On how to avoid a false positive speedup

Ricardo Luis de Azevedo da Rocha
Dept. of Computing Engineering
University of Sao Paulo and
Dept. of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8, Canada
Email: rlarocha@usp.br
azevedod@ualberta.ca

Paul Berube
Dept. of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8, Canada
Email: pberube@ualberta.ca

Bruno Rosa
Dept. of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8, Canada
Email: brosa@ualberta.ca

José Nelson Amaral
Dept. of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8, Canada
Email: amaral@cs.ualberta.ca

***Abstract*—**
The usual way to do research in compilers, moreover in Feedback Directed Optimization is to construct a framework and devise an experiment based on single-run input training and single data testing. Recently some researchers have argued about the reliability of such experiments, and developed other approaches to this problem. Usually using repetition of experiments and collecting data to perform a reliable statistical analysis. This paper also discusses these issues and we aimed to construct an experiment to show a false speedup from actual data. We did this by just ignoring our multiple runs strategy and literally picking parts of our collected data to show that it could happen in a single-run scheme. In conclusion we state that the only way to avoid these problems is to define and use a reliable methodology based on solid statistical measurements. In this paper we also present our methodology called *combined profiling* (CP), and show that using it we can have reliable results. We showed that FDI decisions can be more accurate using CP instead of single-run evaluation.

I. Introduction

Research in compiler transformations often demonstrates heroic efforts in both the identification and abstract analysis of opportunities to improve program efficiency, and in the concrete implementation of these ideas. However, standard practices at the evaluation stage of the scientific process are modest at best, perhaps because code transformations have a long history of providing significant benefits in practical, every-day situations. In most cases, compilers are evaluated using a collection of programs, with each program evaluated using a timing run on a single evaluation input. The deficiencies of this evaluation process are particularly prevalent, and especially disconcerting, when *feedback-directed optimization* (FDI) is used to guide a transformation. In this scenario, instrumentation is inserted into the program during an initial compilation in order to collect a profile of the run-time behavior of the program during one or more training runs. The profile is used in a second compilation of the program to help the compiler assess the benefit of code transformation opportunities. The current standard practice for evaluating an FDI compiler uses the profile of a single-training input to guide transformations, and evaluates the transformed program with a single evaluation input. These standard practices set program inputs as controlled variables. However, performance evaluation should be generalizable to real-world program workloads. Consequently, the program-input dimensions of a rigorous evaluation of compiler performance must be manipulated variables.

Previous work has not addressed the problem of representing and utilizing multi-run profiles. An FDI compiler should not simply add or average profiles from multiple runs, because such a profile does not provide any information about the variations in program behaviors observed between different inputs. [1] uses *Combined Profiling* (CP) to merge the profiles from multiple runs into a distribution model that allows code transformations to consider cross-run behavior variations. Experimental results demonstrate that meaningful behavior variation is present in the program workloads, and that this variation is successfully captured and represented by the CP methodology.

This research uses a different approach and its goal is to assess the results of *combined profiling* (CP). There have been some recent efforts trying to apply multiple profiles to FDI and also to evaluate the performance of a program from multiple inputs. CP can be applied to many different optimization techniques, such as inlining, loop unrolling, etc. We decided to apply CP to inlining as a case study, because it allows many other optimization techniques to be performed afterwards.

Although the usual way to do research in Feedback Directed

This research was done while the nth author was in a sabbatical year at the University of Alberta, supported by grant 2011/17096-5 from the Fundação de Amparo à Pesquisa do Estado de São Paulo – FAPESP.

This research is supported by fellowships and grants from the Natural Sciences and Engineering Research Council of Canada (NSERC), the Informatics Circle of Research Excellence (iCORE), and the Canadian Foundation for innovation (CFI).

Optimization is to perform a single-run input training and single data testing, recently are being developed other approaches to this problem. The main goal of these new approaches is to perform multiple-runs under multiple data, because some questions concerning the single-run approach arose, such as, is this method accurate, or proper, or reliable?

Recent work [2] states that execution time is a key measurement, for example 90 out of 122 papers presented in 2011 at PLDI, ASPLOS, and ISMM, or published in TOPLAS and TACO. As reported by [2], the overwhelming majority of these papers has shown results either impossible to repeat, or didn't demonstrate their performance claims, there were no measure of variation for their results. Our work also focus on execution time, and we expect to reinforce the use of a methodology that allows the researcher to control the measurement errors, or at least to provide sufficient evidence of performance improvement.

This paper discusses these issues by constructing a "false" speedup from actual data, just ignoring our multiple runs strategy and literally picking parts of our collected data to show that many results are possible in a single-run scheme. We also point out that a "false" slowdown can also be picked from our data. This way we reinforce the use of multiple-run methodologies.

Several open questions about the use of profiles collected from multiple runs of a program were addressed and assessed in [3]. Now there are still some questions, as multiple profiles are combined. What is the impact of CP in a controlled case study? FDI decisions can be more accurate using CP instead of single-run evaluation?

This paper addresses these questions by employing a case study of the CP process. As already mentioned the case proposed was for inlining, and we compared the CP process with the single-run process. The application of CP to other situations with multiple profiling instances, such as profiling program phases individually, is not within the scope of this paper.

The main contribution of this paper are:

- *Methodological considerations* The behavior of single-runs and CP-runs are compared and analyzed. We show that single-run methodologies are error-prone.
- *Case study* The case study illustrates that the single-run methodology can induce the researcher to serious errors, and that a methodology like CP is better suited to evaluate performance.

This paper has seven sections, the introduction, where the research problem is posed and the main ideas are shown. We start by describing the inlining transformation in the next section, and then we start the section where we present the "speedup" and also give a notice on a "slowdown" for the same problem. Following this section we analyze the environment and provide sufficient statistical information to explain what happened in the previous section, and also what may happen in experiments using the same methodology. Following the data analysis we employed in the latter section, we show how this problem can be avoided by means of the CP methodology. We end this paper showing the related work, and the conclusion.

II. Function Inlining

Function inlining, or simply inlining, is a classic code transformation that can significantly increase the performance of many programs. A compiler pass that decides which calls to inline, and in which order, is referred to as an inliner. The basic idea of inlining is straightforward: rather than making a function call, replace the call in the originating function with a copy of the body of the to-be-called function. Nonetheless, many inliner designs are possible; [1] describes the existing inliner in `LLVM`, and also the alternative approach used by a new feedback-directed inliner (FDI) that uses CP. All inlining discussed in this paper is implemented in the open-source `LLVM` compiler [4].

Some terminology is required to identify the various functions and calls involved in the inlining process. The function making a call is referred to as the *caller*, while the called function is the *callee*. The representation of a call in a compiler's *internal representation* (IR) is a *call site*; in `LLVM`, a call site is an instruction that indicates both the caller and the callee. Thus, inlining replaces a call site by a copy of that call site's callee. When a call is inlined, the callee may contain call sites, which are copied into the caller to produce new call sites. The call site where inlining occurs is called the *source* call site. A call site in the callee that is copied during inlining is called an *original* call site, and the new copy of the original call site inside the caller is called the *target* call site.

A. Barriers to Inlining

Not every call site can be inlined. Indirect calls use a pointer variable to identify the location of the called code, and arise from function pointers and dynamically-polymorphic call dispatching. These calls cannot be inlined, because the callee is unknown at compiler time. External calls into code not currently available in the compiler, such as calls into different modules or to statically-linked library functions cannot be inlined before link-time because the source representation of the callee is not available in the compiler. Calls to dynamically-linked libraries can never be inlined by definition. Moreover, if a callee uses a `setjump` instruction, it cannot be inlined. A `setjump` can redirect program control flow *anywhere*, including the middle of different function, without using the call/return mechanisms. Inlining the `setjump` could cause any manual stack management at the target of the jump to be incorrect; the inlined version would not be functionally equivalent to the original.

B. Benefits of Inlining

Inlining a call has a small direct benefit. Removing the call reduces the number of executed instructions. The `call` instruction in the caller is unnecessary, as is the `return` instruction in the callee. Furthermore, any parameters passed to the callee and any values returned no longer need to be pushed onto the stack[1].

However, the greatest potential benefit of inlining comes from additional code simplification it may enable by bringing the callee's code into the caller's scope [1]. Many code analysis algorithms work within the scope of a single function; inter-procedural analysis is usually fundamentally more difficult, and always more computationally expensive than intra-procedural analysis, because of the increased scope. A function call inhibits the precision of analyses and is a barrier to code motion because the caller sees the callee as a "black box" with unknown effect.

C. Costs of Inlining

Inlining non-profitable call sites can indirectly produce negative effects. The increased scope provided for analysis by inlining also increases the costs of these analyses. Most algorithms used by compilers have super-linear time complexity. Extremely large procedures may take excessively long to analyze; some compilers will abort an analysis that takes too long. Furthermore, a program must be loaded into memory from disk before it can be executed. A larger executable file size increases a program's start-up time. Finally, developers eschew unnecessarily large program binaries because of the costs associated with the storage and transmission of large files for both the developer and their clients. Therefore, inlining that does not improve performance should be avoided.

D. Inlining-Invariant Program Characteristics

While inlining a call causes a large change in the caller's code, it has a minimal direct impact of the use of memory system resources at run time [1]. Ignoring the subsequent simplifications the inlining enables, inlining proper has no appreciable impact on register use, or data or instruction cache efficiency. Regardless of inlining, the same dynamic sequence of instructions must process the same data in the same order to produce the same deterministic program result.

Inlining should have negligible impact register spills. The additional variables introduced into the caller by inlining place additional demands on the register allocator, and may increase the number of register spills introduced into the caller. However, without inlining, the calling convention requires the caller to save any live registers before making a call, or for the callee to save any registers before it uses them; in both cases, these registers must be restored before resuming execution in the caller. Thus, inlining merely shifts the responsibility for register management from the calling convention to the register allocator.

Similarly, inlining does not change the data memory accesses of a program. Whether in the caller or the callee, the same loads and stores, in the same order, are required for correct computation. Subsequent transformations may reorder independent memory accesses to better hide cache latency, or eliminate unnecessary accesses altogether, but this is not a direct consequence of inlining. Thus, data cache accesses do not change with inlining, and nor does the cache miss rate.

[1]Some calling conventions allow values to pass between the caller and callee in registers.

III. Speedup measured using FDI

We have performed an experiment comparing the performance of the `LLVM` static inliner and the FDI inliner described in [1]. Both inliners are also evaluated with respect to the baseline Never, which means never inline. Our setting that was tested using the programs `bzip2` and `gzip`, was not extracted from the SPEC CPU 2006 benchmark suite, rather than we used the fully-functional "real" versions. Using the real versions of the compressor programs eliminates the unrealistically-simplified profiling situation where mutually-exclusive use cases are combined into a single program run. Consequently, these programs cannot do decompression and compression, or multiple levels of compression, within the same run. These distinct use-cases must be covered by different inputs in the program workload. Our results show a slight improvement over the `LLVM` results. For `gcc` we used the SPEC CPU 2006 benchmark suite. SPEC provides 11 inputs for `gcc` from those we selected 7, `166, Cp-decl, expr, expr2, g23, integrate, bzipR-all, lbm-all, mcf-all`. And we added 3 inputs converted from the SPEC 2000 benchmark, `bzip2`, `LBM`, and `mcf`.

The `bzip2` and `gzip` programs were executed using a representative set of inputs, where compression tasks and decompression tasks were tested under similar inputs. The compression set contains the following inputs, with the compression level shown in parentheses:

- `cards (-4)`: A collection of greeting card layouts in the TIFF (uncompressed) image format.
- `revelation-ogg (-8)`: The audio book "The Revelation of Saint John" in OGG format, from Project Gutenberg[2].

The decompression set for each compressor uses the same base set of files, pre-compressed by the appropriate compressor at the default compression level. The decompression set is composed of:

- `auriel`: The "Auriel's Retreat" land-mass addition mod by lance4791 for the game "The Elder Scrolls IV: Oblivion" from Bethesda Softworks[3].
- `ocal-019`: The Open Clip Art Library archive, version 0.19. The images are primarily in vector-graphics formats[4].
- `proteins-2`: A completely different sample of 157 proteins from the RCSB Protein Data Bank database, each in 6 different file formats.

A. Setting up the experiments

We ran our experiments on 20 Dell Optiplex 755, whose characteristics are:

[2]http://www.gutenberg.org/ebooks/22945
[3]http://planetelderscrolls.gamespy.com/View.php?view=OblivionMods.Detail&id=5949
[4]http://openclipart.org/collections

Input	FDO normalized	LLVM normalized	Speedup
166	0.9494	0.9733	0.9754
c-typeck	0.9097	0.9745	0.9335
Cp-decl	0.9554	0.9849	0.9700
expr	0.9035	0.9552	0.9458
expr2	0.8630	0.9660	0.8934
g23	0.9119	0.9849	0.9259
integrate	0.9811	1.0251	0.9570
bzipR-all	0.9870	1.0092	0.9780
lbm-all	0.8888	1.0000	0.8888
mcf-all	0.9487	1.0000	0.9487
Geomean			0.9412

TABLE I
SUMMARY OF THE DATA COLLECTED DURING THE EXPERIMENT WITH gcc

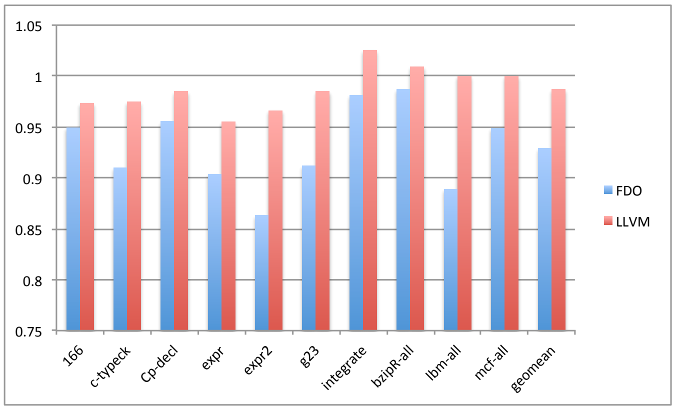


Fig. 1. Running times of the gcc inlined versions, normalized by Never

- Intel Duo Core E6750 2.66 GHz processor;
- 4 GB RAM;
- DVD-RW drive;
- Intel Pro/1000 Gb ethernet;
- Gigabyte GeForce 8600 video cards;
- 250 GB SATA II drive.

B. Presenting the results

We employed a single-run methodology for the experiments and we used three benchmarks to test our hypotheses, bzip2, gzip, and gcc. The results are presented in the following way, we grouped the bzip2 with gzip, and separate gcc to another section. We decided to present the results for bzip2 and gzip together because these programs have similar behavior and gcc has a completely different behavior.

1) Gcc Benchmark: For the gcc benchmark we were able to report a significant result, we measured a 5.88% speedup over LLVM and a 7.09% speedup over Never (no inlining), whereas LLVM achieved a 1.29% speedup over never, using the set of inputs described above. This result is summarized in Table I below. The results are normalized by the baseline Never (no inlining).

The Figure 1 shows that the FDI inliner outperforms Never and LLVM through all the inputs, which explains our speedup. But this particular experiment shows a good result, for the speedup is somewhat high.

Nevertheless, we can show how we can improve this result by just choosing less inputs from the original input set. In this case we can report a speedup of 7.76% over LLVM, as shown in Table II and Figure 2.

Input	FDO normalized	LLVM normalized	Speedup
c-typeck	0.9097	0.9745	0.9335
expr	0.9035	0.9552	0.9458
expr2	0.8630	0.9660	0.8934
g23	0.9119	0.9849	0.9259
lbm-all	0.8888	1.0000	0.8888
mcf-all	0.9487	1.0000	0.9487
Geomean			0.9224

TABLE II
EXTRACT OF THE DATA COLLECTED DURING THE EXPERIMENT WITH gcc

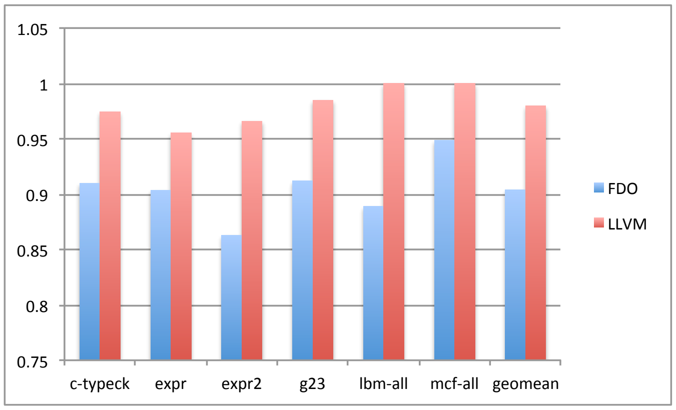


Fig. 2. Extract of the running times of the gcc inlined versions, normalized by Never

2) Compressors / Decompressors: We will start by describing our experiments with the data collected from the bzip2 runs are summarized in Table III. In this table we show that we achieved a slight speedup of 1.36% over LLVM results, and 1.40% over Never (no inlining), whereas LLVM achieved a speedup of 0.04% over Never.

Figure 3 shows the running time normalized by the time of Never (no inlining). We can see that the FDI inliner outperforms Never and LLVM through all the inputs, the same way the former experiments did. The experiment on the program bzip2 shed some light and these results can be fully explored in future research.

The final speedup, despite being a slight improvement, represents that the FDI inliner can actually be employed instead of the LLVM inliner. And this result is significant because the program bzip2 is small, simple, and not particularly fitted to inlining, leading to a conjecture that FDI inliner are better than static ones. Which opens a wide range of experiments with other programs to confirm this conjecture. We ran also with gzip, which happens to be quite similar to bzip2, and confirmed a speedup of 2.09% over LLVM results, and

Input	FDO time (sec)	Never time (sec)	LLVM time (sec)	Speedup
auriel	7.66	7.88	7.94	0.9647
cards	29.52	29.79	29.76	0.9919
ocal	44.94	44.99	45.33	0.9913
proteins-2	79.6	81.11	80.71	0.9862
revelation	5.24	5.31	5.25	0.9980
Geomean				0.9864

TABLE III
SUMMARY OF THE DATA COLLECTED DURING THE EXPERIMENT WITH bzip2

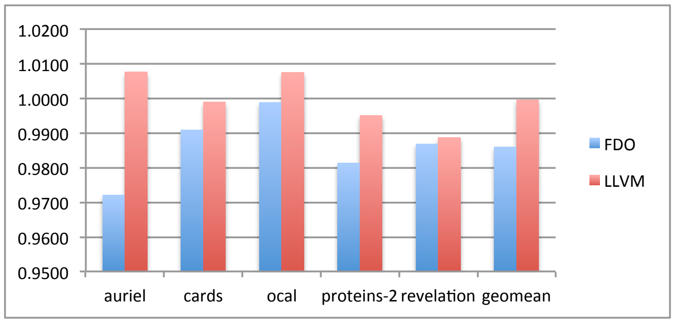


Fig. 3. Running times of the `bzip2` inlined versions, normalized by Never

Input	FDO normalized	LLVM normalized	Speedup
auriel	0.9924	0.9924	1.0000
cards	0.9801	1.0092	0.9712
ocal	0.9914	1.0122	0.9794
proteins-2	0.9905	1.0094	0.9811
revelation	0.9708	1.0072	0.9637
Geomean			0.9790

TABLE IV
SUMMARY OF THE DATA COLLECTED DURING THE EXPERIMENT WITH `gzip`

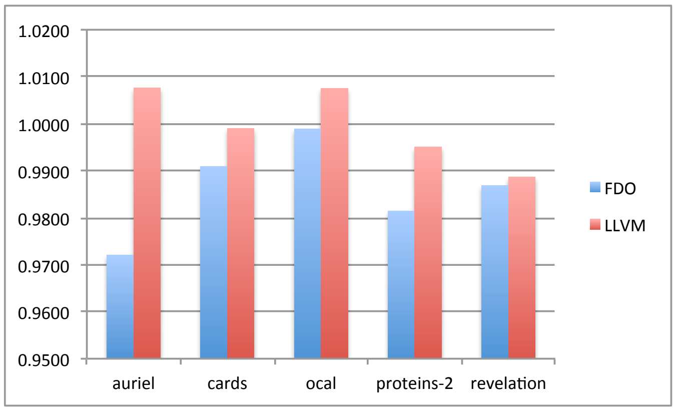


Fig. 4. Running times of the `gzip` inlined versions, normalized by Never

Input	Normalized FDO	Normalized LLVM	Slowdown
auriel	0.9961	1.0025	0.9936
cards	1.0457	0.9882	1.0581
ocal	1.0035	0.9984	1.0051
proteins-2	1.0012	0.9931	1.0080
revelation	1.0359	0.9905	1.0458
Geomean			1.0218

TABLE V
DATA REFLECTING A SLOWDOWN ON `bzip2` – COLLECTED BY OTHER RESEARCH GROUP

Input	Normalized FDO	Normalized LLVM	Slowdown
auriel	1.1278	1.0000	1.1278
cards	1.0052	1.0079	0.9973
ocal	1.1234	0.9987	1.1248
proteins-2	1.0706	1.0081	1.0620
revelation	1.0072	1.0000	1.0072
Geomean			1.0624

TABLE VI
DATA REFLECTING A SLOWDOWN ON `gzip` – COLLECTED BY OTHER RESEARCH GROUP

a speedup of 1.50% over Never (no inlining) and LLVM got a slowdown of 0.61%. These results can be seen in Table IV, where the times are already normalized by the baseline Never (no inlining). Figure 4 shows the normalized running time for `gzip`, and it also outperforms Never and LLVMthrough all inputs.

The results of the experiment are also consistent with other similar findings in the literature, whereas employing single-run experiments does not generate any kind of disturbance in the analysis, and the speedup result are statistically sound. So we can confirm a speedup over the static inliner for the `bzip2` and `gzip` cases.

3) A slowdown: We have to report also that the same experiment was carried out by another group, but unfortunately this group could not confirm our results. The differences in the timing measurements could be considered relevant, and both groups are now trying to understand what went wrong with the latter experiment. They measured a slowdown of 2.14% for `bzip2`, and a slowdown of 5.87% for `gzip`, and a slowdown of 2.06% for `gcc`,as shown in Table V, Table VI, and Table VII.

IV. STATISTICAL CONSIDERATIONS ON SPEEDUPS AND SLOWDOWNS

Every experimental science suffer from the same problem, evaluation of the data already collected. Even the simple idea of collecting data can become a painful task, because the measurement process may introduce errors, or cause distortion in the data. But we are now concerned in analyzing the data already collected.

A measure has itself noise, that may have come from the processes running in background, the operating system calls, interruptions, memory allocation, and other sources of noise, including the measurement process itself. Hence, one of the most important tasks is to have a good understanding of the noise that our system has [2].

This is an important issue, because we know that there is noise, that we cannot avoid it, but we can cope with it. In [2], we may see that the majority of the experimental studies lack a rigorous statistical methodology, which their work tries to fill in. In our case, we also are worried with such issues and we also proposed another way to evaluate and collect data, the CP methodology citeBerubePhD. Therefore we chose to use the initial steps proposed in [2], to further characterize the noise in our experimental data.

Input	FDO normalized	LLVM normalized	Speedup
166	0.9755	0.9755	1.0000
c-typeck	0.9845	0.9845	1.0000
Cp-decl	0.9784	0.9784	1.0000
expr	0.9686	0.9567	1.0124
expr2	0.9686	0.9686	1.0000
g23	1.0574	1.0441	1.0127
integrate	1.0253	1.0000	1.0253
bzipR-all	1.0315	1.0055	1.0258
lbm-all	1.0909	1.0303	1.0588
mcf-all	1.1081	1.0270	1.0789
Geomean			1.0210

TABLE VII
DATA REFLECTING A SLOWDOWN ON `gcc` – COLLECTED BY OTHER RESEARCH GROUP

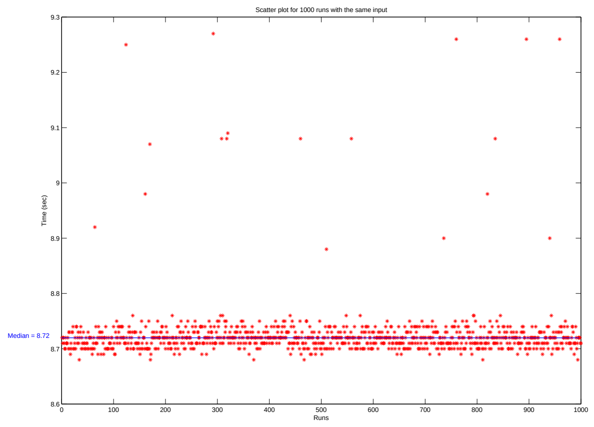
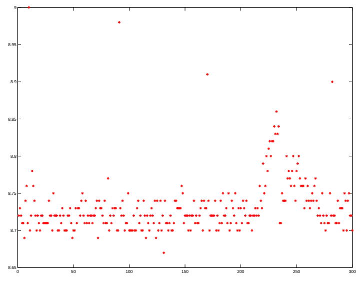


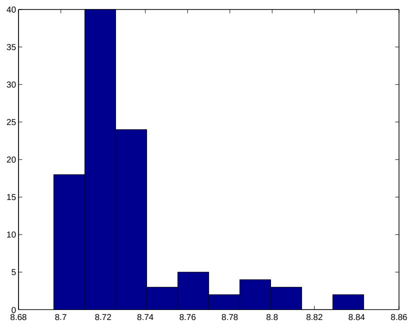
Fig. 5. Running 1000 times the same program with the same input data

Length	Mean	Median	StD Mean	StD Median
10	8.7160	8.7150	0.0100	0.0050
100	8.7328	8.7200	0.0187	0.0100
1000	8.7248	8.7200	0.0197	0.0100

TABLE VIII
SIMPLE STATISTICS ON THE EXPERIMENT



(a) 100-time runs of the 3-consecutive execution of input `ebooks` for program `bzip2`



(b) Histogram for the `auriel` input

Fig. 6. 100-times running 3-consecutive experiment

To show the gaussian variation in the data we collect, Figure 5 depicts a scatter plot of 1000 sequential runs of the program `bzip2`, after being compiled using the Static inliner (LLVM) whose input was `ebooks`. The figure shows a gaussian noise around the median plus a few outliers, generated by the operating system regular use. These outliers are filtered off from the data we are using. They are easily discarded because they have much more variance (more than one deviation from the median).

We ran three independent experiments, the first is the 10-times experiment, then 100-times and, after that, 1000-times, because we needed to confirm that there is no difference on their means, and also that we can discard the outliers. To make sure that we have robust measures, we ran some simple statistics to know the mean, the median, the standard-deviation from the mean (std-mean), and the standard-deviation from the median (std-median), shown in Table VIII. We also ran t-tests on each sample pairs to verify if their means were the same, the results for the t-tests are shown in Table IX below.

The t-tests in Table IX show that the null hypothesis cannot be discarded, as the value 0 in each line of the *t-test* column confirms. The *p-values* illustrate the confidence in the hypothesis, in this case, that the means are different.

Our experiments have also shown that the variance when running the same data just three times in a row is not quite different from the one running 100 times. When we consider each 'input-run' a 3-consecutive run – which means we ran 300 times the same experiment –, and we consider a 'full-run' as running a 3-consecutive run to each input. We ran 100 'full-runs' in this experiment, and we put some extra noise at its end.

Runs	t-test	p-value
(10-100)	0	0.3424
(10-1000)	0	0.6025
(100-1000)	0	0.1528

TABLE IX
T-TESTS APPLIED PAIRWISE TO THE 10, 100, AND 1000 RUNS

What this means is that even though the effect of the noise can mask the correct values, we can treat them in order to assure robustness. This is the way we employed to empirically verify the soundness of the CP methodology. As Figure 6 below shows, the deviation from the mean is not large, but here is a subtle knob increasing the running time of the all programs in the experiment by its end. It was caused by the execution of another system at the same time competing for the same resources. In (Figure 6(a)) we depicted in the y-axis the running time for each program at each 3-consecutive run, which are depicted on the x-axis. It can be also visualised in the histogram (Figure 6(b)). These figures show the 3-consecutive run for the input data `ebooks`, where in the x-axis we depicted the running time for the program and on the y-axis we depicted the number of runs at each bin.

The figures Figure 6 and Figure 5 show that collecting data from single execution can produce erroneous results, even using machines with no other running program, we still have some noise due to operating system activities, interruptions, etc. And also that a simple inclusion of a simple job during the running cycle can perturb the execution time, as can be

Run	Mean	Median	StD Mean	StD Median
1	8.7233	8.72	0.0044	
2	8.71	8.71	0.0067	0.01
3	8.72	8.73	0.02	0.01
4	8.7067	8.7	0.0089	0.00
5	8.71	8.71	0.0067	0.01
6	8.7933	8.74	0.0778	0.01
7	8.73	8.73	0.0067	0.01
8	8.7233	8.71	0.0178	0.00
9	8.73	8.73	0.0067	0.01
10	8.7033	8.71	0.0089	0.00
33	8.71	8.71	0.0067	0.01
34	8.7267	8.73	0.0044	0.00
35	8.71	8.7	0.0133	0.00
36	8.81	8.73	0.1133	0.01
37	8.72	8.72	0.0133	0.02
70	8.72	8.71	0.0133	0.00
71	8.7133	8.72	0.0089	0.00
72	8.7233	8.72	0.0044	0.00
73	8.7233	8.72	0.0044	0.00
74	8.743333	8.74	0.0111	0.01
75	8.7667	8.76	0.0156	0.01
76	8.7967	8.8	0.0111	0.01
77	8.8133	8.82	0.0089	0.00
78	8.83	8.83	0.0067	0.01
79	8.8433	8.84	0.0111	0.01
80	8.74	8.74	0	0.00
81	8.7833	8.78	0.0111	0.01
82	8.77	8.77	0.0067	0.01
83	8.7667	8.76	0.0222	0.02
84	8.79	8.79	0.0067	0.01
85	8.7633	8.76	0.0044	0
86	8.7533	8.76	0.0156	0.01
87	8.7467	8.74	0.0089	0.00
88	8.74	8.74	0.0067	0.01
89	8.7567	8.76	0.0111	0.01
90	8.7267	8.72	0.0156	0.01
91	8.71	8.71	0.0067	0.01
92	8.7133	8.71	0.0044	0
93	8.79	8.75	0.0733	0.03
94	8.7167	8.72	0.0044	0
95	8.72	8.71	0.0133	0
96	8.73	8.73	0.00	0.00
97	8.73	8.74	0.02	0.01
98	8.73	8.74	0.02	0.01
99	8.7133	8.72	0.0089	0
100	8.7367	8.74	0.0178	0.02

TABLE X
DEVIATION FROM THE MEAN AND FROM THE MEDIAN IN THE EXPERIMENT

Runs	t-test	p-value
1	0	0.706108
2	0	0.328462
3	0	0.598565
4	0	0.259765
5	0	0.328462
6	1	0.006947
7	0	0.938929
8	0	0.706426
9	0	0.938929
10	0	0.201735
33	0	0.328462
34	0	0.820524
35	0	0.328682
36	1	0.00085
37	0	0.598316
70	0	0.598233
71	0	0.408107
72	0	0.706108
73	0	0.706108
74	0	0.600263
75	0	0.116071
76	1	0.003654
77	1	0.000274
78	1	0.000013
79	1	0.000001
80	0	0.70832
81	1	0.02056
82	0	0.085091
83	0	0.116484
84	1	0.008985
85	0	0.154594
86	0	0.330314
87	0	0.500169
88	0	0.708384
89	0	0.261142
90	0	0.820684
91	0	0.328462
92	0	0.408
93	1	0.010463
94	0	0.498166
95	0	0.598233
96	0	0.938915
97	0	0.939012
98	0	0.939012
99	0	0.408107
100	0	0.823099

TABLE XI
TEST ON THE MEANS

observed by the knob in Figure 6.

The robustness is achieved when we can statistically assure that the variance on the data is not large. The data used in the experiment are shown in Table X, and the deviations from the mean (and median) to each 3-consecutive run are summarized as the average, minimum, and maximum values, all found on the 300-times experiment.

We also ran the t-tests to confirm that the means are statistically representing the same distribution. This is summarized in Table XI below. We can see very little outliers, except for knob region, because the runtime was being raised during certain amount of time forcing a gradient increasing the time values, and after it what happened was the other way around, decreasing the time values. Both tables Table X and Table XI are shown for the runs.

This experiment brought us confidence in the machine learning method we devised to tune-in the compiler parameters. We considered the possibility of increasing the number of times each individual run need to be performed, in order to achieve low variance in the data; hence we could trust the results. As this experiment has shown, the 3-consecutive run is a good choice, because it does not penalize much the total running time. Also, was shown that single-run testbeds are error-prone because they doesn't take the variance into account.

A. Analyzing the speedup results

In our framework, each program is evaluated using a 15-input workload, as suggested in [1]. `Gcc` is taken from the SPEC CPU 2006 benchmark suite. SPEC provides 11 inputs for `gcc`. In spite of the challenges involved in creating new inputs for this benchmark, four[5] of the SPEC 2000 benchmark programs were converted to the single pre-processed file format. The converted programs are `bzip2`, `LBM`, `mcf`, and `parser`.

As mentioned in Section III, for `bzip2` and `gzip` we did not extract the programs and inputs from the SPEC CPU 2006 benchmark suite, rather than we used the fully-functional "real" versions. The proper original input set for `bzip2` and `gzip` is:

The compression set contains the following inputs, with the compression level shown in parentheses:

- `avernum (-3)`: The installer for the demo version of the game "Avernum: Escape from the Pit" from Spiderweb Software.
- `cards (-4)`: A collection of greeting card layouts in the TIFF (uncompressed) image format.
- `ebooks (-5)`: A collection of ebooks, with and without images, and in a variety of formats, from Project

[5]of seven attempts

Gutenberg[6].

- `potemkin-mp4 (-6)`: The 1925 movie “Bronenosets Potyomkin (Battleship Potemkin)” in MP4 format, from the Internet Archive[7].
- `proteins-1 (-7)`: A sample of 33 proteins from the RCSB Protein Data Bank database. 6 files for each protein, each stored in a different text-based format, provide different characteristics of the protein’s structure[8].
- `revelation-ogg (-8)`: The audio book “The Revelation of Saint John” in OGG format, from Project Gutenberg[9].
- `usrlib-so (-9)`: A collection of shared object (.so) files from `/usr/lib/` of a 32-bit gentoo-linux machine.

The decompression set for each compressor uses the same base set of files, pre-compressed by the appropriate compressor at the default compression level. The decompression set is composed of:

- `auriel`: The “Auriel’s Retreat” land-mass addition mod by lance4791 for the game “The Elder Scrolls IV: Oblivion” from Bethesda Softworks[10].
- `gcc-453`: The source-code archive of the `gcc` compiler, version 4.5.3[11].
- `lib-a`: A collection of library files (.a) from `/lib/` of a gentoo-linux machine. As per the gentoo development guide, a library will be installed in `/lib` (boot critical) or `/usr/lib` (general applications), but not both[12].
- `mohicans-ogv`: The 1920 movie “Last of the Mohicans” in OGV (ogg video) format, from the Internet Archive[13].
- `ocal-019`: The Open Clip Art Library archive, version 0.19. The images are primarily in vector-graphics formats[14].
- `paintings-jpg`: A collection of watercolor paintings, in JPG format.
- `proteins-2`: A completely different sample of 157 proteins from the RCSB Protein Data Bank database, each in 6 different file formats.
- `sherlock-mp3`: The audio book “The Adventures of Sherlock Holmes” in MP3 format, from Project Gutenberg[15].

1) Compressor / Decompressor: After analyzing the inlining environment and having the confidence that we could trust our results, we decided to run the experiment using the program `bzip2`, and we collected data from the same setup (hardware and software) in 18 different settings. Figure 7 shows the data we collected. The vertical axis shows the normalized execution geometric mean time for each setting, the baseline is Never (no inlining), and the horizontal axis shows the settings organized by number. The red “*” represent the normalized geomean time of the FDO inlined program, and the blue “o” represent the normalized geomean for `LLVM` inlined program.

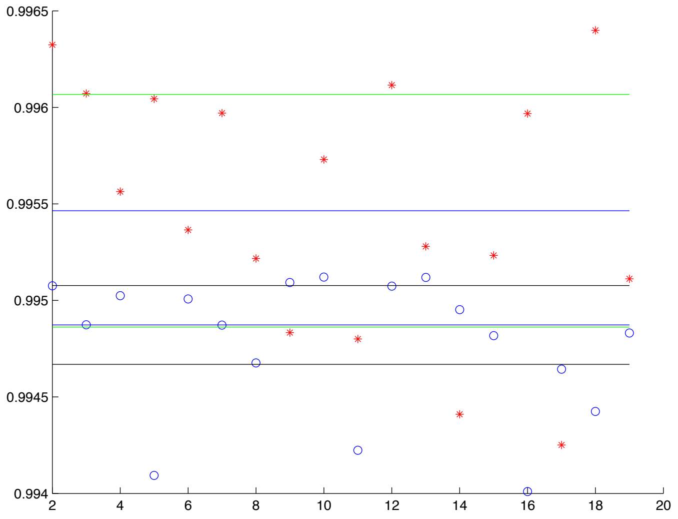


Fig. 7. The 18 different settings for `bzip2` of the same setup

The blue lines in the figure show each median value for the geometric means, the green lines represent one standard deviation from the median for the FDI case, while the black lines represent the standard deviation from the median for the `LLVM` case. As it can be seen, not only the values are too similar, varying only from the fourth decimal digit, but also the medians and their standard deviations overlap, collapse. This is a strong indicator that there is no significant difference between those measures.

So, what we did was to consider that a single-run experiment could have measured any of the strings individually, moreover, a single run may have also collected the best, or the worst values for the actual times of the experiment. Hence, to show a speedup for FDI we collected the worst running time for `LLVM` inlined program, and the best running time for the FDI inlined program.

Even though the data showed a speedup, it was really worthless, only 0.46%. Therefore, as we were not following the SPEC Benchmark suite, we made an ”adjustment”, we left the slowdowns and some of the tiny speedups gathered from the list of inputs outside the final list to be shown. This way we were able to present a tiny, but possibly measurable speedup, as in Section III. To apply a list of inputs can be an issue, as this “speedup” shows, that is why a complete list of inputs containing all the explanations is a requirement when presenting data. The full data for the ”speedup” experiment are shown in table Table XII.

On the other hand, in Section III-B3 we did the opposite, we chose the worst individual running time for the FDI

[6]http://www.gutenberg.org
[7]http://archive.org/details/BattleshipPotemkin
[8]http://www.rcsb.org
[9]http://www.gutenberg.org/ebooks/22945
[10]http://planetelderscrolls.gamespy.com/View.php?view=OblivionMods.Detail&id=5949
[11]http://gcc.gnu.org/gcc-4.5
[12]http://devmanual.gentoo.org/general-concepts/filesystem/index.html
[13]http://archive.org/details/last_of_the_mohicans_1920
[14]http://openclipart.org/collections
[15]http://www.gutenberg.org/ebooks/28733

Input	Normalized FDO	Normalized LLVM	Speedup
auriel	0.9720	1.0076	0.9647
avernum	0.9922	0.9905	1.0017
cards	0.9909	0.9989	0.9919
ebooks	0.9909	0.9920	0.9988
gcc	0.9966	1.0059	0.9907
lib-a	0.9940	0.9970	0.9970
mohicans	1.0000	1.0048	0.9951
ocal	0.9988	1.0075	0.9913
paintings	1.0000	1.0051	0.9949
potemkin	0.9916	0.9887	1.0029
proteins-1	0.9977	0.9910	1.0068
proteins-2	0.9813	0.9950	0.9862
revelation	0.9868	0.9887	0.9980
sherlock	1.0000	1.0020	1.0125
usrlib	1.0000	0.9875	1.0458
Speedup			0.9953 (0.46 %)

TABLE XII
SUMMARY OF THE NORMALIZED DATA USED TO PRODUCE A SPEEDUP FOR `bzip2`

Input	FDO normalized	LLVM normalized	Speedup
166	0.9494	0.9733	0.9754
200	0.9617	0.9735	0.9879
c-typeck	0.9097	0.9745	0.9335
cccp	0.9650	0.9700	0.9948
Cp-decl	0.9554	0.9849	0.9700
expr	0.9035	0.9552	0.9458
expr2	0.8630	0.9660	0.8934
g23	0.9119	0.9849	0.9259
integrate	0.9811	1.0251	0.9570
s04	0.9886	1.0181	0.9710
scilab	0.9945	1.0043	0.9902
bzipR-all	0.9870	1.0092	0.9780
lbm-all	0.8888	1.0000	0.8888
mcf-all	0.9487	1.0000	0.9487
parser-all	0.9945	1.0364	0.9596
Geomean			0.9541 (4.58 %)

TABLE XIII
SUMMARY OF THE NORMALIZED DATA USED TO PRODUCE A SPEEDUP FOR `gcc`

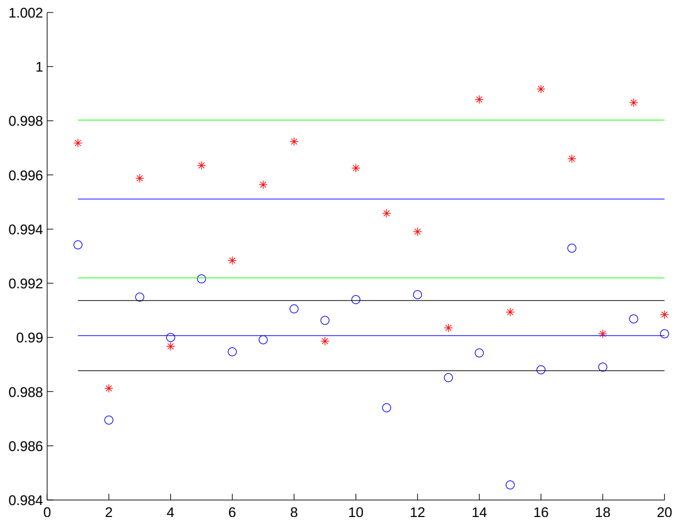


Fig. 8. The 20 different settings for `gzip` of the same setup

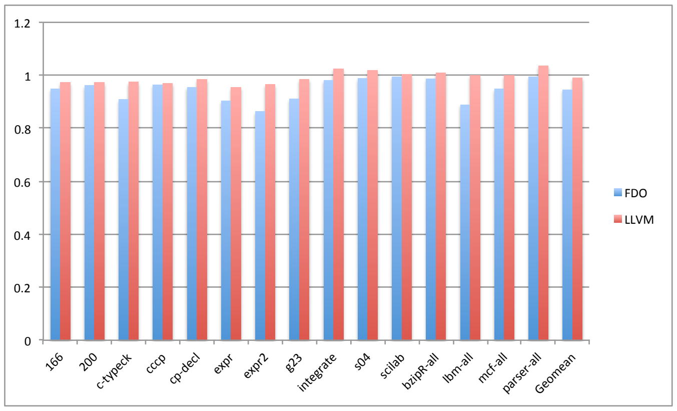


Fig. 9. The 20 different settings for `gcc` of the same setup

inlined program and the best running time for the LLVM inlined program. Proceeding this way we showed that another individual measuring showed a slowdown. And as both results followed the same methodology, they are both correct, and this is unexplainable unless we consider that there is variance on the data.

We used the same process for the `gzip` case using 20 different settings, the Figure 8 shows the data collected in a similar way of Figure 7. In Figure 8 we can notice that there is actually a slowdown compared to LLVM for this setup, and even though we were able to report a speedup.

These cases were artificially constructed using our empirical actual data, considering that if we used a single-run methodology these results could appear. But as we were using CP methodology, we were able to correctly identify that there is no statistical difference between both inliners, for the setup proposed. This result, in a certain way, reinforces the result of [5], where they reported no speedup of $-O2$ over $-O3$ for all benchmarks they analyzed.

2) Analysis of `gcc`*:* The same process was used for the `gcc` case, we selected the best running times for the setting we had at each input, and from this we constructed our speedup experiment. In a single-run framework it is perfectly reasonable that this result can actually appear. But in this case we came up with a result considering a reduced input set, which produced an even better speedup. This was done to raise the question about the proper set of inputs to be employed. In fact, the set we presented in Section III was already an extract from the SPEC CPU 2006 benchmark suite.

In reality we applied the full set and added 4 inputs, which were converted from the SPEC 2000 benchmark, `bzip2`, `LBM`, `mcf`, and `parser`, fulfilling the 15-input set to each program. If we considered the full input-set we would have a different result, the speedup in the case of the best run-times, would be of 4.58%, as shown in Table XIII and in Figure 9.

Therefore, the input-set matters, as much as a sound methodology. To summarize this section we end this with a real outcome of our framework, where we can observe that there was not any speedups, or slowdowns for the case of `gcc`, as can be easily seen from the error bars present in Figure 10. This figure is automatically generated by our system, and reflects the geometric mean of all inputs for twelve different FDI inliners, the LLVM inliner (called static in the figure) and another static inliner called benefit.

In the next section (Section V) we describe in more detail the CP methodology, explaining its use and how we measure the results, in order to avoid the problems highlighted by the example in Section III.

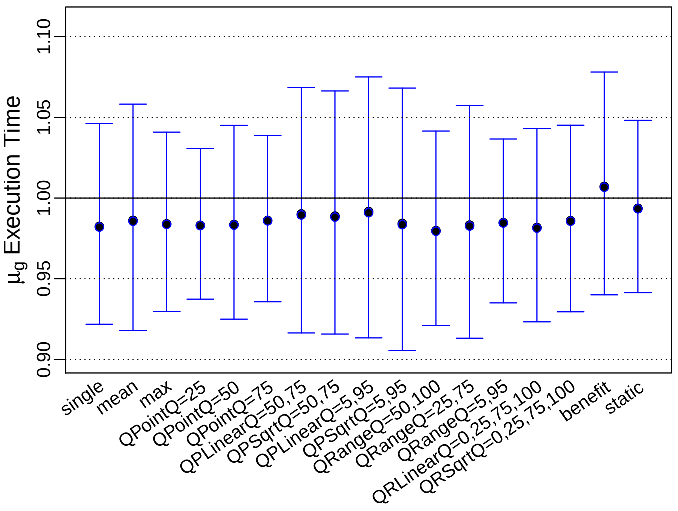


Fig. 10. The actual result for `gcc` returned by our CP framework

V. Combined Profiling Methodology

Capturing behavior variations across inputs is important in the design of an FDO compiler. A number of speculative code transformations are known to benefit from FDO, including speculative partial redundancy elimination [6], [7], trace-based scheduling and others [8], [9].

This section argues that the behavior variations in an application due to multiple inputs should be evaluated by FDO decisions. It also argues that a full parametric estimation of a statistical distribution is not only unnecessary, but it may also mislead FDO decisions if the wrong distribution is assumed or there is insufficient data to accurately estimate the parameters.

A major challenge in the use of traditional single-training-run FDO is the selection of a profiling data input that is representative of the execution of the program throughout its lifetime. For large and complex programs dealing with many use cases and used by a multitude of users, assembling an appropriately representative workload may be a difficult task. Picking a solitary training run to represent such a space is far more challenging, or potentially impossible, if use-cases are mutually-exclusive. While benchmark programs can be modified to combine such use-cases into a single run, this approach is obviously inapplicable to real programs. Moreover, user workloads are prone to change over time. Ensuring stable performance across all inputs in today's workload prevents performance degradation due to changes in the relative importance of workload components.

The *Combined Profiling* (CP) statistical modeling technique presented in [1] produces a *Combined Profile* (CProf) from a collection of traditional single-run profiles, thus facilitating the collection and representation of profile information over multiple runs. The use of many profiling runs, in turn, eases the burden of training-workload selection and mitigates the potential for performance degradation. There is no need to select a single input for training because data from any number of training runs can be merged into a combined profile. More importantly, CP preserves variations in execution behavior across inputs. The distribution of behaviors can be queried and analyzed by the compiler when making code-transformation decisions. Modestly profitable transformations can be performed with confidence when they are beneficial to the entire workload. On the other hand, transformations expected to be highly beneficial on average can be suppressed when performance degradation would be incurred on some members of the workload.

Combining profiles is a three-step process [3]:

1) Collect raw profiles via traditional profiling.
2) Apply *Hierarchical Normalization* (HN) to each raw profile.
3) Apply CP to the normalized profiles to create the combined profile.

CP and HN have been presented in previous work [10], [3]. However, a clearer and expanded version, based on previous versions, can be found in [1], particularly the description of CP's histograms and the discussion of queries.

CP [1] provides a data representation for profile information, but does not specify the semantics of the information stored in the combined profile. Raw profiles cannot be combined naively.

A. Hierarchical Normalization

There is a problem when pairs of measurements are taken under different conditions. Thus, when combining these measurements, all values recorded for a monitor must be normalized relative to a common fixed reference. *Hierarchical normalization* (HN) [1] is a profile semantic designed for use with CP that achieves this goal by decomposing a CFG into a hierarchy of dominating regions.

HN is presented for edge profiling. Vertex profiles are treated identically, but use the domination relationships between vertexes instead of edges. Domination is usually defined in terms of vertexes. In order to use an existing implementation of a vertex dominator-tree algorithm with edge profiles, use the line graph of the CFG instead of the CFG itself. The line graph contains one vertex for each edge in the CFG, and edges in the line graph correspond to adjacencies between the edges of the CFG.

B. Denormalization

The properties of a monitor R_a can only be directly compared to those of a monitor R_b when $dom(a) = dom(b)$. However, more generalized reasoning about R_a may be needed when considering code transformations. Similarly, when code is moved by a transformation, its profile information must be correctly updated. *Denormalization* reverses the effects of hierarchical normalization to lift monitors out of nested domination regions by marginalizing-out the distribution of the dominators above which they are lifted. Denormalization is a heuristic method rather than an exact statistical inference because it assumes statistical independence between monitors.

C. Queries

In an AOT compiler, profiles are used to predict program behavior. Thus, raw profiles are statistical models that use a single sample to answer exactly one question: *"What is the expected frequency of X?"* where X is an edge or path in a CFG or a Call Graph (CG). A CP is a much richer statistical model that can answer a wide range of queries about the measured program behavior. The implementation of CP used in this work provides the following statistical queries as methods of a monitor's histogram:

H.min; H.max
H.mean($incl0s$)
H.stdev($incl0s$)
H.estProbLessThan(v)
H.quantile(q)
H.applyOnRange($F(w, v)$, $vmin$, $vmax$)
H.applyOnQuantile($F(w, v)$, $qmin$, $qmax$)
H.coverage
H.span

CP enables the accurate assessment of the potential performance impact of transformations informed by variable-behavior monitors in a variety of ways, and with adjustable confidence in the result. Concrete examples of this kind of analysis are provided by the implementation of an FDO inliner using CP described in [1].

D. Alternative Usage

The empirical-distribution methodology of CP is orthogonal to the techniques used to collect raw profiles. CP is applicable whenever multiple profile instances are collected, including intra-run phase-based profiles, profiles collected from hardware performance-counter, and sampled profiles. The main issue when combining profiles is how normalization should be done in order to preserve program-behavior characteristics.

VI. Related Work

There are several researchers concerned with the problem of reliability in performance measures. Kalibera *et al.* [2] propose a rigorous methodology for measuring time, and claim that the measurements are still done in reasonable time. Their methodology considers that the environment, consisting of hardware and software, versions of the operating system, versions of the compiler used to measure data, they all change scarcely. For this reason their methodology asserts that before starting to take any measurement the whole environment has to be deeply investigated to find how many repeated iterations are required to achieve an independent state (the execution times of benchmark iterations are statistically independent). They provide means to calculate the number of runs are needed to achieve independent states for a benchmark analysis, also for measuring speedups. They used different benchmarks in their experiments and showed that there are different number of repetition counts for them. Our methodology does not assume that the environment changes scarcely, and we don't need a huge number of repetitions.

Mytkowicz *et al.* [11] ran some experiments using SPEC CPU benchmarks and found significant systematic measurement errors in some sources, that could produce biased results. Their suggestion is to randomise the experimental setup to eliminate the bias. The idea of ramdomising is fully incorporated in Stabiliser [5]. Stabiliser is an LLVM-based compiler and runtime environment for randomisation of code, stack and heap layout. The purpose of randomisation is to reduce the need for repeated execution. Randomising the whole program in fact introduces more variation than in real systems, also some compiler transformations can become useless. Our approach is much less intrusive than theirs and we don't break compiler transformations.

Georges *et al.* [12] shows that different methodologies can lead to different conclusions. They work with Java benchmarks and recommends running multiple iterations of each Java benchmark within a single VM execution, and also multiple VM executions. Our work is not focused in Java, but their recommendation remains true, it is necessary to use a reliable experimental methodology.

A. *FDO-related*

Most compilers take a single-run approach to FDO: a single training run generates a profile, which is used to guide compiler transformations. Some profile file formats support the storage of multiple profiles (*e.g.*, `LLVM`), but when such a file is provided to a compiler, either all profiles except the first are ignored, or a simple sum or average is taken across the frequencies in the collected profiles.

Input characterization and workload reduction are not new problems. However, the similarity metrics used for clustering in [1] are unique in their applicability to workload reduction for an FDO compiler. Most input similarity and clustering work is done in the area of computer architecture, where research is largely simulation-based, thus necessitating small workloads of representative programs using minimally-sized inputs. The architectural metrics of benchmark programs are repeatedly scrutinized for redundancy, while smaller inputs are compared with large inputs. Alternatively, some work bypasses program behavior and examines the inputs directly.

Arnold *et al.*. present an inlining strategy similar to that used in modern compilers [13]. They use a call-site sensitive call graph profile, thus allocating procedure executions frequencies to individual call sites. Using code size expansion as the cost and call site frequency as the benefit, call sites are inlined in decreasing cost/benefit order up to a code expansion limit. They find that a 1% code size expansion limit accounts for 73% of dynamic calls and reduces execution time by 9% to 57%.

Arnold *et al.*. use histograms to combine the profile information collected by a Java JIT system over multiple program runs [14]. The online profiler detects hot methods by periodically sampling the currently-executing method. After each run of a program, histograms for the hot methods stored in a profile repository are updated.

Salverda *et al.* model the critical paths of a program by generating synthetic program traces from a histogram of profiled branch outcomes [15]. To better cover the program's footprint, they do an ad-hoc combination of profiles from SPEC training and reference inputs. In contrast, combined profiling and hierarchical normalization provide a systematic method to combine profile information for multiple runs.

Savari and Young build a branch and decision model for branch data [16]. Their model assumes that the next branch and its outcome are independent of previous branches, an assumption that is violated by computer programs (*e.g.*, correlated branches). One distribution is used to represent *all events* from a run; distributions from multiple runs are combined using relative entropy — a sophisticated way to find the weights for a weighted geometric average across runs. The model cannot provide specific information about a particular branch, which is exactly the information needed by FDO. However, this information is provided by combined profiles because each event is represented separately.

VII. Conclusion

As mentioned in Section I we proposed a case study was for the inlining transformation, and we compared the CP process with the single-run process. We could have chosen any other transformation, because the CP methodology can be applied in all general cases.

In Section III we showed a erroneous speedup measured by a single-run experiment. It was constructed considering that any of the measurements that we ran independently could have happened in a single-run experiment. Hence we searched the collected data trying to find some outliers, or at least some data extreme points. We gathered these data points and defined two specific cases: `Best-runtime` and `Worst-runtime` for our FDO-based inliner, and for a static inliner, in this case the `LLVM` inliner.

With these data points we were able to select the pairs to create the illusion of a speedup and a slowdown:

- `Best-runtime` for FDO and `Worst-runtime` for `LLVM`, creating a speedup;
- `Worst-runtime` for FDO and `Best-runtime` for `LLVM`, creating a slowdown.

With these pairs we could produce a statistical analysis showing a speedup (or slowdown), and we devised these pairs as being representative cases of single-run experiments. Therefore, each pair (speedup or slowdown) can be viewed as a result of a single-run experiment. Even if the researcher is extremely cautious the methodology is error-prone, a bias can be introduced without the knowledge, or intention, of the researcher. So the real message is to define and use a reliable methodology based on solid statistical measurements.

With our experiments we were able to answer some of the open questions posed in the Section I. We know for sure, and showed it in Section IV, that FDI decisions can be more accurate using CP instead of single-run evaluation. For the case of the impact of CP in a controlled case study, we can definitely state that as we run each program more than once, that's the price we have to pay for more reliability, but the impact is acceptable if the number of repetitions is not too high. In our experiments running three times was enough.

A. Future work

For future work our plans can be divided in two different paths:

- *Fine-tuning* Using the CP methodology fine tune our FDI inliner for some different benchmarks. We have already finished some experiments and we are now defining some changes in our algorithms;
- *Apply CP* We are devising a research path whereas we will apply CP to different compiler transformations.

References

[1] PaulBerube, "Methodologies for many-input feedback-directed optimization," Ph.D. dissertation, University of Alberta, 2012.

[2] T. Kalibera and R. Jones, "Rigorous benchmarking in reasonable time," in *Proceedings of the 2013 international symposium on International symposium on memory management*, ser. ISMM '13. New York, NY, USA: ACM, 2013, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/2464157.2464160

[3] P. Berube and J. N. Amaral, "Combined profiling: A methodology to capture varied program behavior across multiple inputs," in *Intern. Symp. on Performance Analysis of Systems and Software (ISPASS)*.

[4] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Code Generation and Optimization (CGO)*, San Jose, CA, USA, March 2004.

[5] C. Curtsinger and E. D. Berger, "Stabilizer: statistically sound performance evaluation," *SIGPLAN Not.*, vol. 48, no. 4, pp. 219–228, Mar. 2013. [Online]. Available: http://doi.acm.org/10.1145/2499368.2451141

[6] F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu, "A new algorithm for partial redundancy elimination based on SSA form," in *Conference on Programming Language Design and Implementation (PLDI)*, Las Vegas, NV, USA, 1997, pp. 273–286.

[7] R. Gupta, D. A. Berson, and J. Z. Fang, "Path profile guided partial redundancy elimination using speculation," in *Intern. Conf. on Computer Languages (ICCL)*, Chicago, IL, USA, May 1998, pp. 230–239.

[8] R. Bodík and R. Gupta, "Partial dead code elimination using slicing transformations," in *Conference on Programming Language Design and Implementation (PLDI)*, Las Vegas, NV, USA, 1997, pp. 159–170.

[9] C. Chekuri, R. Johnson, R. Motwani, B. Natarajan, B. R. Rau, and M. Schlansker, "Profile-driven instruction level parallel scheduling with application to super blocks," in *Intern. Symposium on Microarchitecture (MICRO)*, Paris, France, December 1996, pp. 58–67.

[10] P. Berube, A. Preuss, and J. N. Amaral, "Combined profiling: practical collection of feedback information for code optimization," in *Intern. Conf. on Performance Engineering (ICPE)*. New York, NY, USA: ACM, 2011, pp. 493–498, work-In-Progress Session.

[11] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 265–276. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508275

[12] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 57–76. [Online]. Available: http://doi.acm.org/10.1145/1297027.1297033

[13] M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney, "A comparative study of static and profile-based heuristics for inlining," Boston, Massachusetts, January 2000, pp. 52–64.

[14] M. Arnold, A. Welc, and V. T. Rajan, "Improving virtual machine performance using a cross-run profile repository," San Diego, California, October 2005, pp. 297–311.

[15] P. Salverda, C. Tuker, and C. Zilles, "Accurate critical path prediction via random trace construction," in *Code Generation and Optimization (CGO)*, Boston, MA, USA, 2008, pp. 64–73.

[16] S. Savari and C. Young, “Comparing and combining profiles,” *Journal of Instruction-Level Parallelism*, vol. 2, May 2000.