Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 130 (2018) 1108–1113

Procedia Computer Science

www.elsevier.com/locate/procedia

9th International Conference on Ambient Systems, Networks and Technologies, ANT-2018 and the 8th International Conference on Sustainable Energy Information Technology, SEIT 2018, 8-11 May, 2018, Porto, Portugal

A New Adaptive Algorithm for Inlining: An Experiment on FDO-Based Transformations

Ricardo Luis de Azevedo da Rocha[a,*], Mateus C. M. de Freitas Barbosa[a]

[a]*Department of Computer Engineering, Escola Politécnica, Universidade de São Paulo*
Av. Luciano Gualberto, travessa 3, 380, 05508-900, Sao Paulo, SP, Brazil

Abstract

This paper describes an empirical research focused on inlining, a compiler transformation that explores the idea of expanding a function's code to uncover optimization opportunities. Previous work has not addressed the problem of representing and utilizing multi-run profiles. To produce sound results on *feedback-directed optimization* (FDO), we employ multi-run profiles using Berube'scontribution on *Combined Profiling* (CP). The FDO inliner (FDI) designed by Berube is already an adaptive one. We devised a new algorithm for inlining, a slight modification on Berube'sproduced a speedup over other inliners, including `LLVM`.

Peer-review under responsibility of the Conference Program Chairs.

Keywords: Adaptive Technology, Inlining, Combined Profile, Compiler Transformations, Multi-run Profiles.

1. Introduction

This paper describes an empirical research focused on inlining, a compiler transformation. Inlining explores the idea of expanding a function's code to uncover optimization opportunities. Previous work has not addressed the problem of representing and utilizing multi-run profiles [1,2]. A *feedback-directed optimization* (FDO) compiler should not merely 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.

The deficiencies of this evaluation process are particularly prevalent, and especially disconcerting, when FDO is used to guide a transformation. In this scenario, instrumentation is inserted into the program during an initial compilation 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.

Kalibera [3] 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 employs it. As reported by Kalibera and Jones, the overwhelming majority of these papers has shown results either impossible to repeat or didn't demonstrate their

* Corresponding author. Phone: +55-11-3091-9084; Fax: +55-11-3091-5713.
E-mail addresses: rlarocha@usp.br (Ricardo Luis de Azevedo da Rocha)., mcmfbarbosa@gmail.com (Mateus C. M. de Freitas Barbosa).

1877-0509 © 2018 The Authors. Published by Elsevier B.V.
Peer-review under responsibility of the Conference Program Chairs.
10.1016/j.procs.2018.04.166

performance claims; there was no measure of variation for their results[3]. Berube[4,5] employed *Combined Profiling* (CP) to apply multiple profiles to FDO and also to evaluate the performance of a program from various inputs. Experimental results demonstrate that significant behavior variation is present in the program workloads and that this variation is successfully captured and represented by the CP methodology. This research also focuses on execution time but applying CP.

In this research, CP is applied to inlining as a case study, because it allows many other optimization techniques to be performed afterward. This paper employs a case study of the CP process for inlining, and defining a new inlining strategy over Berube's[4] algorithm, which is already an adaptive one. Our defined algorithm for inlining and the results are shown.

The main contribution of this paper are:

- An algorithm for inlining that produces a speedup;
- An experiment comparing the devised algorithm with regular algorithms for inlining;
- The use of CP-runs in a controlled trial comparing to single-runs.

This paper has six sections, the introduction, where the research problem is posed. The inlining transformation is described in the next section, and then the CP methodology is described. Following starts the section describing the adaptive FDI and the experiments. This paper ends with a discussion on related works and the conclusion.

2. 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 function to be called. Berube describes the existing inliner in `LLVM`and then presents a new feedback-directed inliner (FDI) that uses CP[4]. The FDI inlining strategies proposed by Berube and the `LLVM` inliner are used in this paper to illustrate the need for care when attempting to predict the performance of a FDO transformation with a benchmark-based performance study. All inliners discussed in this paper are implemented in the open-source `LLVM` compiler[6].

Some terminology is required to discuss 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. Inlining inserts a copy of the callee at a call site.

2.1. Feedback-Directed Inlining (FDI)

A commonly held belief amongst compiler designers is that inlining decisions should be sensitive to the frequency of execution of control-flow paths in a program. The premise is that with limited budget a compiler should select the most beneficial call sites to inline and that the most profitable call sites will be the ones that execute most frequently. The limited budget arises from a desire to limit code growth to prevent the scope of non-linear-time static analysis from reaching sizes that would make such analyses impractical. The most common technique still used in compiler research and practice is to estimate the execution frequency of alternate control-flow paths from a single *profiling run* using a single input for a given program. Combined Profiling (CP) is a methodology that allows this prediction to use information collected from multiple executions of a program[2]. Berube developed a new CP-driven Feedback-Directed Inliner that is a worklist algorithm whose decisions are based on tuneable cost/benefit functions[4]. This inliner is shown in Algorithm 1.

3. Combined Profiling Methodology

Working at the UofA CDOL group, under Nelson Amaral supervision, Berube developed the *Combined Profiling* (CP) statistical modeling technique that produces a *Combined Profile* (CProf) from a collection of traditional single-

Algorithm 1 FDI worklist

```
1: Input: Module M: Whole-program IR
2: Input: File cpFile: Combined profile
3: Data: List<call site> candidates, ignored
4: Data: Map<Function → List<call site> > callers
   initialize(M, cpFile) budget = computeCodeGrowthBudget()
   candidates.sort()
5: while budget > 0 AND NOT candidates.empty do
6:     source = candidates.randPop() /* candidates.popBest() */
7:     if source.score ≤ 0 then
8:         break
9:     end if
10:    if source.callee.cannotInline then
11:        ignored.add(source)
12:        continue
13:    end if
14:    if source.expectedCodeGrowth > budget then
15:        ignored.add(source) continue
16:    end if
17:    Try to inline the candidate...
18:    inlineResult = LLVM.inlineIfPossible(source)
19:    if inlineResult.failed then
20:        source.callee.setCannotInline()
21:        ignored.add(source)
22:        continue
23:    end if
24:    Inlining succeeded
25:    budget -= inlineResults.codegrowth
26:    callers[source.getCallee].delete(source)
27:    for caller ∈ callers[source.caller] do
28:        caller.calcScore()
29:    end for
30:    for i = 1 to inlineResults.numInlinedCalls do
31:        target = inlineResults.inlinedCall[i]
32:        original = inlineResults.originalCall[i]
33:        callers[target.getCallee].insert(target)
34:        if ignored.contains(original) > 0 then
35:            target.histogram = 0
36:            ignored.add(target)
37:        else
38:            target.histogram = source.histogram ×
39:            original.histogram
40:            target.calcScore()
41:            candidates.insert(target)
42:        end if
43:    end for
44: end while
```

run profiles, thus facilitating the collection and representation of profile information over multiple runs[4]. 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[2]:

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 provides a data representation for profile information but does not specify the semantics of the information stored in the combined profile[5]. Naive combination of raw profiles, such as simple sums or arithmetic averages, can be very misleading.

4. Adaptive Inliner

FDI is a FDO inliner that can be parameterized, and it also employs the CP methodology. We designed some experiments to show that FDI can achieve better results than well established static inliners. FDI inliner is fully described in Berube's Ph.D. dissertation[4]. The input set was defined as a minimal coverage set, and the training set was defined as the whole input set except for the input under test. The experiment compares the runtime performance of the programs when inlined by `LLVM` static inliner with the performance of the same program when inlined by FDI inliner. The FDI algorithm is shown in Algorithm 1[4].

Algorithm 1 presents an outline of the worklist algorithm used by FDI. The algorithm uses several data structures:

candidates The worklist is a sorted list of candidates. A call site is an inlining candidate if it is a direct call, and if the callee does not contain a `setjump` nor has any previous attempt to inline the callee failed. Furthermore, the call site must have executed at least once during profiling.

ignored A list of call sites that are not inlining candidates. This list is maintained to enable correct and efficient bookkeeping and to allow any copies of these call sites created by inlining their caller to be immediately ignored.

callers A mapping from functions to the call sites that call them. This map allows for the re-scoring of call sites if a call is inlined into their callee. That inlining will change the callee's size and may break the expected simplifications possible if the callee is inlined.

inlineResult A structure returned by inliner that provides summary information regarding the transformation. In particular, it indicates if the attempted inlining failed. FDI enhances the default `LLVM` structure with co-indexed lists identifying the new call sites created in the caller by inlining, and their originating call sites in the callee. This information is required so that profile information can be estimated for the new call sites.

FDI is an `Adaptive Inliner` that modifies (by inlining) the programs according to the acquired profiles, which may change from one acquisition to another. And we improved the algorithm considering the noisy environment because measuring the running time also captures all the events happening on the system. Our approach tries to make a smooth change at each step by applying a limited random choice of the inlining candidates (compare Figures 1a with 1b, the variance is reduced in the latter). This way the sorted list of inlining candidates is not used orderly anymore, but candidates are randomly chosen according to the current budget value. Line 6 of Algorithm 1 was changed from candidates.popBest() to candidates.randPop() reflecting the random choice. We call this setup as `Random-Choice`, and the other as `Original`.

We took some experimental decisions to make a fair comparison on the inliners and have a reasonable and short input set. Firstly, both inliners were also evaluated concerning the baseline `Never`, which means 'never inline'. Second, 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 program `bzipR` is a small subset of the original 15-input set described in Section 4.1. The results show an improvement over `LLVM`.

4.1. Input Workload Description

Rather than using the SPEC benchmark versions of the `bzipR`program, the fully-functional "real" version is used. Using the real versions of the compressor program eliminates the unrealistically-simplified profiling situation where mutually-exclusive use cases are combined into a single program run. Consequently, the program cannot do decompression and compression, or multiple levels of compression, within the same run. Different inputs must cover these distinct use-cases in the program workload. This workload is split in half into a compression set and a decompression set. Several inputs in the compression set have an analogue in the decompression set. However, the file format is usually different, and the source of the data is never the same. For instance, revelation-ogg in the compression set and sherlock-mp3 in the decompression set are both audiobooks, but the audio is recorded in different formats, and the books themselves are different.

The compressor uses a numeric command-line flag to control the tradeoff between compression speed and compression quality. The flags take integer values between 1 (fastest, least compressed) and 9 (slowest, most compressed). The seven inputs in the compression set each use a different compression level, from 3 to 9. Most inputs are collections of files. Each collection is archived (uncompressed) so that the input and output of each run is a single file. In order to minimize the impact of disk access, the output of each run is redirected to `/dev/null`.

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

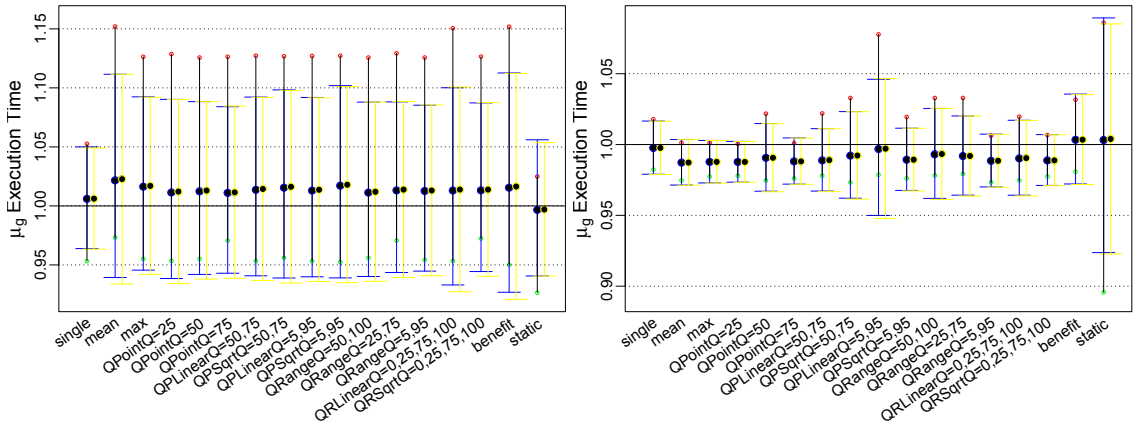
4.2. Results

The experiments were conducted on a Dell Optiplex 755 running Ubuntu Linux 16.04 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 1 TB SATA II drive. This case study uses the SPEC CPU 2006 `gcc` evaluated with fifteen inputs.

Input	FDO normalized	LLVM normalized	Speedup
auriel	0.9886	1.0168	2.77%
avernum	0.9862	1.0037	1.75%
cards	1.0018	0.9970	-0.48%
ebooks	0.9873	1.0194	3.16%
gcc	1.0056	1.0419	3.48%
lib-a	0.9886	0.9979	0.93%
mohicans	0.9807	1.0031	2.22%
ocal	0.9992	1.0029	0.37%
paintings	0.9875	1.0011	1.36%
potemkin	0.9890	1.0105	2.12%
proteins-1	0.9990	0.8956	-11.54%
proteins-2	1.0024	1.0860	7.70%
revelation	0.9823	1.0028	2.05%
sherlock	0.9873	1.0304	4.18%
usrlib	0.9964	0.9502	-4.86%
Geomean			2.26%

Table 1: Running times normalized and FDI geomean speedup over `LLVM` for `bzipR`.

Table 1 shows that FDI outperformed `LLVM` for `bzipR` in this configuration. The data are normalized by baseline `Never` and the geometric mean of the FDI speedup over `LLVM` was 2.26% for `bzipR` with this set of inputs. Figures 1a and 1b show the running time of the programs applying the `Original` FDI algorithm and the running time of the `Random-Choice` FDI algorithm normalized by `Never`.



(a) `bzipR` inlined by the `Original` setup

(b) `bzipR` inlined by the `Random-Choice` setup

Figure 1: Running times of the inlined versions normalized by `Never`

The evaluation of inlining used fourteen different reward functions for the combined-profiling inlining (see Figure 1). The normalized execution time for each of those reward functions uses a 3-fold cross-validation. For instance, to obtain the `single` measurement in the figure, for each input u in the workload $\mathcal{W}$, u is used for training and the generated program is tested using a leave-one-in methodology, *i.e.* the execution times for all inputs in $\mathcal{W}$ except u is obtained, and the speedup for each of these times in relation to the baseline is computed. Then the geometric average of these speedups is computed. Hence, $\mu_g(\mathcal{W})^4$ is the geometric mean of normalized execution times for $\mathcal{W}$, measured by 3-fold cross-validation:

$$\mu_g(\mathcal{W}) = \sqrt[|\mathcal{W}|]{\prod_{i \in \mathcal{W}} \frac{t_u(i)}{t_\emptyset(i)}}$$

Where $t_u(i)$ is the execution time of a FDO version of a program on input i when u is used as the training workload, and $t_\emptyset(i)$ is the execution time for the baseline `Never` running on input i. Both $t_u(i)$ and $t_\emptyset(i)$ are measured as the average of three runs. $\tau_u(i) = \frac{t_\emptyset(i)}{t_u(i)}$.

5. Related Work

There are several researchers concerned with the problem of reliability in performance measures. Kalibera *et al.*[3] 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 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 a different number of repetition counts for them. The methodology we employed does not assume that the environment changes scarcely, and we don't need a considerable number of repetitions.

Mytkowicz *et al.*[7] ran some experiments using SPEC CPU benchmarks and found significant systematic measurement errors in some sources, which 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[8]. 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.*[9] 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 on Java, but their recommendation remains true, it is necessary to use a reliable experimental methodology.

6. Conclusion

This paper has proposed a new algorithm for inlining over Berube's[4] contribution. The `Random-Choice` algorithm produces a speedup over other inliners, including `LLVM`. Our setup applied CP methodology to achieve more statistically sound results. Also, the new `Random-Choice` algorithm for inlining produces a better result and outperforms `LLVM` as shown in Figure 1.

Acknowledgment

The work reported here received support from grant 2017/02111-5, Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP).

References

1. Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J.N., Chang, B.Y.E., et al. In defense of soundiness: A manifesto. *Commun ACM* 2015;**58**(2):44–46. doi:10.1145/2644805. URL `http://doi.acm.org/10.1145/2644805`.
2. Berube, P., Amaral, J.N.. Combined profiling: A methodology to capture varied program behavior across multiple inputs. In: *Intern. Symp. on Performance Analysis of Systems and Software (ISPASS)*. Austin, Texas; 2012, p. 251 – 260.
3. Kalibera, T., Jones, R.. Rigorous benchmarking in reasonable time. In: *Proceedings of the 2013 international symposium on International symposium on memory management*; ISMM '13. New York, NY, USA: ACM. ISBN 978-1-4503-2100-6; 2013, p. 63–74. doi:10.1145/2464157.2464160. URL `http://doi.acm.org/10.1145/2464157.2464160`.
4. PaulBerube, . *Methodologies for Many-Input Feedback-Directed Optimization*. Ph.D. thesis; University of Alberta; 2012.
5. Berube, P., Preuss, A., Amaral, J.N.. Combined profiling: practical collection of feedback information for code optimization. In: *Intern. Conf. on Performance Engineering (ICPE)*. New York, NY, USA: ACM; 2011, p. 493–498. Work-In-Progress Session.
6. Lattner, C., Adve, V.. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Code Generation and Optimization (CGO)*. San Jose, CA, USA; 2004, .
7. Mytkowicz, T., Diwan, A., Hauswirth, M., Sweeney, P.F.. Producing wrong data without doing anything obviously wrong! In: *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*; ASPLOS XIV. New York, NY, USA: ACM. ISBN 978-1-60558-406-5; 2009, p. 265–276. doi:10.1145/1508244.1508275. URL `http://doi.acm.org/10.1145/1508244.1508275`.
8. Curtsinger, C., Berger, E.D.. Stabilizer: statistically sound performance evaluation. *SIGPLAN Not* 2013;**48**(4):219–228. doi:10.1145/2499368.2451141. URL `http://doi.acm.org/10.1145/2499368.2451141`.
9. Georges, A., Buytaert, D., Eeckhout, L.. Statistically rigorous java performance evaluation. In: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*; OOPSLA '07. New York, NY, USA: ACM. ISBN 978-1-59593-786-5; 2007, p. 57–76. doi:10.1145/1297027.1297033. URL `http://doi.acm.org/10.1145/1297027.1297033`.