A Study of Profile Guided Optimizations for the LLVM Compiler Infrastructure

CS243: High Performance Architectures and Their Compilers

Final Project Report

Christopher A. Wood, Julian Lettner, Brian Belleville

December 13th 2013

Table of Contents

Executive Summary

Software performance is tightly coupled to many factors, including the the relative size of the input, the specifications and features of the machine running the code, and interference from other background tasks. It is difficult for a compiler to produce optimal code for every use-case with only the information that is available through compile-time static analysis. Accordingly, there has been a tremendous amount of research over the past two decades focused on integrating details of a program's runtime behavior into the compilation workflow to enable more platform and use-case specific optimizations. This information is generally obtained by generating a profile of the program's execution with a representative input, and optimizations that use this profile are commonly referred to as profile guided optimizations (PGO). Profile guided optimizations can be beneficial for programs that need to achieve high performance for a certain use case, however this specialization is often specific to a particular input and type of machine, and may not be ideal for programs that are intended to be platform- or machine-independent with no "common" use case.

Due to the continued research in this area and potential benefits that can be gained from proper usage of program profile data, we decided to focus our term-long project on integrating a variety of PGO passes into the LLVM compiler infrastructure. In this report we detail our implementation strategy for a such PGO techniques and present the preliminary experimental evaluation of our results. As per the project specification, our goal was not to implement better solutions than existing compilers with respect to resulting program performance and compile-time overhead, but rather to understand the PGO techniques in detail and the many intricacies that are involved with compiler development that make it one of the more challenging areas of computer science.

LLVM and its Optimization Passes

The LLVM compiler infrastructure supports scalar, interprocedural, simple loop, and profile guided optimizations [7]. Scalar optimization passes are based on the data flow of a program and often include techniques such as constant propagation, redundancy elimination, range propagation, and expression simplification. Interprocedural optimization (IPO) is a class of optimization techniques that consider multiple (or all) functions simultaneously. Examples of IPO techniques are function inlining and dead-code elimination. Loop optimizations, as the name implies, focus on improving the execution of loops by rearranging statements in loop bodies and unrolling or merging loops. Profile guided optimizations are any optimization that uses data collected from the program at runtime to guide further optimization. Common examples of optimizations that can be enhanced by profile information are function inlining and loop unrolling. See Figure 1 for an illustration of this two-pass compilation technique. Figure 2 compares standard compilation and compilation with profile guided optimizations.

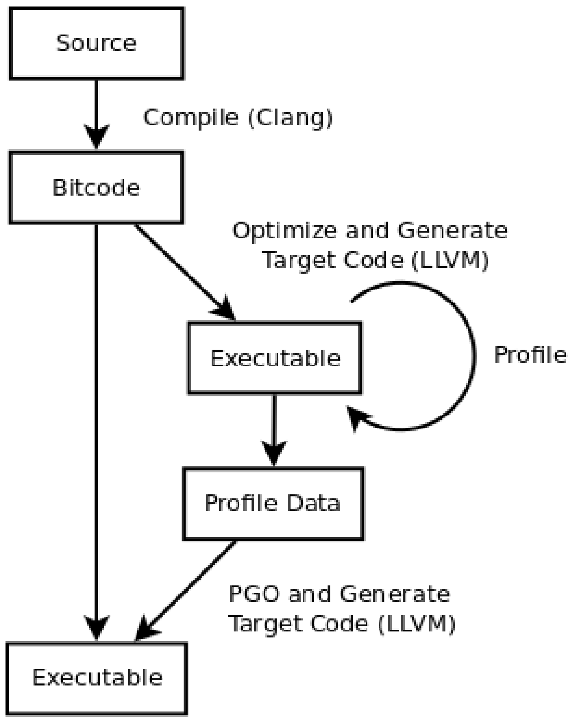
Source
Compile (Clang)
Bitcode
Optimize and Generate Target Code (LLVM)
Executable
Profile
Profile Data
PGO and Generate Target Code (LLVM)
Executable

Figure 1. PGO overview from high-level source to an executable that is compiled in two phases.

The core of the LLVM compiler infrastructure consists of a source- and target-independent optimizer, and code generation support for several different CPU architectures. These tools operate on the LLVM intermediate representation (IR), also called LLVM bitcode [11]. Since the optimizer is source code independent, we also require a compiler that can emit LLVM IR. We chose to use Clang, which is a subproject of LLVM that provides a C, C++, and Objective-C front end [12]. After a program has been compiled to the LLVM intermediate representation, the LLVM optimizer, ***opt***, can be used to run specific optimizations or analyses on the bitcode file [7,11]. Optimization passes are given specific identifiers so that they may easily be invoked from the command line. For example, the identifier “inline” corresponds to the function inlining IPO pass and can be invoked using the -inline flag.

Passes are scheduled and invoked using the LLVM PassManager class. Ultimately, this component is responsible for verifying pass dependencies are satisfied, pipelining the execution of passes on the program for efficient compilation, and sharing analysis results between passes. For example, branch weight metadata that (statically) specifies how likely a particular conditional branch might be the target of a jump can be used in a pass that seeks to rearrange branch conditions based on this property. To implement a new pass for PGO, we must specify the pass dependencies for each and register these passes with the PassManager. Beyond these simple modifications to the LLVM backend and actually implementing the optimization passes using the LLVM API, there are no other changes that need to be made to the compiler. We remark that the very low degree of coupling in the LLVM backend makes the integration of new optimization passes straightforward.

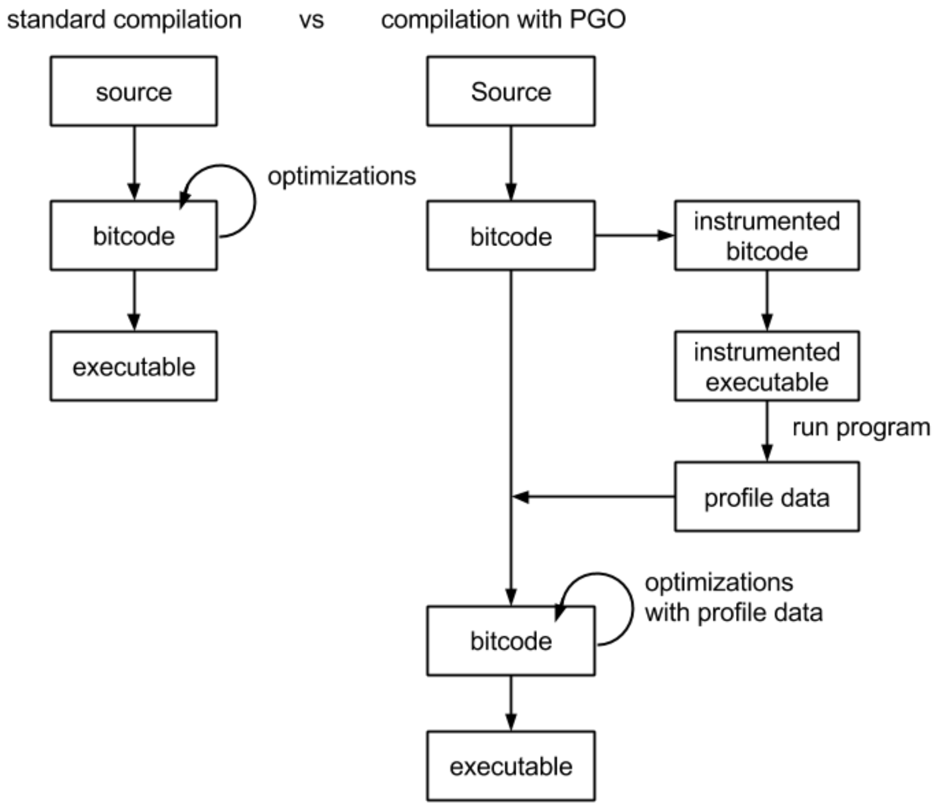

Figure 2. Conceptual comparison of standard compilation and compilation with profile guided optimizations.

Select Profile Guided Optimizations

Profile information can be used for many different optimizations. In this project we chose to focus on four specific cases: function inlining, loop unrolling, basic block placement, and partial redundancy elimination. Profile guided optimization techniques are already a part of released versions of the GCC (internally referred to as feedback driven optimizations, or FDO) and Intel C/C++ compilers. By comparison, the support for PGO in LLVM is far from state of the art. We implemented four profile guided passes that use a 'traditional' compile-profile-recompile workflow on LLVM 3.3. However the LLVM developer community has decided the added development cost needed to collect profile data and then re-run the LLVM optimizer and code generator was too high and going forward has decided to abandon support for this workflow in lieu of an automated, hands-off PGO pass [4]. As of writing this report there is no released version of LLVM that includes the automated PGO techniques. In fact, the current LLVM master development branch does not include any PGO capabilities.

The remainder of this section is dedicated to high-level descriptions of the aforementioned PGO techniques in more detail. Particular implementation details (e.g., algorithmic techniques to realize these optimizations, data structures required, program representation assumptions, etc) are discussed in the subsequent section.

Function Inlining

PGO-based function inlining has the potential to significantly improve the performance of an application by inlining call sites based on how frequently executed or “hot” they are. An example is shown in Figure 3. If it is known that call site P1 for a function F is executed significantly more times then P2 and P3, and the cost of inlining F into P2 and P3 degrades program locality, the PGO function inliner may strive to only expand F into P1.

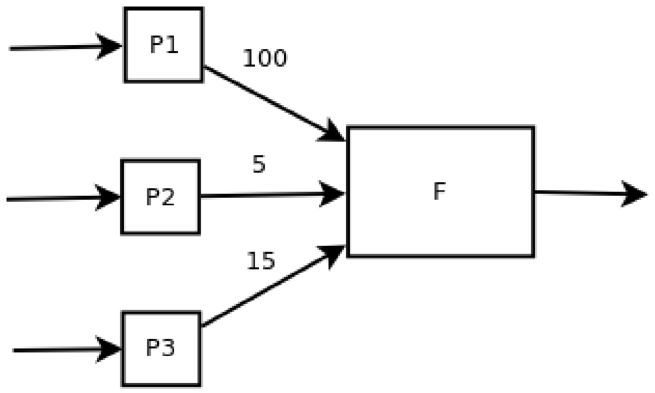

Figure 3. Inlining example based on a sample CFG subgraph.
The size of F is emphasized to reflect the cost of inlining.

Loop Unrolling

Loop unrolling is a common technique to avoid unnecessary conditional checks at every single iteration. However, if the loop termination count is unknown (i.e., it can not be computed at compile-time), then it is not possible to determine the optimal depth of loop unrolling at compile-time. Using information gathered through the profiling process, which hopefully includes the average number of times a particular loop is iterated, this profile guided optimization can unroll the loop to the degree indicated by the average iteration count. While conditional checks will still need to be inserted before every statement block in the unrolled loop, the cost of jumping to the loop conditional test will be decreased.

Basic Block Placement

PGO basic block placement is a form of global code placement that uses program control flow graph (CFG) metadata to extract edge weights, which reflect the frequency with which basic block transitions occurred at runtime, to properly re-arrange the basic block instructions to exploit hardware predictions. Put simply, hot basic block paths through the program CFG are sorted in descending order in such a way so that there are less jumps between unadjacent blocks, or rather, longer sequences of code are executed before branches are taken. The net effect of this placement strategy is that the distance between hot blocks in the CFG is minimized, which, improves cache locality since the code and data for these basic blocks will be located closer together than the code and data for the infrequently traversed blocks. See Figure 4 for a visual depiction of this technique.

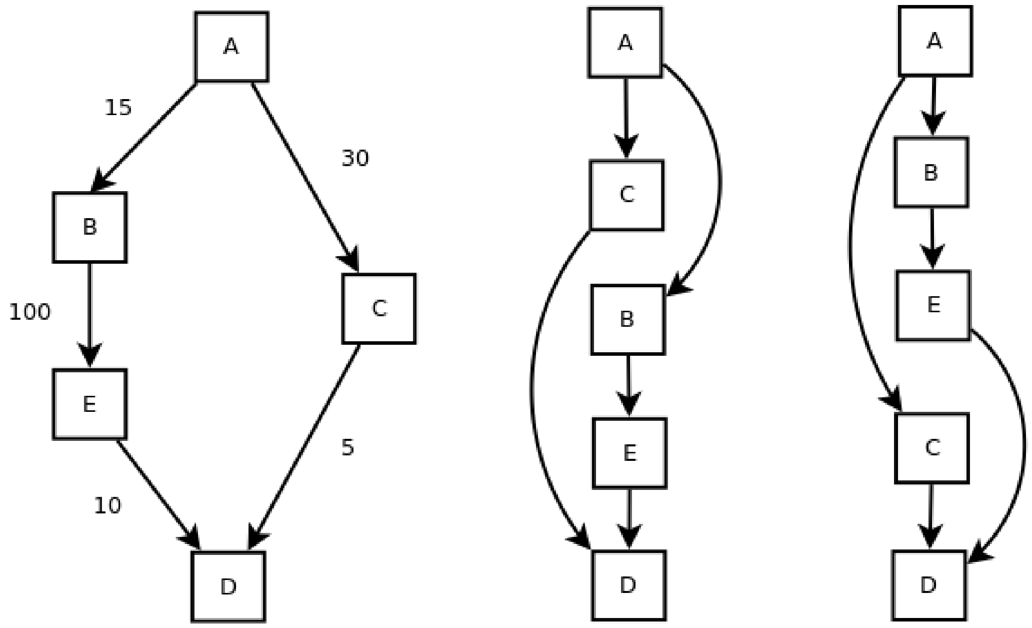

Figure 4. Basic block placement example. The ideal placement strategy is on the far right as it keeps the most frequently traversed blocks close for cache locality.

Partial Redundancy Elimination

Partial redundancy elimination (PRE) is a well-known form of code motion that aims to eliminate redundant expressions on some, but not necessarily all, code paths in a program CFG [10]. In essence, partially redundant expressions are destroyed by moving equivalent expressions into some of the parent nodes in the CFG, storing the result in a new temporary variable, and then using this temporary variable in place of the original redundant expression. This type of code motion is depicted in Figure 5 below.

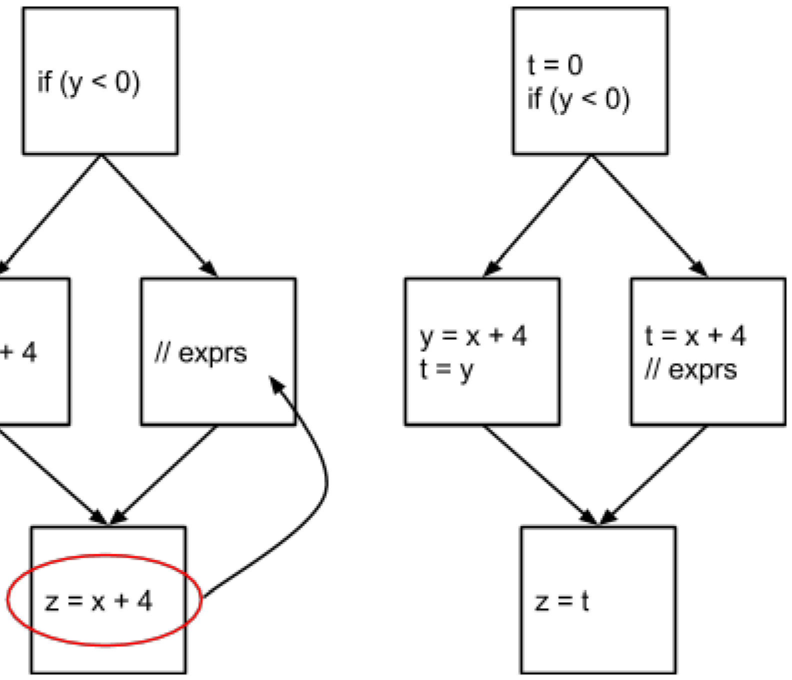

Figure 5. Sample PRE code motion that removes the redundant expression “x + 4” by moving the expression to the parent node that did not contain this expression, storing the result in a temporary, and then using the temporary in place of the original redundant expression.

Observe that the ultimate goal of PRE is to outperform traditional loop invariant code motion and common subexpression elimination. However, with an overly aggressive PRE pass, the resulting code size can actually increase by a significant amount, thus negating any potential benefits to be had. As such, there is no steadfast rule for determining which expressions to eliminate. Rather, such decisions are left to heuristics and compiler designer intuition. In this project, we explore the potential benefits that can be gained from leveraging profile information to make this decision, as we will discuss in the implementation section below.

Implementation Details

In this section we will discuss our implementations of the aforementioned PGO passes. The source code for this project is publicly available on GitHub (https://github.com/yln/llvm-pgo), and the relevant code that we wrote for each pass and our experiments is included in the submission archive alongside this report. This includes our PGO passes, experiment setup and execution scripts, and output parsing scripts. Documentation about the purpose of each file is contained therein.

Function Inlining

Function inlining expands the body of a function in place of a call site. This can speed up the program by removing the cost of the call and return, as well as the cost of the function prologue and epilogue. Inline expansion can also reveal additional optimization opportunities once the function body has been expanded. However, expanding a function will often increase code size, which can have a negative effect on performance if, for example, the code can no longer fit in a cache line. The approach that we took was to enhance the existing function inlining pass of LLVM with a bias based on the profiling information. This bias will cause call sites on hot code paths to be more likely to be inlined, and those on cold paths to be less likely to be inlined. We hope to get a benefit from both types of decisions. Deciding not to inline a function on a cold path can minimize the increase in code size and offset the additional cost to inline a function on the hot path, where the program will spend most of its time.

The existing LLVM inlining pass examines each call site and determines the cost of inlining the function at that call site, and the cost threshold that cannot be exceeded for the call site. The analysis determines these values as abstract units, and inlining is attempted as long as the cost is below the threshold. The main factor the current analysis takes into account is the number of instructions in the function body. In addition, it also assigns bonuses for criteria such as a large percentage of vector instructions and will identify if a function is completely unsuitable for inlining due to the presence of expensive constructs such as allocating a large amount of data onto the stack or recursive function calls. To leverage this existing analysis, our work introduces another factor that is used to determine the inlining threshold. We make use of the profiling information to determine the maximum execution count of any basic block in the program, and compare that to the execution count of a specific call site. This allows us to determine if a call site is hot or cold relative to the rest of the program. We then assign the profile guided threshold bonus based on this information.

The simplest way to do this is by using a linear function that assigns a value based on the simple ratio between the current call site's execution count and the maximum execution count, however execution count of basic blocks are not distributed linearly. To more evenly spread the threshold bonus we use a logarithmic distribution similar to the one proposed by Homescu et al. [6]. The form of the distribution is shown in Equation 1. Call sites with execution counts approaching 0 will receive a penalty of O, while those approaching the maximum execution count will receive a bonus of $M - O$. The constants M and O are in the abstract units that LLVM uses to represent the cost threshold.

$$T_b = \frac{log(1+E)}{log(1+E_{max})} M - O$$

Equation 1. The heuristic used for assigning the profile guided threshold bonus.

To get good results it is important to pick good values for constants M and O. The values we used were determined using the following search procedure. An initial search was performed along the line described by Equation 2, starting from $O = 0$ to $O = 10000$. These were chosen since pairs (M, O) along this line will result in a maximum penalty of $-O$ and a maximum bonus of $+O$. At each point along our search we performed a subset of the benchmarks used in our experimental evaluation and recorded the result. We then the used the best point found in this manner as the start point for the next phase of the search. From that start point we searched for a local optimum point using a hill climb search. In this way we optimized the constant factors in our distribution.

$$M = 2O$$

Equation 2. The line along which the initial search for constants was performed.

Loop Unrolling

Profile data may help to leverage loop unrolling even if the iteration count cannot be computed at compile-time. On the other hand even if the execution count for a loop can be determined at compile-time, the loop might exit early through jumps in its body. Without additional restrictions, this is the case for almost all loops in high level languages because of exceptions, which affect control flow. For simplicity we focus on *counting loops*, that is, loops with a single induction variable and a body that contains no jumps to locations outside the loop. The last instruction in such a body is always a jump to the loop header. This means that the loop will never exit prematurely as the only way to exit the loop is through its header by failing the loop conditional. The following is an example for a simple *counting loop*.

Simple counting loop example

```
for (i = 0; i < count; i++) {
  <body without jumps across loop boundary>;
}
```

Note that also more complex loops can be often canonicalized to this form [2].

Canonicalization example

The `-indvars` transformation pass in the LLVM infrastructure turns
`for (i = 7; i*i < 1000; i++)` into
`for (i = 0; i != 25; i++)`

The primary goal of loop unrolling is to trade increased code size for better performance. The two most significant reasons why loop unrolling can benefit performance are: Fewer jump, and, for compile-time loop unrolling, conditional instructions need to be executed. Hence the ratio of “loop bookkeeping” instructions to actual body instructions is improved. The other reason is to improve loop and instruction level parallelism (ILP). If subsequent iterations can be executed in parallel then loop unrolling may allow for greater ILP. A positive side effect of loop unrolling is that it reduces the impact of branches on the pipeline.

There is a limit to the benefits provided by loop unrolling. According to Amdahl's Law, the amount of amortized overhead decreases with each extra unrolling. In addition, a larger code size may hurt performance due to increased instruction cache miss rate and increased register pressure [3]. Therefore it is not beneficial to fully unroll every single loop even if the execution count is known at compile-time. This is especially true for loops with large bodies and loops contained in recursive functions. Loops with high iteration counts may be unrolled partially to achieve a good trade-off between performance and code size.

Figure 5 compares the conceptual code layout of a standard, statically unrolled and dynamically unrolled loop. Using ***clang*** or ***gcc*** the standard loop layout can be obtained by compiling with an optimization level of -O2 or below. The -O3 flag on the other hand enables optimizations that involve a space and speed trade-off [5]. When this flag is used and the execution count of a loop can be computed at compile-time then the compiler will repeat the loop body n times and eliminate the header altogether. If the execution count is large the loop may be partially unrolled. In this case n is smaller than the execution count and the compiler still needs to emit a rudimentary header.

With the runtime-data acquisition feature of the LLVM infrastructure we can extract the total number of loop iterations k for one or more profiling runs of a program. Although we can measure k accurately, we cannot rely on it being it the same for subsequent executions. Therefore the replicated loop bodies need to be protected by conditional branches for correctness. In theory, profile guided loop unrolling can still benefit performance because we need to execute fewer jump instructions as well as potentially enabling a higher degree of ILP. We investigate this claim in our experimental evaluation.

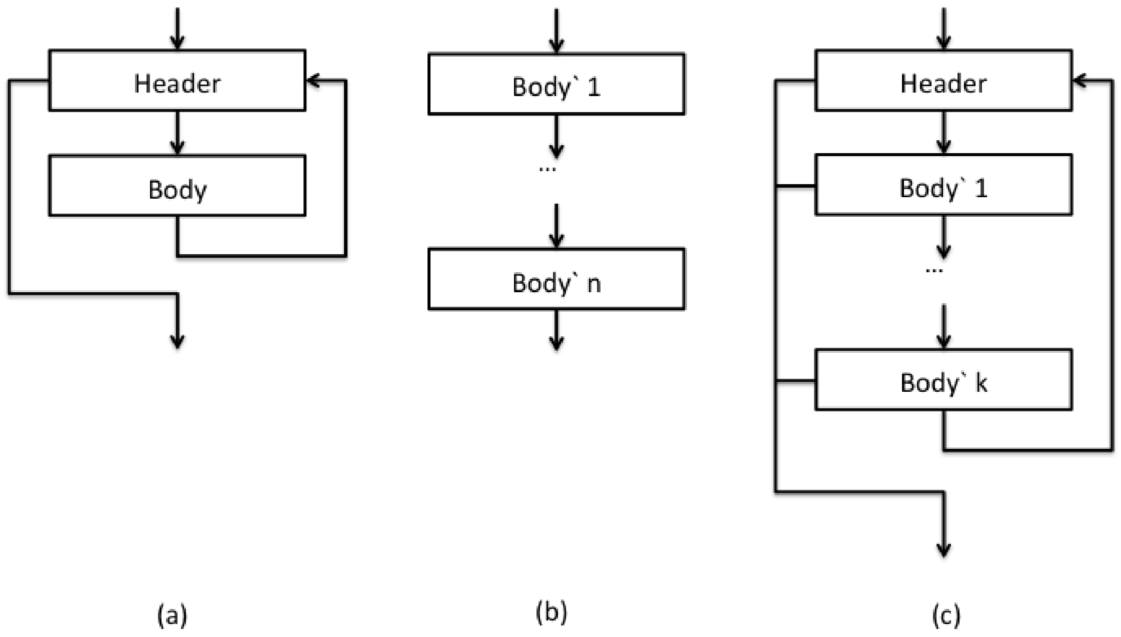

Figure 5. Conceptual code layout for
(a) standard, (b) statically unrolled (c) and dynamically unrolled loops.

Our experimental LLVM loop unroll pass implements the following strategy. Loops are unrolled by the next highest power of 2 with respect to k and up to a threshold of 128. The threshold was determined empirically and is inspired by the default threshold (150) for static loop unrolling in LLVM. If k is smaller than 2 then the loop is not modified. For example, if $k = 0$, then nothing is modified. whereas if $k \in \{5,6,7,8\}$, then the loop is unrolled 8 times.

The described implementation implements dynamically unrolled loops which reduces the number of executed jump instructions over standard loops. Another idea is to also remove the conditional branches that guard each loop body. The purpose of the guards is to ensure that the loop body is not executed more often than it should be. Our idea is to jump directly to the copy of the loop body which has the right number of successors. This eliminates the need for the guards. Figure 6 illustrates this idea. To make it work, we first have to ensure that the loop is a *counting loop* as defined above (single canonicalized induction variable). Then we unroll the loop similarly to dynamic loop unrolling but without guarding replicated loop bodies with conditional branches. Every copy of the loop body has the same size and increments the induction variable. The loop header, instead of checking the loop conditional, computes an offset and jumps to the appropriate body.

Let k be the unroll count (i.e., number of replicated bodies), i the induction variable, and $count$ the runtime iteration count. The number of remaining loop iterations r is then given by $r = count - i$. As long as $r \geq k$, we want to execute all bodies and so $offset = 0$. Otherwise, if

$r < k,$ we want to make sure that we execute only r bodies, so we set $offset = k - r$. We can cover both cases by computing the offset as $offset = k - min(r, k)$ (see Equation 3). Clearly, if $offset = 0$ then we enter the first body, and if $offset = k$ then we jump behind the last body and exit the loop.

$$offset = k - min(count - i,\ k)$$

Equation 3. Offset calculation formula.

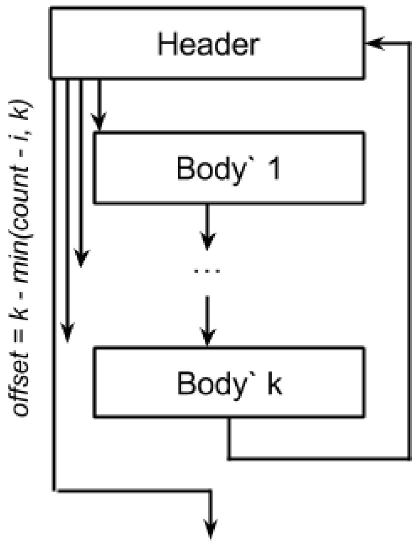

Figure 6. “Offset-jumping” loop layout.

Basic Block Placement

The use of profile data for global code positioning was first investigated by Pettis and Hansen from HP Labs in 1990 [1]. In their seminal work they presented two different algorithms for basic block placement; one uses a top-down breadth-first traversal of the program CFG, and the other is a bottom-up placement method. The last stable release of LLVM implemented the first algorithm, as it conceptually simpler than the latter. We implemented the bottom-up placement method, but for completeness, we also provide a discussion and pseudocode for top-down algorithm here.

The original recursive algorithm for basic block placement is conceptually simpler than the bottom-up iterative approach. The algorithm essentially performs a modified breadth-first traversal of the basic block tree rooted at the entry block for a particular function. In particular, starting with the entry block as the current block, the function iterates over all successors of the current block to identify the one that is visited the most at runtime (i.e. has the highest execution count), and then places that block after the current block. The algorithm then recursively calls itself on the new block and repeats the same procedure. Recursion terminates after all basic blocks have been placed, or simply, after all basic blocks have been visited recursively. The general procedure for this technique is shown in Algorithm 1. We use a stack instead of recursive function calls, but the idea is still the same.

Input: A function *F* with a list of connected basic blocks *BlockList,* and CFG weighted edges *EdgeList*.
Output: Updated function *F* with a list of newly connected basic blocks.

```
1. BlockStack = []
2. visited = []
3. BlockStack.push(F.entryBlock)
4. While BlockStack is not empty
     a. b = BlockStack.pop()
     b. maxCount = 0
     c. b* = Nil
     d. For each b' in b.successors()
          i. If b'.visitCount > maxCount
               1. maxCount = b'.visitCount
               2. b* = b'
     e. Place b* after b
     f. visited.append(b*)
     g. BlockStack.push(b*)
```

Algorithm 1. Recursive top-down basic block placement pseudocode (*algo1* from [1]).

The bottom-up algorithm which we implemented works by iteratively connecting chains of basic blocks together into a straight line according to the frequency with which they occur and their natural order in the program CFG. Specifically, the precedence of basic blocks is such that source blocks, which occur more frequently based on the arc information, should always occur before target blocks in the straight-line representation. Ordering the blocks in the opposite way would cause more jump instructions to be executed at runtime, thus hindering performance. We present the basic pseudocode for this procedure in Algorithm 2 below. The only non-trivial function is *NumberOfConnections*(*c1, c2*), which computes the number of basic block connections between two chains *c1* and *c2*.

Input: A function *F* with a list of connected basic blocks *BlockList,* and CFG weighted edges *EdgeList*.
Output: Updated function *F* with a list of newly connected basic blocks.

```
1. ChainList = Initialize each basic block as a chain c of length 1
2. EdgeList = Sort EdgeList in descending order by weight
3. For each CFG edge e in EdgeList
     a. For each chain c1 in ChainList
          i. For each chain c2 in ChainList - {c1}
               1. If e.head == c1.head and e.tail == c2.head
```

```
                a. Append c1 to c2 in ChainList
4. c' = The chain starting with F.entryBlock
5. visited = []
6. For i = 1 to len(ChainList)
    a. max = 0
    b. c* = Nil
    c. For each chain c in ChainList - visited
        i.  connections = NumberOfConnections(c',c)
        ii. If connections >= max
            1. max = connections
            2. c* = c
    d. Append c* to c', append c* to visited, and set c' = c*
7. Build a list of basic blocks from the chain c' and insert them into F
8. Return F
```

Algorithm 2. Iterative bottom-up basic block placement pseudocode (*algo2* from [1]).

Partial Redundancy Elimination

Profile guided partial redundancy elimination, as introduced in [9], leverages program profile information about the CFG edge execution frequency to determine whether or not code motion should be enabled or disabled for flagged redundant expressions. In PRE terms, candidate expressions for elimination in a basic block B1 are those that are identified as partially available, meaning that it has been evaluated or computed by at least one path leading to B1. As shown in Figure 5, this means that the expression “z = x + 4” is partially available since it was previously evaluated in the left parent node. This expression can be safely hoisted to a higher basic block B2 if it is anticipable in B2, where anticipable means that on all paths from B2 to the final tail block in the CFG the expression is evaluated prior to the redefinition of any of its operands. Intuitively, this means that the expression is “busy” along all CFG paths starting at B2. Putting together the notion of expressions being available and anticipable at certain blocks, we see that a “safe” code motion point is in a block where an expression is both available and anticipable (i.e., down-safety is guaranteed). We omit the formal data-flow equations that can be solved to determine down-safety for brevity, referring the interested reader to [9] for more details.

As per the modified algorithm specified in [9], profile information is incorporated into the optimization pass to determine if code injection points (i.e., a spot between two instructions inside a function) are beneficial based on how frequently they are traversed according to the weighted CFG. Specifically, the *benefit* of moving a particular expression $expr$ to a specific injection point n is defined as the sum of execution frequencies of all available and anticipable paths, deemed the $BenefitPaths$, traversing this code injection point. That is,

$$Benefit_{expr}(n) = \sum_{p \in BenefitPaths} Freq(p) .$$

Analogously, the *cost* of moving a particular expression $expr$ to a specific injection point n is defined as

$$Cost_{exp}(n) = \sum_{p \in CostPaths} Freq(p) .$$

The authors also introduce the following normalized ratios that quantify the cost and benefit of a particular injection point n with respect to a candidate expression $expr$:

$$ProbCost_{expr}(n) = \frac{Cost_{expr}(n)}{Freq(n)}$$
$$ProbBenefit_{expr}(n) = \frac{Benefit_{expr}(n)}{Freq(n)}$$

Finally, code motion is enabled if and only if

$$ProbCost_{exp}(n) < ProbCost_{exp}(n)$$

Therefore, in order to implement this algorithm, we had to compute $Benefit_{exp}(n)$, $Cost_{exp}(n)$, $ProbBenefit(n)$, and $ProbCost_{exp}(n)$. To do so, we performed a breadth-first traversal of the instructions in the candidate program CFG to generate all possible sub-paths through the program. Then, for each pair of instruction $expr$ and basic block n in the program, we compute all available, unavailable, anticipable, and unanticipable subpaths with respect to $expr$ and n. Using these sets, we then construct $BenefitPaths_{exp}(n)$ and $CostPaths_{exp}(n)$. The benefit paths are formed by concatenating all combinations of available and anticipable paths together, whereas the cost paths are formed by concatenating all combinations of unavailable and unanticipable paths together. In doing so, the frequency of each path is also recorded as the minimum edge-weight frequency contained in the path. With these two data structures, the remaining functions are trivial to compute. Clearly, this implementation strategy is infeasible for large and complex programs with many instructions and possible execution paths in the CFG. However, for the programs considered in this project, the compile-time overhead seemed reasonable.

Finally, to leverage these data structures, we modified the existing GVN (global value numbering) pass in LLVM that performs PRE to incorporate this profile guided decision making heuristic. Due to time constraints, code motion of an expression $expr$ to injection point n is performed only in diamond cases (similar to the scenario described in the introduction to this section), and if code motion is enabled at n with respect to $expr$.

Experimental Evaluation

In this section we describe the results from each of our PGO passes. To ensure the accuracy of our results we designed the experiment as follows. For the 20 applications listed in Table 1 we defined five inputs of increasing size. For the smallest input, the execution time t takes less than one second (e.g., $100ms < t < 1s$) on our benchmark system. Our goal for the largest input was to make the program run for about one minute (e.g., $20s < t < 2mins$). The remaining three input sizes fall within this range.

For each input size we execute both versions of the program - baseline and optimized - five times and measure their running times. From the individual times we calculate the average baseline time and the average optimized time and then use these average times to calculate the relative speedup for that input size. Finally we average over the speedup for all input sizes to get the average relative speedup. Unless otherwise specified we refer to this number in the individual discussion sections.

In our experiment, we also considered how closely the profile information matches the input. In our experiment we repeated the procedure described above under three different scenarios. For the first we gather profile data for every input size and separately compile and optimize a program specialized for each input size. The second scenario tested the case where the profile data does not match the input, we gathered profile data just for the smallest input size and only compiled the program once. This version was then used for all of the input sizes. For the third scenario we followed the same procedure as the second, except the profile information was gathered for the largest input size. We intuitively expect that re-compiling with updated profile data for every input size results in the greatest benefit since the profile data exactly matches the characteristics of program execution.

Benchmark Programs

In order to quantitatively assess the effectiveness of our optimization implementations, we collected a wide array of benchmarking applications with different characteristics that, as a whole, stress at least one of the optimizations identified in this project. These benchmarks are implementations of a variety of common algorithms and were obtained from various sources, including the Computer Language Benchmarks Game [8]. Table 1 provides the details for each of these benchmarking applications. Note that all programs are sequential in nature and our optimization passes are not designed to exploit parallelism to increase performance since improving parallelism was not an intended goal of this project.

All of the listed programs are implemented in C and compiled with version 3.3 of ***clang***. The baseline version of a program is compiled the -O3 flag specified except for the measurement of function inlining. This was because much of the benifit from inlining a function comes from the additional optimization opportunities it can reveal, and we wanted to only measure what improvements can be obtained through making inlining decisions based on profile information. The -O3 optimization level enables almost all optimizations including those that involve a space-speed trade-off (inclusive function inlining and static loop unrolling). For the optimized version we apply our own optimization passes on top of the baseline version. In our evaluation of the results we will refer to performance in terms of relative speedup $S = T_b / T_{opt}$, where T_b and T_{opt} are the running times of the baseline and optimized program version respectively. Although we refrain from using absolute timings in our discussion we provide the specification for our benchmarking system for completeness: Ubuntu 13.04 32bit running on an Intel Core i7-3770 CPU (@ 3.4GHz x 8) with 16GB DDR3 memory.

Table 1. Description of benchmarking applications and the input size parameter that defines their behavior.

ID	Program Name	Description	Input Parameter
1	Matrix Multiplication	Computes the product of two square matrices composed of random integers.	Square matrix dimension.
2	Basic Block Test	This program infrequently executes a very computationally expensive block of code based on the input parameter - if the input parameter is large then the block of code is executed more frequently.	Frequency n with which the expensive block of code is executed (n times every 10000)
3	Permutation Generation	Generates all permutations of the numbers in the range $[1, n]$.	Number of integers in the set.
4	Bubble Sort	Sorts a random array of n integers using the bubble sort algorithm.	Number of integers in the random array.
5	Quick Sort	Sorts a random array of n integers using the quick sort algorithm.	Number of integers in the random array.
6	Mandelbrot Set Generation	Generates all points in a mandelbrot set up to n iterations for each point.	Number of iterations for each point in the mandelbrot set.
7	Trial Division Primality Test	Finds all prime numbers from 2 to n using trial division.	Upper bound on the number of integers to test for primality.
8	Combination Generation	Creates the set of all possible distinct, unordered combinations of size k from a total of n numbers.	Input is k, the size of combinations to choose. The total size of the pool will be $C(n, k)$.
9	Conjugate Transpose	Computes the conjugate transpose of a randomly generated $n \, x \, m$ complex valued matrix, and tests if the matrix is hermitian, unitary, or normal.	Two parameters, first is number of rows, second is number of columns. They must be different or else the program will crash.
10	Cholesky Decomposition	Performs the Cholesky decomposition of a positive-definite matrix.	Number of times to repeat the computation of Cholesky decomposition of the matrix.
11	Deconvolution	Performs deconvolution on three different 3-dimensional matrices of different sizes.	Number of times to repeat the computation of the

			deconvolution of the three different matrices.
12	Euler Method	Computes a solution to an ordinary differential equations using euler's method.	Number or times to compute the solution.
13	Integer Factorization	Computes all factors for the numbers in the range $[1, n]$.	Number of integers in the set.
14	Fast Fourier Transform	Recursively computes the discrete Fourier transform using the Cooley–Tukey algorithm for an array of n numbers.	Input is k, the size n of the array of numbers will be $8k$.
15	Huffman (heap)	Creates a Huffman code table using the priority queue construction algorithm. This includes counting the symbol frequencies for the input. The implementation builds the queue by allocating memory on the heap.	Input is k, the real input string is constructed by repeating a template string (40 characters) k times, hence n is $40k$.
16	Huffman (static)	Creates a Huffman code table using the priority queue construction algorithm and encodes and decodes the provided input. This includes counting the symbol frequencies for the input. The implementation builds the queue from static buffers.	Input is k, the real input string is constructed by repeating a template string (40 characters) k times, hence n is $40k$.
17	K-Means Clustering	Divides n points in the Euclidean plane into k clusters using Lloyd's algorithm.	Inputs: n and k. n is the number of points (random and uniformly distributed) and k is the number of clusters.
18	Lucas Primality Test	Finds all prime numbers from 2 to n using the Lucas primality test.	Upper bound on integers to test for primality.
19	2D N-Body simulation	Runs a simple 2D n-body simulation to calculate physical location of a fixed set of bodies in planetary orbit given some initial velocity vector and momentum.	Number of iterations of the n-body simulation.
20	AES-256 Encryption	Encrypts a single 128-bit block of plaintext a specified number of times.	Number of times a piece of plaintext is encrypted.

Function Inlining

The results for profile guided function inlining are shown in Table 2. We compare the execution times of programs compiled with our function inlining pass to the times of programs compiled with the inlining pass present in LLVM 3.3. Function inlining was the only optimization applied for this comparison. The average speedup was around 1.04 , both for the case where the profile data matches the input size and the case where it does not match the input size. This is notable, but not entirely unexpected for function inlining. All of the benchmarks chosen have a simple structure, and the control flow does not dramatically change based on the input size. This means that an inlining scheme that is found to be effective for one input size is expected to be beneficial for other input sizes as well. This is a consequence of the programs we used for benchmarks. We expect that other programs with greater variation in control flow for different input would see a stronger correlation between the benefit of profile guided inlining and how closely the profile data matches the actual input.

The general trend in the results is that most programs experienced a speedup or slowdown of a few percent, with a few benchmarks experiencing speedups greater than 10%. There is also a single negative outlier which was slowed down by over 10%. On average we found there to be a benefit, but that was mostly realized by a few programs that experience significant gains. The largest benefit was found on the Integer Factorization benchmark. This is a very small program which finds all factors of a set of numbers. With the LLVM inlining pass, none of the call sites are inlined since all of the functions in this program are larger than the threshold. Our inlining pass inlines calls that are on the inner loop, and for this micro-benchmark, that small change has a large effect.

The outlier in the opposite direction was AES-256 Encryption. The slowdown was greater than 10% and was unique among the benchmarks. Analyzing the output of the compiler revealed that our pass aggressively inlined calls into the function that performs the encryption, while the LLVM inlining pass chose not to inline the majority of the call sites inside that function. Our pass behaved as intended, all of the calls were on the hot path of the program, but the results indicate that the increase in code size dramatically slowed down the program. Our optimization generally resulted in a larger number of call sites being inlined than the LLVM inlining pass. Most of these benchmarks are small, with a simple call graph, and a single control flow path through the program. Most of the execution counts were clustered near the maximum, and as a result received similar bonuses.

Profile guided function inlining appears to be a worthwhile optimization. The speedups observed in our experiments are modest, but do indicate that on average we were able to speed up the execution compared to programs compiled with the default inlining optimization. An important aspect of function inlining is that it facilitates further optimizations. Once the body of a called function is relocated within the caller, it can then reveal possible optimizations on the transformed function that could not be performed on the original. Our results do not consider this effect, as we only ran the function inlining pass. A further speed up of the benchmarks could be achieved by also running other passes from LLVM, though we tested our optimizations in isolation from one another. That is, we did not test any other optimizations in combination with function inlining since our objective was to measure the impact of profile guided function inlining independent of other optimizations.

Table 2. Speedup results from the profile guided function inlining. Speedups and slowdowns greater than 10% are highlighted.

ID	Program	Speedup with Profile Data	Speedup with Profile Data	Speedup with Profile Data

		for Individual Inputs	**Collected for Small Input**	**Collected for Large Input**
1	Matrix Multiplication	1.124	1.148	1.189
2	Basic Block Test	0.984	0.983	0.968
3	Permutation Generation	1.034	1.032	1.001
4	Bubble Sort	1.089	1.088	1.088
6	Mandelbrot Set Generation	1.092	1.073	1.056
7	Trial Division Primality Test	1.143	1.172	1.142
8	Combination Generation	1.007	0.996	1.006
9	Conjugate Transpose	1.108	0.909	1.127
10	Cholesky Decomposition	1.018	1.009	1.019
11	Deconvolution	0.985	0.983	1.020
12	Euler Method	0.976	1.002	1.023
13	Integer Factorization	1.190	1.191	1.191
14	Fast Fourier Transform	1.010	1.012	1.011
15	Huffman (heap)	0.979	1.054	1.041
16	Huffmap (static)	0.961	1.090	1.047
17	K-Means Clustering	0.985	1.020	1.007
18	Lucas Primality Test	1.094	1.099	1.096
19	2D N-Body Simulation	1.104	0.918	1.042
20	AES-256 Encryption	0.898	0.912	0.900
	Average	**1.040**	**1.034**	**1.047**

Loop Unrolling

As expected, profile guided loop unrolling performs the best when the profile data matches the characteristics of the actual execution. On one hand, this is to be expected for all profile guided optimizations in general, i.e., we want our profile data to match actual execution patterns as close as possible. On the other hand it is especially true for loop unrolling, as atypical profile data affects loop unrolling more than, for example, function inlining which is a conceptually similar optimization.

The speedups achieved by our profile guided LLVM loop unroll pass hover around 1.0 for the most of the benchmarking programs as depicted by Table 3. The average speedup with profile data for individual inputs is 2.3%. The average changes in performance for runs with atypical profile data are negligible (-0.6% and 0.1%). When using profile data that matches the input for compilation 14 out of 20 programs execute faster. This means that loop unrolling degrades performance for 6 of our programs. Most increases and decreases in performance are small, that is, less than 10%.

We now analyze the outliers and some interesting trends observed in these results. The Mandelbrot set generation program shows significant speedups - about 22% and 27% for profile data gathered with individual inputs and smallest input respectively. For profile data gathered by the largest input the speedup dropped down back to 1.0. This is a good example for showing that increased code size can hurt instruction cache locality and on a greater scale may cause memory/page cache misses. Also, the potential gain becomes smaller for increasing unroll counts. We could try to mitigate those negative effects by choosing a smaller maximum unroll count or by making the heuristic for detecting large loop bodies smarter. Of course, thresholds and heuristics that perform well for one set of programs might be a poor choice for others. The significant speedups with profile data gathered from small inputs may be explained by the structure of the program, which is comprised of three nested loops computing the pixels of the Mandelbrot set (loop 1: y-coordinate, loop 2: x-coordinate, loop 3: pixel color). Unrolling, especially of the innermost loop, provides a real performance gain because its body is executed up to 200 times for every pixel. In addition, unrolling may also improve opportunities for improved instruction scheduling.

The most prominent negative outlier is the combination generation program. The optimized versions runs about 12% slower than the baseline version for all three different profile data setups. We can explain the slowdown by looking at the program source code and the LLVM intermediate representation (IR) for both the baseline and the optimized version. The slowdown is a result of an unfortunate combinations of two features. First, the program constructs combinations by means of recursion and prints out complete combinations in a loop. For benchmarking we removed the output to ensure that the program is not artificially slowed down, but added a side effect to make sure that the loop and the containing recursive function is not pruned because of dead code elimination. Second, our profile guided loop unroll pass uses the absolute numbers provided by the LLVM edge count profiling analysis, i.e., we unroll based on the absolute loop execution count and do not take into consideration how many times the containing function is called. This means that we unroll a loop for the maximum number of times (128) inside a highly recursive function, which hurts performance for obvious reasons. We think this is a good demonstration of how an optimization that involves a speed-space tradeoff can also hurt performance. Again we want to emphasize that it is very hard to find heuristics and thresholds that “do the right thing” in every situation. If we exclude the combination generation program from the computation of the average speedup it is bumped up to 3.0%.

Another interesting program is the Cholesky decomposition program. For individual profile data it showed a speedup of approximately 13%, when compiled with profile data for the small input performance decreases by 11%, and compilation with profile data for the large input again yields a speedup of approximately 7%. From these numbers we can observe that the optimizations applied to the program are very sensitive to whether or not the profile data matches the actual execution characteristics. Again, this can be explained by looking at the program structure. In essence, the Cholesky decomposition program consists entirely of matrix operations performed in three nested loops. The observed pattern (best for individual profile data, worst for small input, and in-between for large input) follows our initial intuition of how program performance changes for compilation with different profile data. In retrospect, this guess turned out to be just that - a guess - as many other programs do not follow this pattern.

In our experimental setup and for the considered set of programs, profile guided loop unrolling on its own did not provide significant speedups. It may, however, be still useful in specific scenarios or for enabling further optimizations, which we did not consider in our experiment.

Table 3. Speedup results from the profile guided loop unrolling. Speedups and slowdowns greater than 10% are highlighted.

ID	Program	Speedup with Profile Data for Individual Inputs	Speedup with Profile Data Collected for Small Input	Speedup with Profile Data Collected for Large Input
1	Matrix Multiplication	0.998	1.002	0.996
2	Basic Block Test	0.964	0.901	0.949
3	Permutation Generation	1.016	0.989	1.013
4	Bubble Sort	1.024	1.023	1.021
5	Quick Sort	1.064	0.995	0.967
6	Mandelbrot Set Generation	1.215	1.267	1.003
7	Trial Division Primality Test	1.010	1.010	1.018
8	Combination Generation	0.882	0.882	0.881
9	Conjugate Transpose	1.090	1.018	1.060
10	Cholesky Decomposition	1.126	0.888	1.074
11	Deconvolution	1.029	1.044	1.039
12	Euler Method	1.016	0.967	0.988
13	Integer Factorization	0.996	0.995	0.995
14	Fast Fourier Transform	0.985	0.986	0.985
15	Huffman (heap)	1.007	0.924	0.954

16	Huffmap (static)	0.921	0.971	0.967
17	K-Means Clustering	1.024	0.998	1.049
18	Lucas Primality Test	1.000	1.001	1.000
19	2D N-Body Simulation	1.077	1.021	1.031
20	AES-256 Encryption	1.009	1.008	1.011
	Average	**1.023**	**0.994**	**1.001**

Basic Block Placement

Generally speaking, the profile guided basic block optimization pass did not yield consistently significant speedups across all programs considered. As one can see in Table 4, the relative speedup ratio $S = T_b/T_{opt}$ fluctuated around even (i.e., 1.0) for the majority of the programs considered. Positive outliers included the basic block test, when optimized using profile data collected from small input sizes, and cholesky decomposition programs, when optimized and run on the same input size. Quite surprisingly, the same result for the basic block test program for the remaining optimizations was not observed, and in the case of being collected for large input, actually saw a performance degradation. The remaining negative outliers do not follow any discernible pattern (i.e., programs with a particular property suffered degradation as opposed to those that do not possess such property), which led us to conclude the following: The profile guided basic block placement optimization does not yield any significant performance improvement for “once and for all” recompilations of the (comprehensive set of) programs considered.

Indeed, this observation is supported by the fact that there is a significant lack of research focus on this particular optimization in modern compilers such as LLVM and GCC. Therefore, our results and observation mirror what the compiler community has already found. That is, other profile guided optimization passes will yield more utility than the basic block placement pass, which therefore means that the general belief in the compiler community is that that further improvements of the placement algorithm or implementation strategy are not justified.

Table 4. Speedup results from the profile guided basic block placement optimization using the iterative, bottom-up algorithm from [1]. Speedups and slowdowns greater than 10% are highlighted.

ID	Program	Speedup with Profile Data for Individual Inputs	Speedup with Profile Data Collected for Small Input	Speedup with Profile Data Collected for Large Input
1	Matrix Multiplication	1.000	0.999	1.001

2	Basic Block Test	1.039	1.607	0.891
3	Permutation Generation	0.992	0.979	0.980
4	Bubble Sort	1.022	1.029	1.021
5	Quick Sort	1.002	0.973	1.017
6	Mandelbrot Set Generation	0.933	1.120	0.834
7	Trial Division Primality Test	1.007	1.009	1.013
8	Combination Generation	0.884	0.884	0.881
9	Conjugate Transpose	1.022	0.998	1.054
10	Cholesky Decomposition	1.182	1.055	1.000
11	Deconvolution	1.007	1.000	0.919
12	Euler Method	0.925	0.962	0.872
13	Integer Factorization	0.999	0.999	0.999
14	Fast Fourier Transform	0.985	0.986	0.986
15	Huffman (heap)	1.043	1.010	1.018
16	Huffmap (static)	0.955	0.988	0.933
17	K-Means Clustering	1.004	1.001	1.001
18	Lucas Primality Test	0.999	1.000	1.000
19	2D N-Body Simulation	0.984	1.020	1.003
20	AES-256 Encryption	1.010	1.009	1.007
	Average	**0.999**	**1.031**	**0.971**

Partial Redundancy Elimination

Surprisingly, the PGO PRE pass did not yield any statistically significant speedup or slowdown patterns across all applications considered. That is, the speedups achieved by this pass hover around 1.0 for most of the programs, as shown below in Table 5. The average speedup with profile data for individual inputs is slightly worse than even with an approximate value of -2.1%. Collecting profiling data for the smallest set of data yielded an average speedup of approximately 1.8%, and conversely, profile data captured from the largest input size produced a speedup of -2.0%. Interestingly, there were three programs, the basic block test, combination generation, and mandelbrot fractal generation programs, that served as significant outliers originally causing these averages to shift away from 1.0. After inspecting the code, it is clear that the former program should not be included in the data set (this is why it is crossed out in the table).

Specifically, with respect to the basic block test, there are *no redundancies* in the CFG representation of the program. In this case, the PRE transform was not even applied in the pass.

However, with respect to the combination generation program, there is a clear redundancy (in the current implementation) that takes place during the recursive call to the *combination* function. While this redundancy would have been removed via common subexpression elimination had this particular optimization pass been applied, it was instead removed via the PRE pass. Even with this elimination, we saw performance degradation in all cases considered - two of which fell below the -15% speedup (or slowdown) threshold. This slowdown is the result of the fact that the original program constructs combinations by means of recursion and prints out complete combinations in a loop. For the purposes of obtaining benchmark results we dropped this output to ensure that the program performance was not stifled by I/O overhead; as previously noted, in order to ensure that the complex portions of the code were not marked as dead and removed we added a side effect to maintain the presence of the loop and the recursive function contained therein. The body of this loop has a seemingly redundant expression that was most likely hoisted by the PRE pass since it is executed quite frequently (i.e., speculation of the expression at a “higher” node in the CFG was likely enabled using the PRE benefit and cost functions). Given that this expression was inside a loop and differed based on the loop index, this may have introduced unnecessary variable allocations on the stack prior to entering the loop body, which would explain the program slowdown. Unfortunately, we were not able to analyze the program CFG to verify this conjecture.

Another program that seemed to benefit from PRE was the mandelbrot fractal generation program. At first glance, it appeared as though there were a number of multiplication operations nested at various levels within the main loop that traverses the pixels of an image that could be hoisted to higher nodes in the CFG. However, it turns out that such redundant expressions do not compute the same value, and as a result are not hoisted. Therefore, the speedup we saw in this instance is likely due to better file I/O speed. Consequently, we decided to also omit this program from the collection of average results.

We conclude by noting that in our experimental setup and for the considered set of programs, profile guided PRE provide significant speedups. Computational redundancies typically occur in increasingly complex and scientific code, which is not featured in our set of benchmark programs. Accordingly, we anticipate that the observed speedups and slowdowns are anomalies on of the experimental platform (i.e., jitter introduced from the OS and other processes running), rather than benefits obtained from applying the PRE pass. It was outside the scope of this project to develop code that would benefit from PRE, but doing so may be useful future work to better test the utility of this new optimization pass.

Table 5. Speedup results from the profile guided PRE optimization from [9]. Speedups and slowdowns greater than 10% are highlighted.

ID	Program	Speedup with Profile Data for Individual Inputs	Speedup with Profile Data Collected for Small Input	Speedup with Profile Data Collected for Large Input
1	Matrix Multiplication	0.999	0.998	0.998
~~2~~	~~Basic Block Test~~	~~1.838~~	~~0.940~~	~~0.667~~
3	Permutation Generation	0.960	1.046	0.961
4	Bubble Sort	1.021	1.021	1.022
5	Quick Sort	0.990	0.973	1.011
~~6~~	~~Mandelbrot Set Generation~~	~~1.133~~	~~1.021~~	~~0.989~~
7	Trial Division Primality Test	0.996	1.007	1.016
8	Combination Generation	0.904	0.882	0.881
9	Conjugate Transpose	1.007	1.098	1.006
10	Cholesky Decomposition	0.904	1.090	0.963
11	Deconvolution	0.951	1.036	1.015
12	Euler Method	1.034	0.979	0.875
13	Integer Factorization	0.998	0.999	0.998
14	Fast Fourier Transform	0.990	0.985	0.985
15	Huffman (heap)	1.010	1.016	0.922
16	Huffmap (static)	0.936	1.019	0.921
17	K-Means Clustering	0.957	1.031	0.994
18	Lucas Primality Test	1.000	0.999	0.999
19	2D N-Body Simulation	0.955	1.133	1.061
20	AES-256 Encryption	1.010	1.011	1.008
	Average	**0.979**	**1.018**	**0.980**

Concluding Remarks

Our experimental evaluation is based on the LLVM release 3.3 which was released in June, 2013. As of November 2013, the current master development branch does not include these PGO facilities anymore. Instead, development on a new, developer-transparent version is underway [4]. This indicates that PGO is still an active research topic in the compiler development community.

Through our experimental evaluation we showed that PGO can indeed provide performance gains. It may, however, also degrade performance in certain cases. Great care must be taken to ensure that PGO is used appropriately depending on the application and use case. Out of our three implemented optimization passes, function inlining looks the most promising. As shown by Table 2, it improves performance most consistently with the fewest negative outliers. In their current state we do not consider the other two optimizations to be worthwhile considering that they may also hurt performance.

In addition, the cost of gathering profiling data and re-compiling an optimized version, especially if it is not done transparently, cannot be neglected. Another intrinsic issues with PGO is how to ensure that the gathered profile data matches the actual runtime characteristics. This is not a purely technical question but also depends on organizational aspects and the overall quality of the development, testing and deployment process. Again, this issue is currently being addressed by academic and industry professionals and we can be hopeful to see progress in this area in the near future.

References

[1] Pettis, Karl, and Robert C. Hansen. Profile Guided Code Positioning. *ACM SIGPLAN Notices 25(8)* (1990).

[2] Raymond Lo, Shin-Ming Liu, and Fred Chow. Loop Induction Variable Canonicalization. *In Proceedings of the 1996 Conference on Parallel Architectures and Compilation Technique, IEEE Comput. Soc. Press (1996),* 228-237.

[3] Chau-Wen Tseng. Instruction Level Parallelism 2. *Computer Systems Architecture lecture at University of Maryland* (2012). Available online at: http://www.cs.umd.edu/class/spring2012/cmsc411/lectures/lec08.pdf. Last accessed: 11/20/13.

[4] LLVM Developer Community. RFC - Profile Guided Optimization in LLVM. *LLVM developer mailing list*. 2013. Available online at: https://groups.google.com/forum/#!msg/llvm-dev/UOJqp0f9MBY/lN4MgCT6Q_cJ. Last accessed: 11/20/13.

[5] Optimization Options. *GCC Documentation*. 2013. Available online at: http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Last accessed: 11/20/13.

[6] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile guided Automated Software Diversity. In *2013 International Symposium on Code Generation and Optimization* (CGO 2013), Shenzhen, China (2013).

[7] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *International Symposium on Code Generation and Optimization* (2004).

[8] The Computer Language Benchmarks Game. Available online at: http://benchmarksgame.alioth.debian.org/. Last accessed: 11/20/13.

[9] Gupta, Rajiv, Eduard Mehofer, and Youtao Zhang. Profile guided compiler optimizations. (2002).

[10] Dhananjay M. Dhamdhere. E-path PRE: Partial Redundancy Elimination Made Easy. *ACM SIGPLAN Notices 37(8)* (2002), 53-65.

[11] The LLVM Compiler Infrastructure. Available online at: http://llvm.org. Last accessed: 11/30/13.

[12] clang: a C language family frontend for LLVM. Available online at: http://clang.llvm.org. Last accessed 11/30/13.