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 the profiling makes any difference or not - by means of a random choice of functions to inline.
4) Tune in the constant values of the system.
5) Define which strategy it more appropriate to use in this approach to inlining, simple sort, or a more sophisticated choice, based on the knapsack problem.

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 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. 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 [2]. 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 VIII).
- *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 VII).

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

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 [3], [4], trace-based scheduling and others [5], [6].

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 [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 and HN have been presented in previous work [7], [2]. 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.

Decomposing a CFG into a hierarchy of dominating regions to enable HN is achieved by constructing its dominator tree. Each edge in the CFG is represented by a node in the dominator tree. Denote the immediate proper dominator of CFG edge e by $dom(e)$. Each non-leaf node $n(e)$ in the dominator tree is the head of a region G_e, which, by construction, encompasses any regions entered through descendants of e. To prepare a raw profile for combination with other profiles, the frequency f_e of each non-root node $n(e)$ is normalized against the frequency of its immediate proper dominator, $f_{dom(e)}$. The ratio of these two frequencies is invariant when a branch probability or loop iteration count is (dynamically) constant. This process also prevents variable behavior in an outer loop from masking consistent behaviors within the loop. Normalization proceeds in a bottom-up traversal of the dominator tree, so that the head of a region is normalized to its immediate dominator only after all of its descendents have been normalized. The root of the dominator tree, *i.e.*, the edge representing entry into the procedure, is assigned a "normalized" value of 1.

B. Denormalization

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

C. Queries

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

$H.\text{min}; H.\text{max}$

$H.\text{mean}(incl0s)$

$H.\text{stdev}(incl0s)$

$H.\text{estProbLessThan}(v)$

$H.\text{quantile}(\text{q})$

$H.\text{applyOnRange}(F(w, v), vmin, vmax)$

$H.\text{applyOnQuantile}(F(w, v), qmin, qmax)$

$H.\text{coverage}$

$H.\text{span}$

The mean value of a monitor is analogous to the value provided by a single raw profile, and provides the desired substitutability of a CProf for a raw profile in existing FDO transformations.

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. Extensions and 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.

1) CFG Paths: An algorithm that collects path profiling in a program that contains loops must break cycles. The most commonly used technique to break such cycles is due to Ball and Larus [8]. Given a simple loop, the main idea is to replace the back edge with a set of sub-paths that include (a) a path from a point outside the loop to the end of the first iteration, (b) a path from the loop entry point to a point outside the loop, and (c) a path from the entry point to the exit point in the loop.

Hierarchical normalization must be adapted to work with paths because there are no dominance relationships between paths. Consider two runs of a double-nested loop, where the outer loop L_1 iterates a total of $k = 10,000$ times in the first run, and $k = 100$ times in the second run. In both cases the inner loop L_2 iterates 10 times per iteration of L_1. A combined profile should identify the path of execution within L_1 as consistent across runs, but should indicate that the frequency of the paths into and out of L_1 vary significantly from run to run. The solution is to normalize path frequencies with respect to the frequency of the vertex that starts the path. For instance, P4 should be normalized to the frequency of L_2 to factor out k and preserve the constant nature of the inner loop.

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

```
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]

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 eliminated 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

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

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

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.

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 sum-

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
```

mary 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, $\mathsf{Score}(\mathrm{CS})$, 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 $\mathsf{Reward}(\mathsf{Benefit}(\mathrm{CS}), R_{\mathrm{CS}})$ 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 $\mathsf{Benefit}(\mathrm{CS})$ and a cost function $\mathsf{Cost}(\mathrm{CS})$ described in this section, an inlining candidate's Score is conceptually computed:

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

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.

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 is 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.

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.

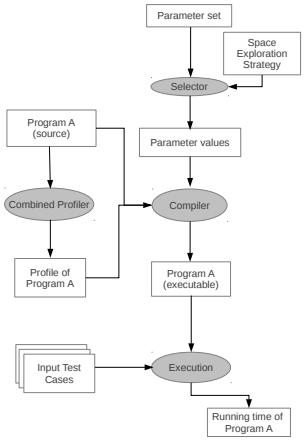
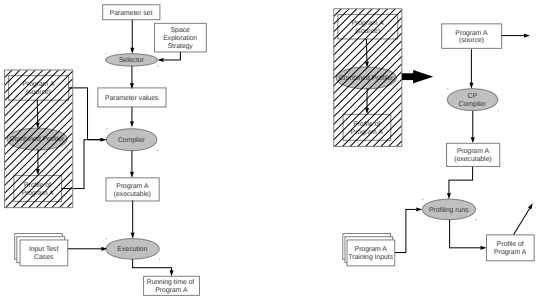


Fig. 2. Generic view of the method



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

Fig. 3. CP compiler and the profile generation

A. Parameter Tuning

Tuning compiler optimization options for a specific program is not an easy task [10]. There exist some tools based on Genetic Algorithms, Simulated Anealling [10], Support Vector Machines [11], etc. For the case of inlining a recently published research pointed to the use of Neural Networks to induce effective inlining heuristics [12]. Parameter tuning is a proficuous area of research.

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

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 [10], or SPSA - Simultaneous Perturbation Stochastic Approximation [13], [14], 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.

VII. 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 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.

VIII. 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.

IX. Related Work

Tuning compiler options is an expanding research area in compilers, as it can be observed in [15], 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 [16]. These ideas are a key motivating factor for our research.

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.

An early attempt to combine profiles is due to Fisher and Freudenberger. They measure instructions per break in control flow and sum profiles to provide better branch prediction [17]. Such summations produce similar results to summing normalized frequencies. While better than single-run profiles, they still yield poor behavior modeling in the presence of multiple program use cases and poor training input selection.

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 [18]. 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 [19]. 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 [20]. 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 [21]. 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 [22]. 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 [23]. 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 near the beginning of execution could greatly benefit from the behavior-variation information collected in a combined profile over many runs.

X. Conclusion

Comparing the ...

References

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

[2] 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)*.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] T. Ball and J. R. Larus, "Efficient path profiling," in *Intern. Symposium on Microarchitecture (MICRO)*, Paris, France, 1996, pp. 46–57.

[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] 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.

[11] 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.

[12] 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.

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

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

[15] 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/

[16] 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

[17] J. A. Fisher and S. M. Freudenberger, "Predicting conditional branch directions from previous runs of a program," in *Intern. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1992, pp. 85–95.

[18] 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.

[19] 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. [Online]. Available: http://doi.acm.org.login.ezproxy.library.ualberta.ca/10.1145/1026474.1026485

[20] 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.

[21] 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.

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

[23] 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. [Online]. Available: http://doi.acm.org.login.ezproxy.library.ualberta.ca/10.1145/1618525.1618531