An Empirical Evaluation of Combined Profiling

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

Rocha: Title is not set, just to fill in the blank; The order of the authors is meaningless now

***Abstract*—A common practice used to evaluate Feedback Directed Optimization (FDO) code transformations is to perform a single-run profile to characterize the behaviour of a program and then to evaluate the performance of the FDO transformation using a single-input run of the program. This research addresses this shortcoming of FDO research using a new methodology, the *combined profiling* (CP), to both inform the FDO decisions and produce a more significant performance evaluation of the code produced with FDO. Combined profiling can be applied to many different optimization techniques, in this paper we apply it to inlining, a simple and general technique that allows many other optimization techniques to be performed afterwards. Besides the application of combining profiling and a proper evaluation of inlining, this work also investigates better strategies for inlining and searches for and better parameter values for the inlining heuristics. The main finds are....**

I. Introduction

Rocha: Purpose of the research:

1) Conduct experiments to study the current status of Combined profiling applied to inlining.
2) Compare the results produced using the methodology with traditional approaches. (here we have to compare the results with the single run experiment) Probably this is the last experiment to be tried.
3) Verify if there are significant statistical differences in the 3-run and 300-run.
4) Verify if the profiling makes any difference or not - by means of a random choice of functions to inline.
5) Tune in the constant values of the system.
6) Define which strategy is more appropriate to use in this approach to inlining, simple sort, or a more sophisticated choice, based on the knapsack problem.
7) Define a method to tune in the parameters of the compiler.

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).

For most useful programs, the program's behaviour during execution depends on the program's input. Consequently, there is seldom an optimally-efficient executable representation of high-level source code that executes in the least possible amount of time for every program input. Even given an optimization criteria, such as processing throughput, and a reasonable set of program inputs, creating an executable whose performance is optimal for that set of inputs is (provably) intractable. Moreover, it is infeasible to approach compiler design as a single problem. Instead, compilers apply a series of *transformations* that attempt to improve the efficiency of a particular region of code in a specific way. For instance, III presents a function-inlining transformation that replaces a function call by a copy of the body of the called function. This transformation improves program efficiency by eliminating the overhead of making function calls, while improving the effectiveness of subsequent transformations.

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 behaviour 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.

Performance evaluation is a challenging, multi-faceted problem. In this research, performance is always assessed in terms of program execution time[1] One dimension of this challenge is the choice between evaluating throughput or latency. Given a collection of tasks (*e.g.*, the programs in a benchmark suite or runs of a single program on a workload of inputs), throughput measures the total time required to complete all tasks sequentially. Conversely, latency considers the tasks in parallel and measures the task that takes the longest. Improving throughput means reducing average execution time; improving latency means reducing worst-case execution time. Both types of performance are important, and both approaches to evaluation are valid. Fortunately, shifting focus is often as simple as changing the weighting used to combine measurements from the individual tasks. Different aspects of this work assume different performance goals and thus perform the weighting in different ways. Consider each of these approaches as one possible option for evaluation, independent of the specific evaluation in which they appear. A real-world application of any of the ideas presented here will have unique performance goals, and can mix-and-match these approaches as appropriate.

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. [1] uses *Combined Profiling* (CP) to merge the profiles from multiple runs into a distribution model that allows code transformations to consider cross-run behavior variations. Experimental results demonstrate that meaningful behavior variation is present in the program workloads, and that this variation is successfully captured and represented by the CP methodology.

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

[1]Other measures of performance include power consumption and code size.

other optimization techniques to be performed afterwards.

The FDO-based inliner presented in [1] demonstrates how a transformation can use the information stored in a combined profile. The feedback-directed inlining framework sorts inlining opportunities according to parameterized reward functions that query a combined profile using distribution quantiles. These components are brought together by performing a thorough cross-validated evaluation of the CP-informed inliner.

Proceeding with our research, we were challenged to define strategies for the inliner, and also to search for better parameter values for it. But we know that this set is not suited to all benchmarks, as reported in Kulkarni *et al.*[2]. To address the former problem we defined three different algorithms to choose a good candidate for inlining. One acting as a best-search at each execution step, another that defines a set of better candidates in a knapsack fashion. The last one performs a random choice which allowed us to effectively compare our results for the strategy.

For the problem of defining the best set point for the parameters, we decided to employ a Machine Learning technique in order to define a ”sweet spot”. This part of our research searched for a technique that could operate on limited data and unknown optimization space. The behavior of the optimization technique had to be steady in any space, because we didn't know whether the space of parameters is convex, or differentiable. And we could not afford to have many values for the parameter set. We decided to use the Simulated Annealing, or the SPSA - Simultaneous Perturbation Stochastic Approximation, because both techniques can deal with our constraints.

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? How the parameters can be tuned in? Which strategy to be used, considering FDO?

This paper addresses these questions by using an empirical validation of the CP process for the inlining case. At first by arguing that the behavior variations in an application due to multiple inputs produces better FDO decisions. It also argues that the parameters can be tuned in using machine learning techniques. As already mentioned the case proposed was for inlining, and we compared the choice of candidate to inline using the best candidate of a sorted list with random choice, and with a more sophisticated choice, based on the knapsack problem. 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:

- *Assessment of combined profiling* (CP), is performed using the case of *inlining* and the behavior of single-runs and CP-runs are analyzed (Section IX).
- *Parameter tuning*, is done by a machine learning algorithm to find the sweetspot for a set of inlining parameters

(Section VI).

- *Inline candidates* can be chosen in many different ways, so analyzing behavior variability for random choice, simple sort and a version of the knapsack problem shed light to this problem (Section VIII).

II. Combined Profiling

Capturing behavior variations across inputs is important in the design of an FDO compiler. A number of speculative code transformations are known to benefit from FDO, including speculative partial redundancy elimination [4], [5], trace-based scheduling and others [6], [7].

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

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

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

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

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

CP and HN have been presented in previous work [8], [3]. However, a clearer and expanded version, based on previous versions, can be found in [1], particularly the description of CP's histograms and the discussion of queries

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

A. Hierarchical Normalization

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

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

B. Denormalization

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

C. Queries

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

$H.\mathrm{min}; H.\mathrm{max}$
$H.\mathrm{mean}(incl0s)$
$H.\mathrm{stdev}(incl0s)$
$H.\mathrm{estProbLessThan}(v)$

H.quantile(q)
H.applyOnRange$(F(w,v), \mathit{vmin}, \mathit{vmax})$
H.applyOnQuantile$(F(w,v), \mathit{qmin}, \mathit{qmax})$
H.coverage
H.span

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

D. Alternative Usage

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

III. 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 [9].

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[2].

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. Figure 1 presents a running example demonstrating the transformations that become possible due to inlining. 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.

For example, after loop unrolling and scalar promotion, the code in Figure 1(d) is transformed into the code in Figure 1(e). Constant propagation in that final version of the code allows the compiler to replace the entire initial computation of fib5 by the constant value 13. However, the analysis does not consider the potential impact of such transformations, but only count instructions directly eliminated because the array's base address is a constant.

The impact of a constant parameter is determined for each formal parameter of each function in advance, by assuming that it takes a constant, but unknown, value. `LLVM`'s IR uses a *single static assignment* (SSA) representation, where each `value` produced is defined exactly once. Data flow is represented by directly linking each `value`[3] to the instructions that use it.

When u is an indirect call for which the callee becomes constant, the call is resolved to a direct call. `LLVM` awards a large bonus for this conversion; FDI counts the conversion separately from other eliminated instructions.

`LLVM` does not include the (now-unconditional) branch instruction in the count of eliminated instructions. FDI counts eliminated branch instructions separately from other elimi-

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

[3]Formal parameters and IR instructions are both `values`.

```
fib5() {
  result = iterative( &fibStep(), 5);
  return(result);
}

iterative( *Step(), n) {
  A = array[n+1];
  for(i=1; i<=n; i++)
    A[i] = *Step(A, i);
  return(A[n]);
}

fibStep(A[], n) {
  if( (n==1)||(n==2) ) return(1);
  return(A[n–2] + A[n–1]);
}
```

(a) Original code fragment

```
fib5() {
  Step() = &fibStep();
  n=5;
  A = array[n+1];
  for(i=1; i<=n; i++)
    A[i] = *Step(A, i);
  ret_iterative = A[n];
  result = ret_iterative;
  return(result);
}

fibStep(A[], n) {
  if( (n==1)||(n==2) ) return(1);
  return(A[n–2] + A[n–1]);
}
```

(b) after inlining `iterative`

```
fib5() {
  A = array[6];
  for(i=1, i<=6; i++)
    A[i] = fibStep(A, i);
  return(A[6]);
}

fibStep(A[], n) {
  if( (n==1)||(n==2) ) return(1);
  return(A[n–2] + A[n–1]);
}
```

(c) after constant and copy propagation

```
fib5() {
  A = array[6];
  for(i=1, i<=6; i++) {
    if( (i==1)||(i==2) ) A[i] = 1;
    else A[i] = A[i–2] + A[i–1];
  }
  return(A[6]);
}
```

(d) after inlining `fibStep` and simplification

```
fib5() {
  A0 = 1;
  A1 = 1;
  A2 = A0 + A1;
  A3 = A1 + A2;
  A4 = A2 + A3;
  A5 = A3 + A4;
  A6 = A4 + A5;
  return(A6);
}
```

(e) after loop unrolling and scalar promotion

Fig. 1. A sequence of transformations on a code fragment that computes Fibonacci numbers, illustrating the code-simplification opportunities enabled by inlining [1]

nated instructions. The analysis does not continue to subsequent successor blocks.

In the evaluation of the inlining benefit of a particular call site, the benefit pre-computed for the callee's formal parameters is retrieved, as appropriate, for each actual parameter that is a constant or a pointer to a stack-allocated array. The impact of each parameter is accumulated to estimate the total code-size reduction enabled by inlining the call site. The LLVM inliner simply adds these values together. FDI adds the counts for each category it measures and then computes a weighted sum of those values, as explained in detail in [1].

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 analyse; 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. 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.

IV. CP-Driven Feedback-Directed Inliner for LLVM

The feedback-directed inlining (FDI) [1] evaluated in this work is fundamentally different than the existing static inliner in LLVM. The static inliner inlines small calls to remove call overhead with minimal increases in code size. FDI attempts to minimize the dynamic number of instructions executed by the program by inlining the most frequent calls. While the static inliner considers call sites on a function-by-function basis, FDI considers the set of inlining opportunities present at global scope in the current state of the program.

What FDI does is to set the problem as one of constrained minimization, the *Substitution Problem* [10]. So, for a particular program, a subset of all invocations, the values of maximum program size, maximum callee size, etc, find a sequence of substitutions that can minimize the expected program execution time. And this is done by following the

Algorithm 1: FDI worklist

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

order of the candidate call sites in the candidates list. The list is ordered by the score of each candidate.

The assumption is that this is an efficient reduction of the Knapsack Problem to the Substitution Problem [10], [11]. The Knapsack Problem is known to be NP-hard [12], so it's intractable. To perform the reduction, we first assume that it is possible to construct an invocation with any integral execution time overhead. The second assumption is that it is possible to construct a procedure of any integral size less than the maximum.

A. *Worklist Algorithm*

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 on the event that a call is inlined into their callee. That inlining will change the callee's size, and may change 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.

At the start of FDI, the CProf is read in, and the histograms are associated with the appropriate call sites (line 1). Every call site is inserted into the callers list of their callee. During initialization, each call site is evaluated, and added to either the candidates or ignored list, as appropriate. When a call site is rejected for inlining, it is immediately and permanently moved from the list of candidates to the ignore list. Transformations such as constant propagation or alias analysis can resolve the callee of an indirect call to a single possibility, thereby making it a direct call. However, if the call is indirect when the call site is first discovered by the inliner, it is placed on the ignore list in spite of the possibility of future inlining resolving the call. Calls to libraries and compiler built-in functions are also immediately ignored because they cannot be inlined.

B. *Candidate Scoring*

The `LLVM` inliner makes inlining decisions at each call site by comparing the expected code growth to a fixed threshold. FDI takes a more directly execution-time-oriented approach to inlining and attempts to achieve the greatest reduction in executed instructions for the least amount of code growth. Therefore, FDI breaks the evaluation of a call site into three components: the expected inlining benefit, the expected code growth, and execution frequency of the call site. Given a call site, CS, the inlining candidate scoring function, Score(CS), combines these three elements so that Algorithm 1 can select the best (highest score) candidates for inlining first. CP provides a rich characterization of execution frequency. Making use of that information is described in detail in [1]; for now, let $\text{Reward}(\text{Benefit}(\text{CS}), R_{\text{CS}})$ represent some function of the estimated (execution-frequency independent) inlining benefit at call site CS and R_{CS}, that call-site's CP monitor. Given the benefit function $\text{Benefit}(\text{CS})$ and a cost function $\text{Cost}(\text{CS})$ described in this section, an inlining candidate's Score is conceptually computed:

$$\text{Score}(\text{CS}) = \frac{\text{Reward}(\text{Benefit}(\text{CS}), R_{\text{CS}})}{\text{Cost}(\text{CS})}$$

More details on Benefit and Cost functions can be found in [1].

C. Finding the Best Candidate

The best candidate to inlining is found by its own score. The way it is done is simply sorting the candidates list by their score values. So every time a new candidate is needed, just take the top one from the ordered list of candidates. Line 05 of Algorithm 1 shows this choice.

But the ordering depends on scoring, and scoring is sensitive to some parameters values. For instance, the expected reduction/expansion for each function is directly dependent of its expected size, and the reduction/ expansion define the benefit and the cost of a function, in other words, its own score. The parameters that can have some effect on the scoring function are the sizes of a instruction, a call, a return, a branch, an allocation, and a block.

Depending on the values of these parameters, the scoring function returns different values for each function, and the ordering may be changed, which makes the inliner take different decisions. The sensitivity of the scoring function have a direct effect on the sorting of the candidates list, which also have a strong impact on inline decisions.

To assure good decisions the system must have a good scoring function, meaning that its result allows a good ordering for the inliner. But, a good ordering depends on how accurate are the estimated scores. One way we can calibrate the scores is to run the system on some known benchmarks. This way the parameters can be calibrated to their best values, those who produce the best estimates.

So there is a need to find a "sweetspot" on these values. The first idea was to apply machine learning techniques to search for it through a space of possible values. But there are also some issues, we cannot afford to have a huge number of acquisition points because the whole system takes a long time to process. Also, optimizing a function without knowing if it is differentiable, or convex, is not a trivial machine learning task.

But the simple choice of a candidate may be done in other ways, for instance considering the current budget value, the expenditure of the candidates, etc. In sectionrefsec:choice we propose and measure the performance of two other different algorithms for the task of choosing the best candidate for the inliner, and compare them with the one used in Algorithm 1.

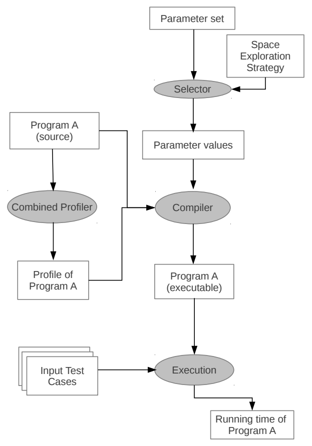
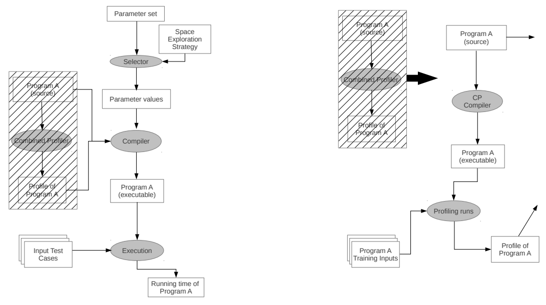


Fig. 2. Generic view of the method



(a) Simplified view of the process (b) Expanded view of the process

Fig. 3. CP compiler and the profile generation

The first is a "more complete" implementation of the knapsack algorithm, and the second represents the null hypotheses, is a random choice. We defined our algorithm empirically, based on the data we have collected.

V. A Method for CP-Driven FDI

The method we employed to use CP can be visualized in a four part fashion, as depicted in Figure 2. The first part, in the upper level of the figure, shows the Selector, which selects a set of values for the inlining parameters based on a strategy. The second part, on the left, shows the Combined Profiling (CP) process applied to a program. The third part, at the bottom of the figure, shows the execution test of a compiled program. And, in the center of the figure we have the compiler.

We can use this method for parameter tuning, and also for a machine learning approach trying to define the best set of parameters for throughput or for latency. The process remains the same, we just have to define properly what is a point of measure. More details on the CP profiling in Figure 3, whereas in Figure 3(a) the simplified view is presented, the hatched area marks the CP compiler; and in Figure 3(b) the CP compiler is shown in more detail.

In the case of parameter tuning, a point of measure is represented by the set of parameter values of the compiler for the program under tuning and its normalized value of the execution time for some test inputs. For the machine learning case, a point of measure for throughput is the sum of the execution times of each of the programs under test with its associated parameter values. On the other hand for latency, a point of measure is the geometric mean of the normalized values.

A. *Parameter Tuning*

Tuning compiler optimization options for a specific program is not an easy task [13]. There exist some tools based on Genetic Algorithms, Simulated Anealling [13], Support Vector Machines [14], etc. For the case of inlining a recently published research pointed to the use of Neural Networks to induce effective inlining heuristics [2].

As illustrated if Figure 2, our method can be effetively used to tune compiler optimization options; in this research we made an empirical evaluation of it in the inlining case.

A set of parameter values that is suited to a benchmark, may achieve poor results when applied to another benchmark. In Kulkarni *et al.*there is a figure that illustrates this problem, in Figure 1 of [2] they show the sensitivity of callee size metric for various benchmarks. Another difficult question to address is related to the interdependencies found in the parameter set. Adjusting one parameter has some effects on another, so it's not simple to tweak a set of parameter values.

Particularly considering the case of the callee size metric, it is commonly used as threshold in many inlining algorithms, and, as mentioned above, the best value for this parameter depends on the benchmark. This is the reason why this parameter was not defined as a constant value in our approach, but as a function that maps the callee size to a threshold value [1].

As any inlining decision can have some impact on subsequent decisions, there is a large space of possible inlining solutions, which can be traversed by slightly changing values, thresholds in any metrics employed. In our case, our space of possibilities can be traversed by changing the values of the inlining parameters.

Referring again to Kulkarni *et al.*, we can actually visualise the problem of searching the space for good solutions. As illustrated by Figure 2 in [2], the space is very large, but the set of good solutions is limited to a small percentage of the whole space. It surely is an indicator to the use of some AI algorithms to search for the optimal, even sub-optimal solution. In our research, we need to find a sweet spot where the parameter values fits best to a program making it run faster than before for some benchmark inputs, that are representative for that program.

Considering the effort usually employed by compiler writers in defining and tuning a heuristic, our work presents a much more general approach to the definition and tuning an inlining heuristic. First, it is automatically parameterized in terms of some common thresholds like the callee size, which is represented by function on the actual callee size, whose output is used to limit the code-growth budget. Second, it allows an automatic procedure for tuning in the parameters of the inliner.

The Hypothesis we use is that most inlining candidates that significantly impact the performance of any run also improve performance across most of the workload. The number of inlining opportunities that are particularly important for only a minority of runs is a small fraction of the total number of inlining opportunities that significantly impact program performance [1].

VI. Machine Learning Methods

To further characterize the problem we are facing, we need to look at the bigger picture. We have some input points from the parameters, and an output, the time spent by one particular program to run. And we can compare a series of runs, with inlining, without inlining, using non-FDO inlining.

With this starting point we have to define an error function and an algorithm to search for the optimal point if it exists, or at least to get as close as we can of it. But we don't really know any information about the space bounding the function. We are to define and then minimize the error function on an unknown space.

Any well known machine learning algorithm such as gradient descent not necessarily will work properly in our environment because, as we mentioned in IV-C, the function to be optimized is not known to be differentiable nor convex. And the space that bounds the function is also unknown.

One possible approach to this problem is to use other kinds of algorithms, such as, Simulated Annealing [13], or SPSA - Simultaneous Perturbation Stochastic Approximation [15], [16], because these algorithms have no supposition on the function and on the space. They are both non-deterministic, and make use of random points trying to avoid being trapped to a local minimum.

Another possibility is to apply an unsupervised learning method, like reinforcement learning, and use its' returned policy to guide the possible changes (up, or down) in the values of the parameters. This approach can be fully automatized and corresponds to another research path.

VII. Robustness

To show the gaussian variation in the data we collect, figure Figure 4 depicts a scatter plot of 1000 sequential runs of the program `bzip2`, after being compiled using the Static inliner (`LLVM`) whose input was `ebooks`. The figure shows a gaussian noise around the median plus a few outliers, generated by the operating system regular use. These outliers are filtered off from the data we are using. They are easily discarded because they have much more variance (more than one deviation from the median).

We ran three independent experiments, the first is the 10-times experiment, then 100-times and, after that, 1000-times, because we needed to confirm that there is no difference on their means, and also that we can discard the outliers. To make sure that we have robust measures, we ran some simple statistics to know the mean, the median, the standard-deviation

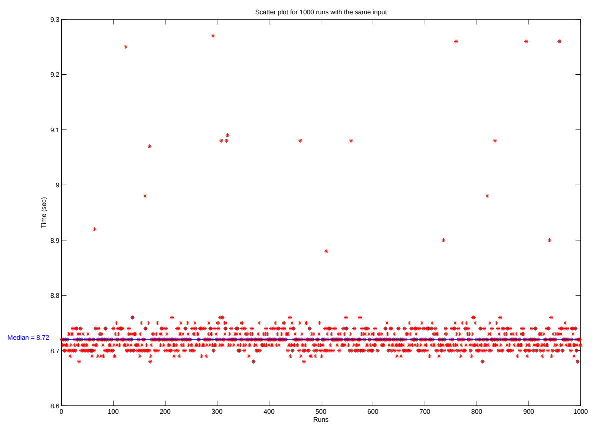


Fig. 4. Running 1000 times the same program with the same input data

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

TABLE I
SIMPLE STATISTICS ON THE EXPERIMENT

from the mean (std-mean), and the standard-deviation from the median (std-median), shown in Table I. We also ran t-tests on each sample pairs to verify if their means were the same, the results for the t-tests are shown in Table II below.

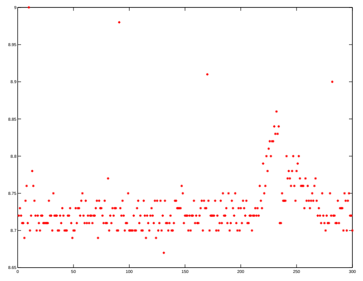
The t-tests in Table II show that the null hypothesis cannot be discarded, as the value 0 in each line of the *t-test* column confirms. The *p-values* illustrate the confidence in the hypothesis, in this case, that the means are different.

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

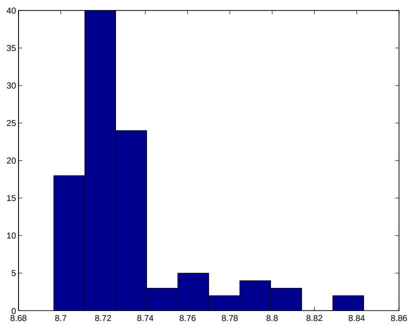
What this means is that even though the effect of the noise can mask the correct values, we can treat them in order to assure robustness. This is the way we employed to empirically verify the soundness of the CP methodology. As figure Figure 5 below shows, the deviation from the mean is not large, but here is a subtle knob increasing the running time of the all programs in the experiment by its end. It was caused by the execution of another system at the same time competing for the same resources. In (Figure 5(a)) we show in the y-axis the running time for each program at each 3-fold

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



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



(b) Histogram for the `auriel` input

Fig. 5. 100-times running 3-fold experiment

run, which are shown on the x-axis. It can be also visualised in the histogram (Figure 5(b)). These figures show the 3-fold run for the input data `ebooks`, where in the x-axis we show the running time for the program and on the y-axis we show the number of runs at each bin.

The figures Figure 5 and Figure 4 show that collecting data from single execution can show erroneous results, even using machines with no other running program, we still have some noise due to operating system activities, interruptions, etc. And also that a simple inclusion of a simple job during the running cycle can perturb the execution time, as can be observed by the knob in figure Figure 5.

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

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

This experiment brought us confidence in the machine learn-

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

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

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

VIII. Choice Methods

The scoring function, as mentioned in IV-B estimates the fitness of a function to be inlined, this way a function whose score is above zero (0) may be inlined, if the budget allows. But the algorithm that decides which function will be inlined is solely based in this information, not in the current budget value, nor in the other scores computed.

In [1] the inlining algorithm sorts the candidates for inlining based on the score function. As the score takes into account estimated values for benefit and cost of inlining, we suggest to use it but aggregating the expenditure, the budget, and other scores besides the most fitted.

The idea is to perform a more compete version of the knapsack algorithm considering all the scores above zero and the budget (the "size" of the knapsack). Proceeding this way we can optimize the amount of budget wasted for inlining.

T	C	Quantiles (%)	Description	Evaluation Name
P	–	25	first quartile	QPointQ=25
P	–	50	estimated median	QPointQ=50
P	–	75	third quartile	QPointQ=75
P	L	50, 75	average and optimistic	QPLinearQ=50,75
P	NL	50, 75		QPSqrtQ=50,75
P	L	5, 95	worst and best w/o outliers	QPLinearQ=5,95
P	NL	5, 95		QPSqrtQ=5,95
R	–	$\langle 50, 100 \rangle$	top half: optimistic	QRangeQ=50,100
R	–	$\langle 25, 75 \rangle$	"central" average	QRangeQ=25,75
R	–	$\langle 5, 95 \rangle$	average w/o outliers	QRangeQ=5,95
R	L	$\langle 0, 25 \rangle$, $\langle 75, 100 \rangle$	pessimistic and optimistic	QRLinearQ=0,25,75,100
R	NL	$\langle 0, 25 \rangle$, $\langle 75, 100 \rangle$		QRSqrtQ=0,25,75,100

TABLE V
RUNNING TIME OF EXPERIMENTS, CONSIDERING 3-TIMES RUN

To make sure that there is some room for improvement in the choice of the next function to be inlined, we are comparing the original choice (just using the highest score) with two different ways of choosing, a random choice, and the "more complete" knapsack choice. Our experiments have shown that there is a best way of choosing the candidates for the inliner, table Table V shows the results for these three algorithms.Rocha: table Table V is dummy

The results shown in figure Figure 6 refer to the geometric mean of all the runs for each program and each input data. We have the running time for each program in the y-axis, and the programs and input data on the x-axis. The performance of the three different algorithms are shown for comparison.Rocha: figure Figure 6 is dummy

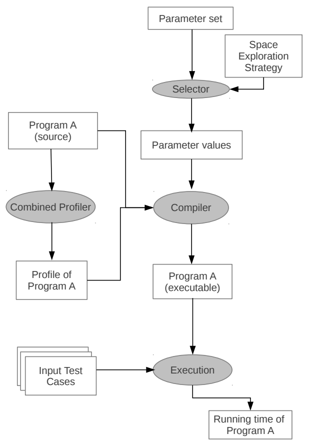


Fig. 6. Algorithms for choosing the candidates

IX. Results

The first experiment was to tune in the value of the compiler parameters. A machine learning algorithm was used to find the sweetspot for a set of inlining parameters. As described in Section VI, we used SPSA - Simultaneous Perturbation Stochastic Approximation because we have no supposition on the function and on the search space.

The second experiment was performed to evaluate and compare two different cases, the case of single-runs, and the case of CPruns. Both cases are analyzed and compared.

The third experiment was conducted to verify and analyze behavior variability for random choice, simple sort and a version of the knapsack problem. Particularly the latter version presented better results.

This study evaluates 14 instantiations of FDI reward functions, along with the Benefit static-reward function, single-profile FDO, and the default LLVM inliner. The version of a program transformed by an FDI inliner is referred to simply by the FDI reward function used by the inliner. For example, the mean version of a program refers to the version of the program created by applying the FDI inliner using the mean reward function. This section outlines how performance is measured and evaluated for these inliners, the concrete FDI reward functions evaluated, and the set of programs and inputs used for the evaluation.

A. *Measuring Single-Run Performance*

Before an inliner's workload performance can be calculated, execution times for runs on individual inputs must be determined. In this study, the execution time of one version of a program on a particular input is computed as the minimum of the execution times of three runs for this pairing. Henceforth, any reference to execution time assumes this three-run measurement.

The baseline execution time used in this study is obtained using the Never static FDI reward function, which unconditionally returns -1.0. Thus, Never will never inline any call site where $\text{Cost}(\text{CS}) > -2$. Given the definition of $\text{Cost}(\text{CS})$ from IV-B, inlining is thus limited to callees containing a single basic block, that are expected to increase code size in the caller by no more than 3 instructions.

The performance of a particular version of a program on one input is computed as the execution time of that version of the program on that input, normalized by the execution time of Never on that input.

B. *Static Inlining*

Two static inliners are evaluated: Benefit, and the default LLVM inliner, Static. These inliners are evaluated by simply taking the geometric mean of their normalized execution times over all inputs in $\mathcal{W}$.

C. *Single-Profile FDO*

Traditional single-profile FDO is referred to in this evaluation as the Single inliner. Inlining is performed using a simple point reward function with FDI, informed by a single profile. Since the profile contains a single value for each monitor, all simple point reward functions are equal. The evaluation uses the leave-one-in methodology in a manner similar to that of [1]. Each input $u \in \mathcal{W}$ is used for training, and then evaluated using $\mathcal{W}/\{u\}$. A performance score for u is calculated by taking the geometric mean of those normalized times:

$$\mathbf{gm}[u] = \sqrt[|\mathcal{W}-1|]{\prod_{i \in \mathcal{W}/\{u\}} \tau_u^{-1}(i)}$$

The final performance for traditional FDO is computed as the geometric mean of each $\mu_\mathbf{g}[u]$:

$$\mu_g = \sqrt[|\mathcal{W}|]{\prod_{u \in \mathcal{W}} \mathbf{gm}[u]}$$

D. *FDI Reward Functions*

The goal of CP-informed inlining is to increase performance across all future program executions by considering the cross-input behavior variations captured in the CProf. In particular, it strives to minimize the worst-case negative impact on the performance of any run. Performance degradation (versus an alternative inlining algorithm) is caused by a failure to inline call sites that are important in one or more program runs. These missed candidates are opportunity costs: the potential benefit of inlining is not realized in those runs because the inlining budget was spent on other candidates. It is the responsibility of the reward function to prioritize inlining candidates such that this opportunity cost is minimized. However, opportunity cost cuts both ways. Inlining a call that is important in only a small proportion of program runs may consequently expend the code-growth budget before more generally-beneficial inlining opportunities can be exploited.

As explained in [1], FDI provides three classes of reward functions that use combined profiles. Results are presented for the mean and max *simple point* rewards, seven concrete instantiations of *quantile point* rewards, and five concrete instantiations of *quantile range* rewards. The quantile points

T	C	Quantiles (%)	Description	Evaluation Name
P	–	25	first quartile	QPointQ=25
P	–	50	estimated median	QPointQ=50
P	–	75	third quartile	QPointQ=75
P	L	50, 75	average and optimistic	QPLinearQ=50,75
P	NL	50, 75		QPSqrtQ=50,75
P	L	5, 95	worst and best w/o outliers	QPLinearQ=5,95
P	NL	5, 95		QPSqrtQ=5,95
R	–	$\langle 50, 100 \rangle$	top half: optimistic	QRangeQ=50,100
R	–	$\langle 25, 75 \rangle$	"central" average	QRangeQ=25,75
R	–	$\langle 5, 95 \rangle$	average w/o outliers	QRangeQ=5,95
R	L	$\langle 0, 25 \rangle, \langle 75, 100 \rangle$	pessimistic and optimistic	QRLinearQ=0,25,75,100
R	NL	$\langle 0, 25 \rangle, \langle 75, 100 \rangle$		QRSqrtQ=0,25,75,100

TABLE VI
CONCRETE QUANTILE-BASED REWARD FUNCTIONS

selected for these example reward functions are listed in Table VI. The type column, **T**, indicates if the reward functions uses (P)oints or (R)anges; this aspect of the reward function is also evident in the Quantiles column, where ranges are listed in angled brackets. The combination column, **C**, indicates how multiple points or ranges are combined: (L)inearly or (N)on-(L)inearly. Evaluation name provides the notation used to identify each inliner in the figures in [1].

The inliners using these reward functions are evaluated according to the 3-fold cross-validation methodology proposed in [1]. The testing and training sets are determined once from a random ordering of the inputs in $\mathcal{W}$. Identical testing and training sets are used in each fold by each inliner. Each input in $\mathcal{W}$ is in exactly one testing set, thus the workload performance of an inliner is determined by taking the usual geometric mean:

$$\mu_g = \sqrt[|\mathcal{W}|]{\prod_{i \in \mathcal{W}} \tau^{-1}(i)}$$

E. Programs and Inputs

This study evaluates the inliners described above using four programs: `bzip2`, `gzip`, `gcc`, and `gobmk`. Each program is evaluated using a 15-input workload, as suggested in [1]. `Gcc` and `gobmk` are taken from the SPEC CPU 2006 benchmark suite. SPEC provides 11 inputs for `gcc`. In spite of the challenges involved in creating new inputs for this benchmark, four[4] of the SPEC 2000 benchmark programs were converted to the single pre-processed file format. The converted programs are `bzip2`, `LBM`, `mcf`, and `parser`. For `gobmk`, SPEC provides 20 inputs. However, only 5 of these inputs come from the `ref` workload; the `train` workload contains 8 inputs, and the `test` workload contains 7 inputs. Many of the inputs from `test` and `train` have very short execution times: 4 inputs take less than 1 second, 6 take 2–9 seconds, 4 take 12–19 seconds, and 1 takes longer than 1 minute. Execution times of less than a few seconds are subject to large proportional timing imprecision, because the Linux `time` command reports times with a resolution of $1/100^{th}$ of a second. Therefore, the 15 longest-running inputs are chosen for $\mathcal{W}$. This set is composed of the `ref` and `train` SPEC workloads, plus connect and dniwog from `test`. The shortest baseline running time in $\mathcal{W}$ is 2.3 seconds, for connect.

The other two programs used in the case study are `bzip2` and `gzip`. However, rather than using the SPEC benchmark versions of these programs, the fully-functional "real" versions are used. Using the real versions of the compressor programs eliminates the unrealistically-simplified profiling situation where mutually-exclusive use cases are combined into a single program run. Consequently, these programs cannot do decompression and compression, or multiple levels of compression, within the same run. These distinct use-cases must be covered by different inputs in the program workload. Both `bzip2` and `gzip` share the same workload of inputs. 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 audio books, but the audio is recorded in different formats, and the books themselves are different.

Both compressors use 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 compression set contains the following inputs, with the compression level shown in parentheses:

- `avernum (-3)`: The installer for the demo version of the game "Avernum: Escape from the Pit" from Spiderweb Software.
- `cards (-4)`: A collection of greeting card layouts in the TIFF (uncompressed) image format.
- `ebooks (-5)`: A collection of ebooks, with and without images, and in a variety of formats, from Project Gutenberg[5].
- `potemkin-mp4 (-6)`: The 1925 movie "Bronenosets Potyomkin (Battleship Potemkin)" in MP4 format, from the Internet Archive[6].
- `proteins-1 (-7)`: A sample of 33 proteins from the RCSB Protein Data Bank database. 6 files for each protein, each stored in a different text-based format, provide different characteristics of the protein's structure[7].
- `revelation-ogg (-8)`: The audio book "The Revelation of Saint John" in OGG format, from Project Gutenberg[8].
- `usrlib-so (-9)`: A collection of shared object (.so) files from `/usr/lib/` of a 32-bit gentoo-linux machine.

The decompression set for each compressor uses the same base set of files, pre-compressed by the appropriate compressor

[4]of seven attempts

[5]http://www.gutenberg.org
[6]http://archive.org/details/BattleshipPotemkin
[7]http://www.rcsb.org
[8]http://www.gutenberg.org/ebooks/22945

at the default compression level. The decompression set is composed of:

- `auriel`: The "Auriel's Retreat" land-mass addition mod by lance4791 for the game "The Elder Scrolls IV: Oblivion" from Bethesda Softworks[9].
- `gcc-453`: The source-code archive of the `gcc` compiler, version 4.5.3[10].
- `lib-a`: A collection of library files (.a) from `/lib/` of a gentoo-linux machine. As per the gentoo development guide, a library will be installed in `/lib` (boot critical) or `/usr/lib` (general applications), but not both[11].
- `mohicans-ogv`: The 1920 movie "Last of the Mohicans" in OGV (ogg video) format, from the Internet Archive[12].
- `ocal-019`: The Open Clip Art Library archive, version 0.19. The images are primarily in vector-graphics formats[13].
- `paintings-jpg`: A collection of watercolor paintings, in JPG format.
- `proteins-2`: A completely different sample of 157 proteins from the RCSB Protein Data Bank database, each in 6 different file formats.
- `sherlock-mp3`: The audio book "The Adventures of Sherlock Holmes" in MP3 format, from Project Gutenberg[14].

X. Related Work

Tuning compiler options is an expanding research area in compilers, as it can be observed in [17], where a full development environment to research on compiler tuning was proposed and developed. The model for compiler tuning uses a probability distribution to predict the actual empirical distribution of the compiler under study [18]. These ideas are a key motivating factor for our research.

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.

Krintz and Calder annotate Java bytecode with the optimization decisions made in previous program runs so that the JVM can exploit the benefits of those decisions immediately in subsequent runs [19]. However, this approach largely negates the inherent input-sensitivity of dynamic compilation. Sandya guides dynamic compilation with an off-line profile, but requires the user to specify a confidence level for the accuracy of that profile [20]. The N hottest methods in the off-line profile are candidates for dynamic compilation, and are compiled when a hotness threshold is exceeded. With a high confidence level, the frequency stored in the off-line profile is used to initialize each method's hotness. As the confidence level decreases, a reduced proportion of that frequency is used. On-line profiling contributes to each method's hotness during execution until a threshold is exceeded and the method is compiled. Thus, the input-sensitivity of the JIT can be maintained by setting a low confidence level for the off-line profile, but this approach largely negates the utility of supplying the off-line profile.

Arnold *et al.*. use histograms to combine the profile information collected by a Java JIT system over multiple program runs [21]. 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 [22]. 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 [23]. 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.

Shen *et al.*. investigate the sensitivity of Java garbage collectors to the inputs given to programs [24]. They find that varying program inputs can dramatically change the impact of garbage collection on program performance, and that the best garbage collector for a program is not consistent across inputs. These results suggest that a JIT that attempts to select the best garbage collector for the executing program at or

[9]http://planetelderscrolls.gamespy.com/View.php?view=OblivionMods.Detail&id=5949

[10]http://gcc.gnu.org/gcc-4.5

[11]http://devmanual.gentoo.org/general-concepts/filesystem/index.html

[12]http://archive.org/details/last_of_the_mohicans_1920

[13]http://openclipart.org/collections

[14]http://www.gutenberg.org/ebooks/28733

near the beginning of execution could greatly benefit from the behavior-variation information collected in a combined profile over many runs.

B. Inline-related

Arnold *et al.*. present an inlining strategy similar to that used in modern compilers [10]. 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%.

Zhao and Amaral investigate FDO inlining in the Open Research Compiler (ORC) [25]. The original benefit of inlining a call site in the ORC is measured by a *temperature* metric that takes the ratio of execution time spent in the callee compared to the total program execution time, weighted by the normalized frequency of the call compared to the invocation frequency of the caller. Temperature is intended to identify frequently-executed call sites to small callees, but infrequent calls to functions containing high trip-count loops will also result in a high temperature. An improved metric corrects this deficiency. A second improvement proposed in this work is adaptive inlining, which allows the temperature threshold required to inline a call site to vary with program size. Thus, smaller programs, which tend to benefit more from inlining, allow more aggressive inlining than larger programs, which tend to suffer more from the negative effects of excessive code growth.

Kulkarni *et al.*. considered the automatic construction of heuristics for inlining using Machine Learning techniques [2]. In this research they used an unsupervised learning model based on NEAT (Neuro-Evolution of Augmenting Topologies) method to generate new heuristics for inlining in two different platforms: The Maxine VM, and the HotSpot VM. They reported a speedup of 11 % on average on the Maxine VM, and 15 % on average on the Java HotSpot VM.

XI. Conclusion

Comparing the ...

References

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

[2] S. Kulkarni, J. Cavazos, C. Wimmer, and D. Simon, “Automatic construction of inlining heuristics using machine learning,” in *Code Generation and Optimization (CGO)*, Shenzhen, China, February 2013.

[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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] R. W. Scheifler, “An analysis of inline substitution for a structured programming language,” *Commun. ACM*, vol. 20, no. 9, pp. 647–654, Sep. 1977. [Online]. Available: http://doi.acm.org/10.1145/359810.359830

[12] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[13] S. Zhong, Y. Shen, and F. Hao, “Tuning compiler optimization options via simulated annealing,” in *Future Information Technology and Management Engineering, 2009. FITME ’09. Second International Conference on*, dec. 2009, pp. 305 –308.

[14] R. Nabinger-Sanchez, J. N. Amaral, D. Szafron, M. Pirvu, and M. Stoodley, “Using machines to learn method-specific compilation strategies,” in *Code Generation and Optimization (CGO)*, Chamonix, France, April 2011.

[15] J. C. Spall, *Stochastic Optimization: Stochastic Approximation and Simulated Annealing*, 1st ed. New York, NY, USA: Wiley, 1999, vol. 20, pp. 529–342.

[16] ——, *Stochastic Optimization*, 2nd ed. Heildelberg, Germany: Springer-Verlag, 2012, pp. 173–201.

[17] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather, C. Williams, M. O’Boyle, P. Barnard, E. Ashton, E. Courtois, and F. Bodin, “MILEPOST GCC: machine learning based research compiler,” in *Proceedings of the GCC Developers’ Summit*, Jul. 2008. [Online]. Available: http://hal.inria.fr/inria-00294704/fr/

[18] E. V. Bonilla, C. K. I. Williams, F. V. Agakov, J. Cavazos, J. Thomson, and M. F. P. O’Boyle, “Predictive search distributions,” in *Proceedings of the 23rd international conference on Machine learning*, ser. ICML ’06. New York, NY, USA: ACM, 2006, pp. 121–128. [Online]. Available: http://doi.acm.org/10.1145/1143844.1143860

[19] C. Krintz and B. Calder, “Using annotations to reduce dynamic optimization time,” in *Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, 2001, pp. 156–167.

[20] S. M. Sandya, “Jazzing up JVMs with off-line profile data: Does it pay?” *SIGPLAN Notices*, vol. 39, no. 8, pp. 72–80, Aug. 2004.

[21] 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.

[22] 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.

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

[24] X. Shen, F. Mao, K. Tian, and E. Z. Zhang, “The study and handling of program inputs in the selection of garbage collectors,” *SIGOPS Opererating Systems Review*, vol. 43, no. 3, pp. 48–61, Jul. 2009.

[25] P. Zhao and J. N. Amaral, “To inline or not to inline? Enhanced inlining decisions,” in *Workshop on Languages and Compilers and Parallel Computing (LCPC)*, October 2003.