminate the impact of comparing on distinct data sets. While the performance obtained under the realistic experimental conditions is slightly below the unrealistic case, we find that it is often close for most programs, and barely different on average.

We also compare the realistic case against a case where all combinations are compared, though using distinct data sets, in order to discriminate the impact of evaluating a restricted set of combinations. Since we do not have $\frac{300\times299}{2} = 44850$ data sets, we approximate this scenario by reusing data sets (i.e., for comparing each pair of combinations, we randomly pick a data set), and report the result in Figure 13, see '`300 combs, comparisons`'. We again find that the performance of the realistic case is very close to the performance of the less realistic case.

This result is significant. It implies that, even under realistic experimental conditions, it is possible to achieve almost the same performance as the upper bound performance defined by less realistic experimental conditions. In other words, the potential of iterative optimization can indeed be achieved in practice, even in a post-shipping scenario.

4.7 Analysis: Key compiler optimizations

We now delve a little deeper in the composition of program-optimal combinations. These combinations may contain more than 50 optimization flags, making it difficult to understand which optimizations really impact performance. We therefore prune the program-optimal combinations by simply removing optimization flags one by one, randomly, as long as they do not change the overall speedup. Table 5 shows the top performing combinations after pruning across all programs and data sets for ICC. (There are fewer pruned combinations than programs: program-optimal combinations may prune to the same combinations or the baseline.) This list of top combinations underscores that, even though modern compilers have powerful optimizations such as vectorization, inter-procedural analysis, etc., the heuristics in charge of activating these different optimizations and selecting their parameters may not be effective. For example, the `-fast` optimization level in ICC includes inlining and vectorization by default, but both optimizations may degrade performance. Iterative optimization does a better job at applying these optimizations to particular programs: e.g., only apply inlining to programs with small functions, or turn off vectorization when it is detrimental to performance.

5. Discussion on scope of the results

The results in this paper are obviously tied to the experimental setup. In this section, we aim at gaining insight into how general the results and conclusions are.

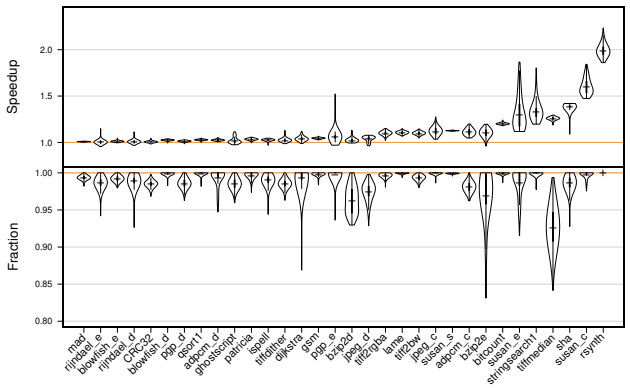
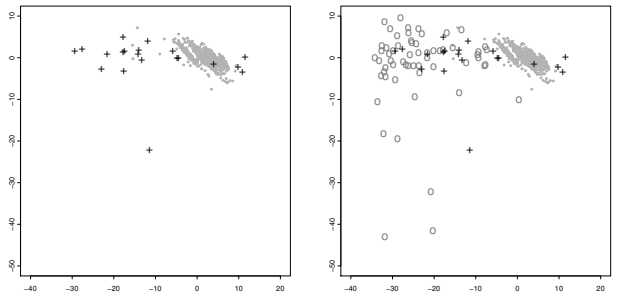


Figure 14. *Speedup (top graph) and fraction of the data set optimal speedup (bottom graph) for the program-optimal combinations for GNU's GCC.*



(a) *initial 1000 data sets* (b) *including small images*

Figure 15. *Program behavior space of* `tiffmedian`.

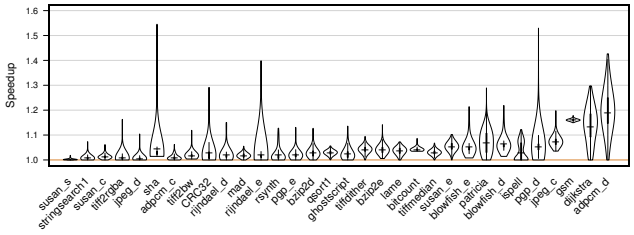


Figure 16. *Data set optimal speedups relative to* `-fast` *for 8000 combinations.*

5.1 Results for GNU's GCC

We used Intel's ICC compiler throughout the paper. We now evaluate whether the conclusions also apply to other compilers. We therefore consider GNU's GCC compiler v4.4.1. Figure 14 reports the data set optimal speedups (with respect to `-O3`) that can be achieved using GCC. Compared to Figure 9, we observe that GCC achieves higher speedups than ICC — this is relative to the `-O3` and `-fast` baseline combinations, respectively. It is also interesting to note that the best speedups are achieved for different programs for the most part. More importantly, we find again that, for each program, a single combination can achieve 83% of the data set optimal speedup for all data sets, with most programs standing at 90% or higher. Therefore, our conclusions are valid for two of the most widely used compilation platforms.

5.2 Data sets coverage

There is no guarantee that our collection of 1000 data sets covers the entire program behavior space — there may still be parts of the space that are unrepresented by KDataSets. For instance, our initial attempt at collecting data sets for program `tiffmedian` resulted in the distribution shown in Figure 15(a). The 20 data sets did in fact a better job at covering the program behavior space than our initial 1000 data sets. This turned out to be due to the lack of very small images ($< 29kB$). This is because these small images exhibit a significantly different L1, L2 and DTLB behavior than large images ($29kB - 1.5MB$), resulting in significantly different and more spread out program behavior. After increasing the mix of small and large images, the coverage of the 1000 data sets was eventually superior to that of the 20 data sets, as shown in Figure 15(b); the circles in Figure 15(b) correspond to the data sets added to our initial set, the withdrawn redundant data sets do not show. However, even the notion of 'covering the program behavior space' is subjective because a given user may be interested in a limited type of data sets only.

5.3 Compiler combinations

There is also no guarantee that the 300 combinations we explored covers the entire compiler optimization space. To see whether 300 combinations is too small a number, we conducted an additional experiment in which we consider 8000 combinations. However, to complete it in a reasonable amount of time, we had to run each combination on 10 randomly chosen data sets instead of all the 1000 data sets. The data set optimal speedups across these 8000 combinations are reported in Figure 16. Compared to Figure 2(a), except for a few data sets (e.g., in `sha` and `adpcm_d`), we observe no significant difference overall, which suggests that 300 combinations may be enough to represent the optimization space that can be explored in a reasonable time window.

5.4 Platforms and benchmarks

More generally, we cannot assert that our conclusions generalize beyond our benchmark suite, compiler platforms (Intel ICC and GNU GCC) and target architecture. There are also obvious examples where the performance of program segments is data set sensitive; auto-tuned libraries like FFTW [25] or ATLAS [31] are examples where data set dependent tuning is useful. Nevertheless, our results suggest that, in general, the data set sensitivity problem may have been overstated. We also note that our results are consistent across all benchmarks so far.

5.5 Fine-grain iterative optimization

For now, we only considered whole-program optimizations, i.e., a combination of compiler optimizations is applied across the entire program. However, a more fine-grain iterative optimization approach may apply different combinations to different sections of the code (e.g., functions or loops). Fine-grain iterative optimization is more complicated because it introduces optimization dependences (and possibly ripple effects): e.g., applying a combination to function F1 can affect the behavior of another function F2, and thus the choice of its best combination. This dependence complicates the problem as well as largely increases the design space. We will explore practical ways to tackle that issue as part of our future work.

5.6 Measurement bias

Recent work by Mytkowicz [26] raises the issue of measurement bias, and provides evidence for two sources of measurement bias,

namely link order and environment size. We believe that link order should not be considered measurement bias in the context of iterative optimization. Link order should rather be viewed as another opportunity for iterative optimization. Environment size affects only one fourth of the benchmark programs in [26] by a small margin only (within $\pm 2\%$ using ICC). In our study, 73% of program/dataset pairs have a speedup that is higher than 1.02, and are thus relatively immune to this source of measurement bias.

6. Related Work

We already mentioned in the introduction that several studies have investigated the impact of data sets on program behavior and performance. And some of these have even looked how data sets affect program optimization decisions. Most of these works remain limited by the restricted number of data sets available in existing benchmark suites. The SPEC CPU2000 suite contains 3 input sets per benchmark, and most benchmarks have less than 10 data sets in the SPEC CPU2006. The embedded EEMBC benchmark suite [1] also contains less than 10 data sets for most benchmarks. The recently introduced parallel PARSEC benchmark suite [7] contains 6 data sets for each benchmark. A large number of data sets is not only useful for compiler research and workload characterization research (e.g., [21]), many architecture studies rely on profile-based optimization techniques as well [23, 28], and may benefit from having more data sets in order to study data set sensitivity.

Several studies have investigated redundancy among data sets and how to find a minimal set of representative programs and inputs for architecture research [11]. We use several of the proposed statistical techniques and feature characterization approaches in this article to investigate our own data set suite.

Finally, we have mentioned the broad set of iterative/adaptive compilation techniques which attempt to find the best possible compiler optimizations by stochastically scanning the set of all possible combinations [4, 10, 12, 17, 19, 22, 29]. They have 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. Many of these research works have shown how machine-learning and statistical techniques [4, 9, 12, 22, 29] can be used to select or tune program transformations based on program features. Most of these works also require a large number of training runs. For instance, Stephenson et al. [30] and Arnold et al. [5] collect profile information across multiple runs of a Java program to selectively apply run-time optimizations.

7. Conclusions and Future Work

Using KDataSets, a collection of 1000 data sets for 32 programs, we investigate a fundamental issue in iterative optimization, which could not be thoroughly evaluated up to now: whether it is possible to learn the best possible compiler optimizations across distinct data sets. We conclude that the issue seems significantly more simple than previously anticipated, with the ability to find a near-optimal combination of compiler optimizations across all data sets. We outline a process for selecting the program-optimal combination, and we investigate the impact of performing iterative optimization in an unrealistic context.

For now, we have investigated whole-program optimizations, and we intend to study whether the same conclusions are sustained for fine-grain optimizations. We also intend to apply our conclusions to datacenters where, typically, a few programs are run a very large number of times, and where any execution time reduction translates into proportional gains in datacenter equipment and operating costs.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. Lieven Eeckhout is supported through the FWO projects G.0232.06, G.0255.08, and G.0179.10, and the UGent-BOF projects 01J14407 and 01Z04109. Olivier Temam is supported by the IST FP7 HiPEAC2 contract No. IST-217068. Olivier Temam and Grigori Fursin are supported by EU FP6 MILEPOST project contract No. IST-35307. The rest of the authors are supported by National Natural Science Foundation of China under grant No. 60873057 and 60921002, and National Basic Research Program of China under grant No. 2005CB321602.

References

[1] EEMBC: The Embedded Microprocessor Benchmark Consortium. `http://www.eembc.org`.

[2] cBench: Collective Benchmarks. `http://www.ctuning.org/cbench`.

[3] PAPI: A Portable Interface to Hardware Performance Counters. `http://icl.cs.utk.edu/papi`.

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

[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)*, pages 297–311, October 2005.

[6] P. Berube and J. Amaral. Aestimo: a feedback-directed optimization evaluation tool. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 251–260, March 2006.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, October 2008.

[8] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 185–197, 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, July 1999.

[10] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. ACME: adaptive compilation made efficient. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 69–77, July 2005.

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

[12] B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded programs. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 78–86, July 2005.

[13] G. Fursin and O. Temam. Collective optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC)*, pages 34–49, January 2009.

[14] 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)*, pages 245–260, January 2007.

[15] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE Fourth Annual International Workshop on Workload Characterization (WWC)*, pages 3–14, December 2001.

[16] M. Haneda, P. Knijnenburg, and H. Wijshoff. On the impact of data input sets on statistical compiler tuning. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006.

[17] K. Hoste and L. Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 165–174, April 2008.

[18] K. Hoste and L. Eeckhout. Comparing benchmarks using key microarchitecture-independent characteristics. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 83–92, October 2006.

[19] K. Hoste, A. Georges, and L. Eeckhout. Automated just-in-time compiler tuning. In *Proceedings of the Eighth Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, April 2010.

[20] W. C. Hsu, H. Chen, P. C. Yew, and D.-Y. Chen. On the predictability of program behavior using different input data sets. In *Proceedings of the Sixth Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT)*, pages 45–53, February 2002.

[21] Y. Jiang, E. Z. Zhang, K. Tian, F. Mao, M. Gethers, X. Shen, and Y. Gao. Exploiting statistical correlations for proactive prediction of program behaviors. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, April 2010.

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

[23] G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 14–27, June 2003.

[24] F. Mao, E. Z. Zhang, and X. Shen. Influence of program inputs on the selection of garbage collectors. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 91–100, March 2009.

[25] 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 (ICASSP)*, volume 3, pages 1381–1384, May 1998.

[26] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 265–276, February 2009.

[27] 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, March 2006.

[28] K. Sankaranarayanan and K. Skadron. Profile-based adaptation for cache decay. *ACM Transactions on Architecture and Code Optimization (TACO)*, 1:305–322, September 2004.

[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, June 2003.

[30] M. W. Stephenson. *Automating the Construction of Compiler Heuristics Using Machine Learning*. PhD thesis, MIT, USA, January 2006.

[31] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimization of software and the atlas project. In *Parallel Computing*, March 2001.

[32] Y. Zhong, X. Shen, and C. Ding. Program locality analysis using reuse distance. *Transactions on Programming Languages and Systems (TOPLAS)*, 31(6):1–39, Aug. 2009.