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 aims to construct an experiment to show a false speedup from actual data. This was done by just ignoring the multiple run strategy and literally selecting parts of the collected data to show that, in a single-run scheme, it can happen. As conclusion the paper states that the only way to avoid these problems is to define and use a reliable methodology based on solid statistical measurements. In this paper the methodology called *combined profiling* (CP) is also presented, and it is shown that employing it can generate more reliable results. FDI decisions are shown to be more accurate using CP instead of single-run evaluation.**

I. Introduction

This paper describes an empirical research focused on the confidence of speedups (or slowdowns) results. This problem arises in every empirical research, and specially in compiler research this is a crucial matter, because it is usual to report smaller speedups than other areas. But, because compilers have to optimize code for various different kinds of applications, another major concern is the input set that should be used to test the improvements achieved for some transformation. Not only the size of the inputs employed, but mainly the type of input and the type of behavior the program will be expected to have. The main issue though is on the methodology commonly applied for empirical research on compiler systems, the single-run for training and testing the programs.

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* (FDO) 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 FDO 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 FDO 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. Berube uses *Combined Profiling* (CP) to merge the profiles

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).

from multiple runs into a distribution model that allows code transformations to consider cross-run behavior variations [1]. 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. Recently other approaches to this problem are being developed. 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 Kalibera and Jones, 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 [2]. This research also focus on execution time, but showing the need of a methodology that allows the researcher to control the measurement errors, or at least to provide sufficient evidence of performance improvement.

There have been some recent efforts trying to apply multiple profiles to FDO and also to evaluate the performance of a program from multiple inputs. CP methodology addresses this problem and can be applied to many different optimization techniques, such as inlining, loop unrolling, etc. In this research CP is applied to inlining as a case study, because it allows many other optimization techniques to be performed afterwards.

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 the use of multiple-run methodologies is reinforced.

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? FDO 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 studies* The cases studied illustrate 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 eight sections, the introduction, where the research problem is posed and the main ideas are shown. The inlining transformation is described in the next section, and then the next section describes the problem and the whole setting of this research. Following starts the section where the "speedup" is presented and also has a notice on a "slowdown" for the same problem. After this section, the environment is analyzed and provides 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 employed in the latter section, the next section shows how this problem can be avoided by means of the CP methodology. This paper ends with a discussion on 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.

III. The Perils of a Poor Experimental Methodology

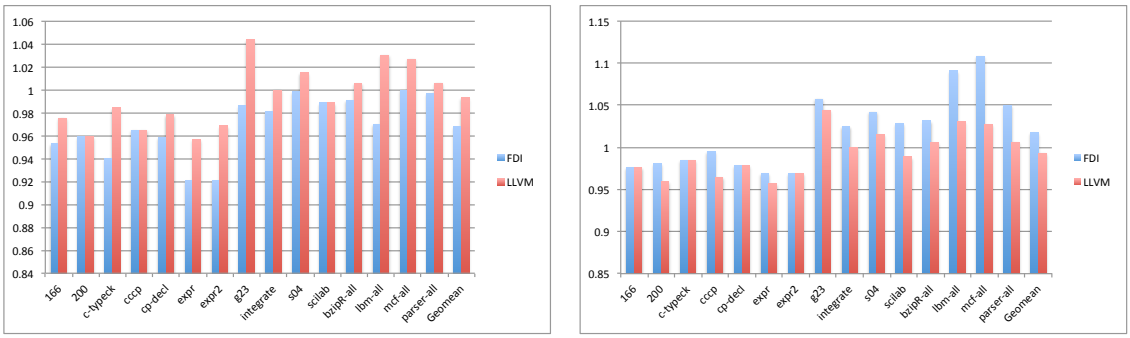
Benchmark-based evaluation is often used to predict the effect of a set of code transformations on the performance of actual applications that resemble the benchmark used in the evaluation. An issue with many of the performance evaluations of FDO-based code transformations published in the literature is the lack of exploration of the effect of different data input on the reported results. An interesting question is how misleading a performance prediction that uses a single data input may be.

The goal of this section is to investigate the potential error in the prediction for the case of FDI using combined profiling. The following experiment compares an FDI with the standard inliner from `LLVM`: (1) Select a reasonable set of data inputs for a given benchmark; (2) Execute all combinations of single-input profiling/single-input testing for the FDO inliners, repeating each test run a number of times that is sufficient to capture runtime variances;[2] (3) Run the `LLVM` inliner on all inputs — the same number of times as in (2) for each input; (4) To illustrate the best performance of FDI that could be reported from the data, select the best run amongst all profiling/testing combinations for a given test input and compare with the worst run for the `LLVM` inliner; (5) To demonstrate the worst performance of FDI, do the opposite, look for the worst FDI run and the best `LLVM` run for a given test input; (6) To find what the actual comparison is, use all but the test input to generate a combined profile and use this combined profile in FDI; (7) execute this binary the necessary number of times and compare the average of these runs with the average of the same number of runs using the `LLVM` inliner.

This performance evaluation uses an infrastructure based on the `LLVM` development framework. This infrastructure includes a set of C++ programs and a set of scripts to control the machine-learning training, the compilation and the execution of performance runs. This single infrastructure offers the option of performing both single-run-training/single-run-testing FDO and CP-based FDO with multiple-run performance evaluation. The number of runs used for CP and for the evaluation are parameters set by the experimenter [1].

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

[2]For the experiments described in this paper an empirical statistical study using 1000 runs revealed that three runs were sufficient.



(a) FDI is faster than LLVM;

(b) FDI is slower than LLVM;

Fig. 1. Performance Study for gcc. (a) Best runs of FDI compared with worst runs of LLVM. (b) Worst runs of FDI compared with best runs of LLVM. Bar heights represent running time normalized to running time of Never

The experiments were conducted on 20 Dell Optiplex 755 running Slackware Linux 2.6.32.39 each equipped with Intel Duo Core E6750 2.66 GHz processors, 4 GB RAM, DVD-RW drive, Intel Pro/1000 Gb ethernet, Gigabyte GeForce 8600 video cards, and 250 GB SATA II drive.

A. Experimental Results

The first case study uses the SPEC CPU 2006 gcc evaluated with fifteen inputs. The eleven inputs distributed with SPEC CPU 2006 are augmented with four SPEC 2000 benchmark programs used as input: bzip2, LBM, mcf, and parser. To be used as inputs these programs had to be converted to the single pre-processed file format required by the gcc benchmark. Figure 1 presents the result of the comparison between LLVMand FDI. The objective of this figure is to illustrate how an experimental evaluation that performs a single execution can result in misleading results and conclusions. For each test input the FDI measurement uses a leave-one-in methodology where all inputs, except the one used for testing, are used for training. is and LLVM. The baseline for comparison is Never which is a version of the compiler that uses an inlining cost function that limits inlining to callers containing a single basic block and that are expected to increase the code size in the caller by at most three instructions [1].

For all runs on individual inputs for a given version of the compiler, the reported execution time is the minimum of the execution time of three runs. This measuring method is selected to minimize interference from machine activity, such as network transactions or operating system interrupts, that are unrelated with the inlining strategy under study. Each bar in Figure 1 is the result of comparing the time obtained in a single execution. These results indicate the range of performance that could be reported by a careless experimental evaluation that uses a single execution of each version to report performance variations. The conclusion could vary from saying that FDI is 2.4% faster than LLVM to saying that LLVM is 2.8% faster than FDI. What is the true relative performance between the two versions? The next section provides a description of the Combined Profiling methodology that will be used to determine the actual performance comparison between FDI and LLVM.

IV. Combined Profiling Methodology

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 for 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.

Berube developed the *Combined Profiling* (CP) statistical modelling technique that 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 [1] . 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 described before [10], [3]. Berube's Ph.D. thesis presents a comprehensive description [1]. CP 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.

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:

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

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.

V. Determining the Number of Runs Required for Statistical Significance

The contradictory conclusions that arise from the experimental results presented in Figure 1 are caused by two issues: (1) the representation of a space of program behaviours by a single point in that space; and (2) the modelling of the effect of uncontrolled variables on the result of the experiments. The use of CP with a leave-one-out evaluation methodology leads to a more appropriate evaluation of the space of behaviour variations due to data input. The repetition of each experiment a reasonable number of times and the reporting of the average of these runs with a corresponding confidence interval to inform about this variation leads to a better accounting for he effect of uncontrolled variables that affect the results of the experiments. The result is a more accurate prediction of performance from the benchmark-based evaluation.

Uncontrolled variables include processes running in background, operating system calls, interruptions, memory allocation, and other sources, including the measurement process itself. Hence, it is important to have a good understanding of the sources of performance disturbances in the system [2]. Kalibera and Jones state that the majority of the experimental studies lack a rigorous statistical methodology [2]. A methodology to deal with the effect of uncontrolled variables is to examine the distribution of the data and identify measurements that can safely be eliminated because they are tainted by the effect of uncontrolled variables. For instance, Figure 2 depicts a scatter plot of 1000 sequential runs of the program `bzip2` compiled using the Static inliner (`LLVM`) and run with the `ebooks` input. For `bzip2` and `gzip`the code used is not the one distributed by SPEC, but rather fully-functional versions of these programs. Using these versions 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. The inputs for compression include images, ebooks in a variety of formats, movies in MP4 format, textual representation of proteins, audio books, and object files [1].

Nelson: We must state what is the benchmark run to produced the scatter plot shown in Figure Figure 2 and also what was the input used for those runs.

Ricardo: It is written in the phrase that first makes the reference to Figure 2 that the program is `bzip2`, that it was

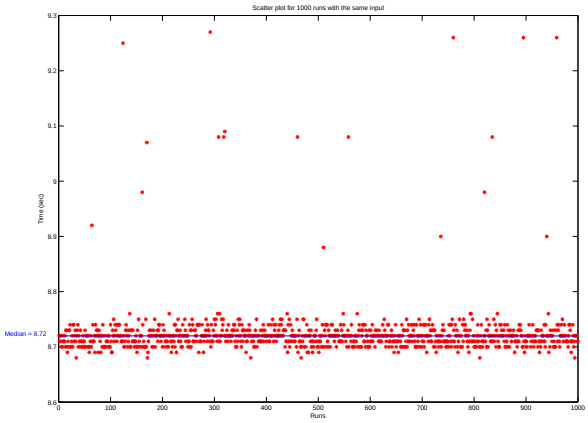


Fig. 2. Scatter plot of the execution times from 1000 runs of `bzip2` with `ebooks` data input. The execution times appear in the graph in the order in which they were measured.

compiled using the Static inliner, and the input was `ebooks`. Do you think it's somewhat hidden?

Nelson: What is our rationalization for concluding that we should see the same pattern of execution for all experiments based on the results shown in Figure Figure 2? Have we also done this experiment for other benchmarks and are only showing the result for one? Or are we assuming that the results should generalize because the conditions for all experiments are the same?

Ricardo: We have other benchmarks, but not all of them, and we are also assuming the results should generalize.

Figure 2 reveals a gaussian noise around the median plus some outliers that are the result of regular operating system activity. These outliers can safely be filtered out from the data set. They are easily discarded because they are more than one standard deviation above the median.

Nelson: Make fonts much bigger in Figure 2, also change "runs" to "run" in the horizontal axis because "runs" could be the number of runs represented by a point. Ricardo: The scatter plots were generated by Matlab, I am trying to correct this one, but I'm not being successful. Probably the connection, or the server, are overloaded today, I'll have to try again later.

Three independent experiments, with 10 runs, 100 runs, and 1000 runs, reinforces that outliers can be discarded because there is high probability that the measured means have minimum difference among them, and also the behaviour of the program remained unchanged. Ricardo: changed in response to your comments Nelson: What do these simple statistics tell us? Simple statistics — mean, median, standard-deviation from the mean (std-mean) and standard-deviation from the median (std-median) — shown in Table I lead to the conclusion that the three distributions are quite similar. Ricardo: completed in response to your comment/question Nelson: Is this the correct interpretation of the result of the t-tests? Ricardo: As we discussed by email, we only have evidence that the results point to failing to reject the null hypotheses. But there is still a possibility of making a type II error stating that the means are the same. So we can only say that since the null hypothesis is reject there is a probability (based on the p-values) that the means may be same. What you are saying is correct, we cannot state that the means are the same, they have a p-value probability of being the same. Also the results of t-tests run on each sample pairs, and shown in Table II, confirm that the means have high probability of being the same.

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 I
SIMPLE STATISTICS ON THE EXPERIMENT

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

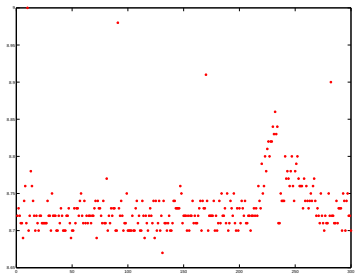
TABLE II
T-TESTS APPLIED PAIRWISE TO THE 10, 100, AND 1000 RUNS

Nelson: We need a clearer and more complete description of the t-tests that were run. Did we run a one-sample or a two-sample t-test? (see http://en.wikipedia.org/wiki/Student's_t-test). What does the value 0 in all rows of the table mean and how should they be interpreted? Why these p-values prevent us from discarding the null hypothesis? Our logic seems to be that because we did not discard the null hypothesis, we must conclude that the means are similar. Is that a correct reasoning?

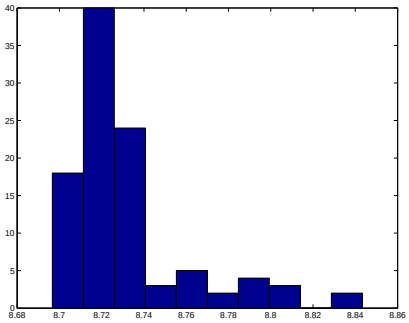
Ricardo: We suppose that the distribution is normal, but if the data are substantially non-normal and the sample size is small, the t-test can give misleading results. This is a risk that we are willing to take. We ran the two-sample t-test. About the value 0, they were returned by MatLab to show that the null hypotheses cannot be discarded it can be read as a 'false' as well. The p-values, except for the case 10-1000 are low, so the confidence is higher. In the 10-1000 case the p-value gives us a confidence of less than 40%. Concerning the logic applied, as mentioned earlier, we can still make mistakes, but we assume that the distributions tend to be normal and we believe that we cannot discard the null hypotheses.

The t-tests in Table II demonstrates that the null hypothesis cannot be discarded, as the value 0 in each line of the *t-test* column confirms, which means that the three means are similar. The *p-values* show the confidence in the hypothesis, in this case that the means are different. As the values are not high, the confidence is very low.

Another experiment has 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. This experiment was constructed by exercising each 'input-run' 3 times, 3-consecutive runs for each input, and the whole experiment was run 100 timesA 'full-run' in this experiment is a 3-consecutive run for each input, hence the experiment ran 100 'full-runs'. Nevertheless the behavior of the system was



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



(b) Histogram for the `auriel` input

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

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 III
DEVIATION FROM THE MEAN AND FROM THE MEDIAN IN THE EXPERIMENT

stressed through the addition of extra noise, simulating an 'uncontrolled variable' [2]. The extra noise was injected by the end of the experiment.

The purpose of the extra noise was to verify if the system was robust, even though the effect of the noise can mask the correct values, these data can treated assuring robustness. This way the CP methodology was empirically verified with respect to soundness. As can be seen in Figure 3, the deviation from the mean is not large, and also that there is a subtle knob, which increases the running time of the all programs. It was caused by the execution of another system at the same time competing for the same resources. The running time of each program at each 3-consecutive run can be found in the y-axis of Figure 3(a), and the order number of each full-run is depicted on the x-axis. The extra noise can also be visualized in the histogram of Figure 3(b), where the x-axis depicts the running time for the program and the y-axis depicts the number of runs at each bin. These two figures show the 3-consecutive run for the input data `ebooks`.

Figure 3 and Figure 2 show that collecting data from single execution can produce erroneous results, even using machines with no other running program. This happens because of the very nature of the empirical experiments, there is some noisy data distribution caused by regular operating system activities, interruption calls, etc. Also, an inclusion of a simple task during the running cycle can perturb the execution time of the program under evaluation, as can be observed by the knob in Figure 3.

The data collected in the experiment are shown in Table III, and the deviations from the mean (and median) to each 3-consecutive run are summarized as the average, minimum, and maximum values.

To confirm that the means are statistically representing the same distribution the t-tests were also run. This is summarized in Table IV below. It is easy to see that the number of outliers is little, except for the knob region.

To obtain low variance one possibility is to increase the number of consecutive runs for each individual input. Nevertheless, these experiments have shown that 3-consecutive run is a good choice, because it does not penalize much the total running time of the system.

The experiments also have shown that single-run testbeds are error-prone because they don't take the variance into account. The results of an experiment (speedups, or slowdowns) are robust only if there is statistical assurance that the variance on the data is not large.

A. Analyzing the speedup results

As aforementioned, the actual experiments collected 3 different run-times for each program at each input, hence it was just a matter of choosing least and greatest values to exhibit a speedup or a slowdown. In a single-run methodology any combination would be plausible to occur.

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 IV
TEST ON THE MEANS

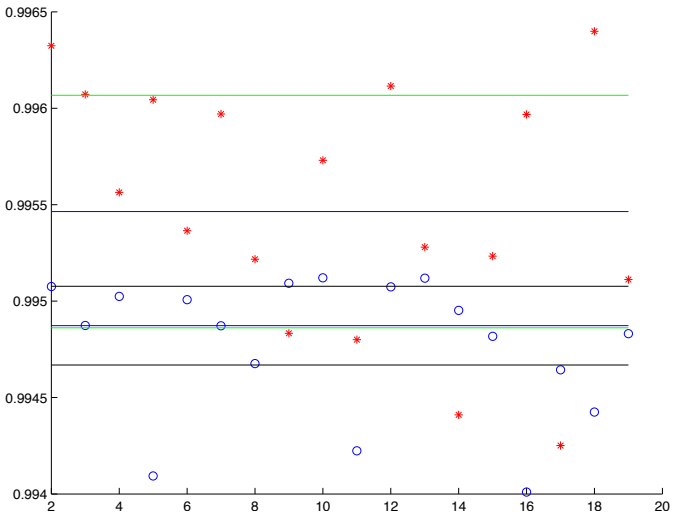


Fig. 4. The 18 different settings for `bzip2` of the same setup

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

The complete and correct values are described below. This section end with a figure that was generated by our framework, where the error bars are clearly depicted in it, showing the variance on the speedup geometric means.

1) Analysis of `bzip2` *and* `gzip`*:* After analyzing the inlining environment and having the confidence that the results are trustful, the first program studied was `bzip2`. Collecting data from the same setup (hardware and software) in 18 different settings was the first step. Figure 4 shows the data 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 FDI inlined program, and the blue "o" represent the normalized geomean for LLVM inlined program.

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.

Now just consider that a single-run experiment could have measured any one of the 3-consecutive run values individually, moreover, a single run may have also collected the best, or the worst values for the actual times of the experiment. Hence, to outcome a speedup for FDI the experiment could have collected the worst running time for LLVM inlined program, and the best running time for the FDI inlined program.

Even though this biased data showed a speedup, it was really worthless, only 0.46%. Therefore, to reinforce that the input set is also a big issue, the data were "adjusted", leaving the slowdowns and some of the tiny speedups gathered from the set of inputs out of the final list to be shown. This way a tiny, but possibly measurable speedup, was presented in Section VI. Nevertheless, defining a list of inputs is an issue and has to be treated as part of the experiment design, as this "speedup" have shown. The full data for the "speedup" experiment are shown in table Table V.

On the other hand, in Section VI-B the opposite was performed, choosing the worst individual running time for the FDI inlined program and the best running time for the LLVM inlined program. Proceeding this way it was easy to present, from "a different" individual measuring, a slowdown. And as both results followed the same methodology, they are both correct, and this is unexplainable unless considering that there is variance on the data collected.

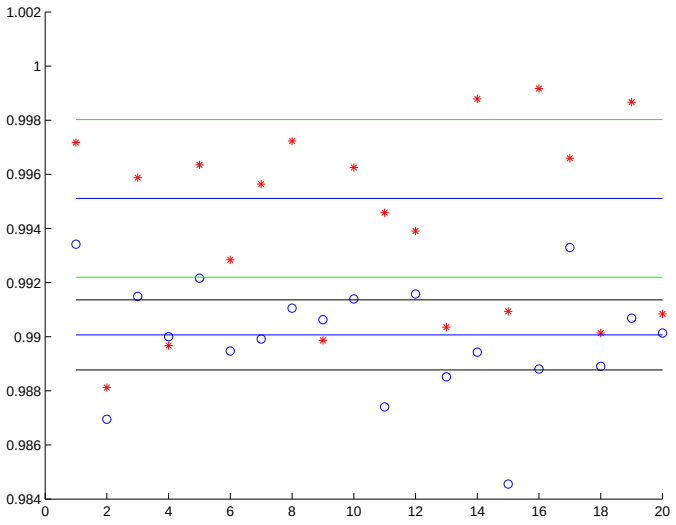


Fig. 5. The 20 different settings for `gzip` of the same setup

Input	FDO normalized	LLVM normalized	Speedup
13x13	0.9922	0.9983	0.9938
arb	0.9939	0.9969	0.9969
arend	0.9894	1.0017	0.9877
arion	0.9934	0.9989	0.9945
atari_atari	0.9838	1.0000	0.9838
buzco	0.9912	0.9970	0.9941
connect	0.9881	1.0118	0.9766
connection	0.9881	1.0039	0.9843
dniwog	0.9924	0.9977	0.99470
nicklas2	0.9980	1.0019	0.9960
nicklas4	0.9896	0.9960	0.9936
nngs	0.9905	0.9989	0.9915
score2	0.9775	0.9958	0.9816
trevorc	0.9928	1.0004	0.9923
trevord	0.9895	1.0025	0.9870
Geomean			0.9899 (1.01 %)

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

The same process was employed for the `gzip` case, using 20 different settings. Figure 5 presents the `gzip` data in the same way of Figure 4. From Figure 5 there can be seen no evidence of speedup for this setup, and even though a speedup was reported.

Although these cases were artificially constructed from empirical data, if a single-run methodology was employed these results could appear. But employing CP methodology allows a researcher to correctly identify the statistical variance on the data and to discard speedups or slowdowns. This result, in a certain way, reinforces the result of Curtsinger and Berger, reporting no speedup of $-O2$ over $-O3$ for all benchmarks they analyzed, when code randomization is applied [5].

2) Analysis of `gobmk`*:* For `gobmk` the input set was chosen in a similar way to that for `bzip2` and `gzip` cases. But, in reality the full 15-input set was applied, and if the full input-set is taken, the experiment would have produced a different result, the speedup would be of 1.01%, as shown in Table VI and in Figure 6.

3) Analysis of `gcc`*:* For `gcc` there were two different outcomes, one with the input set chosen, similar to the `bzip2`, `gzip`, and `gobmk` cases, and another one with a somewhat even more reduced input set, which produced an even better

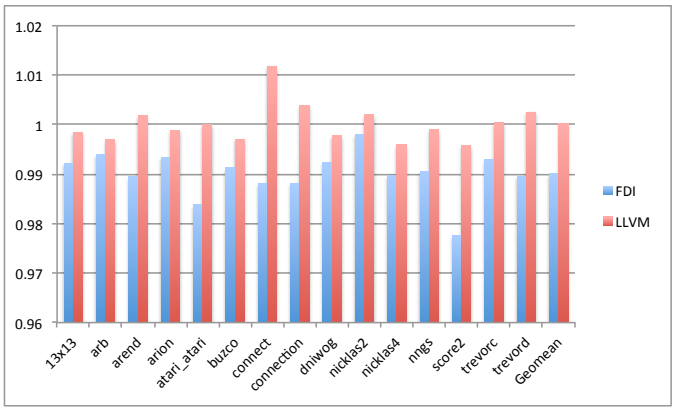


Fig. 6. The complete data of the speedup for `gobmk`

Input	FDO normalized	LLVM normalized	Speedup
166	0.9532	0.9755	0.9771
200	0.9594	0.9594	1.0000
c-typeck	0.9400	0.9845	0.9548
cccp	0.9646	0.9646	1.0000
Cp-decl	0.9589	0.9784	0.9800
expr	0.9208	0.9567	0.9624
expr2	0.9208	0.9686	0.9506
g23	0.9860	1.0441	0.9443
integrate	0.9810	1.0000	0.9810
s04	0.9987	1.0153	0.9836
scilab	0.9886	0.9886	1.0000
bzipR-all	0.9907	1.0055	0.9852
lbm-all	0.9696	1.0303	0.9411
mcf-all	1.0000	1.0270	0.9736
parser-all	0.9970	1.0059	0.9911
Geomean			0.9748 (2.52 %)

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

speedup. Again, this was done to raise the question about the proper set of inputs to be employed.

In reality the full 15-input set was applied, and if the full input-set is taken, the experiment would have produced a different result, the speedup would be of 2.52%, as shown in Table VII and in Figure 7.

Therefore, the input-set matters, as much as a sound methodology. To summarize this section and illustrate the outcomes of the framework employing CP methodology, Figure 8 presents one of the figures automatically generated by the

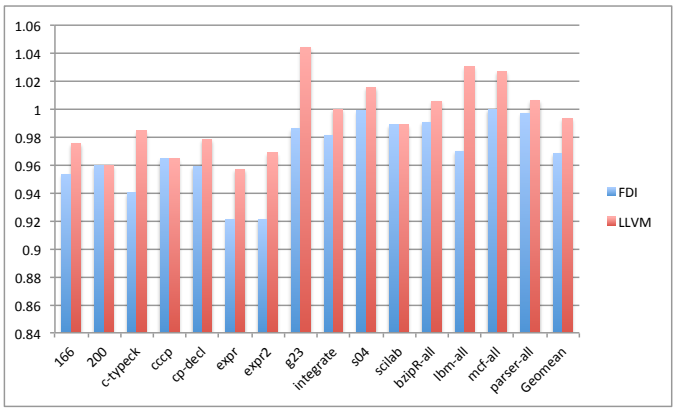


Fig. 7. The complete data of the speedup for `gcc`

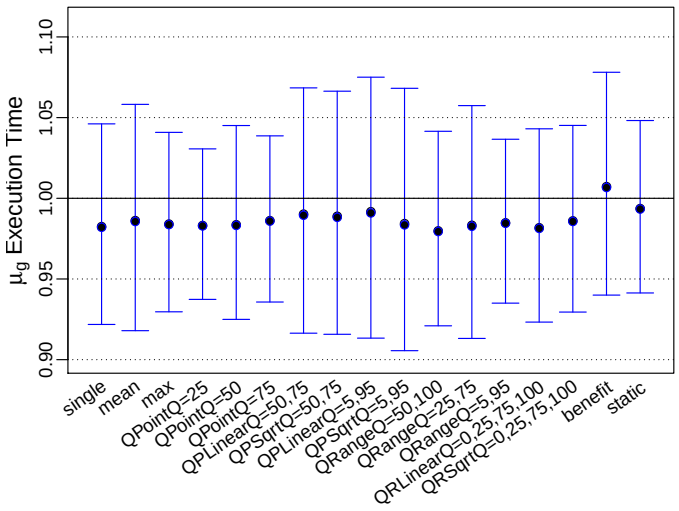


Fig. 8. The actual result for `gcc` returned by our CP framework

framework. In this figure it can be observed that there was no speedups, nor slowdowns for the `gcc` case, the error bars present in Figure 8 demonstrates the level of confidence in the geometric mean results.

This figure reflects the geometric mean of all inputs for one single-run FDI inliner (called single in the figure), twelve different FDI inliners, the LLVM inliner (called static in the figure) and another static inliner called benefit.

Next section (Section IV) describes the CP methodology in more detail, explaining its use and how to measure the results, in order to avoid the problems highlighted by the example in Section VI.

VI. Reporting a speedup measured using FDI

To make a fair comparison on the inliners and have a reasonable and short input set, some experimental decisions were taken. Both inliners were also evaluated with respect to the baseline Never, which means never inline. The input set for each program was defined to be representative for the entire set of inputs, and are described as follows. The input set for the programs `bzip2`, `gzip`, `gobmk`, and `gcc` is a small subset of the original 15-input set described in Section III. The results show a slight improvement over LLVM.

In the figures throughout this section, all results are presented in a bar graph where the values used for each of the columns are calculated using the actual running times divided by the actual running time of Never, normalizing the results presented. Whenever the values are less than one, then these particular programs have a better (faster) performance than the same program without inlining. Also, the speedup columns in the tables shown in this section are calculated from the ratio between FDI and LLVM.

A. Presenting the speedup results

As aforementioned the data points were selected as representing a single-run methodology for the experiments, and three benchmarks were used to test the hypotheses, `bzip2`, `gzip`, and `gcc`. The points selected were the best-run times for FDI and the worst-run times for LLVM. The results are presented in the following way, `bzip2` and `gzip`, are grouped, while `gcc` is separately described in other section, because the program behavior is completely different from the other two benchmarks.

Input	FDO normalized	LLVM normalized	Speedup
166	0.9532	0.9755	2.28%
200	0.9594	0.9594	0.00%
c-typeck	0.9400	0.9845	4.52%
cccp	0.9646	0.9646	0.00%
cp-decl	0.9589	0.9784	2.00%
expr	0.9208	0.9567	3.76%
expr2	0.9208	0.9686	4.94%
g23	0.9860	1.0441	5.57%
integrate	0.9810	1.0000	1.90%
s04	0.9987	1.0153	1.63%
scilab	0.9886	0.9886	0.00%
bzipR-all	0.9907	1.0055	1.48%
lbm-all	0.9696	1.0303	5.88%
mcf-all	1.0000	1.0270	2.63%
parser-all	0.9970	1.0059	0.88%
Geomean			2.52%

TABLE VIII
SUMMARY OF THE DATA COLLECTED DURING THE EXPERIMENT WITH `gcc`

Input	FDO normalized	LLVM normalized	Speedup
auriel	0.9720	1.0076	3.53%
avernum	0.9922	0.9905	-0.18%
cards	0.9909	0.9989	0.81%
ebooks	0.9909	0.9920	0.11%
gcc	0.9966	1.0059	0.92%
lib-a	0.9940	0.9970	0.30%
mohicans	1.0000	1.0048	0.48%
ocal	0.9988	1.0075	0.86%
paintings	1.0000	1.0051	0.51%
potemkin	0.9916	0.9887	-0.29%
proteins-1	0.9977	0.9910	-0.68%
proteins-2	0.9813	0.9950	1.38%
revelation	0.9868	0.9887	0.19%
sherlock	1.0000	1.0020	0.21%
usrlib	1.0000	0.9875	-1.26%
Geomean			0.46%

TABLE IX
SUMMARY OF THE DATA COLLECTED DURING THE EXPERIMENT WITH `bzip2`

1) Case Study 1 `gcc`*:* For the `gcc` benchmark the results show a speedup of 2.52% over LLVM and a 3.17% speedup over Never, whereas LLVM achieved a 0.67% speedup over Never, for the whole input set. This result is summarized in Table VIII below. The results are normalized by the baseline Never (no inlining).

The Figure 7 shows that the FDI inliner outperforms Never and LLVM through all the inputs, which explains the speedup.

2) Case Study 2: `bzip2` *and* `gzip`*:* For the `bzip2` and `gzip` cases, the experiments showed a slight speedup over LLVM. The data collected from the `bzip2` runs are summarized in Table IX. In this table the speedup achieved was a slight one, 0.46% over LLVM results, and 0.71% over Never (no inlining), whereas LLVM achieved a speedup of 0.25% over Never.

Figure 9 shows the running time normalized by the time of Never. And again the FDI inliner outperforms Never and LLVM through all the inputs, the same way the former experiments did.

The final result for the speedup, despite being a slight improvement, represents that the FDI inliner can actually be employed instead of the LLVM inliner. And this result is also significant because the program `bzip2` is small, simple, and

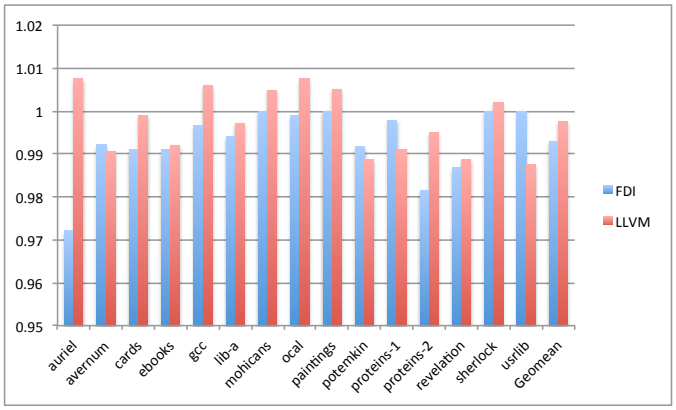


Fig. 9. Running times of the `bzip2` inlined versions, normalized by Never

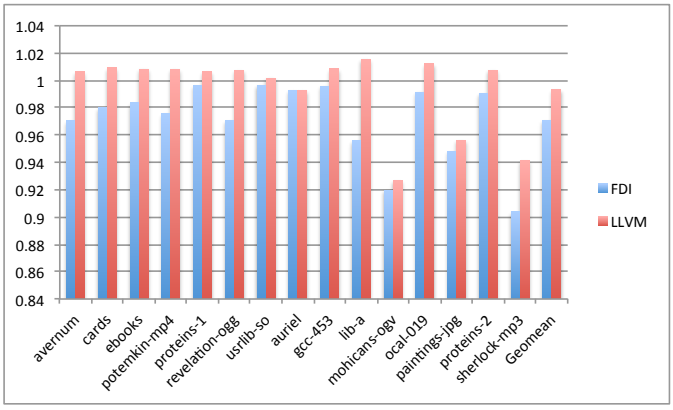


Fig. 10. Running times of the `gzip` inlined versions, normalized by Never

Input	FDO normalized	LLVM normalized	Speedup
avernum	0.9703	1.0062	3.57%
cards	0.9801	1.0092	2.88%
ebooks	0.9836	1.0081	2.44%
potemkin-mp4	0.9755	1.0079	3.21%
proteins-1	0.9959	1.0064	1.04%
revelation-ogg	0.9708	1.0072	3.62%
usrlib-so	0.9966	1.0016	0.51%
auriel	0.9924	0.9924	0.00%
gcc-453	0.9957	1.0085	1.27%
lib-a	0.9558	1.0151	5.84%
mohicans-ogv	0.9195	0.9269	0.80%
ocal-019	0.9914	1.0122	2.05%
paintings-jpg	0.9478	0.9561	0.87%
proteins-2	0.9905	1.0074	1.68%
sherlock-mp3	0.9038	0.9411	3.97%
Geomean			2.26%

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

Input	FDO normalized	LLVM normalized	Speedup
13x13	0.9922	0.9983	0.62%
arb	0.9939	0.9969	0.30%
arend	0.9894	1.0017	1.23%
arion	0.9934	0.9989	0.55%
atari_atari	0.9838	1.0000	1.61%
buzco	0.9912	0.9970	0.58%
connect	0.9881	1.0118	2.34%
connection	0.9881	1.0039	1.57%
dniwog	0.9924	0.9977	0.53%
nicklas2	0.9980	1.0019	0.39%
nicklas4	0.9896	0.9960	0.64%
nngs	0.9905	0.9989	0.84%
score2	0.9775	0.9958	1.84%
trevorc	0.9928	1.0004	0.76%
trevord	0.9895	1.0025	1.30%
Geomean			1.01%

TABLE XI
SUMMARY OF THE DATA COLLECTED DURING THE EXPERIMENT WITH `gobmk`

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.

The experiment with `gzip` was a starting point to test the conjecture, and the results are quite similar to those from `bzip2`, and confirmed a speedup of 2.26% over LLVM results, and a speedup of 2.90% over Never (no inlining) and LLVM got a speedup of 0.66% over Never. These results can be seen in Table X, where the times are already normalized by the baseline Never (no inlining). Figure 10 shows the normalized running time for `gzip`, and it also outperforms Never and LLVM through 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) Case Study 3 `gobmk`*:* For the `gobmk` benchmark the results show a speedup of 1.01% over LLVM and a 0.99% speedup over Never, whereas LLVM had a 0.02% slowdown over Never, for the short input set used. This result is summarized in Table XI. The results are normalized by the baseline Never (no inlining).

The Figure 11 shows that the FDI inliner outperforms Never and LLVM through all the inputs, which explains the speedup.

B. Presenting the slowdown results

Proceeding as described in Section III, the data points for these experiments were selected as the worst-run times for FDI and the best-run times for LLVM. This time the single-run experiments report a slowdown.

1) Case Study 1 `gcc`*:* The slowdown over LLVM measured is of 2.43% for `gcc`, as shown in Table XII. Figure 1(b) shows the normalized running time for the slowdown measured for `gcc`.

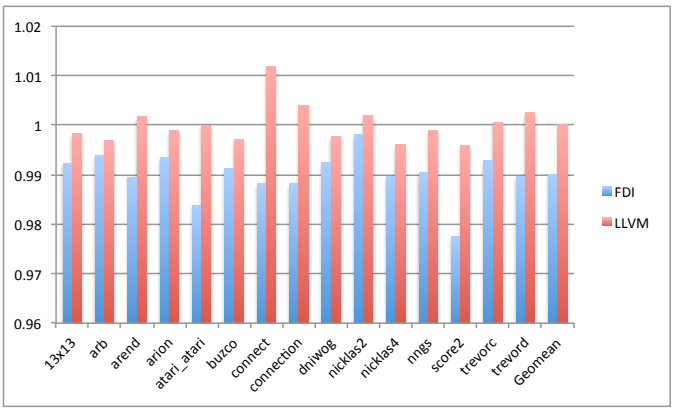


Fig. 11. Running times of the `gobmk` inlined versions, normalized by Never

Input	FDO normalized	LLVM normalized	Speedup
166	0.9755	0.9755	0.00%
200	0.9807	0.9594	2.17%
c-typeck	0.9845	0.9845	0.00%
cccp	0.9949	0.9646	3.05%
cp-decl	0.9784	0.9784	0.00%
expr	0.9686	0.9567	1.23%
expr2	0.9686	0.9686	0.00%
g23	1.0574	1.0441	1.26%
integrate	1.0253	2.47%	
s04	1.0420	1.0153	2.56%
scilab	1.0281	0.9886	3.84%
bzipR-all	1.0315	1.0055	2.52%
lbm-all	1.0909	1.0303	5.56%
mcf-all	1.108108	1.0270	7.32%
parser-all	1.049839	1.0059	4.18%
Geomean			2.43%

TABLE XII
DATA REFLECTING A SLOWDOWN ON gcc

Input	Normalized FDO	Normalized LLVM	Slowdown
auriel	0.9961	1.0025	-0.64%
avernum	0.9922	0.9905	0.18%
cards	1.0457	0.9882	5.50%
ebooks	0.9909	0.9920	-0.11%
gcc	0.9966	1.0059	-0.93%
lib-a	0.9940	0.9970	-0.30%
mohicans	1.0000	1.0048	-0.48%
ocal	1.0035	0.9984	0.51%
paintings	1.0000	1.0051	-0.51%
potemkin	0.9916	0.9887	0.29%
proteins-1	0.9977	0.9910	0.68%
proteins-2	1.0012	0.9931	0.80%
revelation	1.0359	0.9905	4.39%
sherlock	1.0000	1.0020	-0.21%
usrlib	1.0000	0.9875	1.24%
Geomean			0.71%

TABLE XIII
DATA REFLECTING A SLOWDOWN ON bzip2

2) Case Study 2: bzip2 *and* gzip*:* The slowdown over LLVM measured is of 0.71% for bzip2, 8.84% for gzip, as shown in Table XIII, Table XIV.

3) Case Study 3: gobmk*:* The slowdown over LLVM measured is of 0.99% for gobmk, as shown in Table XV.

VII. 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

Input	Normalized FDO	Normalized LLVM	Slowdown
avernum	1.0093	1.0062	0.31%
cards	1.0092	1.0092	0.00%
ebooks	1.0081	1.0081	0.00%
potemkin-mp4	1.0052	1.0079	-0.26%
proteins-1	1.0203	1.0064	1.37%
revelation-ogg	1.0072	1.0072	0.00%
usrlib-so	1.0651	1.0016	5.96%
auriel	1.1363	0.9924	2.67%
gcc-453	1.1111	1.0085	9.23%
lib-a	1.0735	1.0151	5.44%
mohicans-ogv	1.2835	0.9269	27.78%
ocal-019	1.1412	1.0122	11.31%
paintings-jpg	1.1826	0.9561	19.15%
proteins-2	1.0780	1.0074	6.55%
sherlock-mp3	1.2692	0.9411	25.85%
Geomean			8.84%

TABLE XIV
DATA REFLECTING A SLOWDOWN ON gzip

Input	FDO normalized	LLVM normalized	Speedup
13x13	1.0057	0.9987	0.70%
arb	1.0049	0.9984	0.64%
arend	1.0111	1.0019	0.91%
arion	1.0087	1.0000	0.87%
atari_atari	1.0000	0.9892	1.08%
buzco	1.0145	0.9970	1.72%
connect	1.0177	1.0059	1.16%
connection	1.0078	1.0039	0.39%
dniwog	1.0067	0.9984	0.82%
nicklas2	1.0156	1.0000	1.54%
nicklas4	1.0047	0.9960	0.87%
nngs	1.0123	0.9984	1.37%
score2	1.0044	0.9960	0.84%
trevorc	1.0140	0.9998	1.39%
trevord	1.0078	1.0030	0.48%
Geomean			0.99 %

TABLE XV
DATA REFLECTING A SLOWDOWN ON gobmk

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.

VIII. Conclusion

As mentioned in Section I a case study was proposed for the inlining transformation. The experiment was designed to make a clear point about applying single-run methodologies and also about the definition of the input-set. The experiments compared the CP process with the single-run process. Any other transformation could have been chosen, because the CP methodology can be applied in all general cases.

In Section VI it was shown an erroneous speedup, considering that it was measured by a single-run experiment. The speedup was constructed considering that any of the measurements that ran independently could have happened in a single-run experiment. Hence, searching the collected data for some outliers, or at least some data at extreme points was not a hard task. So, gathering these data points and defining two specific cases: `Best-runtime` and `Worst-runtime` for the FDO-based inliner, and for the `LLVM` inliner.

With these data points just selecting the ideal pairs it is possible 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 and assuming a single-run methodology, a statistical analysis showing a speedup (or slowdown) was produced for `bzip2`, `gzip`, `gobmk`, and `gcc`. 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 these experiments some of the open questions posed in the Section I can be answered. It is surely known, and was shown in Section V, 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, that CP is more reliable and its results are meaningful. Notwithstanding each program has to be run more than once, that is a small price to pay for more reliability, and the impact is acceptable if the number of repetitions is not too high. In the experiments carried out in this research running three times was enough.

A. Future work

There are two different paths for future work planning:

- *Fine-tuning* Using the CP methodology fine tune the FDI inliner for some different benchmarks. Some experiments have already finished, and some changes in the algorithms are being introduced.;
- *Apply CP* Applying CP to different compiler transformations is another research path.

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.