ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Compiler Flag Optimization using Fourier-sparse Set Functions

Master Thesis

Manuel Nowack

September 4, 2022

Advisors: Prof. Dr. Markus Püschel, Chris Wendler
Department of Computer Science, ETH Zürich

Abstract

Automatic compiler optimizations play an important part in the software development process because they often provide a notable performance improvement at almost no cost. However, the standard optimization setting provided by compilers is often suboptimal for a specific program. We demonstrate that Fourier-sparse set function learning can reliably find a selection of GCC compiler flags with better performance than both a random search and the highest standard optimization setting. Then, we reaffirm that considering interactions between three or more separate optimizations is important by comparing the general Fourier-sparse set function learning implementation to its degree-two variant. Finally, we verify in synthetic experiments that Fourier-sparse set function learning can almost perfectly learn the best selection of compiler flags if the target function is well approximated by a Fourier-sparse set function even when confronted with a very large search space, noisy runtime measurements, and training data that is randomly sampled rather than iteratively selected during the learning process.

Contents

Chapter 1

Introduction

Compilers usually feature a variety of optimizations to improve performance at the expense of longer compilation time. For many optimizations the cost is small enough so that most developers heavily rely on optimizations provided by the compiler. Modern compilers, such as GCC or LLVM, implement optimizations as multiple independent heuristics that transform a program into a semantically equivalent program. Most of these optimizations can be enabled or disabled by command line flags. For example, when passing the flag `-finline-functions` to GCC, all functions are considered for inlining, even if they are not declared inline. Therefore, throughout this thesis we use the term compiler flag and optimization interchangeably. Since most of the optimizations are independent, multiple flags can be passed at the same time. We call the combination of all optimizations used to compile a program an *optimization setting*. Naturally, we want to apply to each program the optimization setting that yields the fastest runtime. However, even the most advanced compilers frequently fail to deliver the best optimization settings for individual applications [6, 13, 3, 10, 8]. Finding a suitable optimization setting for a given program is a very hard problem for several reasons.

- Not all attempted optimizations are beneficial for all programs. Sometimes an optimization transformation can even be detrimental to performance [14, 7].
- Modern compilers offer hundreds of optimizations, e.g. GCC has over 250 optimization flags. Since most of the optimizations are independent, the number of possible combinations thereof grows exponentially. Even simply assuming binary flags (e.g. enabled/disabled), there will be 2^{250} optimization settings which are almost the number of atoms in the universe.
- The best optimization setting for one program may not be optimal – or even close to optimal – for another program [6, 3]. Even for the same

program the same optimization setting can have a wildly varying effect on different machines because the effectiveness of each optimization depends on the underlying hardware too.

- The optimization space contains numerous irregularly distributed local optima [11] and due to its exponential size traversing a significant fraction of it is practically impossible.

Therefore, compilers usually offer a default optimization setting that is expected to work well enough for a typical program. For example, the highest standard optimization level in GCC is `-O3`. However, this default optimization setting often underperforms for a specific program.

The process of finding a suitable optimization setting for a program is referred to as compiler autotuning [2]. Autotuning consists of three main problems:

1. what optimizations to use
2. which set of parameters to choose from (e.g., loop tiling size, etc.)
3. in which order to apply the optimizations

Some compilers such as GCC do not allow the user to chose the order in which optimizations are applied, while others such as LLVM do.

1.1 Related work

Due to the huge numbers of possible optimizations and the different behaviour on different target platforms, it is challenging to determine why a given optimization setting performs better than another optimization setting. This makes it difficult to predict the performance impact of each optimization as well as companion optimizations that it would have synergy with. To handle this challenge, prior tuning methods often ignore or oversimplify the estimation of an optimization's impact and its relationship with other optimizations [6, 13]. In reality, optimization decisions are deeply related, so without proper consideration of their relationships, these approaches are ignoring information that could be used to better guide the tuning process and improve overall quality.

SRTuner [12] is a tuning method that specifically addresses this challenge and focuses on accurate optimization impact estimation and inter-optimization relationships as a means to navigate a compiler's optimization space.

A recent work [9] has taken a different angle and models autotuning by minimizing a Fourier-sparse poset function. The space of all optimization settings is modeled as a poset and there is a poset function that maps each poset element (=optimization setting) to its runtime. Then, we can derive a Fourier transform for this domain and learn a sparse approximation of the function,

i.e., most of the Fourier coefficients will be zero. Finally, we minimize the learned Fourier-sparse approximation and return the optimization setting with the smallest runtime. This model can in principle capture arbitrarily complex inter-optimization relationships.

1.2 Our Work

Unfortunately, the previous effort to solve autotuning by Fourier-sparse poset function learning did not succeed when evaluated on LLVM. The minimum of the learned function did not outperform the best optimization setting from the training data. Based on that result, this thesis focuses on applying more narrow Fourier-sparse poset function learning methods to autotuning and seeks to provide explanations why the broader models fail.

While working on these questions, we encountered a significant obstacle in the duration of evaluating the various tuning methods because obtaining accurate runtime measurements takes a sufficient number of repetitions. Furthermore, we need multiple repetitions after any change in the tuning methods to be confident we did not merely encounter a lucky or unlucky run. Therefore, waiting for the tuning methods to terminate became a bottleneck during development. As a consequence, we decided to first build a parallel benchmarking infrastructure that produces reliable measurements even when many different benchmarks are concurrently running on the target machine.

Then, we demonstrate that when disregarding the order of optimizations and therefore reducing the problem to Fourier-sparse set function learning, we can reliably find a selection of GCC compiler flags with better performance than both a random search and the highest standard optimization setting. Our implementation is based on code from [15] for fast enumeration of set elements. However, we find it is necessary to allow to iteratively choose the training data, rather than using only randomly sampled optimization settings as the training data. Otherwise, the minimum of the learned function still does not outperform the best optimization setting from the randomly sampled training data.

Furthermore, we reaffirm that considering interactions between three or more flags is important by comparing the general Fourier-sparse set function learning implementation to its degree-two variant. We show that our Fourier-sparse set function learning implementation can keep up with SRTuner, another tuning method that focuses on accurate optimization impact estimation and inter-optimization relationships, and with BOCS, a low-degree implementation of Fourier-sparse set function learning featuring a more sophisticated acquisition function.

Finally, we were able to verify in synthetic experiments that Fourier-sparse set function learning can almost perfectly learn the best optimization setting if

the target function is well approximated by a Fourier-sparse set function even when confronted with a very large search space, noisy runtime measurements, and training data that is randomly sampled rather than iteratively selected during the learning process.

Chapter 2

Measurement Setup

2.1 Collective Knowledge

A major problem of practical research in computer science is the lack of opportunities to reproduce empirical results and compare them with other published techniques. Published research rarely includes the full experiment specification required to be able to reproduce the results. In fact, it is not unheard of to come across a research paper and find no trace of its source code.

Collective Knowledge (CK) is an open-source framework and methodology that aims to solve these problems [5]. The CK concept is to decompose research projects into reusable components that encapsulate research artifacts and provide unified application programming interfaces (APIs), command-line interfaces (CLIs), meta descriptions and common automation actions for related artifacts. The goal is to automate AI, ML and System research, accelerate innovation, and simplify its adoption in the real world. CK has been used in production for more than 5 years, including companies such as Arm and General Motors.

CK is a continuation of the cTuning Foundation which was developed with the intention to accelerate the very time-consuming autotuning process and help our compiler to learn the most efficient optimizations across real programs, datasets, platforms, and environments. Autotuning remains a main use case for CK. To this end, it exposes a wide array of benchmark programs and datasets and utilities to compile, run, and benchmark programs through both a Python and command-line interface.

2.2 Parallel Measurements

While CK is valuable building block of our benchmark infrastructure to obtain accurate runtime measurements, we have found its default configuration of runtime measurement insufficient for our purposes.

The previous effort to approach autotuning by Fourier-sparse poset function learning assumed offline training data, i.e., the tuning method receives randomly sampled pairs of optimization settings and runtime measurements and learns from this data. This assumption facilitates sequentially sampling training data in one sweep and storing it so that the tuning method can be rerun after incremental changes without sampling new training data. However, the tuning methods we work with assume online training data, i.e., the tuning method can iteratively query the runtime of its chosen optimization setting and make use of all previous queries when choosing the next optimization setting.

Since there are very limited options to speed up runtime measurements, especially not without sacrificing accuracy (the runtime of an optimization setting is the aggregate over many repetitions as the runtime of a single run is inherently volatile), the obvious improvement is parallelization. Autotuning different programs is embarrassingly parallelizable because tuning each program is an isolated problem. However, running benchmarks in a parallel environment introduces challenges not taken into account by CK.

2.2.1 Problems of parallelizing CK queries

First, let us define the implementation of a query in CK. A query returns the runtime of a program when compiled with a given optimization setting. Each query is the aggregate over several repetitions of running the compiled program. The number of repetitions is dynamically determined by an exponential search so that the overall execution time matches a fixed value (4 seconds by default in CK).

When running multiple programs in parallel, they may compete over the available cores and even if the number of cores exceeds the number of programs, one program may evict cached data used by another program.

To assess the reliability of runtime measurements we make repeated queries of the same optimization setting for many different programs in parallel and analyze the result.

One distinct pattern of highly variant measurements is that for the final queries the reported runtime becomes significantly smaller, as illustrated in Figure 2.1. The issue here is that towards the end some threads have already finished all their queries and no longer compete over system resources with other programs. The reason some threads are able to finish earlier is due to

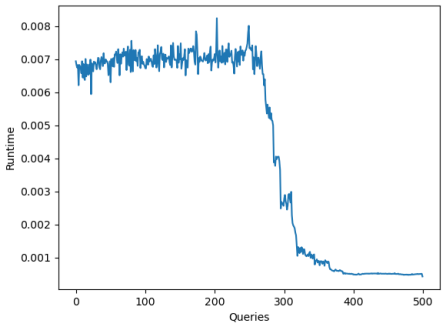

Figure 2.1: 500 consecutive queries of the CK program consumer-tiff2bw:image-tiff-0005

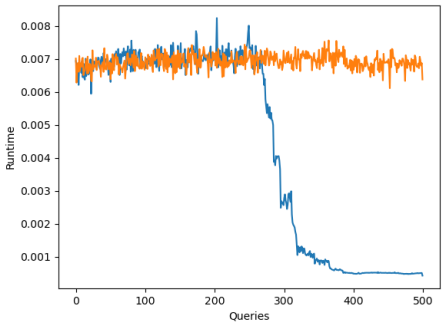

Figure 2.2: 500 consecutive queries of the CK program consumer-tiff2bw:image-tiff-0005; The orange queries are the result of our improvement

different compilation times, preprocessing overhead, and minor divergences from the targeted overall execution time of 4 seconds (programs with a high runtime and therefore a small number of repetitions may not be able to match a specific overall execution time exactly), which accumulate over the course of hundreds of queries. Therefore, programs particularly prone to interference are suddenly able to run much faster during their final queries. Measurement noise in this order of magnitude impedes any earnest attempt at learning from the training data.

A fairly effective solution for this problem is for each thread to continue running unmonitored queries after it finished its computations until all threads have finished their computations. The result is shown in Figure 2.2. It should be noted that Figure 2.2 is an extreme case of susceptibility to interference and we ended up excluding this particular program from the final evaluations because even in a single-threaded environment it did not produce sufficiently reproducible measurements.

Another distinct pattern of variant measurements can be seen in Figure 2.3.

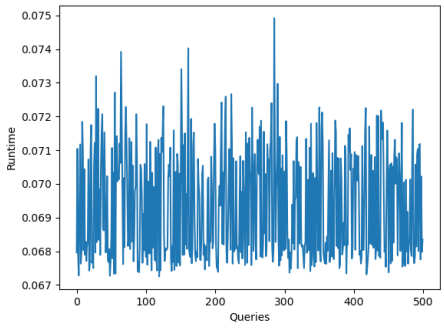

Figure 2.3: 500 consecutive queries of the CK program automotive-bitcount:number-0001:

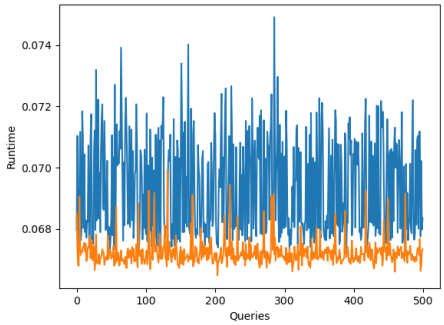

Figure 2.4: 500 consecutive queries of the CK program automotive-bitcount:number-0001:; The orange queries are the result of our improvement

There is a clear lower bound the programs do not exceed, but some queries report a notably higher runtime. Our solution to this phenomenon is not to make one query that targets an overall execution time of 4 seconds, but to make 4 subqueries that target an overall execution time of 1 second and then return the minimum over these subqueries. (In our implementation we ended up making 5 subqueries of 1 seconds.) The result is shown in Figure 2.4.

Combining all our improvements yields highly accurate measurements for most programs even in a parallel environment, as visible in Table 3.1 in the next chapter. However, some programs retained an unfavorably high variance despite our best efforts and this noise does not disappear in a single threaded environment either. Therefore, we did not run any final evaluations on such programs.

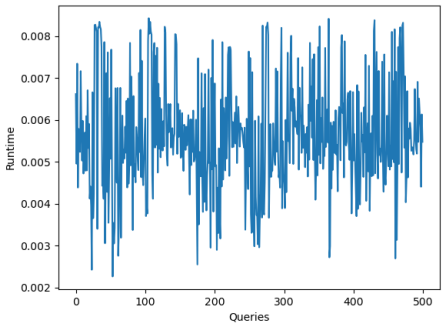

Figure 2.5: 500 consecutive queries of the CK program cbench-consumer-jpeg-c:image-ppm-0005:

2.3 Improving Collective Knowledge

Even though CK has been used in production for more than 5 years, it is still a proof-of-concept prototype requiring further improvements and standardization, as acknowledged by its authors.

In the following we note some inconveniences we came across that we believe deserve some attention.

- There is a forced pause of 0.5 seconds after preprocessing a query but before executing the actual measurement. We fail to see a tangible benefit of this pause, while it prolongs the query duration quite a bit. Therefore, we removed this pause in our evaluation to save time.
- The number of repetitions required to obtain a fixed overall execution time is recomputed for each query. Especially for programs with a high runtime and therefore a small number of repetitions, this preprocessing overhead ends up being not too far away from the subsequent overall execution time. This is a quite annoying if you want to run the same optimization setting multiple times and it becomes a more serious problem in a parallel evaluation because the slowest program determines when the whole evaluation is finished. We cut this time by precomputing the number of repetitions for the baseline optimization setting (`-O3` for GCC) and use this value for all future queries. While this behaviour should not become enabled by default, it would be very useful to have a builtin option to enable such functionality.
- Several benchmark programs fail to compile or cause a segfault on our target machine using GCC 9. Some of these issues date back to the original cBench, but other programs such as `bzip2` used to work on our target machine when using the original cBench.
- Programs such as `office-stringsearch2` that take two input files do not allow to specify the second input file, which renders the Python

interface defunct. The command line interface appears to hardcode the second input file.

- The Python interface takes arguments as one dictionary where each argument is specified as a key-value pair. This is a very strange and cumbersome way of passing arguments when keyword arguments are a widely used Python feature.
- The various CK functions print verbose output to the standard output without offering an opt-out. Usually, this messages are of no concern and especially in the Python interface it congests our own output. Since CK executes benchmark programs by a call to the deprecated `os.system` function instead of the modern `subprocess` standard module, it is not even possible to redirect the standard output prior to calling a CK function. We worked around this issue by not using the standard output at all, but rather write everything to our own log files on disk.

Chapter 3

Results

The source code is available at https://gitlab.ethz.ch/mnowack/compiler-flags. Installation instructions are detailed in the repository's readme file.

All empirical evaluations have been performed on an AMD EPYC 7742 64-core processor using Ubuntu 20.04.4 and GCC 9.4.0. We disabled the dynamic CPU frequency boost and fixed the processor to its base frequency of 2.25 GHz. This tweak is a necessary condition to ensure accurate measurements.

All results in this chapter have been obtain by running the python script `main.py [--tuner TUNER] [--search_space SEARCH_SPACE] [--budget BUDGET] [--rerun RERUN] [--simulation] [--parallel PARALLEL] [modules ...]` at the root directory.

- The `tuner` argument specifies which tuning method(s) are evaluated. This value is always clearly denoted in the ensuing experiments.
- The `search_space` argument points to a text file that contains the compiler flags and their possible values which form the search space of the tuning methods. The text files can be found in the repository inside the directory `gcc_flags`. The largest file features 98 flags which are the same 100 flags SRTuner was evaluated on minus `-fdelayed-branch`, which is not supported by our target machine, and `ftree-parallelize-loops`, which takes numerical values which is not supported by a set function. The smallest file features the 20 flags that had the highest individual impact over our benchmark programs (check out `flag_selection.py` for the exact selection criteria). This value is always clearly denoted in the ensuing experiments.
- The `budget` argument specifies the number of queries the tuning method(s) have available. This value is always clearly denoted in the ensuing experiments.
- The `rerun` argument is a convenience utility that is equivalent to se-

quentially running `main.py` this many times. Passing this argument does not imply any form of aggregation over the individual runs.

- The `simulation` argument instructs to use simulated measurements. This redefines the meaning of the `modules` argument to specify the seed used to initialize the Fourier-sparse set function representing the program runtimes.
- The `parallel` argument specifies the number of threads to use. Consequently, this value is always set to match the number of modules arguments.
- The `modules` argument specifies which CK programs and datasets will be evaluated. The format of a module is `data_uoa:dataset_uoa:cmd_key` which each denote a CK entry. They correspond to the program, dataset, and command key. Since all programs we use for the evaluations have the prefix "cbench-", we trimmed this prefix from all tables in this thesis to save horizontal space. All experiments were run on the same 59 modules which can be seen in e.g. Table 3.1.

Furthermore, all results are the average over 10-32 repetitions, depending on what is feasible considering the duration of one repetition.

The output files are put under version control in the directories `evaluation`, `simulation`, and `stability`. We analyze these output files to obtain the results in this chapter by running the python script `analysis/main.py` with the root directory as the current working directory.

3.1 Accuracy of runtime measurements

First of all, we demonstrate the reliability of our parallel benchmark infrastructure. Every experiment includes a measurement of `-O3`, the highest standard optimization level of GCC, to enable a comparison to a default baseline. Table 3.1 shows the mean, standard deviation, and relative standard deviation for all 255 `-O3` queries made over the course of these experiments.

Table 3.1: Mean, standard deviation, and relative standard deviation of -O3

	μ	σ	σ/μ
automotive-bitcount:number-0001:	0.067200	0.000129	0.001924
automotive-qsort1:cdataset-qsort-0001:	0.019548	0.000072	0.003702
automotive-qsort1:cdataset-qsort-0005:	0.012260	0.000040	0.003303
automotive-qsort1:cdataset-qsort-0009:	0.017622	0.000053	0.002995
automotive-qsort1:cdataset-qsort-0013:	0.014720	0.000039	0.002639
automotive-qsort1:cdataset-qsort-0017:	0.014128	0.000052	0.003672
automotive-qsort1:cdataset-qsort-0020:	0.022195	0.000070	0.003168
automotive-susan:image-pgm-0001:corners	0.006451	0.000074	0.011502
automotive-susan:image-pgm-0005:corners	0.014077	0.000038	0.002685
automotive-susan:image-pgm-0009:corners	0.028631	0.000075	0.002619
automotive-susan:image-pgm-0013:corners	0.002028	0.000004	0.001991
automotive-susan:image-pgm-0017:corners	0.006526	0.000007	0.001064
automotive-susan:image-pgm-0020:corners	0.009907	0.000009	0.000929
automotive-susan:image-pgm-0001:edges	0.016644	0.000011	0.000680
automotive-susan:image-pgm-0005:edges	0.030187	0.000031	0.001043
automotive-susan:image-pgm-0009:edges	0.087775	0.000102	0.001164
automotive-susan:image-pgm-0013:edges	0.009062	0.000010	0.001101
automotive-susan:image-pgm-0017:edges	0.019399	0.000014	0.000745
automotive-susan:image-pgm-0020:edges	0.028464	0.000021	0.000721
automotive-susan:image-pgm-0001:smoothing	0.084407	0.000195	0.002310
automotive-susan:image-pgm-0005:smoothing	0.223750	0.000502	0.002244
automotive-susan:image-pgm-0009:smoothing	0.433747	0.000397	0.000915
automotive-susan:image-pgm-0013:smoothing	0.016277	0.000035	0.002162
automotive-susan:image-pgm-0017:smoothing	0.071849	0.000179	0.002496
automotive-susan:image-pgm-0020:smoothing	0.119156	0.000179	0.001500
network-dijkstra:cdataset-dijkstra-0001:	0.000001	0.000000	0.024939
network-dijkstra:cdataset-dijkstra-0005:	0.012131	0.000013	0.001051
network-dijkstra:cdataset-dijkstra-0009:	0.129342	0.000091	0.000705
network-dijkstra:cdataset-dijkstra-0013:	0.563824	0.000471	0.000836
network-patricia:cdataset-patricia-0001:	0.000065	0.000001	0.013364
network-patricia:cdataset-patricia-0005:	0.000306	0.000002	0.005303
network-patricia:cdataset-patricia-0009:	0.001396	0.000006	0.004400
network-patricia:cdataset-patricia-0013:	0.006769	0.000038	0.005571
network-patricia:cdataset-patricia-0017:	0.034910	0.000147	0.004196
network-patricia:cdataset-patricia-0020:	0.126620	0.000714	0.005638
telecom-adpcm-c:pcm-0001:	0.000659	0.000001	0.001134
telecom-adpcm-c:pcm-0005:	0.005963	0.000005	0.000882
telecom-adpcm-c:pcm-0009:	0.000749	0.000001	0.000816
telecom-adpcm-c:pcm-0013:	0.004840	0.000005	0.000943

Continued on next page

Table 3.1: Mean, standard deviation, and relative standard deviation of `-O3`

	μ	σ	σ/μ
telecom-adpcm-c:pcm-0017:	0.004995	0.000004	0.000808
telecom-adpcm-c:pcm-0020:	0.007744	0.000006	0.000791
telecom-adpcm-d:adpcm-0001:	0.000449	0.000000	0.000779
telecom-adpcm-d:adpcm-0005:	0.003601	0.000003	0.000869
telecom-adpcm-d:adpcm-0009:	0.000514	0.000001	0.001153
telecom-adpcm-d:adpcm-0013:	0.003334	0.000004	0.001214
telecom-adpcm-d:adpcm-0017:	0.003348	0.000004	0.001058
telecom-adpcm-d:adpcm-0020:	0.005148	0.000004	0.000836
telecom-crc32:pcm-0001:	0.001065	0.000003	0.002675
telecom-crc32:pcm-0005:	0.008509	0.000028	0.003247
telecom-crc32:pcm-0009:	0.001219	0.000005	0.004490
telecom-crc32:pcm-0013:	0.007875	0.000032	0.004102
telecom-crc32:pcm-0017:	0.007912	0.000036	0.004498
telecom-crc32:pcm-0020:	0.012148	0.000048	0.003956
telecom-gsm:au-0001:	0.011500	0.000040	0.003460
telecom-gsm:au-0005:	0.092707	0.000233	0.002509
telecom-gsm:au-0009:	0.013122	0.000061	0.004631
telecom-gsm:au-0013:	0.080315	0.000195	0.002431
telecom-gsm:au-0017:	0.086517	0.000223	0.002578
telecom-gsm:au-0020:	0.128225	0.000345	0.002692

We also created density plots for all these queries and observed that the measurements roughly adhere to a normal distribution. The plots are available in the `analysis/plots/default_runtime` directory of the repository.

3.2 Offline Fourier-sparse set function learning

Since the previous effort to solve autotuning by (offline) Fourier-sparse poset function learning did not succeed when taking both the selection and order of optimizations into account, our first experiment explores if offline Fourier-sparse set function learning succeeds when applied to GCC which ignores the order of flags. Unfortunately, we find that even in this simpler setting, offline Fourier-sparse set function learning still fails to find a suitable optimization setting both for small and large search spaces. Table 3.2 displays the fraction over 20 repetitions that succeeded in finding an optimization setting that has faster runtime than the best optimization setting from the training data. Interestingly enough, Fourier-sparse set function learning performs slightly better in the larger search space.

Table 3.2: Probability that the minimum of the learned Fourier-sparse set function is better than the best optimization setting from the training data; 500 queries and 20/98 flags in the search space

	20	98
automotive-bitcount:number-0001:	0.05	0.10
automotive-qsort1:cdataset-qsort-0001:	0.00	0.05
automotive-qsort1:cdataset-qsort-0005:	0.00	0.00
automotive-qsort1:cdataset-qsort-0009:	0.00	0.00
automotive-qsort1:cdataset-qsort-0013:	0.00	0.00
automotive-qsort1:cdataset-qsort-0017:	0.00	0.00
automotive-qsort1:cdataset-qsort-0020:	0.00	0.05
automotive-susan:image-pgm-0001:corners	0.00	0.30
automotive-susan:image-pgm-0005:corners	0.00	0.45
automotive-susan:image-pgm-0009:corners	0.00	0.40
automotive-susan:image-pgm-0013:corners	0.00	0.00
automotive-susan:image-pgm-0017:corners	0.00	0.00
automotive-susan:image-pgm-0020:corners	0.00	0.35
automotive-susan:image-pgm-0001:edges	0.05	0.40
automotive-susan:image-pgm-0005:edges	0.15	0.40
automotive-susan:image-pgm-0009:edges	0.00	0.35
automotive-susan:image-pgm-0013:edges	0.45	0.15
automotive-susan:image-pgm-0017:edges	0.25	0.30
automotive-susan:image-pgm-0020:edges	0.00	0.25

Continued on next page

Table 3.2: Probability that the minimum of the learned Fourier-sparse set function is better than the best optimization setting from the training data; 500 queries and 20/98 flags in the search space

	20	98
automotive-susan:image-pgm-0001:smoothing	0.00	0.00
automotive-susan:image-pgm-0005:smoothing	0.00	0.00
automotive-susan:image-pgm-0009:smoothing	0.00	0.00
automotive-susan:image-pgm-0013:smoothing	0.00	0.00
automotive-susan:image-pgm-0017:smoothing	0.00	0.00
automotive-susan:image-pgm-0020:smoothing	0.00	0.00
network-dijkstra:cdataset-dijkstra-0001:	0.35	0.35
network-dijkstra:cdataset-dijkstra-0005:	0.35	0.10
network-dijkstra:cdataset-dijkstra-0009:	0.20	0.15
network-dijkstra:cdataset-dijkstra-0013:	0.20	0.15
network-patricia:cdataset-patricia-0001:	0.00	0.00
network-patricia:cdataset-patricia-0005:	0.00	0.00
network-patricia:cdataset-patricia-0009:	0.05	0.10
network-patricia:cdataset-patricia-0013:	0.00	0.20
network-patricia:cdataset-patricia-0017:	0.00	0.30
network-patricia:cdataset-patricia-0020:	0.00	0.25
telecom-adpcm-c:pcm-0001:	0.00	0.20
telecom-adpcm-c:pcm-0005:	0.00	0.00
telecom-adpcm-c:pcm-0009:	0.00	0.15
telecom-adpcm-c:pcm-0013:	0.00	0.15
telecom-adpcm-c:pcm-0017:	0.00	0.10
telecom-adpcm-c:pcm-0020:	0.00	0.05
telecom-adpcm-d:adpcm-0001:	0.00	0.30
telecom-adpcm-d:adpcm-0005:	0.05	0.35
telecom-adpcm-d:adpcm-0009:	0.00	0.45
telecom-adpcm-d:adpcm-0013:	0.00	0.55
telecom-adpcm-d:adpcm-0017:	0.05	0.60
telecom-adpcm-d:adpcm-0020:	0.00	0.40
telecom-crc32:pcm-0001:	0.00	0.00
telecom-crc32:pcm-0005:	0.00	0.05
telecom-crc32:pcm-0009:	0.00	0.00
telecom-crc32:pcm-0013:	0.00	0.00
telecom-crc32:pcm-0017:	0.00	0.00
telecom-crc32:pcm-0020:	0.00	0.05
telecom-gsm:au-0001:	0.25	0.55
telecom-gsm:au-0005:	0.15	0.40
telecom-gsm:au-0009:	0.15	0.25
telecom-gsm:au-0013:	0.05	0.30

Continued on next page

Table 3.2: Probability that the minimum of the learned Fourier-sparse set function is better than the best optimization setting from the training data; 500 queries and 20/98 flags in the search space

	20	98
telecom-gsm:au-0017:	0.25	0.35
telecom-gsm:au-0020:	0.05	0.30

3.2.1 Validation score

To better understand the failure of offline Fourier-sparse set function learning, we look at the R^2 score of the learned Fourier-sparse set functions on a validation set of 10000 random queries. Furthermore, we use this experiment to verify our chosen strength of the α value (regularization term). As we can see in Table 3.3, $\alpha = 0.01$ is optimal for almost all programs. Therefore, we make this parameter static for all programs. Strangely enough, the validation score is quite high for most programs even though the minimum of the learned Fourier-sparse set function is worse than the minimum of the training data. Another interesting observation can be made in Table 3.4: The quality of the minimum of the learned Fourier-sparse set function and the α (and therefore the validation score) are not strongly related.

The best validation score and speedup in Table 3.3, 3.4 are colored in green.

Table 3.3: R^2 validation score of the learned Fourier-sparse set function for different α; 500 queries and 20 flags in the search space

	1e-1	1e-2	1e-3	1e-4	1e-5
automotive-bitcount:number-0001:	0.62	0.77	0.66	0.52	0.43
automotive-qsort1:cdataset-qsort-0001:	0.76	0.88	0.83	0.74	0.72
automotive-qsort1:cdataset-qsort-0005:	0.76	0.88	0.83	0.76	0.71
automotive-qsort1:cdataset-qsort-0009:	0.76	0.88	0.82	0.73	0.70
automotive-qsort1:cdataset-qsort-0013:	0.76	0.88	0.82	0.74	0.70
automotive-qsort1:cdataset-qsort-0017:	0.76	0.88	0.83	0.74	0.71
automotive-qsort1:cdataset-qsort-0020:	0.76	0.88	0.83	0.74	0.70
automotive-susan:image-pgm-0001:corners	0.75	0.80	0.69	0.53	0.47
automotive-susan:image-pgm-0005:corners	0.81	0.85	0.77	0.67	0.60
automotive-susan:image-pgm-0009:corners	0.81	0.85	0.77	0.66	0.61
automotive-susan:image-pgm-0013:corners	0.26	0.39	-0.07	-0.48	-0.58
automotive-susan:image-pgm-0017:corners	0.65	0.72	0.54	0.34	0.25
automotive-susan:image-pgm-0020:corners	0.70	0.76	0.62	0.41	0.37
automotive-susan:image-pgm-0001:edges	0.85	0.93	0.90	0.85	0.83
automotive-susan:image-pgm-0005:edges	0.84	0.92	0.89	0.82	0.80
automotive-susan:image-pgm-0009:edges	0.78	0.84	0.75	0.60	0.57

Continued on next page

Table 3.3: R^2 validation score of the learned Fourier-sparse set function for different α; 500 queries and 20 flags in the search space

	1e-1	1e-2	1e-3	1e-4	1e-5
automotive-susan:image-pgm-0013:edges	0.89	0.98	0.98	0.96	0.95
automotive-susan:image-pgm-0017:edges	0.87	0.96	0.94	0.91	0.90
automotive-susan:image-pgm-0020:edges	0.85	0.92	0.89	0.83	0.80
automotive-susan:image-pgm-0001:smoothing	0.42	0.47	0.09	-0.21	-0.38
automotive-susan:image-pgm-0005:smoothing	0.42	0.46	0.16	-0.27	-0.36
automotive-susan:image-pgm-0009:smoothing	0.42	0.46	0.13	-0.26	-0.36
automotive-susan:image-pgm-0013:smoothing	0.42	0.46	0.09	-0.24	-0.38
automotive-susan:image-pgm-0017:smoothing	0.42	0.46	0.14	-0.22	-0.38
automotive-susan:image-pgm-0020:smoothing	0.42	0.46	0.13	-0.24	-0.35
network-dijkstra:cdataset-dijkstra-0001:	0.61	0.74	0.62	0.46	0.41
network-dijkstra:cdataset-dijkstra-0005:	0.64	0.84	0.81	0.72	0.65
network-dijkstra:cdataset-dijkstra-0009:	0.64	0.83	0.80	0.72	0.65
network-dijkstra:cdataset-dijkstra-0013:	0.64	0.84	0.80	0.74	0.65
network-patricia:cdataset-patricia-0001:	0.97	0.99	0.98	0.97	0.96
network-patricia:cdataset-patricia-0005:	0.93	0.97	0.96	0.93	0.92
network-patricia:cdataset-patricia-0009:	0.84	0.87	0.79	0.69	0.65
network-patricia:cdataset-patricia-0013:	0.75	0.78	0.64	0.47	0.39
network-patricia:cdataset-patricia-0017:	0.71	0.74	0.56	0.36	0.28
network-patricia:cdataset-patricia-0020:	0.73	0.75	0.57	0.38	0.31
telecom-adpcm-c:pcm-0001:	0.37	0.63	0.42	0.21	0.13
telecom-adpcm-c:pcm-0005:	0.34	0.57	0.32	0.12	0.01
telecom-adpcm-c:pcm-0009:	0.37	0.63	0.42	0.19	0.08
telecom-adpcm-c:pcm-0013:	0.38	0.64	0.42	0.21	0.13
telecom-adpcm-c:pcm-0017:	0.38	0.63	0.42	0.17	0.07
telecom-adpcm-c:pcm-0020:	0.35	0.59	0.35	0.09	0.03
telecom-adpcm-d:adpcm-0001:	0.89	0.97	0.96	0.93	0.93
telecom-adpcm-d:adpcm-0005:	0.89	0.97	0.96	0.93	0.93
telecom-adpcm-d:adpcm-0009:	0.89	0.97	0.96	0.93	0.93
telecom-adpcm-d:adpcm-0013:	0.88	0.96	0.94	0.91	0.92
telecom-adpcm-d:adpcm-0017:	0.88	0.97	0.95	0.93	0.92
telecom-adpcm-d:adpcm-0020:	0.88	0.96	0.94	0.90	0.91
telecom-crc32:pcm-0001:	0.63	0.89	0.91	0.81	0.84
telecom-crc32:pcm-0005:	0.64	0.89	0.90	0.84	0.83
telecom-crc32:pcm-0009:	0.64	0.89	0.92	0.84	0.83
telecom-crc32:pcm-0013:	0.64	0.90	0.93	0.88	0.87
telecom-crc32:pcm-0017:	0.64	0.90	0.94	0.88	0.88
telecom-crc32:pcm-0020:	0.64	0.90	0.94	0.88	0.89
telecom-gsm:au-0001:	0.73	0.94	0.93	0.90	0.88

Continued on next page

Table 3.3: R^2 validation score of the learned Fourier-sparse set function for different α; 500 queries and 20 flags in the search space

	1e-1	1e-2	1e-3	1e-4	1e-5
telecom-gsm:au-0005:	0.74	0.94	0.93	0.90	0.88
telecom-gsm:au-0009:	0.74	0.94	0.93	0.90	0.88
telecom-gsm:au-0013:	0.75	0.94	0.92	0.88	0.87
telecom-gsm:au-0017:	0.74	0.94	0.93	0.90	0.88
telecom-gsm:au-0020:	0.74	0.94	0.93	0.90	0.88

Table 3.4: Speedup of the minimum of the learned simulated Fourier-sparse set function over the best optimizating setting from the training data for different α; 500 queries and 20 flags in the search space

	1e-1	1e-2	1e-3	1e-4	1e-5
automotive-bitcount:number-0001:	0.962	0.925	0.936	0.930	0.931
automotive-qsort1:cdataset-qsort-0001:	0.960	0.977	0.983	0.983	0.976
automotive-qsort1:cdataset-qsort-0005:	0.962	0.973	0.978	0.976	0.976
automotive-qsort1:cdataset-qsort-0009:	0.962	0.978	0.979	0.974	0.974
automotive-qsort1:cdataset-qsort-0013:	0.964	0.978	0.977	0.978	0.977
automotive-qsort1:cdataset-qsort-0017:	0.963	0.975	0.972	0.971	0.978
automotive-qsort1:cdataset-qsort-0020:	0.961	0.976	0.983	0.977	0.977
automotive-susan:image-pgm-0001:corners	0.967	0.977	0.977	0.981	0.972
automotive-susan:image-pgm-0005:corners	0.970	0.972	0.977	0.975	0.976
automotive-susan:image-pgm-0009:corners	0.976	0.979	0.975	0.977	0.975
automotive-susan:image-pgm-0013:corners	0.975	0.981	0.978	0.976	0.978
automotive-susan:image-pgm-0017:corners	0.970	0.984	0.987	0.980	0.975
automotive-susan:image-pgm-0020:corners	0.972	0.980	0.981	0.977	0.977
automotive-susan:image-pgm-0001:edges	0.980	0.993	0.994	0.987	0.984
automotive-susan:image-pgm-0005:edges	0.974	0.992	0.986	0.990	0.990
automotive-susan:image-pgm-0009:edges	0.971	0.983	0.986	0.974	0.980
automotive-susan:image-pgm-0013:edges	0.987	0.999	0.996	0.991	0.992
automotive-susan:image-pgm-0017:edges	0.983	0.994	0.993	0.992	0.989
automotive-susan:image-pgm-0020:edges	0.981	0.994	0.992	0.989	0.985
automotive-susan:image-pgm-0001:smoothing	0.928	0.909	0.921	0.898	0.878
automotive-susan:image-pgm-0005:smoothing	0.931	0.912	0.905	0.909	0.912
automotive-susan:image-pgm-0009:smoothing	0.932	0.919	0.890	0.915	0.887
automotive-susan:image-pgm-0013:smoothing	0.927	0.909	0.895	0.914	0.914
automotive-susan:image-pgm-0017:smoothing	0.929	0.913	0.905	0.908	0.901
automotive-susan:image-pgm-0020:smoothing	0.930	0.911	0.897	0.935	0.889
network-dijkstra:cdataset-dijkstra-0001:	0.974	0.989	0.966	0.984	0.950
network-dijkstra:cdataset-dijkstra-0005:	0.917	0.969	0.968	0.964	0.965

Continued on next page

Table 3.4: Speedup of the minimum of the learned simulated Fourier-sparse set function over the best optimizating setting from the training data for different α; 500 queries and 20 flags in the search space

	1e-1	1e-2	1e-3	1e-4	1e-5
network-dijkstra:cdataset-dijkstra-0009:	0.927	0.969	0.971	0.960	0.951
network-dijkstra:cdataset-dijkstra-0013:	0.937	0.972	0.950	0.970	0.954
network-patricia:cdataset-patricia-0001:	0.903	0.943	0.932	0.921	0.936
network-patricia:cdataset-patricia-0005:	0.961	0.976	0.977	0.972	0.973
network-patricia:cdataset-patricia-0009:	0.973	0.985	0.975	0.977	0.975
network-patricia:cdataset-patricia-0013:	0.978	0.981	0.972	0.979	0.978
network-patricia:cdataset-patricia-0017:	0.980	0.984	0.984	0.977	0.975
network-patricia:cdataset-patricia-0020:	0.982	0.989	0.978	0.979	0.980
telecom-adpcm-c:pcm-0001:	0.996	0.976	0.974	0.982	0.980
telecom-adpcm-c:pcm-0005:	0.990	0.976	0.985	0.988	0.976
telecom-adpcm-c:pcm-0009:	0.997	0.980	0.977	0.979	0.983
telecom-adpcm-c:pcm-0013:	0.997	0.983	0.976	0.980	0.978
telecom-adpcm-c:pcm-0017:	0.997	0.974	0.970	0.981	0.975
telecom-adpcm-c:pcm-0020:	0.987	0.962	0.963	0.966	0.963
telecom-adpcm-d:adpcm-0001:	0.998	0.997	0.995	0.994	0.995
telecom-adpcm-d:adpcm-0005:	0.998	0.996	0.997	0.997	0.995
telecom-adpcm-d:adpcm-0009:	0.998	0.998	0.995	0.994	0.995
telecom-adpcm-d:adpcm-0013:	0.998	0.998	0.997	0.994	0.995
telecom-adpcm-d:adpcm-0017:	0.998	0.997	0.999	0.996	0.998
telecom-adpcm-d:adpcm-0020:	0.997	0.997	0.995	0.991	0.996
telecom-crc32:pcm-0001:	0.933	0.865	0.928	0.959	0.955
telecom-crc32:pcm-0005:	0.934	0.891	0.930	0.928	0.926
telecom-crc32:pcm-0009:	0.933	0.827	0.933	0.923	0.903
telecom-crc32:pcm-0013:	0.936	0.830	0.937	0.938	0.917
telecom-crc32:pcm-0017:	0.935	0.868	0.965	0.960	0.929
telecom-crc32:pcm-0020:	0.936	0.887	0.968	0.963	0.932
telecom-gsm:au-0001:	0.959	0.990	0.986	0.992	0.989
telecom-gsm:au-0005:	0.965	0.986	0.987	0.986	0.985
telecom-gsm:au-0009:	0.962	0.990	0.989	0.987	0.980
telecom-gsm:au-0013:	0.962	0.984	0.983	0.983	0.988
telecom-gsm:au-0017:	0.958	0.989	0.983	0.991	0.987
telecom-gsm:au-0020:	0.970	0.981	0.990	0.990	0.983

3.3 Simulated Fourier-sparse set function learning

Next, we scrutinize the soundness of offline Fourier-sparse set function learning from a different angle. Instead of making real runtime measurements, we construct a synthetic Fourier-sparse set function and answer queries by evaluating the synthetic function.

For every program we construct a Fourier-sparse set function with 60 non-zero coefficients, subdivided into 20 degree-one coefficients, 16 degree-two coefficients, 12 degree-three coefficients, 8 degree-four coefficients, and 4 degree-five coefficients. Half the coefficients are strictly positive, i.e., they represent flags in the standard optimization setting that successfully improve performance and should not be disabled. The other half of the coefficients can be either negative or positive and they may obtain higher values, i.e., they represent impactful flags. We select the coefficients uniformly at random and set the ranges of their values so that a Fourier-sparse set function with 20 flags usually has a maximum speedup uf 6% to 25% and with 120 flags a maximum speedup of 8% to 60%. To ensure reproducible results the name of the program is used as the seed of the random number generator.

3.3.1 Quality

As we can see in Table 3.5, 3.6, 3.7 when confronted with a perfect Fourier sparse set function, offline Fourier-sparse set function learning already succeeds with 100 queries, yields a noticeable speedup over the best optimization setting from the training data with 500 queries, and nearly perfectly learns the function with 1000 queries.

Table 3.5: Probability that the minimum of the learned simulated Fourier-sparse set function is better than the best optimization setting from the training data; 100-1000 queries and 20-120 flags in the search space

	20	30	40	50	60	70	80	90	100	110	120
100	0.69	0.66	0.75	0.75	0.75	0.72	0.69	0.75	0.78	0.72	0.69
200	0.94	0.88	0.88	0.91	0.84	0.81	0.91	0.88	0.94	0.84	0.91
300	1.00	0.97	0.97	0.94	0.94	0.97	0.97	0.94	0.97	0.97	0.91
400	1.00	0.94	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.97	1.00
500	0.94	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.94	1.00
600	0.97	0.97	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
700	0.94	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
800	0.97	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
900	0.94	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1000	0.94	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 3.6: Speedup of the minimum of the learned simulated Fourier-sparse set function over the best optimizating setting from the training data; 100-1000 queries and 20-120 flags in the search space

	20	30	40	50	60	70	80	90	100	110	120
100	1.013	1.016	1.030	1.022	1.018	1.022	1.019	1.025	1.023	1.023	1.016
200	1.042	1.043	1.056	1.064	1.062	1.056	1.055	1.050	1.070	1.069	1.057
300	1.065	1.071	1.071	1.080	1.100	1.088	1.106	1.103	1.123	1.115	1.097
400	1.065	1.093	1.114	1.132	1.121	1.130	1.132	1.126	1.154	1.152	1.121
500	1.060	1.102	1.130	1.152	1.162	1.166	1.172	1.168	1.178	1.174	1.168
600	1.060	1.096	1.133	1.159	1.176	1.188	1.188	1.189	1.196	1.212	1.193
700	1.049	1.099	1.134	1.160	1.179	1.198	1.195	1.189	1.208	1.227	1.211
800	1.057	1.097	1.131	1.162	1.180	1.201	1.198	1.195	1.218	1.232	1.218
900	1.050	1.097	1.129	1.162	1.180	1.200	1.192	1.190	1.216	1.231	1.218
1000	1.053	1.095	1.122	1.163	1.180	1.200	1.192	1.191	1.213	1.229	1.216

Table 3.7: Speedup of the optimal optimization setting over the minimum of the learned simulated Fourier-sparse set function; 100-1000 queries and 20-120 flags in the search space

	20	30	40	50	60	70	80	90	100	110	120
100	1.086	1.132	1.155	1.188	1.217	1.230	1.236	1.224	1.250	1.267	1.255
200	1.042	1.091	1.112	1.127	1.149	1.170	1.172	1.179	1.177	1.197	1.190
300	1.017	1.049	1.087	1.101	1.100	1.130	1.112	1.112	1.111	1.139	1.144
400	1.009	1.022	1.037	1.045	1.074	1.082	1.083	1.087	1.077	1.096	1.114
500	1.008	1.009	1.015	1.026	1.033	1.044	1.040	1.043	1.050	1.072	1.064
600	1.005	1.011	1.010	1.014	1.017	1.020	1.022	1.022	1.031	1.033	1.039
700	1.016	1.007	1.006	1.010	1.013	1.009	1.013	1.017	1.017	1.019	1.021
800	1.005	1.007	1.007	1.006	1.009	1.005	1.009	1.010	1.009	1.011	1.012
900	1.011	1.006	1.006	1.005	1.009	1.005	1.007	1.010	1.007	1.008	1.008
1000	1.007	1.005	1.007	1.004	1.008	1.005	1.005	1.008	1.006	1.008	1.007

3.3.2 Validation score

We repeat the experiment from Section 3.2.1 and look at the R^2 score of the learned Fourier-sparse set functions on a random validation set of 10000 queries and verify the chosen α value. Table 3.8, 3.4 reaffirm $\alpha = 0.01$ as a well chosen value.

Table 3.8: R^2 validation score of the learned simulated Fourier-sparse set function for different α; 500 queries and 20-100 flags in the search space

	20	30	40	50	60	70	80	90	100
1e-1	0.61	0.53	0.47	0.46	0.42	0.39	0.38	0.37	0.37
1e-2	0.95	0.93	0.91	0.88	0.85	0.82	0.77	0.76	0.75
1e-3	0.99	0.96	0.91	0.85	0.80	0.75	0.69	0.69	0.68
1e-4	0.97	0.88	0.83	0.72	0.66	0.62	0.54	0.58	0.56
1e-5	0.99	0.90	0.76	0.67	0.57	0.50	0.43	0.42	0.37

Table 3.9: Speedup of the minimum of the learned simulated Fourier-sparse set function over the best optimizating setting from the training data for different α; 500 queries and 20-100 flags in the search space

	20	30	40	50	60	70	80	90	100
1e-1	0.966	0.966	0.979	1.000	1.002	1.009	1.020	1.016	1.021
1e-2	1.060	1.102	1.130	1.152	1.162	1.166	1.172	1.168	1.178
1e-3	1.046	1.103	1.074	1.081	1.083	1.095	1.076	1.082	1.122
1e-4	1.060	1.061	1.074	1.084	1.089	1.083	1.065	1.072	1.068
1e-5	1.064	1.077	1.069	1.090	1.077	1.067	1.058	1.039	1.051

3.3.3 Noise

Another potential explanation for the failure of offline Fourier-sparse set function learning could be that even small noise from runtime measurements trips up the learning process. However, the results in Table 3.10, 3.11, 3.12 reject this hypothesis. Adding Gaussian noise $\sim \mathcal{N}(0, 0.01^2)$ to the runtimes that are slightly smaller than 1 second does not notably impair the ability to find a good minimum. The real measurements displayed in Table 3.1 appear to have less noise than that. It takes Gaussian noise $\sim \mathcal{N}(0, 0.05^2)$ for the learning process to fail.

Table 3.10: Speedup of the minimum of the learned simulated Fourier-sparse set function over the best optimizating setting from the training data; 100-1000 queries with Gaussian noise $\sim \mathcal{N}(0, 0.005^2)$ and 20-120 flags in the search space

	20	30	40	50	60	70	80	90	100	110	120
100	1.013	1.016	1.030	1.022	1.018	1.022	1.019	1.025	1.023	1.023	1.016
200	1.042	1.043	1.056	1.064	1.062	1.056	1.055	1.050	1.070	1.069	1.057
300	1.065	1.071	1.071	1.080	1.100	1.088	1.106	1.103	1.123	1.115	1.097
400	1.065	1.093	1.114	1.132	1.121	1.130	1.132	1.126	1.154	1.152	1.121
500	1.060	1.102	1.130	1.152	1.162	1.166	1.172	1.168	1.178	1.174	1.168

Continued on next page

Table 3.10: Speedup of the minimum of the learned simulated Fourier-sparse set function over the best optimizating setting from the training data; 100-1000 queries with Gaussian noise $\sim \mathcal{N}(0, 0.005^2)$ and 20-120 flags in the search space

	20	30	40	50	60	70	80	90	100	110	120
600	1.060	1.096	1.133	1.159	1.176	1.188	1.188	1.189	1.196	1.212	1.193
700	1.049	1.099	1.134	1.160	1.179	1.198	1.195	1.189	1.208	1.227	1.211
800	1.057	1.097	1.131	1.162	1.180	1.201	1.198	1.195	1.218	1.232	1.218
900	1.050	1.097	1.129	1.162	1.180	1.200	1.192	1.190	1.216	1.231	1.218
1000	1.053	1.095	1.122	1.163	1.180	1.200	1.192	1.191	1.213	1.229	1.216

Table 3.11: Speedup of the minimum of the learned simulated Fourier-sparse set function over the best optimizating setting from the training data; 100-1000 queries with Gaussian noise $\sim \mathcal{N}(0, 0.01^2)$ and 20-120 flags in the search space

	20	30	40	50	60	70	80	90	100	110	120
100	1.001	1.014	1.024	1.008	1.007	1.015	1.017	1.025	1.032	1.018	1.016
200	1.034	1.027	1.060	1.045	1.060	1.052	1.042	1.034	1.068	1.071	1.070
300	1.043	1.051	1.063	1.068	1.083	1.092	1.093	1.076	1.115	1.096	1.087
400	1.059	1.066	1.102	1.107	1.099	1.119	1.122	1.106	1.140	1.141	1.114
500	1.056	1.088	1.117	1.138	1.134	1.148	1.151	1.144	1.168	1.155	1.162
600	1.044	1.081	1.121	1.144	1.155	1.171	1.167	1.166	1.178	1.189	1.181
700	1.042	1.092	1.125	1.150	1.166	1.186	1.185	1.174	1.196	1.204	1.196
800	1.052	1.092	1.126	1.151	1.166	1.191	1.190	1.183	1.201	1.213	1.209
900	1.052	1.089	1.124	1.153	1.172	1.195	1.184	1.178	1.208	1.221	1.207
1000	1.051	1.089	1.119	1.152	1.176	1.193	1.187	1.180	1.206	1.222	1.206

Table 3.12: Speedup of the minimum of the learned simulated Fourier-sparse set function over the best optimizating setting from the training data; 100-1000 queries with Gaussian noise $\sim \mathcal{N}(0, 0.05^2)$ and 20-120 flags in the search space

	20	30	40	50	60	70	80	90	100	110	120
100	0.854	0.882	0.912	0.908	0.910	0.928	0.932	0.926	0.946	0.946	0.934
200	0.892	0.881	0.890	0.894	0.903	0.925	0.909	0.938	0.954	0.938	0.950
300	0.894	0.887	0.893	0.884	0.890	0.885	0.925	0.933	0.953	0.937	0.950
400	0.916	0.890	0.928	0.867	0.898	0.882	0.917	0.935	0.942	0.943	0.934
500	0.902	0.903	0.913	0.916	0.942	0.893	0.900	0.910	0.921	0.957	0.937
600	0.912	0.914	0.936	0.931	0.934	0.892	0.888	0.889	0.929	0.917	0.938
700	0.917	0.922	0.941	0.933	0.932	0.951	0.911	0.927	0.876	0.913	0.951

Continued on next page

Table 3.12: Speedup of the minimum of the learned simulated Fourier-sparse set function over the best optimizating setting from the training data; 100-1000 queries with Gaussian noise $\sim \mathcal{N}(0, 0.05^2)$ and 20-120 flags in the search space

	20	30	40	50	60	70	80	90	100	110	120
800	0.913	0.920	0.943	0.930	0.932	0.929	0.951	0.925	0.905	0.939	0.934
900	0.920	0.925	0.935	0.949	0.923	0.944	0.956	0.911	0.985	0.950	0.914
1000	0.927	0.932	0.951	0.942	0.936	0.923	0.927	0.915	0.948	0.964	0.922

3.4 Online Fourier-sparse set function learning

As offline Fourier-sparse set function learning does not succeed at finding a suitable optimization setting, this experiment explores if switching to online Fourier-sparse set function learning, i.e., allowing to iteratively choose the training data, brings remedy. Our acquisition function works as follows. First, we use 20% of the query budget to make random queries. Second, we fit and minimize a Fourier-sparse set function on this initial training data. If the resulting minimum optimization setting is not already in our training data, we choose it as the next query. Otherwise, the next query will be another random optimization setting. Then, we repeat the second step until the query budget is used up.

As it can be seen in Table 3.13 this approach looks more promising. The Dijkstra benchmark displays a noticeable speedup over the best flags from randomly sampled training data. Furthermore, although the speedup is minuscule, Fourier-sparse set function learning beats random samples for 47 out of 59 benchmark programs with a significance level of 20 repetitions. Since online Fourier-sparse set function learning is significantly slower than the offline version, we were unable to run the experiment with the large search space of 98 flags.

Table 3.13: Speedup of online Fourier-sparse set function learning over the best optimizating setting from randomly sampled training data; 500 queries and 20 flags in the search space

	Speedup
automotive-bitcount:number-0001:	1.001606
automotive-qsort1:cdataset-qsort-0001:	1.000185
automotive-qsort1:cdataset-qsort-0005:	1.003432
automotive-qsort1:cdataset-qsort-0009:	0.998361
automotive-qsort1:cdataset-qsort-0013:	1.002487
automotive-qsort1:cdataset-qsort-0017:	0.999163

Continued on next page

Table 3.13: Speedup of online Fourier-sparse set function learning over the best optimizating setting from randomly sampled training data; 500 queries and 20 flags in the search space

	Speedup
automotive-qsort1:cdataset-qsort-0020:	1.002016
automotive-susan:image-pgm-0001:corners	0.993455
automotive-susan:image-pgm-0005:corners	0.999218
automotive-susan:image-pgm-0009:corners	0.997379
automotive-susan:image-pgm-0013:corners	1.001209
automotive-susan:image-pgm-0017:corners	1.001323
automotive-susan:image-pgm-0020:corners	1.000328
automotive-susan:image-pgm-0001:edges	1.004254
automotive-susan:image-pgm-0005:edges	1.003030
automotive-susan:image-pgm-0009:edges	1.003601
automotive-susan:image-pgm-0013:edges	1.005629
automotive-susan:image-pgm-0017:edges	1.002312
automotive-susan:image-pgm-0020:edges	1.005048
automotive-susan:image-pgm-0001:smoothing	1.001532
automotive-susan:image-pgm-0005:smoothing	1.001701
automotive-susan:image-pgm-0009:smoothing	1.002164
automotive-susan:image-pgm-0013:smoothing	1.001680
automotive-susan:image-pgm-0017:smoothing	1.003319
automotive-susan:image-pgm-0020:smoothing	1.001717
network-dijkstra:cdataset-dijkstra-0001:	1.027766
network-dijkstra:cdataset-dijkstra-0005:	1.033165
network-dijkstra:cdataset-dijkstra-0009:	1.031030
network-dijkstra:cdataset-dijkstra-0013:	1.026990
network-patricia:cdataset-patricia-0001:	0.995036
network-patricia:cdataset-patricia-0005:	0.999813
network-patricia:cdataset-patricia-0009:	1.003497
network-patricia:cdataset-patricia-0013:	1.006286
network-patricia:cdataset-patricia-0017:	1.001415
network-patricia:cdataset-patricia-0020:	1.000067
telecom-adpcm-c:pcm-0001:	0.999788
telecom-adpcm-c:pcm-0005:	1.000469
telecom-adpcm-c:pcm-0009:	1.001507
telecom-adpcm-c:pcm-0013:	1.001136
telecom-adpcm-c:pcm-0017:	1.000434
telecom-adpcm-c:pcm-0020:	1.000692
telecom-adpcm-d:adpcm-0001:	1.000235
telecom-adpcm-d:adpcm-0005:	0.999589
telecom-adpcm-d:adpcm-0009:	0.999936

Continued on next page

Table 3.13: Speedup of online Fourier-sparse set function learning over the best optimizating setting from randomly sampled training data; 500 queries and 20 flags in the search space

	Speedup
telecom-adpcm-d:adpcm-0013:	1.000407
telecom-adpcm-d:adpcm-0017:	1.000491
telecom-adpcm-d:adpcm-0020:	1.000305
telecom-crc32:pcm-0001:	0.998429
telecom-crc32:pcm-0005:	1.002180
telecom-crc32:pcm-0009:	0.998474
telecom-crc32:pcm-0013:	1.002181
telecom-crc32:pcm-0017:	1.002142
telecom-crc32:pcm-0020:	1.002345
telecom-gsm:au-0001:	1.005498
telecom-gsm:au-0005:	1.004027
telecom-gsm:au-0009:	1.005938
telecom-gsm:au-0013:	1.004220
telecom-gsm:au-0017:	1.007095
telecom-gsm:au-0020:	1.000129

3.5 Low-degree Fourier-sparse set function learning

In this experiment we compare our online Fourier-sparse set function learning implementation to two other Fourier-sparse set function learning methods that are constrained to low-degree terms. The first tuning method is identical to our online Fourier-sparse set function learning implementation except it is restricted to at most degree-two terms. The second implementation is the well known BOCS (Bayesian Optimization of Combinatorial Structures) [4] which is also restricted to degree-two terms but uses a more sophisticated acquisition function.

The results can be seen in Table 3.14. The tuning method(s) with the highest speedup is colored in green. Online Fourier-sparse set function learning performs best 29 times and BOCS performs best 25 times, out of 59 benchmark programs in total. The degree-two variant of our implementation performs unimpressive which reaffirms the necessity to consider complex interactions between different optimizations and alleviate concerns the unrestricted Fourier-sparse set function learning will overfit.

Table 3.14: Speedup of the best optimization setting learned from different Fourier-sparse set functions over -O3; 500 queries and 20 flags in the search space

	Fourier	Fourier (degree-two)	BOCS
automotive-bitcount:number-0001:	1.223	1.226	1.226
automotive-qsort1:cdataset-qsort-0001:	1.029	1.032	1.032
automotive-qsort1:cdataset-qsort-0005:	1.040	1.038	1.037
automotive-qsort1:cdataset-qsort-0009:	1.033	1.033	1.037
automotive-qsort1:cdataset-qsort-0013:	1.036	1.032	1.036
automotive-qsort1:cdataset-qsort-0017:	1.033	1.034	1.037
automotive-qsort1:cdataset-qsort-0020:	1.036	1.032	1.038
automotive-susan:image-pgm-0001:corners	1.002	1.011	1.000
automotive-susan:image-pgm-0005:corners	1.007	1.006	1.006
automotive-susan:image-pgm-0009:corners	1.007	1.008	1.006
automotive-susan:image-pgm-0013:corners	1.008	1.008	1.007
automotive-susan:image-pgm-0017:corners	1.007	1.005	1.005
automotive-susan:image-pgm-0020:corners	1.005	1.004	1.003
automotive-susan:image-pgm-0001:edges	1.057	1.055	1.058
automotive-susan:image-pgm-0005:edges	1.051	1.049	1.051
automotive-susan:image-pgm-0009:edges	1.053	1.052	1.054
automotive-susan:image-pgm-0013:edges	1.076	1.073	1.076
automotive-susan:image-pgm-0017:edges	1.063	1.061	1.063
automotive-susan:image-pgm-0020:edges	1.059	1.054	1.056
automotive-susan:image-pgm-0001:smoothing	1.085	1.084	1.083
automotive-susan:image-pgm-0005:smoothing	1.080	1.078	1.082
automotive-susan:image-pgm-0009:smoothing	1.078	1.077	1.079
automotive-susan:image-pgm-0013:smoothing	1.086	1.085	1.087
automotive-susan:image-pgm-0017:smoothing	1.081	1.075	1.082
automotive-susan:image-pgm-0020:smoothing	1.083	1.084	1.084
network-dijkstra:cdataset-dijkstra-0001:	1.002	1.001	1.001
network-dijkstra:cdataset-dijkstra-0005:	1.018	1.015	1.021
network-dijkstra:cdataset-dijkstra-0009:	1.013	1.013	1.013
network-dijkstra:cdataset-dijkstra-0013:	1.010	1.011	1.010
network-patricia:cdataset-patricia-0001:	1.016	1.017	1.048
network-patricia:cdataset-patricia-0005:	1.041	1.040	1.048
network-patricia:cdataset-patricia-0009:	1.026	1.024	1.028
network-patricia:cdataset-patricia-0013:	1.024	1.019	1.023
network-patricia:cdataset-patricia-0017:	1.016	1.016	1.018
network-patricia:cdataset-patricia-0020:	1.017	1.014	1.018
telecom-adpcm-c:pcm-0001:	1.022	1.022	1.022
telecom-adpcm-c:pcm-0005:	1.140	1.139	1.138

Continued on next page

Table 3.14: Speedup of the best optimization setting learned from different Fourier-sparse set functions over −03; 500 queries and 20 flags in the search space

	Fourier	Fourier (degree-two)	BOCS
telecom-adpcm-c:pcm-0009:	1.019	1.017	1.018
telecom-adpcm-c:pcm-0013:	1.025	1.024	1.024
telecom-adpcm-c:pcm-0017:	1.032	1.031	1.031
telecom-adpcm-c:pcm-0020:	1.061	1.060	1.060
telecom-adpcm-d:adpcm-0001:	1.016	1.015	1.015
telecom-adpcm-d:adpcm-0005:	1.016	1.016	1.016
telecom-adpcm-d:adpcm-0009:	1.016	1.015	1.016
telecom-adpcm-d:adpcm-0013:	1.017	1.015	1.015
telecom-adpcm-d:adpcm-0017:	1.017	1.016	1.015
telecom-adpcm-d:adpcm-0020:	1.016	1.015	1.015
telecom-crc32:pcm-0001:	1.736	1.735	1.732
telecom-crc32:pcm-0005:	1.740	1.739	1.726
telecom-crc32:pcm-0009:	1.737	1.736	1.736
telecom-crc32:pcm-0013:	1.741	1.739	1.719
telecom-crc32:pcm-0017:	1.740	1.738	1.727
telecom-crc32:pcm-0020:	1.742	1.739	1.724
telecom-gsm:au-0001:	1.253	1.248	1.256
telecom-gsm:au-0005:	1.253	1.249	1.254
telecom-gsm:au-0009:	1.257	1.254	1.261
telecom-gsm:au-0013:	1.234	1.232	1.238
telecom-gsm:au-0017:	1.260	1.258	1.264
telecom-gsm:au-0020:	1.232	1.237	1.242

We have also integrated a different algorithm [1] to efficiently learn low-degree Fourier-sparse set functions based on the Walsh-Hadamard transform into our codebase, but we did not include it in our evaluation because its implementation does not allow to set a limit on the number of query.

3.6 Comparison to other tuning methods

In this final experiment we compare online Fourier-sparse set function learning to SRTuner and two basic tuning methods. *Random* simply makes uniformly random queries and returns the best optimization setting among them. This is the baseline any sensible tuning method needs to be able to beat (unless random is already very close to the global optimum). *Greedy* makes one query for every possible optimization setting that differs in exactly one optimization and returns the best optimization setting among them. This process is repeated until the budget of queries is used up. For example with a budget of 500 queries and 100 binary flags in the search space, there will be 5 rounds.

For the two basic tuning methods the reported speedup may be slightly below 1. This does not imply a defunct in the evaluation process, but is expected to happen for programs where the standard optimization setting is already close to optimal. The crux lies in how we evaluate the return value of a tuning method. After a tuning method returns its predicted minimum, we measure its runtime one last time with an extra number of repetitions because we want to be very confident in the accuracy of this value before adding it to the final output (the tuning method does not get this feedback). Therefore, when the standard optimization setting is already close to optimal, the basic tuning methods are biased towards selecting an optimization setting that includes negative noise which disappears on a more through measurement. For a single optimization setting the cost of additional repetitions is negligible in the big picture, but doing this for all measurements would significantly increase the duration of the tuning method.

The results for a small search space can be seen in Table 3.15. The tuning method(s) with the highest speedup is colored in green. Online Fourier-sparse set function learning performs best 25 times and SRTuner performs best 22 times, out of 59 benchmark programs in total. However, it is irritating that neither online Fourier-sparse set function learning nor SRTuner manage to beat random queries by a notable margin and that the confidence intervals overlap. This doesn't match what has been reported from the SRTuner paper which was also evaluated on cBench programs.

Table 3.15: Speedup of the best optimization setting learned from different tuning methods over -O3; 500 queries and 20 flags in the search space

	Random	Greedy	Fourier	SRTuner
automotive-bitcount:number-0001:	1.220	1.225	1.222	1.219
automotive-qsort1:cdataset-qsort-0001:	1.029	1.026	1.029	1.028
automotive-qsort1:cdataset-qsort-0005:	1.036	1.037	1.040	1.033
automotive-qsort1:cdataset-qsort-0009:	1.035	1.036	1.034	1.031
automotive-qsort1:cdataset-qsort-0013:	1.033	1.031	1.035	1.024
automotive-qsort1:cdataset-qsort-0017:	1.033	1.031	1.032	1.028
automotive-qsort1:cdataset-qsort-0020:	1.033	1.036	1.036	1.029
automotive-susan:image-pgm-0001:corners	1.006	1.005	0.999	1.002
automotive-susan:image-pgm-0005:corners	1.007	1.005	1.006	1.003
automotive-susan:image-pgm-0009:corners	1.008	1.005	1.005	0.999
automotive-susan:image-pgm-0013:corners	1.007	1.001	1.008	1.001
automotive-susan:image-pgm-0017:corners	1.006	0.999	1.007	1.000
automotive-susan:image-pgm-0020:corners	1.005	1.002	1.005	1.001
automotive-susan:image-pgm-0001:edges	1.052	1.055	1.057	1.048
automotive-susan:image-pgm-0005:edges	1.047	1.048	1.050	1.049
automotive-susan:image-pgm-0009:edges	1.049	1.052	1.052	1.048
automotive-susan:image-pgm-0013:edges	1.069	1.077	1.075	1.061
automotive-susan:image-pgm-0017:edges	1.060	1.062	1.062	1.054
automotive-susan:image-pgm-0020:edges	1.053	1.057	1.059	1.050
automotive-susan:image-pgm-0001:smoothing	1.082	1.083	1.084	1.083
automotive-susan:image-pgm-0005:smoothing	1.078	1.079	1.080	1.076
automotive-susan:image-pgm-0009:smoothing	1.075	1.076	1.078	1.077
automotive-susan:image-pgm-0013:smoothing	1.083	1.083	1.085	1.081
automotive-susan:image-pgm-0017:smoothing	1.077	1.080	1.081	1.082
automotive-susan:image-pgm-0020:smoothing	1.081	1.085	1.083	1.083
network-dijkstra:cdataset-dijkstra-0001:	0.975	1.000	1.002	0.992
network-dijkstra:cdataset-dijkstra-0005:	0.986	1.000	1.018	1.022
network-dijkstra:cdataset-dijkstra-0009:	0.983	1.000	1.013	0.982
network-dijkstra:cdataset-dijkstra-0013:	0.983	1.000	1.010	0.985
network-patricia:cdataset-patricia-0001:	1.012	1.000	1.007	1.072
network-patricia:cdataset-patricia-0005:	1.036	1.035	1.036	1.078
network-patricia:cdataset-patricia-0009:	1.022	1.024	1.026	1.124
network-patricia:cdataset-patricia-0013:	1.018	1.010	1.025	1.115
network-patricia:cdataset-patricia-0017:	1.016	1.018	1.017	1.109
network-patricia:cdataset-patricia-0020:	1.016	1.017	1.016	1.099
telecom-adpcm-c:pcm-0001:	1.022	1.022	1.022	1.046
telecom-adpcm-c:pcm-0005:	1.139	1.134	1.139	1.163
telecom-adpcm-c:pcm-0009:	1.017	1.018	1.018	1.043

Continued on next page

Table 3.15: Speedup of the best optimization setting learned from different tuning methods over -O3; 500 queries and 20 flags in the search space

	Random	Greedy	Fourier	SRTuner
telecom-adpcm-c:pcm-0013:	1.024	1.025	1.025	1.043
telecom-adpcm-c:pcm-0017:	1.031	1.030	1.032	1.053
telecom-adpcm-c:pcm-0020:	1.060	1.060	1.060	1.071
telecom-adpcm-d:adpcm-0001:	1.016	1.016	1.016	1.014
telecom-adpcm-d:adpcm-0005:	1.016	1.015	1.015	1.014
telecom-adpcm-d:adpcm-0009:	1.016	1.015	1.016	1.014
telecom-adpcm-d:adpcm-0013:	1.016	1.015	1.016	1.014
telecom-adpcm-d:adpcm-0017:	1.016	1.015	1.016	1.015
telecom-adpcm-d:adpcm-0020:	1.016	1.015	1.016	1.015
telecom-crc32:pcm-0001:	1.738	1.713	1.736	1.742
telecom-crc32:pcm-0005:	1.739	1.720	1.743	1.738
telecom-crc32:pcm-0009:	1.739	1.715	1.737	1.750
telecom-crc32:pcm-0013:	1.739	1.719	1.743	1.743
telecom-crc32:pcm-0017:	1.738	1.719	1.742	1.741
telecom-crc32:pcm-0020:	1.740	1.720	1.744	1.738
telecom-gsm:au-0001:	1.245	1.254	1.251	1.252
telecom-gsm:au-0005:	1.249	1.259	1.254	1.260
telecom-gsm:au-0009:	1.248	1.258	1.255	1.260
telecom-gsm:au-0013:	1.229	1.234	1.234	1.243
telecom-gsm:au-0017:	1.251	1.254	1.260	1.266
telecom-gsm:au-0020:	1.231	1.241	1.232	1.246

The results for a large search space can be seen in Table 3.16. The tuning method(s) with the highest speedup is colored in green. Since online Fourier-sparse set function learning is significantly slower than the offline version, we were unable to include it in this experiment. Greedy performs best 42 times and SRTuner performs best 11 times, out of 59 benchmark programs in total. However, it is again irritating that all tuning methods are this close to each other.

Table 3.16: Speedup of the best optimization setting learned from different tuning methods over -O3; 500 queries and 98 flags in the search space

	Random	Greedy	SRTuner
automotive-bitcount:number-0001:	1.179	1.234	1.195
automotive-qsort1:cdataset-qsort-0001:	1.028	1.038	1.028
automotive-qsort1:cdataset-qsort-0005:	1.036	1.043	1.036
automotive-qsort1:cdataset-qsort-0009:	1.034	1.042	1.033
automotive-qsort1:cdataset-qsort-0013:	1.032	1.038	1.033

Continued on next page

Table 3.16: Speedup of the best optimization setting learned from different tuning methods over -O3; 500 queries and 98 flags in the search space

	Random	Greedy	SRTuner
automotive-qsort1:cdataset-qsort-0017:	1.035	1.039	1.034
automotive-qsort1:cdataset-qsort-0020:	1.033	1.040	1.035
automotive-susan:image-pgm-0001:corners	0.993	0.997	1.001
automotive-susan:image-pgm-0005:corners	1.002	1.004	1.008
automotive-susan:image-pgm-0009:corners	1.003	1.001	1.004
automotive-susan:image-pgm-0013:corners	1.000	1.001	0.999
automotive-susan:image-pgm-0017:corners	0.993	1.000	0.995
automotive-susan:image-pgm-0020:corners	0.994	1.001	0.998
automotive-susan:image-pgm-0001:edges	1.105	1.078	1.102
automotive-susan:image-pgm-0005:edges	1.132	1.107	1.131
automotive-susan:image-pgm-0009:edges	1.091	1.077	1.092
automotive-susan:image-pgm-0013:edges	1.103	1.077	1.106
automotive-susan:image-pgm-0017:edges	1.108	1.075	1.105
automotive-susan:image-pgm-0020:edges	1.091	1.079	1.094
automotive-susan:image-pgm-0001:smoothing	1.056	1.077	1.074
automotive-susan:image-pgm-0005:smoothing	1.052	1.079	1.072
automotive-susan:image-pgm-0009:smoothing	1.051	1.074	1.073
automotive-susan:image-pgm-0013:smoothing	1.056	1.081	1.079
automotive-susan:image-pgm-0017:smoothing	1.052	1.077	1.076
automotive-susan:image-pgm-0020:smoothing	1.055	1.079	1.077
network-dijkstra:cdataset-dijkstra-0001:	0.987	1.016	0.988
network-dijkstra:cdataset-dijkstra-0005:	1.018	1.041	1.034
network-dijkstra:cdataset-dijkstra-0009:	0.993	1.020	1.001
network-dijkstra:cdataset-dijkstra-0013:	0.992	1.019	1.006
network-patricia:cdataset-patricia-0001:	1.049	1.032	1.053
network-patricia:cdataset-patricia-0005:	1.096	1.107	1.101
network-patricia:cdataset-patricia-0009:	1.168	1.182	1.167
network-patricia:cdataset-patricia-0013:	1.165	1.174	1.164
network-patricia:cdataset-patricia-0017:	1.151	1.173	1.153
network-patricia:cdataset-patricia-0020:	1.139	1.145	1.140
telecom-adpcm-c:pcm-0001:	1.272	1.301	1.297
telecom-adpcm-c:pcm-0005:	1.561	1.548	1.574
telecom-adpcm-c:pcm-0009:	1.256	1.291	1.272
telecom-adpcm-c:pcm-0013:	1.397	1.443	1.431
telecom-adpcm-c:pcm-0017:	1.271	1.308	1.290
telecom-adpcm-c:pcm-0020:	1.329	1.356	1.351
telecom-adpcm-d:adpcm-0001:	1.327	1.363	1.353
telecom-adpcm-d:adpcm-0005:	1.355	1.373	1.364

Continued on next page

Table 3.16: Speedup of the best optimization setting learned from different tuning methods over -O3; 500 queries and 98 flags in the search space

	Random	Greedy	SRTuner
telecom-adpcm-d:adpcm-0009:	1.356	1.376	1.372
telecom-adpcm-d:adpcm-0013:	1.410	1.441	1.429
telecom-adpcm-d:adpcm-0017:	1.355	1.375	1.371
telecom-adpcm-d:adpcm-0020:	1.359	1.382	1.377
telecom-crc32:pcm-0001:	1.750	1.717	1.744
telecom-crc32:pcm-0005:	1.748	1.720	1.749
telecom-crc32:pcm-0009:	1.751	1.718	1.744
telecom-crc32:pcm-0013:	1.749	1.720	1.747
telecom-crc32:pcm-0017:	1.748	1.720	1.754
telecom-crc32:pcm-0020:	1.750	1.721	1.752
telecom-gsm:au-0001:	1.325	1.341	1.339
telecom-gsm:au-0005:	1.336	1.350	1.345
telecom-gsm:au-0009:	1.327	1.344	1.337
telecom-gsm:au-0013:	1.311	1.325	1.321
telecom-gsm:au-0017:	1.340	1.353	1.349
telecom-gsm:au-0020:	1.312	1.331	1.315

Chapter 4

Conclusion

We present a benchmarking infrastructure that yields accurate measurements in a parallel environment which greatly accelerates the autotuning process. We use this benchmarking infrastructure to demonstrate that Fourier-sparse set function learning can reliably find a selection of GCC compiler flags with better performance than both a random search and the highest standard optimization setting.

We reaffirm that considering interactions between three or more flags is important by comparing the general Fourier-sparse set function learning implementation to its degree-two variant. We show that our Fourier-sparse set function learning implementation can keep up with SRTuner, another tuning method that focuses on accurate optimization impact estimation and inter-optimization relationships, and with BOCS, a low-degree implementation of Fourier-sparse set function learning featuring a more sophisticated acquisition function.

We verify in synthetic experiments that Fourier-sparse set function learning can almost perfectly learn the best selection of compiler flags if the target function is well approximated by a Fourier-sparse set function even when confronted with a very large search space, noisy runtime measurements, and training data that is randomly sampled rather than iteratively selected during the learning process.

4.1 Limitations and future work

The definition of our set does not take the order of flags into account which limits its portability to other compilers such as LLVM. Furthermore, support for parametric flags is very limited. For example, the compiler flag `fvect-cost-model` takes one of the values “unlimited”, “dynamic”, or “cheap”. We currently model this as three separate items in the set, i.e., for

each value the flag `fvect-cost-model=<value>` is present or missing from the compilation flags. This relies on GCC's behavior of only considering the last duplicate flag, but this also implies a more complex learning process because the tuning method has to realize enabling two values at the same time is pointless. Other tuning methods like SRTuner fully support parametric flags.

In the comparisons of the various tuning methods we observe all results are very close to each other which does not match our expectations from previous work. One possible explanation is that we are already close to the global optimum. To confirm or reject this guess, one would have to lower the search space to a size where it is feasible to traverse the full search space to compute the global optimum. Another possibility is an unexpected distortion of the runtime measurements in our parallel environment that escapes our stability checks.

In a similar vein, we have not yet found a conclusive explanation why Fourier-sparse set function learning performs well in a synthetic setting but fails in practice when the training data is randomly sampled rather than iteratively selected during the learning process.

Bibliography

[1] Andisheh Amrollahi, Amir Zandieh, Michael Kapralov, and Andreas Krause. Efficiently learning fourier sparse set functions. *Advances in Neural Information Processing Systems*, 32, 2019.

[2] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)*, 51(5):1–42, 2018.

[3] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. Cobayn: Compiler autotuning framework using bayesian networks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(2):1–25, 2016.

[4] Ricardo Baptista and Matthias Poloczek. Bayesian optimization of combinatorial structures. In *International Conference on Machine Learning*, pages 462–471. PMLR, 2018.

[5] Grigori Fursin. Collective knowledge: organizing research projects as a database of reusable components and portable workflows with common interfaces. *Philosophical Transactions of the Royal Society A*, 379(2197):20200211, 2021.

[6] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, et al. Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39(3):296–327, 2011.

[7] Kyriakos Georgiou, Craig Blackmore, Samuel Xavier-de Souza, and Kerstin Eder. Less is more: Exploiting the standard compiler optimization levels for better performance and energy consumption. In *Proceedings*

of the 21st International Workshop on Software and Compilers for Embedded Systems, pages 35–42, 2018.

[8] Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 165–174, 2008.

[9] Tierry Hörmann. Compiler flag optimization using fourier-sparse poset functions. Master's thesis, ETH Zurich, 2022.

[10] Zhelong Pan and Rudolf Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 12–pp. IEEE, 2006.

[11] Eunjung Park, Sameer Kulkarni, and John Cavazos. An evaluation of different modeling techniques for iterative compilation. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, pages 65–74, 2011.

[12] Sunghyun Park, Salar Latifi, Yongjun Park, Armand Behroozi, Byungsoo Jeon, and Scott Mahlke. Srtuner: effective compiler optimization customization by exposing synergistic relations. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 118–130. IEEE, 2022.

[13] Rogier PJ Pinkers, Peter MW Knijnenburg, Masayo Haneda, and Harry AG Wijshoff. Statistical selection of compiler options. In *The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings.*, pages 494–501. IEEE, 2004.

[14] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I August. Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 204–215. IEEE, 2003.

[15] Eliza Wszola. *Machine Learning on Manycore CPUs*. PhD thesis, ETH Zurich, 2022.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Compiler flag optimization using Fourier-sparse Poset Functions

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Manuel	Nowack

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 30.08.2022	M. Nowack

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.