Deconstructing Iterative Optimization

YANG CHEN, SHUANGDE FANG, and YUANJIE HUANG, State Key Laboratory of Computer Architecture, ICT, CAS, China
LIEVEN EECKHOUT, Ghent University, Belgium
GRIGORI FURSIN, INRIA and Paris South University, France
OLIVIER TEMAM, INRIA, Saclay, France
CHENGYONG WU, State Key Laboratory of Computer Architecture, ICT, CAS, China

Iterative optimization is a popular compiler optimization approach that has been studied extensively over the past decade. In this article, we deconstruct iterative optimization by evaluating whether it works across datasets and by analyzing why it works.

Up to now, most iterative optimization studies are based on a premise which was never truly evaluated: that it is possible to learn the best compiler optimizations across datasets. In this article, we evaluate this question for the first time with a very large number of datasets. We therefore compose KDataSets, a dataset suite with 1000 datasets for 32 programs, which we release to the public. We characterize the diversity of KDataSets, and subsequently use it to evaluate iterative optimization. For all 32 programs, we find that there exists at least one combination of compiler optimizations that achieves at least 83% or more of the best possible speedup across *all* datasets on two widely used compilers (Intel's ICC and GNU's GCC). This optimal combination is program-specific and yields speedups up to 3.75× (averaged across datasets of a program) over the highest optimization level of the compilers (-O3 for GCC and -fast for ICC). This finding suggests that optimizing programs across datasets might be much easier than previously anticipated.

In addition, we evaluate the idea of introducing compiler choice as part of iterative optimization. We find that it can further improve the performance of iterative optimization because different programs favor different compilers. We also investigate why iterative optimization works by analyzing the optimal combinations. We find that only a handful optimizations yield most of the speedup. Finally, we show that optimizations interact in a complex and sometimes counterintuitive way through two case studies, which confirms that iterative optimization is an irreplaceable and important compiler strategy.

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

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Compiler optimization, Benchmarking, Iterative optimization

---

This article is an extension of a paper published in PLDI2010 [Chen et al. 2010].
G. Fursin is a HiPEAC member.
L. Eeckhout is supported by the FWO projects G.0255.08 and G.0179.10, the UGent-BOF projects 01J14407 and 01Z04109, the ICT Department of Ghent University, and the European Research Council the under the European Community's Seventh Framework Programme (FP7/2007-2013)/ERC Grant agreement no. 259295. O. Temam is supported by HiPEAC-2 NoE under grant European FP7/ICT 217068 and INRIA YOUHUA associated team funding. The rest of the authors except for G. Fursin are supported by the National Natural Science Foundation of China under grants No. 60873057, 60921002, and 61033009; the National Basic Research Program of Chian undern grant No. 2011CB302504; and National Science and Technology Major Project of China under grants No. 2009ZX01036-001-002 and 2011ZX01028-001-002.
Author's address: Y. Chen, email:chenyang.ict@gmail.com.

DOI 10.1145/2355585.2355594 http://doi.acm.org/10.1145/2355585.2355594

ACM Reference Format:
Chen, Y., Fang, S., Huang, Y., Eeckhout, L., Fursin, G., Temam, O., and Wu, C. 2012. Deconstructing iterative optimization. ACM Trans. Architec. Code Optim. 9, 3, Article 21 (September 2012), 30 pages.
DOI = 10.1145/2355585.2355594 http://doi.acm.org/10.1145/2355585.2355594

1. INTRODUCTION

Iterative optimization has become a popular optimization strategy [Cooper et al. 2005; Franke et al. 2005; Kulkarni et al. 2004; Agakov et al. 2006; Stephenson et al. 2003] because of its simplicity while yielding substantial performance gains. It is essentially based on repeatedly trying out a large number of compiler optimizations until a well-performing combination of compiler optimizations is found for a particular program. Over the years, many researchers [Cooper et al. 2005; Franke et al. 2005; Fursin et al. 2007; Fursin and Temam 2010; Kulkarni et al. 2004; Agakov et al. 2006; Stephenson et al. 2003; Cooper et al. 1999; Cavazos et al. 2007; Pan and Eigenmann 2004, 2006b] have made progress on various aspects of iterative optimization. In this article, we focus on the following two fundamental questions:

(1) *Whether* iterative optimization works across a large number of datasets?
(2) *Why* the optimization combinations found by iterative optimization work?

Answering the first question is key in order to confidently use iterative optimization in practice. This question is about how dataset-dependent iterative optimization is. More precisely, if one selects a combination of optimizations based on runs over one or a few datasets, will that combination still be the best for other datasets? This question is difficult to answer based on prior research because there is no benchmark suite with a large enough number of datasets.

Several researchers have investigated how program behavior and performance varies across datasets. Zhong et al. [2009] present two techniques to predict how program locality is affected across datasets; they consider 6 programs using up to 6 datasets each. Hsu et al. [2002] compare the profiles generated using the 3 datasets of some SPECint2000 programs, and they observe significant program behavior variability across datasets for some programs. Several studies [Berube and Amaral 2006; Haneda et al. 2006; Mao et al. 2009; Fursin et al. 2007] focus on the impact of datasets on compiler optimization parameterization and selection. In particular, Berube and Amaral [2006] collect 17 inputs per program on average for 7 SPECint2000 programs to study inlining. Fursin et al. [2007] collect 20 datasets for each of the MiBench benchmarks to evaluate the dataset sensitivity of compiler optimizations. These iterative optimization studies and others [Cooper et al. 2005; Franke et al. 2005; Kulkarni et al. 2004; Agakov et al. 2006; Stephenson et al. 2003] underscore the fact that a significant number of iterations[1] (tens or hundreds) are required to find the best combination of compiler optimizations. However, to the best of our knowledge, there is no benchmark suite available with several hundreds of distinct datasets per program. As a result, not only are researchers forced to evaluate iterative optimization using an unrealistic experimental setting using one or a few datasets, they are unable to answer the aforementioned fundamental question about whether iterative optimization is effective across datasets.

To address this key question, we collect 1000 datasets for 32 programs, mostly derived from the MiBench benchmark suite [Guthaus et al. 2001]. We call this collection KDataSets, which we make publicly available. Our results using KDataSets and two state-of-the-art compilers (GCC and ICC) show that, for each benchmark, there is

[1]One iteration is the evaluation of one combination of compiler optimizations.

at least one combination of compiler optimizations that achieves 83% or more of the maximum speedup (i.e., the speedup obtained with the best possible combination per dataset) across all datasets. This optimal combination is program-specific and yields speedups (averaged across datasets) up to 3.75× over the highest optimization level of the compilers (-O3 for GCC and -fast for ICC). This result has a significant implication: it means that, in practice, iterative optimization is largely dataset insensitive.

Then, we focus on the second question about *why* iterative optimization works. In our setting, iterative optimization uses combinations of compiler options (optimization options) to control compiler optimizations. We conduct detailed analyses of the optimal combinations to obtain insight in which optimization options are important and why. We find that only a handful of compiler options that control optimizations have a significant performance impact, and that most combinations actually require just a few compiler options. We also show, through two detailed case studies, how these compiler options improve performance and that their interactions can be complex, counterintuitive and elusive. This suggests that the results of iterative optimization, that is, broadly searching for combinations of compiler options, cannot be easily replicated through a manual and analytical process.

Through the process of deconstructing iterative optimization we explore one more opportunity for improving iterative optimization. We evaluate the idea of introducing compiler choice itself as part of iterative optimization, that is, a form of iterative meta-optimization. We find that it can further improve the performance of iterative optimization because different programs favor different compilers.

This article 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 datasets per benchmark, and we show that a large population of datasets is indeed necessary to capture a sufficiently broad range of program behavior. In Section 4, we then evaluate iterative optimization using the 1000 datasets of KDataSets. In Section 5, we illustrate the complex manner in which iterative optimization operates using two case studies. We discuss the scope and the general applicability of the results obtained in this article in Section 6. Finally, we summarize related work (Section 7) and conclude (Section 8).

2. KDATASETS: A 1000-DATASET SUITE

As mentioned in the introduction, we have collected 1000 datasets for each of our 32 benchmarks. Most of these benchmarks come from the MiBench benchmark suite. MiBench [Guthaus et al. 2001] 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, for instance, the x86 Intel Atom processor is increasingly used for embedded devices), which calls for a broad enough benchmark suite. In fact, we use a modified version of the MiBench suite plus `bzip2` (both the compressor and the decompressor), which was evaluated across different datasets and architectures [Fursin et al. 2007; cBench]. The benchmarks are listed in Table I; the number of source lines ranges from 130 lines for kernels, for instance, `crc32`, to 99,869 lines for large programs, for instance, `ghostscript`.

Table I also summarizes the various datasets in KDataSets; it describes the range in file size along with a description of how these datasets were obtained. The datasets vary from simple numbers and arrays, to text files, postscript files, pictures, and audio files in different formats. Some datasets, such as the numbers for `bitcount` as well as the numbers in the array for `qsort`, are randomly generated. For other programs,

Table I. KDataSets Description

Program	# source lines	Dataset file size	Dataset 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_d (converted)
tiff_2bw	15477	9K-137M	TIFF image: from JPEG images by ImageMagick converter (converted)
tiff_2rgba	15424		
tiff_dither	15399		
tiff_median	15870		
susan_c	1376	12K-46M	PGM image: from jpeg images by ImageMagick converter (converted)
susan_e	1376		
susan_s	1376		
mad	2358	28K-27M	MP3 audio: varied length, styles (ringtone, speech, music)
lame	14491	167K-36M	WAVE audio: output of mad (converted)
adpcm_c	210		
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	0.1K-42M	Text file: varied size, contents (novel, prose, poem, technical writings, etc.)
rsynth	4111		
stringsearch1	338		
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

such as `dijkstra` and `patricia`, we built datasets 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 by studying the benchmarks' source code.

The text files are collected from the Gutenberg project (`gutenberg.org`) and `python.org`. Postscript files are collected from various web sites: `somethingconstructive.net`, `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 appropriate formats (WAVE, ADPCM, Sun/NeXT). For programs with miscellaneous files as inputs (e.g., compression, encryption), we use a mix of the aforementioned files. In some cases, the output of some programs are inputs to other programs, e.g., `mad`/`adpcm_c`. The entire dataset suite corresponds to 27 GB of data.

All datasets within KDataSets are publicly available at `kdatasets.appspot.com`.

Table II. Definitions and Terminology

datasets: $d \in D, \|D\| = 1000$.
optimization combinations: $o \in O, \|O\| = 300$.
speedup of o on d: s_d^o.
dataset optimal speedup of d: $s_d^{optimal} = max\{s_d^o, o \in O\}$.
***fraction* of dataset 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 minimum *fraction***: $o_{opt} = o/f_d^o = max_{o \in O} min\{f_d^o, d \in D\}$.
o_{opt} **with highest minimum *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***: $o_{opt} = o/s_d^o = max_{o \in O} mean\{s_d^o, d \in D\}$.

3. KDATASETS CHARACTERIZATION

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

3.1. Experimental Setup

We first briefly describe our experimental setup. We consider a configuration of 3GHz Intel Xeon dual-core processor (E3110 family) with 2×3 MB L2 cache and a 2GB of RAM. 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 [PAPI]. We use the GNU GCC v4.4.1 compiler. We also present experiments with another compiler (namely Intel's ICC) and more platforms in later sections to show that our main conclusion is not limited to this specific configuration.

The GCC compiler features a large number of optimizations. We selected 132 compiler options for GCC, namely inlining, unrolling, vectorization, scheduling, register allocation, constant propogation, among many others. We use a random optimization strategy for creating combinations of optimization options: a combination is based on the uniform random selection of options; the number of options is itself randomly selected. Random selection of compiler optimization combinations is a popular and effective approach for exploring the compiler optimization design space [Pan and Eigenmann 2006a; Cavazos et al. 2007; Fursin et al. 2007]. Random combinations sometimes expose compiler bugs that lead to buggy program binaries that produce incorrect results on some or all of the datasets. We call them buggy combinations. We filter these buggy combinations as a first step out.[2] We consider 300 combinations[3] of compiler optimizations for each program/dataset pair throughout the article. (We study the results' sensitivity to the number of combinations in Section 6 by considering 8,000 combinations). For each combination and each dataset, we measure the program's execution time (wall clock time). We 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. We repeated each execution three times and report the average of the three measurements in this article. We use harmonic mean when reporting average speedup numbers. We also collect 66 microarchitecture-independent characteristics using the MICA toolset [Hoste and Eeckhout 2006], such as instruction mix, ILP, branch predictability, memory access patterns, and working set size.

Finally, Table II 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.

[2]For GCC, 4.6% of the random combinations were found to be buggy and replaced with ones that produce bug-free program binaries.

[3]The 300 combinations are the same for all the datasets of one program, not across programs.

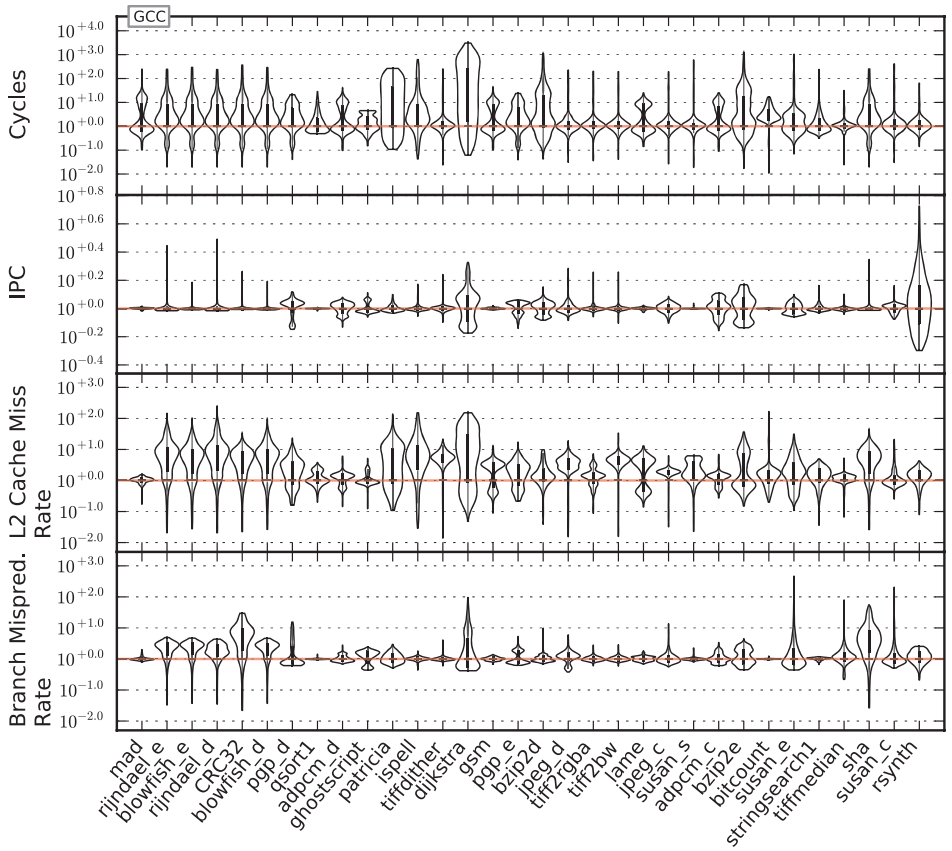

Fig. 1. Distribution of (normalized) performance characteristics across datasets (on a log scale) from top to bottom: number of cycles, instruction per cycle, L2 cache miss rate and branch misprediction rate.

3.2. Performance Characterization

Figure 1 summarizes the performance characteristics for all programs and all datasets. The violin graphs show the distribution of the normalized execution time, IPC as well as other performance characteristics that relate to cache and branch behavior across all datasets per program. All graphs use a logarithmic scale on the y-axis.

The goal of this characterization is to underscore the differences among the datasets: this illustrates the broad range of program behavior that these datasets generate. The execution time distribution (top graph) shows that the amount of work varies greatly across datasets: while the standard deviation can be low for some benchmarks, for instance, `tiffmedian`, it is fairly high for most other benchmarks. The other graphs in Figure 1 show that not only does the amount of work vary wildly across datasets, but also performance in terms of IPC and several other performance characteristics (L1 and L2 miss rates, and branch prediction rate) exhibit significant variability.

3.3. How Programs React to Compiler Optimizations Across Datasets

We now investigate how programs react to compiler optimizations across datasets. To this end, we compile each program with each of the 300 randomly generated combinations of compiler optimizations, and we run each of these 300 program versions with

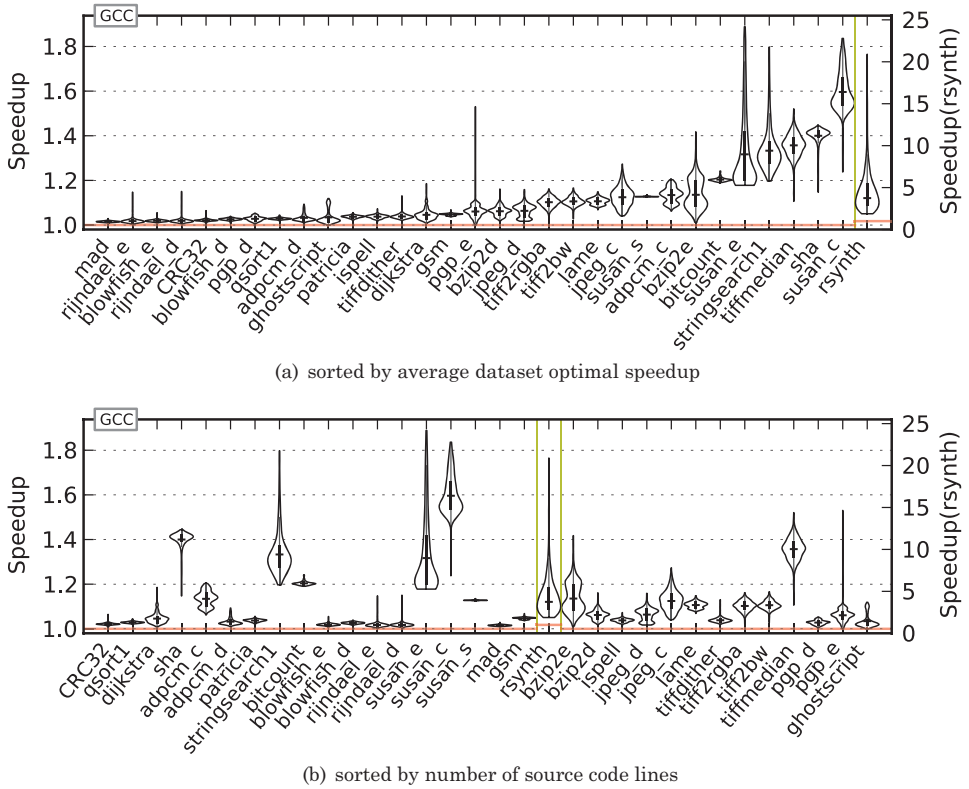

Fig. 2. Dataset optimal speedups relative to `-O3` for KDataSets (the horizontal line in each violin shows the mean value of the distribution). `rsynth` uses a separate y axis (right one) because of its high speedups.

each of the 1000 datasets. We then record the best possible speedup for each dataset, and we will refer to this speedup as the *dataset optimal speedup*, see also Table II for a formal definition. The dataset optimal speedup is defined as the best possible speedup relative to GCC `-O3` for a given dataset. Because we have 1000 datasets per program (and hence 1000 dataset optimal speedup numbers), we report these results as a distribution (see Figure 2): the violin plots show the distribution of the dataset optimal speedup for each benchmark. Figure 2(a) sorts the various benchmarks by average dataset 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 dataset optimal speedups are within $\pm 1\%$ of the average for more than 98% of the datasets, that is, there is hardly any discrepancy among the datasets. However, at the other end of the spectrum, twelve programs exhibit significant variation in performance improvements, that is, the deviation from their average performance ranges from 10% up to 463%. The unusually high speedups achieved on the speech synthesis program `rsynth` are due to an optimization that directs the CPU to speed up certain floating-point operations at the cost of a slight loss in precision. It is often used by programmers to optimize precision-loss tolerant media processing applications [Slingerland and Smith 2001]. Note that this optimization does not necessarily result in program output changes because media applications usually have a quantization setup which is insensitive to slight floating-point inaccuracy; it

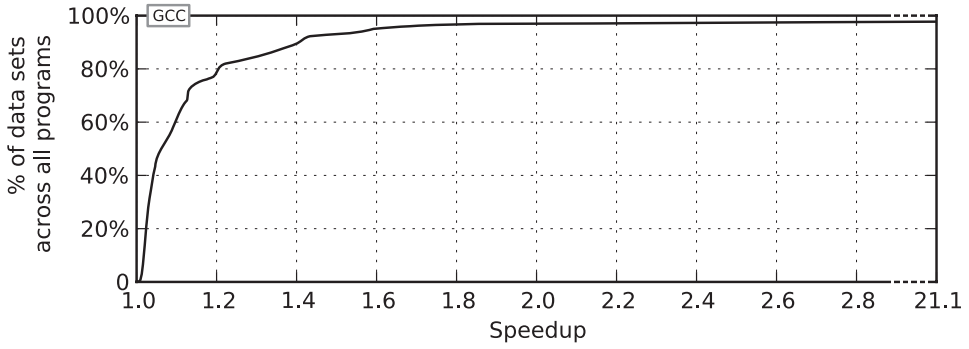

Fig. 3. Distribution of the dataset optimal speedups across all datasets and all programs.

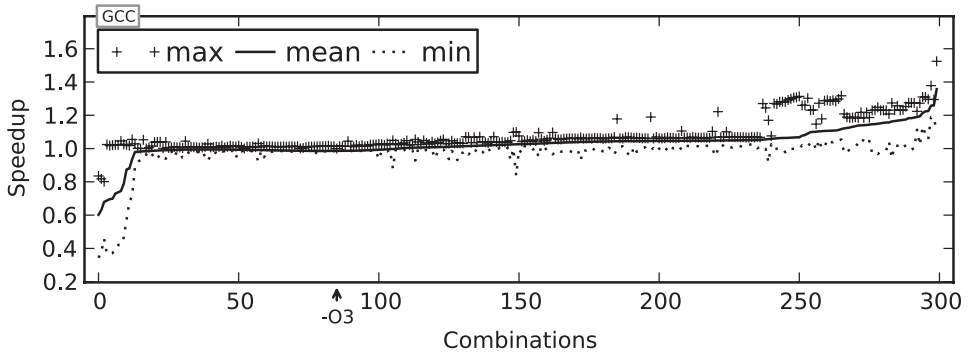

Fig. 4. Reactions to compiler optimizations for `tiffmedian`.

is the case for `rsynth`: we have verified that all the outputs of all recompiled programs, including `rsynth`, are the same as the outputs produced by programs compiled using the default optimization level (GCC's `-O3`).

One can also note that, even for programs with a relatively small average performance improvement, there are some datasets for which the performance improvement is significant. Figure 3 illustrates this further. Here we have sorted all datasets and all programs according to the dataset optimal speedup they achieve. For 39% of the datasets, iterative optimization can achieve a speedup of 1.1× or higher, and for more than half of the datasets, it can achieve a speedup of 1.05× or higher. Overall, iterative compilation yields substantial performance improvements across programs and datasets, and the magnitude of the improvement varies across datasets.

It is also interesting to note that dataset sensitivity depends on the combination of compiler optimizations. This is illustrated for `tiffmedian` in Figure 4: we plot the mean, and maximum and minimum speedup for each compiler optimization, sorted by increasing mean speedup. Due to space constraints, we only show the figure for `tiffmedian` as a typical example. This graph shows that the speedups achieved by a given combination can vary widely across datasets.

In summary, we find that the reactions to compiler optimizations are nontrivial, both across programs, and across datasets for the same program.

3.4. KDataSets Versus MiDataSets

We now compare KDataSets (1000 datasets) against MiDataSets (20 datasets) as previously proposed by Fursin et al. [2007]. The reason for doing so is to determine whether

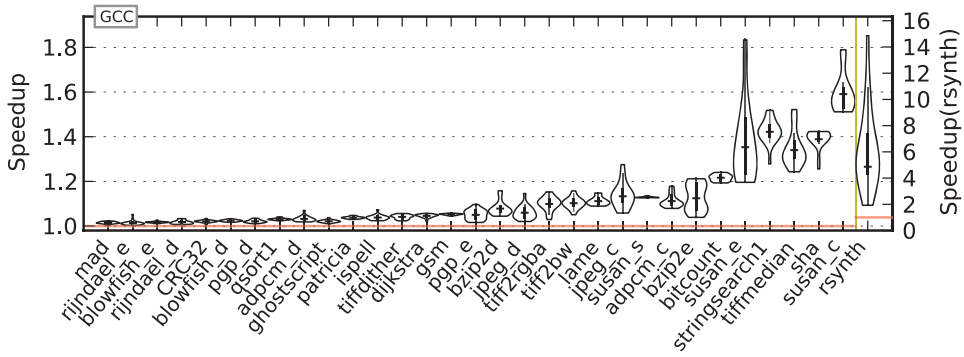

Fig. 5. Dataset optimal speedups relative to -O3 for MiDataSets.

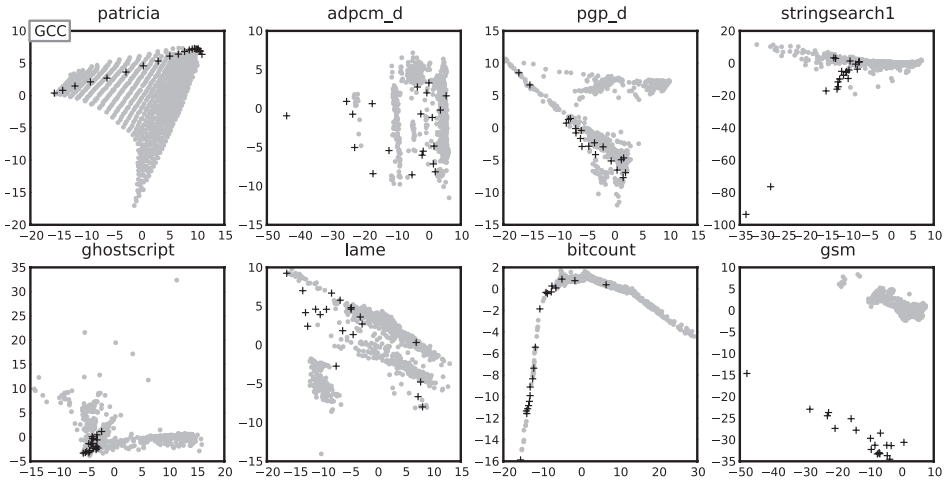

Fig. 6. Dataset characterization using principal component analysis: the features are a set of microarchitecture-dependent and microarchitecture-independent characteristics; KDataSets is shown using gray dots, and MiDataSets is shown using black crosses. The horizontal and vertical axes show the first and second principal component, respectively. Each principal component is a weighted linear combination of all the characteristics.

the 20 datasets of MiDataSets cover the same program behavior as the 1000 datasets of KDataSets. The question is not obvious considering that the dataset optimal speedups of KDataSets (see Figure 2) and MiDataSets (see Figure 5), are comparable.

Using a broad set of both microarchitecture-dependent and microarchitecture-independent characteristics, we find that KDataSets covers a significantly broader span of program behavior than MiDataSets. We have collected the 66 microarchitecture-independent features provided by the MICA tool v0.22 [Hoste and Eeckhout 2006] 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 datasets, following the methodology by Eeckhout et al. [2003]. We then plot the datasets along the two most significant principal components in Figure 6; these principal components capture the most significant dimensions; we show plots for only a few representative programs due to space 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 the

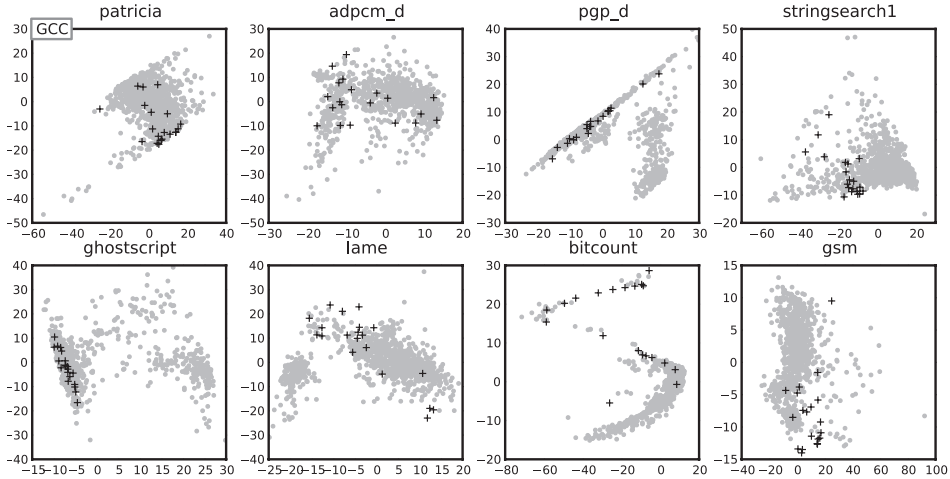

Fig. 7. Dataset characterization using principal component analysis: the features are the speedup numbers across the 300 combinations of compiler optimizations; KDataSets is shown using gray dots, and MiDataSets is shown using black crosses. The horizontal and vertical axes show the first and second principal component, respectively. Each principal component is a weighted linear combination of all the characteristics.

32 programs, see for example `patricia`, `ghostscript`, `bitcount` and `pgp_d`. KDataSets substantially expands the space covered compared to MiDataSets for 4 out of the 32 programs, for instance, `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 differently to compiler optimizations. We therefore conduct another analysis using PCA where the features are the speedups obtained for each of the 300 combinations of compiler optimizations (see Figure 7). 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. DATASET SENSITIVITY OF ITERATIVE OPTIMIZATION

Now that we have shown that KDataSets exhibits diverse behavior, we can evaluate whether iterative optimization works across a large number of datasets.

4.1. Program-Optimal Combinations

In order to investigate how sensitive the selection of combinations is to datasets, we first determine the dataset optimal speedup, that is, this is the highest speedup that can be obtained for a given dataset. Then, for each program, we retain the combination that yields the best overall performance across all datasets. We term this combination the *program-optimal* combination (see Table II for a formal definition); and we will clarify the exact selection process in the next section. In Figure 8, we report the performance for all datasets and for each program compiled with its program-optimal combination. The distribution of speedups (relative to `-O3`) across datasets is shown in the top graph, while the distribution of the *fraction of the dataset optimal speedup* is shown in the bottom graph; the fraction of the dataset optimal speedup is the ratio of the program-optimal combination speedup over the dataset optimal speedup (see also Table II) 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 83% of the dataset optimal speedup for *all*

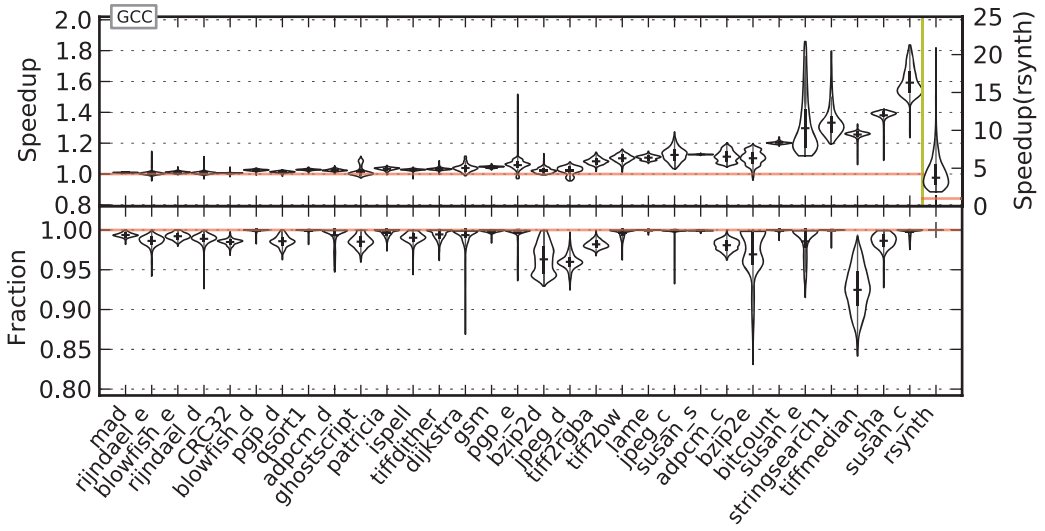

Fig. 8. Speedup (top graph) and fraction of the dataset optimal speedup (bottom graph) for the program-optimal combinations.

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

(a)

	Speedup			Fraction dataset optimal speedup		
Comb.	$D1$	$D2$	Avg.	$D1$	$D2$	Min
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
dataset optimal	1.30	1.10	1.20	1.00	1.00	1.00

(b)

	Speedup			Fraction dataset optimal speedup		
Comb.	$D1$	$D2$	Avg.	$D1$	$D2$	Min
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
dataset optimal	1.30	1.01	1.16	1.00	1.00	1.00

datasets, with most programs standing at a minimum of 90% of the dataset optimal speedup. The consequences of this observation are significant. This result confirms that iterative optimization is robust across datasets: after learning over a sufficient number of datasets, the selected best tradeoff combination is likely to perform well on yet unseen datasets.

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 III(a), we show an illustrative example with two datasets $D1$ and $D2$ and three combinations $comb1$, $comb2$ and $comb3$ plus the baseline (`-O3` for GCC). For each program, we evaluate every combination on every dataset, we deduce the dataset optimal speedup, and, in the next two columns, we compute the fraction of the dataset

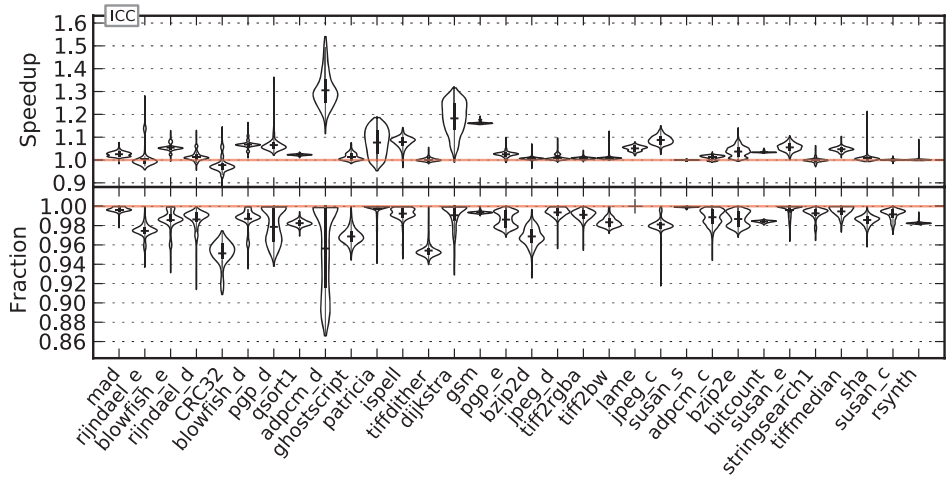

Fig. 9. Speedup (top graph) and fraction of the dataset optimal speedup (bottom graph) for the program-optimal combinations for ICC.

optimal speedup achieved by every combination on every dataset. We then want to select a combination that maximizes performance across all datasets. So we find the minimum fraction of the dataset optimal speedup achieved by each combination (rightmost column), and we pick the combination with the highest minimum fraction of the dataset optimal speedup.

Program-optimal combinations may sometimes, though rarely, induce slowdowns compared to the baseline. Let us define the dataset optimal speedup for each dataset as B; and let us define the fraction of the dataset optimal speedup that the program-optimal speedup achieves as M. If the dataset optimal speedup is small, it may be the case that $B \times M < 1$, that is, 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 III(b): because the dataset optimal speedup for dataset $D2$ is small, the fraction of the dataset optimal speedup for *comb*2 and *comb*3 is high even though it induces a slight slowdown. As a result, *comb*2 gets selected as the combination with the highest minimum fraction of the dataset 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 8, we can see that this phenomenon happens (in the top graph, violin part below speedup equal to one), though infrequently. For instance, programs such as `bzip2e` exhibit slowdowns for a few of their datasets. It is relatively more frequent for a couple of programs such as `jpeg_d`.

4.3. Results for Another Compiler: Intel's ICC

So far, we have used GNU's GCC compiler. We now evaluate whether the conclusions also apply to other compilers. We therefore consider Intel's ICC compiler v11.1. We selected 53 optimization options for ICC, and we use the random approach, as described in Section 3.1, to create 300 combinations of compiler options.[4] From Figure 9, we observe that the best speedups are achieved for different programs in most cases. More importantly, we find again that, for each program, a single combination can achieve 86% of the dataset optimal speedup for all datasets, with most programs standing

[4]For ICC, 3.7% of the randomly generated combinations were found to be buggy and replaced with ones that produce bug-free program binaries.

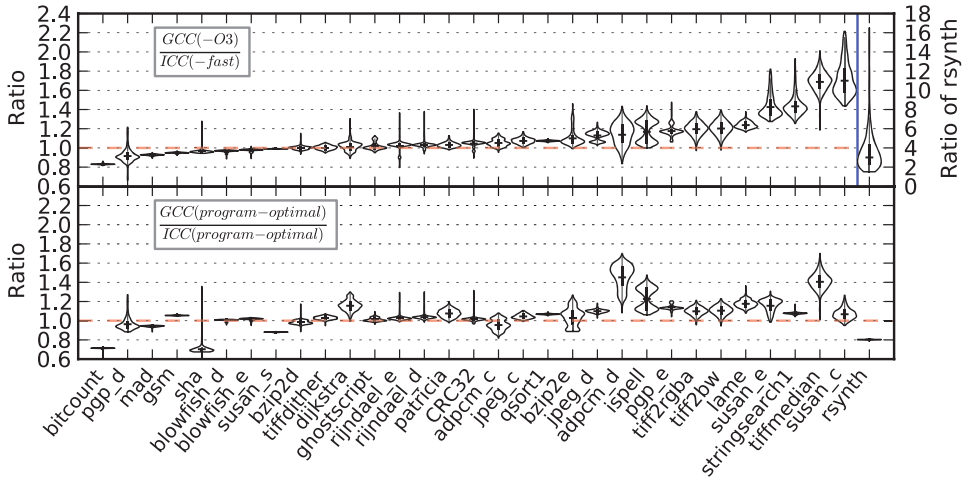

Fig. 10. Comparing the performance of GCC and ICC. The top graph shows the performance of GCC -O3 versus ICC -fast. The bottom graph compares the performance of the program-optimal combinations for the two compilers. A ratio greater than 1 means ICC is better, below 1 means GCC is better.

at 90% or higher. Therefore, our conclusions stand for two of the most widely used compilation platforms.

4.4. Compiler Choice as Part of the Iterative Optimization

In the previous sections, we have considered ICC and GCC separately. In this section, we go one step further and we introduce compiler choice as part of the iterative optimization, that is, we not only try different optimization combinations of a compiler, but we also try different compilers in the iterative optimization process. To the best of our knowledge, this is the first time this idea of iterative metaoptimization is evaluated in the literature.

In order to show that it makes sense to choose different compilers for different programs, we first compare the performance of GCC `-O3` and ICC `-fast` on every dataset. In the top graph of Figure 10, we report the ratio of the runtime of GCC `-O3` over that of ICC `-fast` for each dataset. A ratio greater than 1 means ICC `-fast` yields better performance, while a value below 1 means GCC `-O3` yields better performance. We observe that ICC `-fast` achieves better performance for 23 out of the 32 programs, while GCC `-O3` performs better for the remaining 9 benchmarks. It means that the highest default optimization level of ICC is generally better than that of GCC.

However, as we have observed in Section 4.3, the relative speedups achieved by the program-optimal combinations of GCC (`-O3` as baseline) are generally higher than those of ICC (`-fast` as baseline). In the bottom graph of Figure 10, we compare their absolute performance for every dataset. Comparing the two graphs in Figure 10, we observe that, although iterative optimization improves performance of GCC significantly on several programs such as `sha`, `susan_c`, `rsynth`, ICC still yields better average performance on 24 of the 32 programs.

This result suggests that it makes sense to introduce compiler choice as part of the iterative optimization process. Figure 11 shows the performance improvements we achieve over the program-optimal combinations for GCC. We observe significant speedups for most of the programs, up to 1.70×. Figure 12 shows the performance improvements with ICC as the baseline. It confirms that although ICC is generally better, GCC is still able to bring further speedups for some of the programs.

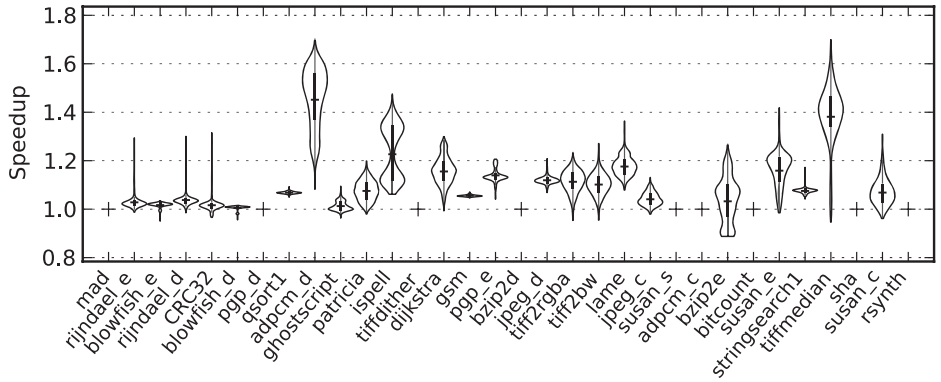

Fig. 11. Performance improvement over the program-optimal combination of GCC after introducing compiler choice as part of the iterative optimization.

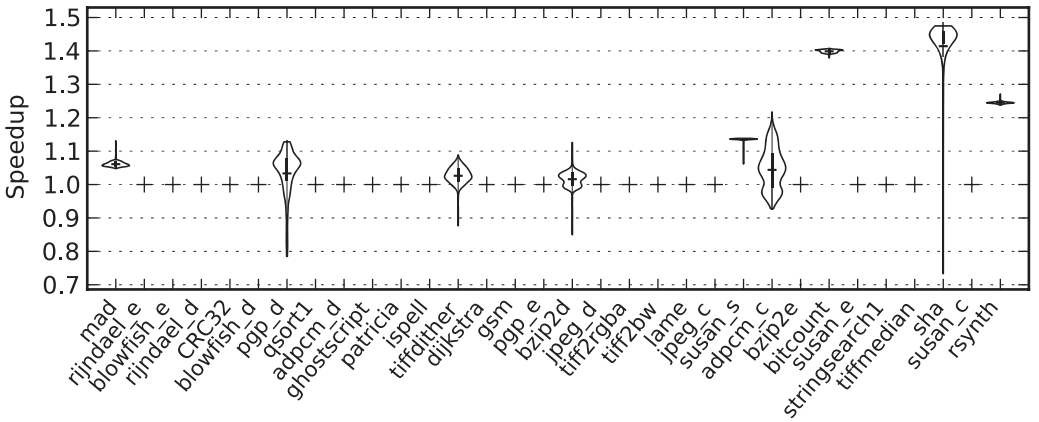

Fig. 12. Performance improvement over the program-optimal combination of ICC after introducing compiler choice as part of the iterative optimization. The big outlier of `sha` is introduced by a dataset that is very small.

We do similar experiments as in the previous section to investigate the dataset sensitivity of iterative optimization in the current context. As shown in Figure 13, we find that, for each program, a single combination can achieve at least 77% of the dataset optimal speedup across all datasets, with most programs standing at 90% or higher. Therefore, our main conclusions are still valid after we introduce compiler choice as part of the iterative optimization process.

5. ANALYZING PROGRAM-OPTIMAL COMBINATIONS

Having the notion of a program-optimal combination of compiler optimizations, we now focus on the second part of our analysis: understanding why iterative optimization operates in practice. We especially focus on understanding which compiler options, or combinations of compiler options, are important, and why.

Table IV lists the program-optimal combinations for the top 15 MiBench programs that are selected out of the 32 programs because they achieve significant speedups (more than 1.03) over GCC `-O3`. In this section, we analyze these combinations in detail. Note that, as `-O3` is the baseline, we represent all the combinations in a unified form of `-O3` plus zero or more options. Optimization levels other than `-O3` are represented as

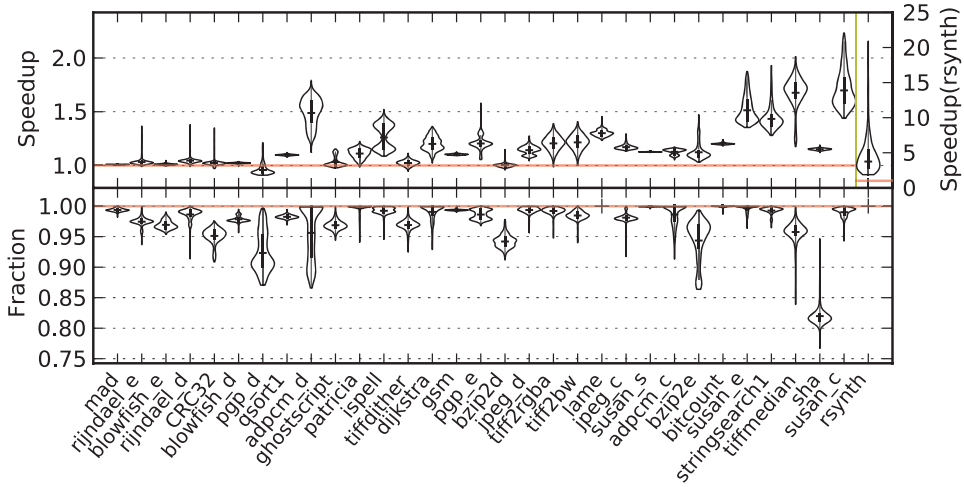

Fig. 13. Speedup (top graph) and fraction of the dataset optimal speedup (bottom graph) for the program-optimal combination out of 300 GCC combinations + 300 ICC combinations.

`-O3 -Oy` because GCC lets the last optimization level override previous ones, that is, `-Ox -Oy` equals `-Oy`. For example, `-O2 -ftracer` is represented as `-O3 -O2 -ftracer`. As a result, we omit `-O3` when we count options in the following sections. We only consider the 50 datasets exhibiting the highest speedups in order to reduce the total experimentation time.

5.1. Analysis of Useful Options

We first prune out options that have little or no performance impact. Second, we evaluate the performance contribution of each of the remaining options to the corresponding program-optimal combination. Third, we examine each combination to see whether most of the speedup is achieved by only one option or a combination of options. Finally, we show some non-trivial interactions among the options in some combinations.

5.1.1. Pruning Combinations of Options. As shown in Table IV, program-optimal combinations found using random search can contain tens of compiler options. In order to know which of these options are really useful, we remove the options one by one as long as the resulting combination has the same performance (within $\pm 1\%$) as the program-optimal combination.

The pruned combinations are listed in Table V. Comparing it to the unpruned ones in Table IV, we observe a drastic decrease in the number of options in each combination. For instance, `rsynth` drops from 20 to 4 compiler options; `tiffmedian` drops from 28 to 4; `tiff2rgba` drops from 13 to 1. As a result, the two longest combinations have 5 options, while another five have 4 options, three have 3 options, three have 2 options, two have only 1 option.

Overall, the number of useful options is actually rather small (and never exceeds 5). In other words, most of the options can be pruned out without affecting performance. Most of the pruned options either disable optimizations that are enabled by `-O3`, or alter parameters of optimization algorithms. This suggests that, for a particular program, turning on or off some of the optimizations implied by `-O3` does not have a measurable performance impact, and that some optimizations are not very sensitive to their parameters.

Table IV. Program-Optimal Combinations of the Top 15 MiBench Programs for GCC

Program	program-optimal combination
rsynth	-O3 –param large-stack-frame-growth=1716 -fcse-skip-blocks -fipa-cp -fipa-reference -fno-branch-count-reg -fno-if-conversion -fno-peephole2 -fno-prefetch-loop-arrays -fno-tree-fre -fno-tree-pre -fno-tree-reassoc -fno-unswitch-loops -fno-variable-expansion-in-unroller -foptimize-sibling-calls -fpeephole -fstrict-aliasing -ftree-ch -ftree-dce -ftree-dse -ftree-vectorize -funsafe-math-optimizations
susan_c	-O3 –param ira-max-loops-num=122 –param max-unrolled-insns=1561 -falign-labels=14 -falign-loops -fforward-propagate -fno-ipa-struct-reorg -fno-ivopts -fno-move-loop-invariants -fno-schedule-insns2 -fno-tree-builtin-call-dce -fno-tree-pre -fsched-interblock -fsplit-ivs-in-unroller -ftree-sra
sha	-O3 -fcaller-saves -fcprop-registers -fira-coalesce -fno-conserve-stack -fno-gcse-sm -fno-if-conversion2 -fno-inline-functions -fno-loop-block -fno-merge-constants -fno-rtl-abstract-sequences -fno-tree-dce -fno-tree-sra -fno-tree-ter -fno-tree-vrp -fno-unsafe-loop-optimizations -fregmove -fsched-stalled-insns-dep=1 -fsched-stalled-insns=64 -fsee -fselective-scheduling -funroll-all-loops -fvect-cost-model
tiffmedian	-O3 –param max-inline-insns-recursive-auto=772 -fgcse -fipa-cp-clone -fno-align-loops -fno-btr-bb-exclusive -fno-cprop-registers -fno-gcse-sm -fno-inline-functions-called-once -fno-ipa-cp -fno-ivopts -fno-optimize-sibling-calls -fno-rerun-cse-after-loop -fno-rtl-abstract-sequences -fno-split-wide-types -fno-tree-loop-optimize -fno-tree-vect-loop-version -fno-web -fprefetch-loop-arrays -fsched-spec -fsched-stalled-insns-dep=16 -fsched-stalled-insns=16 -fselective-scheduling -fselective-scheduling2 -fthread-jumps -ftree-ch -ftree-sink -funroll-all-loops -funswitch-loops
stringsearch1	-O3 –param large-stack-frame-growth=1741 -falign-functions=8 -fgcse-sm -finline-functions -finline-functions-called-once -fipa-cp -fno-align-labels -fno-btr-bb-exclusive -fno-guess-branch-probability -fno-ivopts -fno-sched-spec-load-dangerous -fno-tree-loop-distribution -fno-tree-reassoc -fpeephole2 -fsignaling-nans -ftree-pre -funroll-all-loops
susan_e	-O3 -fif-conversion -fira-share-save-slots -fmove-loop-invariants -fno-expensive-optimizations -fno-forward-propagate -fno-gcse -fno-ivopts -fno-modulo-sched -fno-optimize-sibling-calls -fno-prefetch-loop-arrays -fno-reorder-functions -fno-sched-interblock -fno-selective-scheduling -fno-tree-dce -fno-tree-loop-ivcanon -fno-unroll-all-loops -fpeel-loops -fpredictive-commoning -frerun-cse-after-loop -fstrict-overflow -ftracer -ftree-parallelize-loops=51
bitcount	-O2 -ffunction-cse -fno-align-functions -fno-branch-count-reg -fno-ipa-pta -fno-loop-block -fno-see -fno-strict-aliasing -fno-thread-jumps -fno-tree-dce -fno-tree-vect-loop-version -fno-unsafe-math-optimizations -fsplit-wide-types -ftree-ch -ftree-loop-distribution -ftree-pre -ftree-reassoc -funroll-all-loops
bzip2e	-O3 -fconserve-stack -fmerge-constants -fmodulo-sched-allow-regmoves -fno-cprop-registers -fno-gcse-lm -fno-ira-coalesce -fno-ivopts -fno-move-loop-invariants -fno-sched-spec-load -freorder-blocks -ftree-reassoc -ftree-switch-conversion -funroll-all-loops
adpcm_c	-O2 -fipa-struct-reorg -fno-align-loops -fsee -ftracer -ftree-copyrename -ftree-fre
susan_s	-O3 -fno-if-conversion -fno-peephole2 -fregmove -fsel-sched-pipelining -funroll-all-loops
jpeg_c	-O2 –param ira-max-loops-num=122 –param max-unroll-times=8 -fcse-follow-jumps -fdefer-pop -fgcse-las -fipa-pta -fira-share-save-slots -fno-align-jumps -fno-conserve-stack -fno-guess-branch-probability -fno-loop-block -fno-modulo-sched -fno-peel-loops -fno-sched-spec -fno-sel-sched-pipelining -fno-tree-loop-optimize -fpeephole2 -fregmove -frerun-cse-after-loop -freschedule-modulo-scheduled-loops -fsched-stalled-insns-dep=11 -ftree-ch -ftree-dse -ftree-vect-loop-version -funroll-all-loops
lame	-O3 -fcaller-saves -fcprop-registers -fira-coalesce -fno-conserve-stack -fno-gcse-sm -fno-if-conversion2 -fno-inline-functions -fno-loop-block -fno-merge-constants -fno-rtl-abstract-sequences -fno-tree-dce -fno-tree-sra -fno-tree-ter -fno-tree-vrp -fno-unsafe-loop-optimizations -fregmove -fsched-stalled-insns-dep=1 -fsched-stalled-insns=64 -fsee -fselective-scheduling -funroll-all-loops -fvect-cost-model
tiff2bw	-O3 –param ira-max-loops-num=140 –param max-unrolled-insns=921 -falign-jumps=53 -falign-loops=1 -fearly-inlining -fgcse-las -fif-conversion2 -fipa-pure-const -fipa-struct-reorg -floop-strip-mine -fno-align-labels -fno-cse-skip-blocks -fno-inline-functions -fno-inline-small-functions -fno-peel-loops -fno-tree-dce -fno-tree-pre -fno-tree-sra -fno-tree-vectorize -fno-unsafe-loop-optimizations -fprefetch-loop-arrays -freorder-blocks -freschedule-modulo-scheduled-loops -fsee -fselective-scheduling -fselective-scheduling2 -ftree-dominator-opts -ftree-loop-im -ftree-loop-ivcanon
tiff2rgba	-O3 –param max-unrolled-insns=394 -fgcse -fgcse-las -fipa-cp -fno-cx-limited-range -fno-omit-frame-pointer -fno-sched-spec -fno-thread-jumps -fno-tree-loop-distribution -fsched-stalled-insns-dep=2 -fsched2-use-superblocks -ftree-loop-im -fweb
jpeg_d	-O2 –param max-inline-insns-recursive=470 -falign-functions=35 -falign-jumps=20 -fgcse-sm -finline-functions -floop-strip-mine -fno-delete-null-pointer-checks -fno-loop-block -fno-modulo-sched -fno-regmove -fno-sel-sched-pipelining -fno-thread-jumps -fno-tree-copy-prop -fpeephole -freorder-blocks-and-partition -fsched-stalled-insns-dep=47 -ftree-parallelize-loops=35 -funroll-all-loops

Table V. Program-Optimal Combinations after Pruning.

Program	program optimal combination
rsynth	-O3 -fno-tree-fre -fno-tree-pre -fno-tree-reassoc -funsafe-math-optimizations
susan_c	-O3 -fno-ivopts -fno-move-loop-invariants -fno-schedule-insns2 -fno-tree-pre
sha	-O3 -fira-coalesce -fno-inline-functions -fno-merge-constants -fno-tree-ter -funroll-all-loops
tiffmedian	-O3 -fno-cprop-registers -fno-inline-functions-called-once -fno-tree-loop-optimize -funroll-all-loops
stringsearch1	-O3 -fno-guess-branch-probability -fno-ivopts -funroll-all-loops
susan_e	-O3 -fno-ivopts -ftracer
bitcount	-O3 -O2 -fno-align-functions -fno-tree-dce -funroll-all-loops
bzip2e	-O3 -fno-ivopts -fno-move-loop-invariants -funroll-all-loops
adpcm_c	-O3 -O2 -ftracer
susan_s	-O3 -funroll-all-loops
jpeg_c	-O3 -O2 -fno-guess-branch-probability -fno-tree-loop-optimize -funroll-all-loops
lame	-O3 -fira-coalesce -funroll-all-loops
tiff2bw	-O3 –param max-unrolled-insns=921 -falign-jumps=53 -fno-inline-small-functions -fprefetch-loop-arrays -fselective-scheduling2
tiff2rgba	-O3 -fno-omit-frame-pointer
jpeg_d	-O3 -O2 -finline-functions -funroll-all-loops

5.1.2. Which Options Are Important and How Much Do They Contribute?. We now examine the options in the pruned combinations individually. To understand the performance contribution of an option to a combination, we compare the performance of the combination with and without that option. More precisely, the performance contribution of an option (e.g., `-O2`) to a combination (e.g., `-O3 -O2 -ftracer`) is defined as the speedup achieved by adding the option (`-O2`) to the combination of the remaining options (`-O3 -ftracer`), that is, *runtime*(`-O3 -ftracer`)/*runtime*(`-O3 -O2 -ftracer`).

In Table VI, we report the speedup as defined above, averaged across all datasets, for each option in the pruned combinations.[5] If an option appears in the combinations for more than one program, its speedup is averaged again over the programs. In Table VI, the options are in the first column, the number of occurrences of each option are in the second column, the average speedups are in the third column, and the fourth column explains the effects of the options. The options are sorted by decreasing speedups.

As shown in Tables V and VI, the 15 pruned combinations have 47 options in total, of which 27 are distinct. Out of the 27, only 15 options have more than 1.03 speedup contributions to the corresponding program-optimal combinations, 11 options have more than 1.10 contributions, 3 have more than 1.20 contributions. The option `-funroll-all-loops` has a performance impact for a maximum of 9 combinations, followed by `-fno-ivopts` and `-O2`, each of which appears in 4 combinations. Another 5 options appear in 2 combinations. The remaining 17 options appear in only one combination.

As shown by the fourth column, most of the options start with `-fno-`$\cdots$. They disable one or several optimizations implied by `-O3`. A few of the options, such as `-funroll-all-loops` and `-ftracer`, turn on optimizations that are not implied by `-O3`. Another few, such as `-falign-jumps` and `-fselective-scheduling2`, change parameters or algorithms used by optimization passes.

In summary, by examining the individual performance contributions of these options, we find that only a handful of options have a significant performance impact in the program-optimal combinations.

5.1.3. How Many Optimizations Are Important Within a Combination?. In this section, we examine each pruned combination to see whether its performance improvement is contributed mostly by one dominant option or several options.

[5]The speedups shown here may look different from those in Figure 8 because in this section we evaluate the options only in a selected subset of the 1000 datasets, as explained at the start of this section.

Table VI. Performance Contribution of Each Individual Option

Option	#combs	Speedup	Description
-funsafe-math-optimizations	1	11.66	Floating-point optimizations, may violate IEEE or ANSI standards.
-fno-guess-branch-probability	2	1.35	Don't guess branch probabilities using heuristics.
-fno-ivopts	4	1.25	Disable induction variable optimizations on trees.
-fno-tree-loop-optimize	2	1.18	Disable loop optimizations on trees.
-fno-inline-functions	1	1.14	Disable optimization that inline all simple functions.
-funroll-all-loops	9	1.14	Unroll all loops, including loops with uncertain number of iterations.
-fno-omit-frame-pointer	1	1.14	Don't keep the frame pointer in a register for functions that don't need one.
-falign-jumps=53	1	1.12	Align branch targets to a power-of-two boundary, skipping up to 53 bytes.
-fselective-scheduling2	1	1.12	Schedule instructions using selective scheduling algorithm.
-fno-inline-small-functions	1	1.11	Don't inline functions without mark `inline` and whose body is smaller than expected function call code.
-fno-tree-pre	1	1.10	Disable partial redundancy elimination on trees.
-ftracer	2	1.09	Perform tail duplication to enlarge superblock size.
-fno-move-loop-invariants	2	1.08	Disable the loop invariant motion pass in the RTL loop optimizer.
-O2	4	1.04	-O2
-fno-tree-ter	1	1.03	Disable temporary expression replacement in SSA-to-normal phase.
-fprefetch-loop-arrays	1	1.02	Prefetch memory to accelerate loops that access large arrays.
–param max-unrolled-insns=921	1	1.02	The upper limit of instructions that a loop should have if that loop is unrolled.
-fno-inline-functions-called-once	1	1.02	Don't inline the static functions that are called once and without mark `inline`.
-fira-coalesce	2	1.02	Do optimistic register coalescing.
-fno-cprop-registers	1	1.02	Disable copy-propagation after register allocation and post-register allocation instruction splitting.
-finline-functions	1	1.01	Inline all simple functions to their callers.
-fno-schedule-insns2	1	1.01	Disable instruction scheduling after register allocation.
-fno-align-functions	1	1.01	No align the start of functions to the a power-of-two boundary.
-fno-tree-dce	1	1.01	Disable dead code elimination on trees.
-fno-merge-constants	1	1.00	Don't attempt to merge identical constants across compilation units.

From all the options in a program-optimal combination, we enumerate all subsets of $N(\geq 1)$ options and evaluate their performance. In Figure 14, we report the maximum performance achieved by all possible groups of N options with bar graphs, which are grouped per program. The performance is reported as a percentage of the speedup achieved by the whole program-optimal combination. Recall that the number of options in each pruned combination varies, which leads to a different number of bars for each program. Overall, there are 8 programs for which one option is able to achieve most (more than 95%) of the maximum performance (the program-optimal speedup). In the remaining 7 programs, `rsynth` and `jpeg_c` require 4 options to pass the 95% threshold; `tiff2bw` and `bzip2e` require 3; `sha`, `tiffmedian`, and `susan_c` require 2. For `tiff2bw`, `bzip2e` and `susan_c`, a combination of any two options does not improve much upon one option. For these programs, only a specific combination of three options can bring noticeable improvement over one option. In summary, although one option is enough for

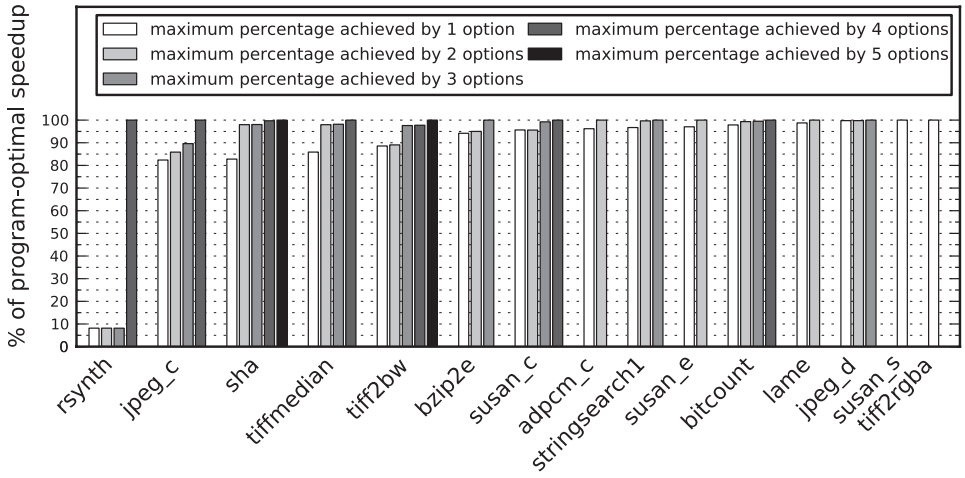

Fig. 14. The maximum percentage of program-optimal speedup achieved by subsets of the options.

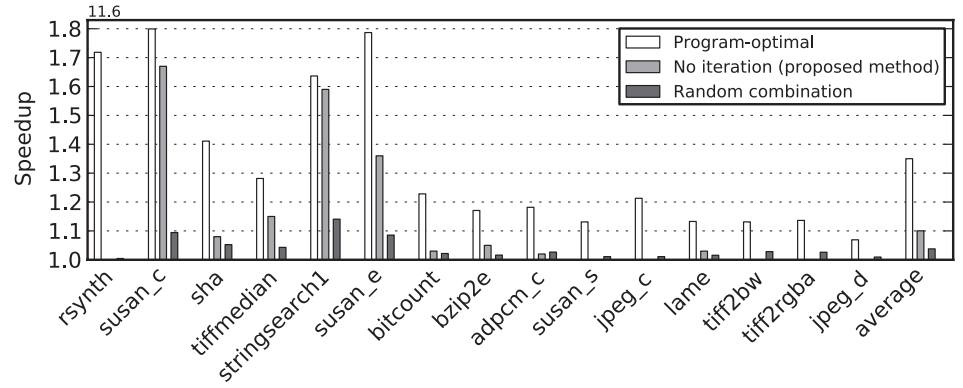

Fig. 15. The speedups achieved by a strategy deduced from our analysis observations without any exploration.

most programs, there are still several programs that require a combination of several options in order to achieve near program-optimal speedup.

Based on the observation that one option is enough for most programs, and the observation of Section 5.1.2 that about 15 options bring most of the performance contribution, we can deduce a simple strategy that would bring a significant share of the benefits of iterative optimization without any combination exploration (iteration) at all, for the cases where the exploration time is a vital criterion. The strategy is that, for a new program, we use the combination that is formed out of all the useful options of the program-optimal combinations seen so far. We can expect from the above reasoning that at least some of the useful options for the new program are in such a combination.

Figure 15 reports the results of applying the method to the 15 programs we consider in this section. We use leave-one-out cross validation in the experiments, which means that for each program, we form the combination based on the data of the other 14 programs. We only use the options that have significant speedup contributions ($>1.05\times$). The bars labeled "Program-optimal" show the best achievable speedups; the bars labeled "No iteration" show the speedups achieved by the our proposed combination; and the bars labeled "Random" report speedup achieved by just using pure random

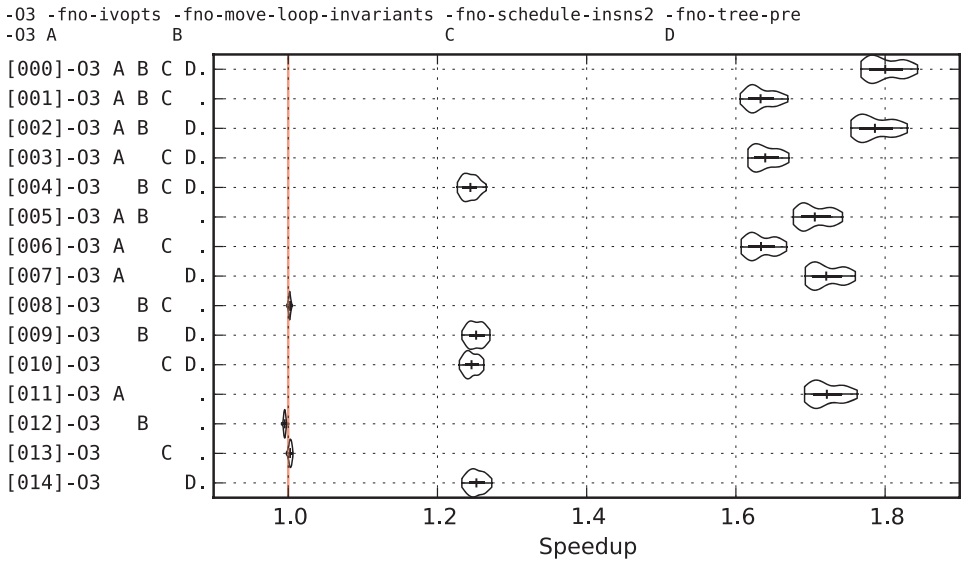

Fig. 16. An example: the interactions of the options for `susan_c`.

combinations. As shown, the proposed method achieves an average speedup of 1.1×. It outperforms the 1.03 average speedup of a random combination.

5.1.4. Nontrivial Interactions among Options in Combinations. After pruning, all the remaining options within a combination have a positive performance effect. However, there are still non-trivial interactions among some of them. As an example, Figure 16 shows the interactions of the four options in the program-optimal combination for `susan_c`. Along the vertical axis, we show the performance for all possible combinations of the four options. The four options, that is, `-fno-ivopts`, `-fno-move-loop-invariants`, `-fno-schedule-insns2` and `-fno-tree-pre` are renamed to be A, B, C, D, respectively, for space reasons. For each combination, we use a violin plot to show the distribution of the speedups it achieves over `-O3` across 50 datasets. Combinations 0 to 4 in Figure 16 show that adding any one of the four options to the remaining three to form the program-optimal combination (`-O3 A B C D`) results in performance improvement. However, by comparing combinations 11 (`-O3 A`) and 7 (`-O3 A D`), we observe that, without BC, adding option D to A decreases performance slightly, although D alone is able to achieve a speedup of more than 1.2 over the baseline as shown by combination 14. Similarly, combinations 11 and 1 (`-O3 A B C`) show that, without D, adding BC to A also decreases performance, although B, C alone or together seem to exhibit no significant effect.

To show a global view of such interactions over all the programs, we evaluate all possible interactions between x option(s) and y option(s), and report the results in Figure 17. For each $x+y$, we draw a violin plot to show the distribution of the speedups or slowdowns over all possible instances of adding y option(s) to x. For example, adding options `B,C` to `-O3 A` is an instance of adding 2 options to 1. The total number of instances in each violin is shown in parentheses below the corresponding label. The speedups or slowdowns shown in these violin plots are relative to `-O3 + x (options)`. Thus, a value greater than 1 means that adding y options over x options improves performance, while a value smaller than 1 means a decrease. Figure 17 shows that the options in the program-optimal combinations interact in different ways. Their combined effects range from slowdowns to high speedups. We also observe that, after

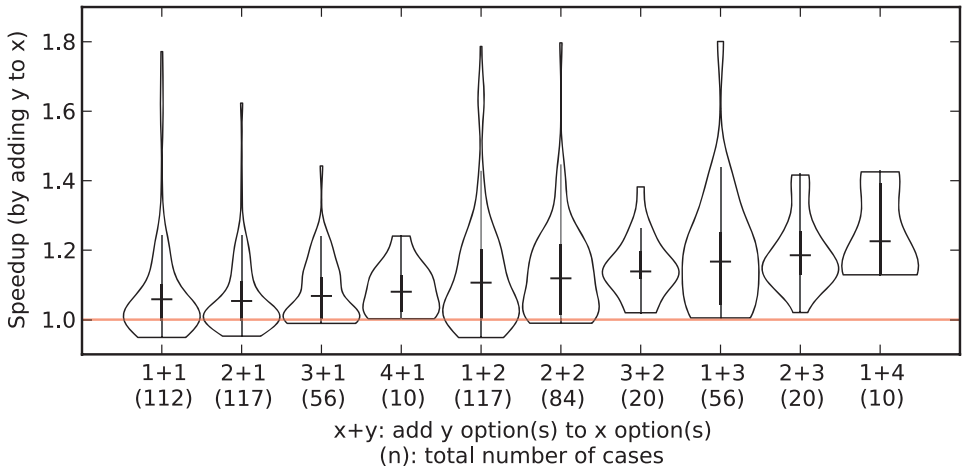

Fig. 17. Quantifying how compiler options interact. Each violin shows the distribution of the speedups or slowdowns over all possible instances of adding y option(s) to x option(s) across all programs. For example, adding options `B,C` to `-O3 A` is an instance of adding 2 options to 1. The total number of instances in each violin is shown in parentheses below the corresponding label. The speedups or slowdowns shown in these violin plots are relative to `-O3 + x (options)`. Thus, a value greater than 1 means that adding y options over x options improves performance, while a value smaller than 1 means a decrease.

pruning, increasing the number of options consistently increases overall performance, up to 5 options, which confirms the same trend shown in Figure 14.

5.2. Illustrating How Iterative Optimization Can Unveil Complex Interactions Among Compiler Optimization Options: Two Case Studies

It is clear from the previous sections that iterative optimization yields combination comprising compiler options that interact in complex ways. We now analyze two of these program-optimal combinations in more detail.

In Section 5.2.1, we analyze the program-optimal combination for `bzip2e`. In this case study, we show that the interactions of compiler options can be complex and counterintuitive. In Section 5.2.2, we analyze the program-optimal combination for `tiff2rgba`. This case study shows an example in which the code produced by iterative optimization interacts with the underlying hardware in a way that is hard to understand without knowing the hardware implementation details. Moreover, these hardware details are hard to describe in the hardware model of a compiler to guide its operations. These two case studies suggest that iterative optimization is an irreplaceable and important compiler optimization strategy.

5.2.1. Bzip2e: Counterintuitive Optimizations And Dataset Sensitivity. The program `bzip2e` is a widely used compression utility. As shown in Figure 18, the program-optimal combination for `bzip2e` has three options `-fno-ivopts -fno-move-loop-invariants -funroll-all-loops`. The three options together achieve an average speedup of 1.17 across all datasets. We observe that leaving any of them out degrades performance on only some of the datasets, not all. The reason is that, as shown in Figure 19, the program has two hot loops, `loop a` and `loop b`. Some datasets spend more time in `loop a`, while others spend more time in `loop b`. The first two options mainly improve `loop a`. The last option mainly improves `loop b`.

-fno-ivopts -fno-move-loop-invariants on loop a. As explained in Table VI, the two options disable optimizations on loop induction variables and loop invariants. The source code of `loop a` in Figure 19 shows that the operations on induction variables, `i1`

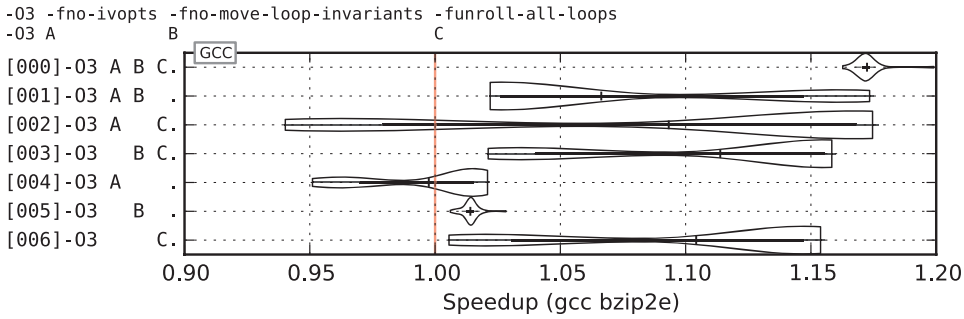

Fig. 18. The interactions among the options in the program-optiaml combination for `bzip2e`.

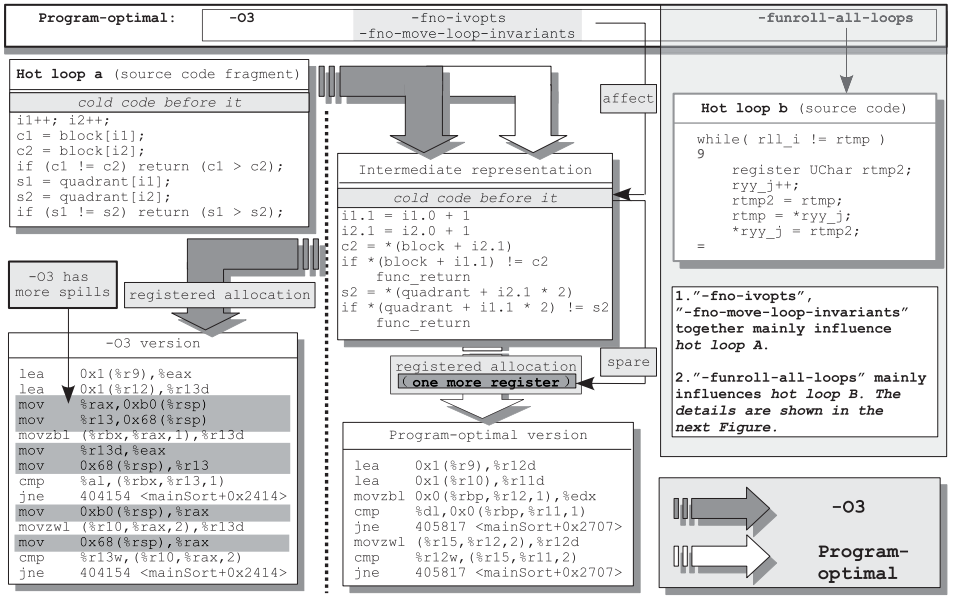

Fig. 19. Optimizations on `loop a` of bzip2e.

and `i2`, are simple enough and there is no invariant that can be moved outside the loop. Thus, turning the optimizations off has no immediate impact on the hot loop. Instead, some cold loops before the hot loop are less optimized, which causes the heuristics of the register allocation pass to spare one more register for `loop a`. As shown in Figure 19, this results in fewer instructions for moving register contents to and from memory, which improves performance.

-funroll-all-loops on loop b. As explained in Table VI, this option instructs the compiler to unroll loops that are otherwise not considered for unrolling because their loop bounds are unknown at compile time. `loop b` is such a `while` loop, as shown in Figure 20. Its body is copied 8 times in the optimization pass `loop2_unroll` enabled by this option. In the subsequent pass `web`, which is also enabled by this option, the variables (or pseudo registers) that are reused several times among the copies of the original loop body are given a new name in each copy. Due to this renaming, the copy

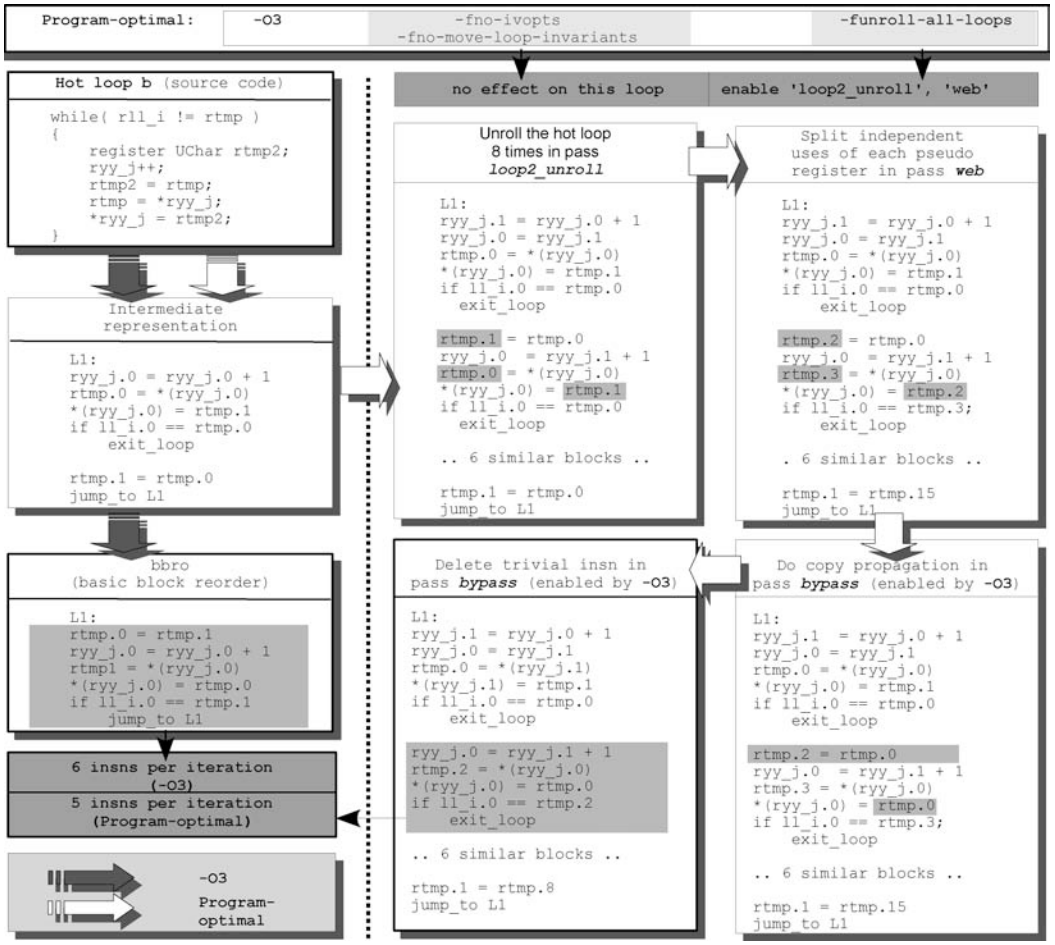

Fig. 20. Optimizations on `loop b` in bzip2e.

propagation optimization, employed by a later pass `bypass`,[6] makes one instruction redundant in each copy of the loop body. These instructions are then detected and deleted in that pass, resulting in performance improvement. The details are shown in Figure 20.

The case for `loop a` shows that the compiler options in program-optimal combinations sometimes interact in an counterintuitive way. The case for `loop b` is more intuitive but still complex. The optimizations for the two loops must work together to cover all the datasets.

5.2.2. Tiff2rgba: Hitting the Micro-architecture Wall.. The program `tiff2rgba` converts the color model of a TIFF image into the RGB model. As shown in Table V, the program-optimal combination has only one option `-fno-omit-frame-pointer`. It achieves an average speedup of 1.14. The speedup mainly comes from the runtime reduction of the only hot function in this program `LZWDecode`.

[6] `bypass is enabled by -O3.`

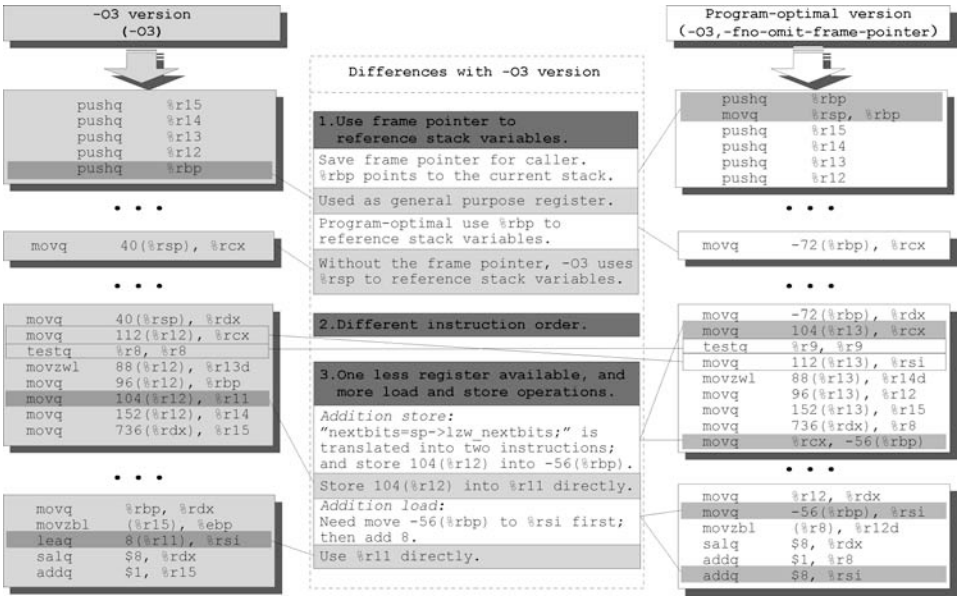

Fig. 21. The differences between code produced by `-O3` and the program-optimal combination for `tiff2rgba`.

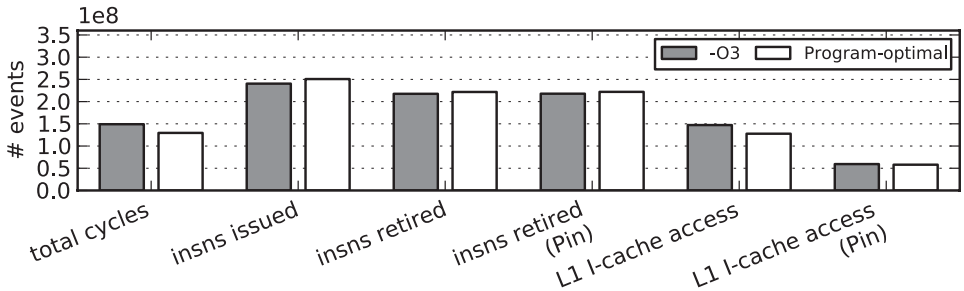

Fig. 22. The performance events reported by hardware counters and Pin for `tiff2rgba`.

-fno-omit-frame-pointer results in more instructions. As explained in Table VI, `-fno-omit-frame-pointer` instructs the compiler to reserve a dedicated register, for each function, pointing to the starting address of its frame (activation record) on the stack. The local variables of a function are stored in the frame, and can thus be referenced by adding an offset to the frame pointer. However, the default behavior is to omit such a frame pointer for functions that do not need it. In these functions, the stack register is never changed before the return instruction. As a result, it can be used to reference local variables, which eliminates the need for a dedicated frame pointer register. As shown in Figure 21, this omission not only avoids the instructions to save, set up and restore the frame pointer, but also makes an extra register available which leads to fewer spills. In short, the option in the program-optimal combination disables this frame pointer optimization, resulting in more instructions.

More instructions but shorter execution time? To understand why, we examine more than 400 hardware events supported by performance counters built into the CPU we use. As shown by the bars "insns issued," "insn retired" in Figure 22, the extra static instructions caused by `-fno-omit-frame-pointer` do lead to more dynamic instructions issued and retired (executed) at runtime. This option also leads to a slight increase in

the events related to the load and store instructions and memory accesses. However, counterintuitively, as shown by the bars "total cycles," this option does improve performance. We examine all the events supported by the hardware, and find that the only one that can be used to explain the runtime reduction is the number of L1 instruction cache accesses, as shown in Figure 22. Note that, we only observe a reduction in the number of accesses, not misses.

The instruction fetch algorithm might be the cause. One possible explanation for the reduction in instruction cache accesses is that the extra instructions inadvertently cause a more compact cache layout for the frequently executed instructions. To validate this hypothesis, we use the Pin tool [Luk et al. 2005] to trace the execution of every instruction and count the number of instruction cache accesses. Although the Intel processor we use has 64-byte cache lines, the data path between the instruction cache and the fetch buffer is only 16-byte wide. As a result, one access can at most fetch one fourth of a cache line. We reflect this in the counting. The bars "insns retired (Pin)" in Figure 22 show that Pin reports the same number of dynamic instructions as the hardware performance counters. The bars "L1 I-cache access (Pin)" show that the number of L1 instruction cache accesses reported by Pin is also less than that of `-O3`. However, their difference and the absolute access numbers reported by Pin are much less than those reported by the hardware counters. The reason is that the exact implementation details of the instruction fetch algorithm for the processor are unknown; and the accounting as described above does not capture the complex fetching mechanism accurately. The Intel Developer's Manual [Intel 2011] suggests that to decode an instruction the processor may fetch the same cache line multiple times. The manual also states that even for instructions that are fully contained in one cacheline, the processor may still need to fetch additional cachelines. We ran the same binaries on an open source cycle-accurate x64 simulator PTLsim [Yourst 2007] to obtain deeper understanding of this issue. However, the runtime reduction is not observed in this situation. We also ran the same binaries on an AMD processor which supports the same ISA. The performance improvement is also not observed. All of these experiments suggest that the performance difference is really caused by subtle implementation details of the Intel processor at the micro-architecture level.

6. DISCUSSION ON SCOPE OF THE RESULTS

The results in this article are obviously tied to the experimental setup. In this section, we discuss how general the results and conclusions are.

6.1. Compiler Combinations

There is no guarantee that the 300 combinations we explored represent the entire compiler optimization space well. 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 datasets instead of all the 1000 datasets. The dataset optimal speedups across these 8000 combinations are reported in Figure 23. Compared to Figure 2(a), we observe no significant difference overall except for only a few datasets (the average speedup difference is 4.6%), which suggests that the 300 combinations considered in this article capture some of the best possible speedups and combinations.

6.2. Platforms and Benchmarks

More generally, we cannot assert that our conclusions generalize beyond our benchmark suites (MiBench), compiler platforms (Intel ICC and GNU GCC) and target architecture (Intel). There are also obvious examples where the performance of program segments is dataset sensitive; auto-tuned libraries like FFTW [Matteo and Johnson 1998] or

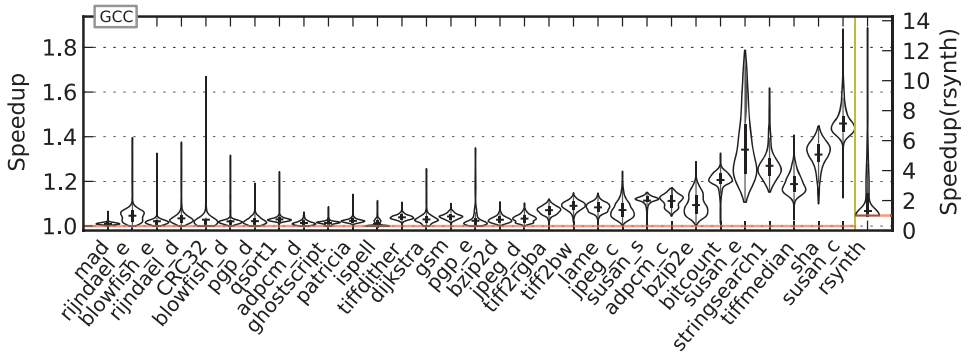

Fig. 23. Dataset optimal speedups relative to -O3 for 8000 combinations.

Table VII. Datasets Descriptions for Three PARSEC Benchmarks

Program	# source lines	Dataset file size	Dataset description
ferret	10946	636K-17.3M	Groups of query images constructed using 1,166,657 JPEG files downloaded from 130 different web sites.
freqmine	2189	5.7M-183M	Lists of transactions randomly generated using the IBM Quest Market-Basket Synthetic Data Generator [Agrawal and Srikant 1994].
x264	42070	2.7M-165M	Yuv4mepg videos converted from MKV and RMVB files downloaded from the Internet.

ATLAS [Whaley et al. 2001] are examples where dataset dependent tuning is useful. Nevertheless, our results suggest that, in general, the dataset sensitivity problem may have been overstated. We note that our results are consistent across all benchmarks so far.

In addition, we collected 1000 datasets (see Table VII) for 3 more programs from the PARSEC benchmark suite [Bienia et al. 2008]: `ferret`, `freqmine`, `x264`. These programs represent three emerging workloads: content similarity search, frequent itemset mining, and H.264 video encoding. We evaluated the three programs on three hardware platforms: Intel, AMD and Loongson, respectively. The Intel platform has already been introduced. The AMD platform contains the 2.6 GHz AMD Opteron processor (8218), 16 GB RAM, 2×1 MB L2 cache, running Red Hat Enterprise Linux 4.4. The Loongson platform contains the 800 MHz Loongson 2F, 512 MB RAM, 512 KB L2 cache, Redflag Linux 6.0. Loongson 2F [Loongson] is a MIPS-compatible general purpose CPU. The compiler used is GCC. As shown in Figure 24, our conclusions are confirmed again: for each program, a single combination can achieve at least 80% of the dataset optimal speedup for all datasets, with most programs standing at 90% or higher.

6.3. Measurement Bias

Recent work by Mytkowicz et al. [2009] 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 Mytkowicz et al. [2009] by a small margin only (within ±2% using ICC). In our study, 80% of program/dataset pairs have a speedup that is higher than 1.02, and are thus relatively immune to this source of measurement bias.

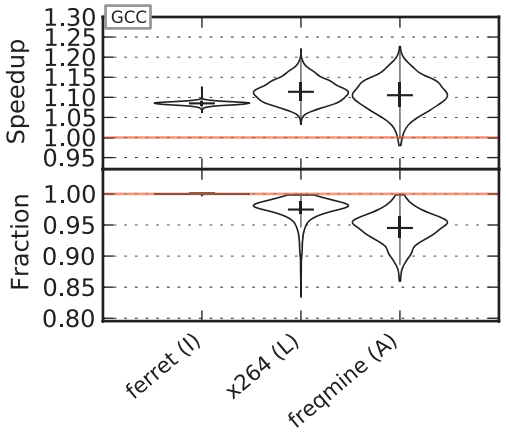

Fig. 24. Speedup (top graph) and fraction of the dataset optimal speedup (bottom graph) for the program-optimal combinations for PARSEC programs on GCC. These programs are evaluated across three platforms as indicated by the markers after their names: (I) for Intel, (A) for AMD, (L) for Loongson.

7. RELATED WORK

We already mentioned in the introduction that several studies have investigated the impact of datasets on program behavior and performance. Some of these have even looked at how datasets affect program optimization decisions. Most of these works remain limited by the restricted number of datasets available in existing benchmark suites. The SPEC CPU2000 suite contains 3 input sets per benchmark, and most benchmarks have less than 10 datasets in the SPEC CPU2006. The embedded EEMBC benchmark suite [EEMBC] also contains less than 10 datasets for most benchmarks. The recently introduced parallel PARSEC benchmark suite [Bienia et al. 2008] contains 6 datasets for each benchmark. A large number of datasets is not only useful for compiler research and workload characterization research [Jiang et al. 2010] many architecture studies rely on profile-based optimization techniques as well [Magklis et al. 2003; Sankaranarayanan and Skadron 2004], and may benefit from having more datasets in order to study dataset sensitivity.

Several studies have investigated redundancy among datasets and how to find a small set of representative programs and inputs for architecture research [Eeckhout et al. 2003]. We use several of the proposed statistical techniques and feature characterization approaches in this article to characterize KDataSets.

The analysis results we report in Section 5 are consistent with the observations made by several previous studies. Hoste and Eeckhout [2008] use iterative optimization to do multi-objective search for optimization combinations that have better performance and/or faster compilation speed than the default optimization levels of the GCC compiler. They find that only about a third of all the optimizations enabled by `-O3` appear in the combinations they found. We make similar observations in Section 5.1.1 and 5.1.2. Several studies [Cavazos et al. 2007; Agakov et al. 2006] report that simply exploring the optimization space with random compiler option combinations can already achieve a significant amount of speedup that can be achieved by more sophisticated heuristics, genetic algorithms, or machine learning approaches. Our findings in Section 5 also suggest that this approach can be quite effective.

Some researchers propose to use compiler writer's knowledge [Triantafyllis et al. 2003] or involve the application developer [Kulkarni et al. 2003] in the optimization process to speed it up. Our analyses of the program-optimal combinations can inform programmers and compiler writers which optimizations are more likely to have a

significant performance impact. While analyzing the causes of measurement bias due to different link orders and UNIX environment sizes, Mytkowicz et al. [2009] also hit the micro-architecture wall as we did in Section 5.2.2. They call for more information and cooperation from the hardware manufactures, which we agree with. However, as shown in Section 5.2.2, iterative optimization can adapt a program to a specific hardware implementation even without the knowledge of its details.

8. CONCLUSIONS

In this article, we deconstructed iterative optimization by first composing KDataSets, a collection of 1000 datasets for 32 MiBench programs, to evaluate a fundamental issue in iterative optimization: whether it is possible to learn the best possible compiler optimizations across distinct datasets. We conclude that the issue seems significantly simpler than previously anticipated, with the ability to find a program-optimal combination of compiler optimizations across datasets. Then, we evaluate the idea of introducing compiler choice as part of iterative optimization. We find that it further improves the performance of iterative optimization as different programs favor different compilers. Moreover, we conduct an in-depth analysis of the program-optimal combinations, and we find that few options have a significant impact, and that most combinations require only a few options. Still, we also exhibit several cases where the interplay between the different options of a combination are fairly complex, which underscores that iterative optimzation is an irreplaceable and important compiler optimization strategy.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback.

REFERENCES

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

Agrawal, R. and Srikant, R. 1994. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases.* 487–499.

Berube, P. and Amaral, J. 2006. Aestimo: A feedback-directed optimization evaluation tool. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. 251–260.

Bienia, C., Kumar, S., Singh, J. P., and Li, K. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. 72–81.

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

cBench. cBench: Collective Benchmarks. http://www.cTuning.org/tools.

Chen, Y., Huang, Y., Eeckhout, L., Fursin, G., Peng, L., Temam, O., and Wu, C. 2010. Evaluating iterative optimization across 1000 datasets. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, 448–459.

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*. 1–9.

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

Eeckhout, L., Vandierendonck, H., and De Bosschere, K. 2003. Quantifying the impact of input data sets on program behavior and its applications. *J. Instruct.-Level Parallel. 5*, 1–33.

EEMBC. EEMBC: The Embedded Microprocessor Benchmark Consortium. http://www.eembc.org.

FRANKE, B., O'BOYLE, M., THOMSON, J., AND FURSIN, G. 2005. Probabilistic source-level optimisation of embedded programs. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. 78–86.

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*. 245–260.

FURSIN, G. AND TEMAM, O. 2010. Collective optimization: A practical collaborative approach. *ACM Trans. Archit. Code Optim. 7*, 20:1–20:29.

GUTHAUS, M., RINGENBERG, J., ERNST, D., AUSTIN, T., MUDGE, T., AND BROWN, R. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual International Workshop on Workload Characterization*. 3–14.

HANEDA, M., KNIJNENBURG, P., AND WIJSHOFF, H. 2006. On the impact of data input sets on statistical compiler tuning. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*.

HOSTE, K. AND EECKHOUT, L. 2006. Comparing benchmarks using key microarchitecture-independent characteristics. In *Proceedings of the IEEE International Symposium on Workload Characterization*. 83–92.

HOSTE, K. AND EECKHOUT, L. 2008. COLE: Compiler optimization level exploration. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 165–174.

HSU, W. C., CHEN, H., YEW, P. C., AND CHEN, D.-Y. 2002. On the predictability of program behavior using different input data sets. In *Proceedings of the 6th Annual Workshop on Interaction between Compilers and Computer Architectures*. 45–53.

INTEL. 2011. *Intel 64 and IA-32 Architectures Software Developers Manual*. Vol. 3.

JIANG, Y., ZHANG, E. Z., TIAN, K., MAO, F., GETHERS, M., SHEN, X., AND GAO, Y. 2010. Exploiting statistical correlations for proactive prediction of program behaviors. In *Proceedings of the International Symposium on Code Generation and Optimization*.

KULKARNI, P., HINES, S., HISER, J., WHALLEY, D., DAVIDSON, J., AND JONES, D. 2004. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 171–182.

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*. 12–23.

Loongson. LOONGSON. Loongson 2f. http://www.loongson.cn/EN/product info.php?id=34.

LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI '05)*. ACM, New York, NY, 190–200.

MAGKLIS, G., SCOTT, M. L., SEMERARO, G., ALBONESI, D. H., AND DROPSHO, S. 2003. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*. 14–27.

MAO, F., ZHANG, E. Z., AND SHEN, X. 2009. Influence of program inputs on the selection of garbage collectors. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 91–100.

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. 1381–1384.

MYTKOWICZ, T., DIWAN, A., HAUSWIRTH, M., AND SWEENEY, P. F. 2009. Producing wrong data without doing anything obviously wrong! In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. 265–276.

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. 2006a. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization*. 319–332.

PAN, Z. AND EIGENMANN, R. 2006b. Fast automatic procedure-level performance tuning. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*. IEEE.

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

SANKARANARAYANAN, K. AND SKADRON, K. 2004. Profile-based adaptation for cache decay. *ACM Trans. Archit. Code Optim. 1*, 305–322.

SLINGERLAND, N. AND SMITH, A. J. 2001. Performance analysis of instruction set architecture extensions for multimedia. In *Proceedings of the 3rd Workshop on Media and Stream Processor*.

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*. 77–90.

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*. 204–215.

Whaley, R. C., Petitet, A., and Dongarra, J. 2001. Automated empirical optimization of software and the ATLAS project. *Parallel Comput 27*.

Yourst, M. 2007. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proceedings of the IEEE International Symmposium on Performance Analysis of Systems and Software*. 23–34.

Zhong, Y., Shen, X., and Ding, C. 2009. Program locality analysis using reuse distance. *ACM Trans. Program. Lang. Syst. 31*, 6, 1–39.

Received September 2011; revised March 2012; accepted June 2012