A Study of Phased Branch Behavior in C++ Applications

Vinay Rajagopalan
Clemson University
Computer Science Dept
Clemson, SC USA
vrajago@cs.clemson.edu

Brian A. Malloy
Clemson University
Computer Science Dept
Clemson, SC USA
malloy@cs.clemson.edu

ABSTRACT

We investigate the notion of phases as they occur in object-oriented programs. The focus of our investigation is on C++ programs and our test suite includes various kinds of object-oriented programs including scientific and general-purpose applications. We focus on individual phased branch behavior and attempt to capture information and generalize about the frequency of phased behavior. We provide guidance about the advantages of path profiling over edge profiling in determining phase change in various program executions.

General Terms

profiling, edge profiling, path profiling, simulation, program optimization

1. INTRODUCTION

The tremendous interest and growth in the internet has brought with it a demand for mobile code that is compact and efficient. The idea behind mobile code is that applications can be distributed quickly across computer networks and automatically executed upon arrival. For these applications to run efficiently, they must adapt to the run-time environment of the target architecture. In view of the demand for speed, there is evidence that traditional static optimizations may not provide the efficiency required for modern mobile applications. Moreover, there is increased use of object-technology in these mobile applications with a concurrent belief that object-oriented code, with its frequent use of dynamic binding, is less amenable to traditional, static optimizations.

One optimization in common use today is feedback-directed optimization, FDO, which uses program characteristics, obtained at runtime, to attempt to improve the performance of the application. Most FDO approaches use profile guided compilation, which facilitates determination of those parts of a program to modify to maximize performance[?]. Profiling entails instrumenting an application to obtain a cost metric such as time spent in a method or the number of times a method executes. This metric can indicate those parts of a program where optimization is likely to achieve the most benefit[?]. However, there are some significant drawbacks to profile guided optimization.

One drawback results from the static nature of feedback-directed optimization techniques. Usually a program is instrumented for profiling, executed, and optimized based on characteristics seen during the program run. The resulting optimized program tends to perform well when the same characteristics that were present during profiling continue to exist. Recent work has shown that changes in the input of a program can dramatically effect the behavior of common commercial applications[?]. If the input to a program changes, the optimizations performed on a previous execution may no longer be beneficial [?]. In fact, the performance of a statically optimized program may degrade appreciably, so that performance is worse than the un-optimized version of the program[?].

A second drawback with FDO relates to the determination of the instrumentation strategy that is likely to provide the most information at a tolerable cost. Traditional approaches to profile guided optimization construct a graphical representation of the program that describes the flow of control, and then place counters at vertices or edges in the graph. Edge profiles record an aggregate count of the frequency of edge executions that provide general information about the behavior of a program.

However, reference [?] argues for the collection of more detailed information than aggregate edge information to describe a program's run-time characteristics, introducing the notion that the run-time behavior of a program cycles through a series of *phases*. For example the branch prediction and cache miss rate might be more accurately described over the execution of a program using phases[?]. Phase profiling might address the shortcomings of edge profiling if one could detect changes in phase and apply an optimization more suited for a particular phase and later apply a different one for a different phase.

To illustrate the impact that phased behavior might have on optimization, consider the graph in Figure 1, which models the run time branching behavior of an application. The branch shown in the figure is biased towards "taken" 55% of

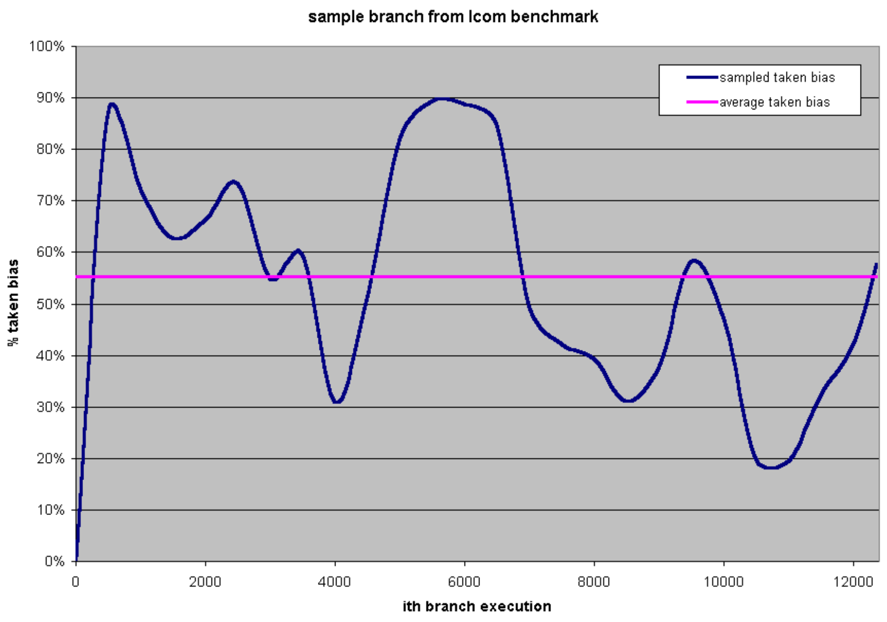


Figure 1: *Phased Behavior.* This is a branch taken from the lcom benchmark which exhibits phased behavior. The pink line at y = 55% represents the average bias for the branch. The blue line represents the sampled taken bias for the branch. The sampled bias seems to interleave between periods of being strongly taken and being weakly taken. An optimization based purely on the aggregate bias would adversely affect performance during the period of very weak taken bias.

the time; that is, aggregate information would lead the profiler to conclude that the branch is likely to be "taken" more often than not. However, the sampled taken bias interleaves between periods of strong bias towards "taken" and periods of "not taken". An optimization based strictly on the 55.5% average bias would perform poorly during the phases with significant bias towards "not taken".

In this paper, we investigate the notion of phases as they might occur in object-oriented programs. The focus of our investigation is on C++ programs and our test suite includes various kinds of object-oriented programs including scientific and general-purpose applications. We focus on individual phased branch behavior and attempt to capture information and generalize about the frequency of phased behavior. We provide guidance about the advantages of using path profiling over edge profiling for improving the performance of programs that exhibit phased branch behavior.

The remainder of this paper is organized as follows. In the next section we provide background about various approaches to program profiling, including the notion of *superblock* scheduling. In Section 3 we describe the simulation tool that we exploit to investigate phased behavior and in Section 4 we describe our case study that forms the basis of our conclusions about phased behavior. In Section 5 we discuss the results we obtained from analyzing phased behavior for individual branches. In Section 6 we discuss the impact that our study might have on profiling research and we overview related work in Section 7. In Section 8 we discuss ways to enhance and extend our work. Finally, in Section 9 we summarize and conclude.

2. BACKGROUND

In this section, we provide definitions of terms and background about profiling. We discuss both edge and path profiling including a greedy algorithm for computing edge profile information. We conclude this section with a discussion of phases and phased behavior, including the impact of phased behavior on superblock scheduling.

2.1 Profiling

One of the common uses of profiles is to derive paths taken during program execution for use in path based optimizations[?]. *Edge profiling* records the execution count of transitions between basic blocks[?] or *edges*. Since edge profiles summarize path execution using aggregate edge counts[?; ?], programs paths must be derived from a profile. Generally path based optimizations are performed on *hot paths*, heavily executed paths, in order to gain the best performance gain with limited resources.

One technique for deriving heavily executed paths from an edge profile is to use a *greedy algorithm* which follows the edge with the maximum execution frequency out of a basic block[?]. This greedy algorithm does not always produce accurate results; more specifically, the greedy algorithm may misidentify which path significantly contributes to overall control flow of a program. For example, given an edge profile for a CFG such as Figure 2a, a greedy algorithm would incorrectly identify ACDEF instead of ABDEF as the most frequently executing path.

Path profiling addresses the shortcomings of edge profiles by recording the actual paths taken within an application[?]. Looking at Figure 2a, path profiling can disambiguate between the two paths and determine which path actually contributed significantly to the control flow of the program. Since the frequency counts for actual paths are recorded instead of aggregate edge counts, the correct hot path ABDEF was found for the CFG in Figure 2a.

The additional accuracy provided by path profiling can be beneficial for certain optimizations based on run-time profile information. Young and Smith demonstrated improvements which can be made to superblock schedulers using path profiling[?; ?]. A *superblock scheduler* attempts to improve program performance by increasing the amount of instruction-level parallelism (ILP) extractable within the program code[?] by using superblocks. A *superblock* is a sequence of basic blocks with one entry and one or more exit points[?; ?]. A superblock allows a compiler to locate ILP beyond the bounds of a single basic block. Side entrances to the superblock are eliminated using *tail duplication. Tail duplication* consists of copying the basic blocks from the destination of a side entry to the end of the path to new basic blocks. The side entry is changed to point to the newly copied blocks instead of the superblock[?]. For example, if the nodes A, B, D, and E from Figure 2a are chosen to form a superblock, then the edge from C to D would represent a side entry. The *tail* of the path, nodes D and F, would be copied to new basic blocks and the node C would point to the newly created node for D.

One of the optimizations superblock schedulers perform is loop unrolling by representing an unrolled loop bodies as superblocks. To get the maximum performance benefit out of loop unrolling using superblocks, side exits must be avoided or at least kept minimal since side exists could causes misses in the instruction cache[?]. To avoid side exits, superblocks are generally created for hot paths[?; ?].

A superblock scheduler based on edge profiles for detecting hot paths would unroll the loop in Figure 2a as $(ACDEF)^r$ where r is an unrolling factor. For example, an unrolling factor of 3 would create the superblock ACDEFACDEFACDEF. The superblock created using edge profiling would take an early exit at node A to node B about 120 times based on the path profile in Figure 2a. A superblock scheduler which used the path profile in Figure 2a would be able to detect the hot loop path ABDEF and unroll the loop as follows $(ABDEF)^r$.

Even though path profiling seems more attractive than edge profiling due to the more detailed profiles, path profiling incurs a more significant overhead than edge profiling[?]. Ball and Larus's path profiling algorithm adds about 31% overhead to a program; whereas, edge profiling adds about 16% overhead to a program[?]. Using statistical sampling of the program counter, the overhead associated with profiling can be reduced to about 1 - 3%[?].

For path profiling to be more useful than edge profiling, path profiling has to be able to amortize its overhead throughout the execution of a program by recognizing a significant amount of program behavior which is not possible by using edge profiling techniques. Young and Smith describe the cause of inaccuracies within edge profiles as being due to intersecting frequently executing paths[?]. In Figure 2a this happens at node D. Whether the path ACDEF executes the most depends on how much flow ABDEF takes from ACDEF along the DE edge.

Edge profiling needs the path ACDEF to be the most frequently executing path in all path profiles in order to be preferable to path profiling contrary to what actually occurs in Figure 2a. Thomas Ball, et. al. describe the behavior needed for edge profiles to able to identify hot paths by defining the *definite* and *potential* flow of a path[?].

The *definite flow* of a path within a program represents a lower bound on the execution frequency of the path. If a path within a program has a definite flow of f, then the execution count assigned to that path from any path profile must be at least f[?]. Path profiling is not needed to determine definite flow for paths within a program. To compute the definite flow of a path, the edges that join into the path are subtracted from the flow at the target of the path's last edge. For example, to compute the definite flow for the path ACDEF in Figure 2a, the edge count for edges BD and DF would be subtracted from the flow at node F. The resulting definite flow for path ACDEF would be $50(50 = 270 - ((270 - 150) + (270 - 170)))$.

The *potential flow* of a path determines the maximum execution count a path can have in any path profile for a program execution[?]. The potential flow of a path can be found by looking for the edge in the path with the lowest execution count. The potential flow for the path ACDF in Figure 2 is 100 since the edge DF has the lowest frequency count on the path.

Depending on the amount of potential or definite flow present within a program, one can determine whether an edge profile can adequately find heavily executed paths. If there is a large amount of definite flow in an application, edge profiling provides an adequate level of detail to produce heavily executed paths within the program since definite flow is simultaneously recognizable in all path profiles.

Looking at the CFG in Figure 2a, we can see the path ACDEF has a definite flow of 50(50 = 270 - ((270-150) + (270-170))) which means all frequency assignments for path ACDEF will have at least an execution count of 50. The definite flow of 50 for ACDEF is relatively small compared to the potential flow(lowest execution count for edge along a path) for the remaining paths. The potential flows for paths ACDF, ABDF, and ABDEF are 100, 100, and 120 which gives some room for the paths ACDF, ABDF, and ABDEF to steal flow away from ACDEF.

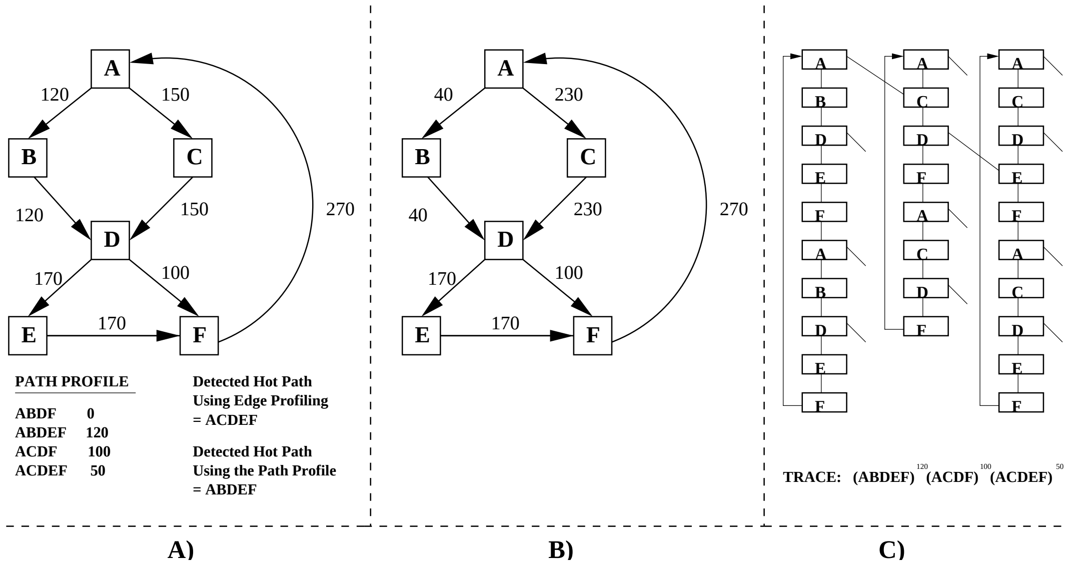

A
B
C
D
E
F
120
150
120
150
270
170
100
170
PATH PROFILE
ABDF 0
ABDEF 120
ACDF 100
ACDEF 50
Detected Hot Path Using Edge Profiling = ACDEF
Detected Hot Path Using the Path Profile = ABDEF
40
230
40
230
270
170
100
170
TRACE: (ABDEF)^120 (ACDF)^100 (ACDEF)^50
A)
B)
C)

Figure 2: *Example CFGs.* A) This example CFG demonstrates that an edge profile can sometimes misidentify the most frequently executed path within a program. A commonly used greedy algorithm which follows the most frequently executed edge out of a basic block would identify path ACDEF as the hot path while ABDEF is actually the hot path. B) This example of a CFG in which an edge will be able to find the hot path ACDEF. C) This is an example of a set of superblocks created(unrolled using an unrolling factor of two) using the path profile in A) and taking the phased behavior of the branches at node A and D into account.

If the edge counts for the CFG in Figure 2a were adjusted such as in Figure 2b, the definite flow for the path ACDEF is 130(130 = 270 - ((270-230) + (270-170))) which higher than the potential flows for the remaining paths(ABDF = 40, ACDF = 100, ABDEF = 40). Since the definite flow for the path ACDEF is higher than the potential flows for the remaining paths, path profiling is not needed.

2.2 Phased Behavior

We intended to analyze the phased behavior in individual branches of some C++ benchmarks to determine whether path profiling was preferable to edge profiling in terms of identifying hot paths within a program. We originally thought that phased behavior of individual branches can be used as an indicator of the amount of definite flow within a program. If a conditional branch had only one phase(highly biased towards one direction) , we thought that the branch had a large amount of definite flow. If a branch had a large number of phases, we conjectured that this indicated a small amount of definite flow for either edge of the branch.

Examining the CFGs in Figure 2a and Figure 2b we can see why our assumptions may not be correct for acyclic paths as used in Larus and Ball's path profiling algorithm. For the CFG in Figure 2a the conditional branch at node A could just consist of one phase biased for 150 out of 270 executions towards the AC edge which we would have incorrectly thought indicated a large amount of definite flow. Also, for the CFG in Figure 2b the conditional branch at node A could consist of a large amount of insignificant(small amount of executions) phases which interleave the executions of the edges AB and AC resulting in a misclassification of the paths having a small amount of definite flow. A more accurate measure might be to look for conditional branches which phases that have relatively equal length(number of executions). Also a good indicator might be just to look for nodes with side entry and exit points where hot paths intersect[**?**].

On the other hand, having branches that exhibit phased behavior can be a good indicator that additional optimization opportunities exist in less dominant paths not just in the dominant hot path. Suppose a trace such as $(ABDEF)^{120}(ACDF)^{100}(ACDEF)^{50}$ induced the edge and path profile in Figure 2a. We can see that the conditional branches at node A and D have phased behavior. For the first 120 executions, the branch at node A is biased towards B. For the remaining 150 executions, node A is biased towards C. The branch at node D interleaves between being biased towards E and F.

A superblock scheduler based on path profiling which ignores the phases present at node A and D would create a superblock such as $(ABDEF)^r$. The superblock $(ABDEF)^r$ would take an early exit to node C for the remainder of the trace after 120 executions and would taken an early exit at node D to node E for 100 iterations.

A superblock scheduler which takes phases into account could create three superblocks $(ABDEF)^r$, $(ACDF)^r$, and $(ACDEF)^r$. The side exit from node A in superblock

$(ABDEF)^r$ could linked to node C in super block $(ACDF)^r$ to account for the phase change n the branch at node A. For the side exit from node D in super block could be link to node E in superblock $(ACDEF)^r$ to account for the phase change in node D. The resulting superblocks are shown in Figure 2c. By taking phases into account, a superblock scheduler was able to reduce the number of early exits to only two occurences. One at node A for the transition to the superblock $(ACDF)^r$ and one at node D for the transition to superblock $(ACDEF)^r$.

In order to expose the the less dominant paths, path profiling is needed since edge profiles store an aggregate count of the paths that execute within a program. Adding the executions counts for the edges along the paths in the path profile contained in Figure 2a gives us the edge profile in Figure 2a. Since this edge profile contains an aggregate count for the paths in Figure 2a, the exact contribution of each path to the total flow of the CFG can not be determined by simply looking at the edge profile[?; ?]. To gain benefits from phased branch behavior, path profiling must be used to isolate paths other than the dominant hot path.

3. THE SIMULATION TOOL

In this section, we describe our overall approach toward the detection of phased behavior. We describe Shade, the simulation tool that we used.

To evaluate phased behavior in C++ programs, all the conditional branches within an application are traced. For each branch, the sequence of taken or not taken bits are stored. To look at the changes in *branch phases* and the tendency for a branch to stay biased in one direction for a significant period of time over the execution of a program, a *sampled bias* is computed. After a specified number of executions (*granularity*), the amount of a branch's bias towards *taken*, *not taken*, or towards neither direction is stored as the sample's bias. C++ applications were written to process the trace files containing sampled biases and generate plots to describe the phased behavior present in individual branches in C++ programs.

To trace individual branches with a program, we used a software simulator called Shade[?]. *Shade* is an execution profiling framework(see Figure 3) that allows programmers to build analyzers that can retrieve per instruction information about an application such as the instruction address, effective address, taken bit, and opcode. Software simulation has the advantage of flexibility, permitting the level of traces to be modified at runtime. For example using EEL[?] or Machsuif[?] would require code to place instrumentation around branches to call tracing functions. Shade places instrumentation for calling tracing functions and for storing trace information on behalf of the programmer.

Shade provides instruction-level access to information in the form of trace records. Users can specify the type of instruction information needed and the class of instructions to trace such as conditional branches or memory operations. For each desired instruction that shade encounters, shade stores per instruction information within the trace record. The user can specify the number of instructions to be traced at a time along with an array of trace records to store the information. If the default trace information provided by shade is not sufficient, users can specify functions that are called before and after an instruction executes.

We used Shade to obtain a conditional branch trace. We developed a shade analyzer that specified conditional branches to be traced. The analyzer plugs into the Shade framework to produce an application tracer that is shown in Figure 3. The analyzer recorded the instruction address(for identification) of the branch and whether the branch was taken or not. In order to obtain information about the phases that a branch may go through, individual branches were sampled at an adjustable level(granularity). Sampling was used in order to show changes in branch phase that may not be necessarily reflected by a profile that stores an aggregate edge count[?]. Looking at Figure 1 we can see that sampling of the taken bias shows interleaving patterns of high and low taken bias that can not be determined by just looking at the average taken bias. The *granularity* or sample size determines the number of branch executions contained in a sample. A granularity level of 1000 would represent 1000 executions of a particular branch. Also sampling reduced the size of the trace files obtained by running the analyzer.

In order to determine how a branch was biased during a sample,the bias of a branch was represented as a *fuzzy set*[?]. A *fuzzy set* allows partial membership in multiple sets. For our purposes we used three sets for a branch's bias: *taken*, *not taken*, or *neither* taken or not taken. We allowed partial membership between the neither set and taken or not taken sets. We did not allow partial membership between the taken and not taken sets since we used the neither set to represent a transition period when a branch is not significantly biased towards taken or not taken. Each set has a *membership grade* associated with it that determines the strength of a sample's membership in a particular set. The set with the highest membership grade, *height function*[?], is chosen to be the sample's bias. This allows a sample to be labeled mostly taken, mostly not taken, or not highly biased in either direction rather having to be a 100% member of a particular set.

To compute the membership grade for each sample of a branch, we maintain a fuzzy value for each branch. We use the fuzzy value to compute the membership grade to a particular set for each sample. Initially the fuzzy value for each branch is set to zero. A hash table was used to map each branch to its corresponding fuzzy value. Whenever a particular branch occurs during program execution the branch's instruction address is used as the hash key to obtain the branch's fuzzy value. If the branch was taken, the fuzzy value is incremented. If the branch was not taken, the fuzzy value is decremented. This process continues until the granularity level(sample size) is reached. Once the granularity level is reached the fuzzy value for the branch is divided by the granularity level, which puts the fuzzy value between -1 and 1. Next the membership functions for each of the sets are applied to the branch's fuzzy value. The

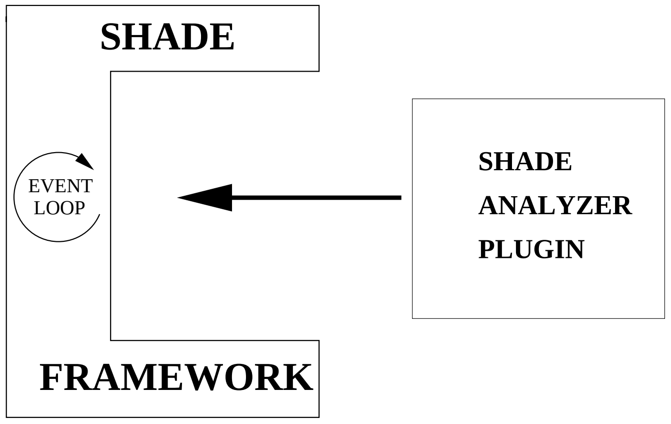
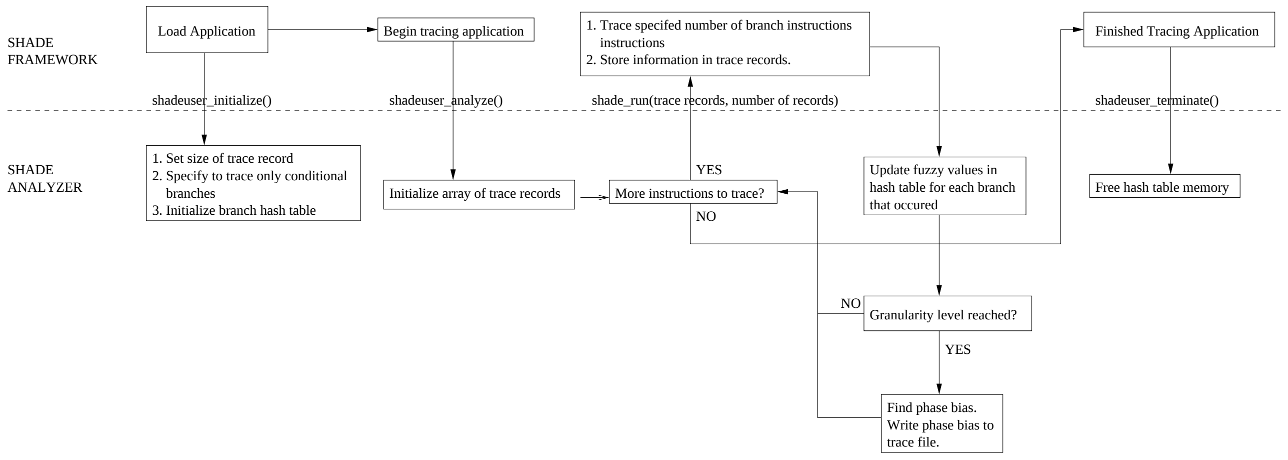


Figure 3: *Shade Framework.* Shade is a framework for tracing applications. Users build analyzers and plug the analyzer into the shade framework.



SHADE FRAMEWORK
SHADE ANALYZER
Load Application
Begin tracing application
1. Trace specifed number of branch instructions instructions
2. Store information in trace records.
Finished Tracing Application
shadeuser_initialize()
shadeuser_analyze()
shade_run(trace records, number of records)
shadeuser_terminate()
1. Set size of trace record
2. Specify to trace only conditional branches
3. Initialize branch hash table
Initialize array of trace records
More instructions to trace?
YES
NO
Update fuzzy values in hash table for each branch that occured
Free hash table memory
NO
Granularity level reached?
YES
Find phase bias. Write phase bias to trace file.

Figure 4: *Conditional Branch Tracing Using Shade.* The figure above describes how our shade analyzer interacts with the shade framework to trace conditional branches.

membership functions use the fuzzy value to determine the membership grade the sample has in each set (taken, not taken, and neither).

The membership functions are as follows:

$$\text{taken, f(x)} = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\text{not taken, f(x)} = \begin{cases} -x & \text{if } x < 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\text{not taken, f(x)} = -|x| + 1 \qquad \text{for all } x$$

The set with the highest(height function[?]) membership grade represents the sample's most significant bias. For example if we had a trace of 20 branch executions such as 11101110111110111111(1 = taken, 0 = not taken), the fuzzy value would be computed as follows:

```
Not Taken   = 0
Taken       = 1
Granularity = 20
Branch Trace : 11101110111110111111

fuzzy value = 1+1+1-1+1+1+1-1+1+1+1+1+1-1
              +1+1+1+1+1+1 = 14
fuzzy value = 14/(granularity) = 14 / 20 = 0.7
```

Next, the 0.7 fuzzy value would be fed into each of the membership functions:

```
taken membership grade     = f(0.7) = 0.7
not taken membership grade = f(0.7) = 0
neither membership grade   = f(0.7) = 0.3

sample's bias  = maximum(taken membership grade,
                not taken membership grade,
                neither membership grade)
      = maximum(0.7, 0, 0.3)
      = 0.7
      = mostly taken
```

Visually we can see how the membership functions map the branch sample's fuzzy value to its resulting bias in Figure 5.

Next the branch's resulting bias, instruction address, length of the sample, and the membership grades for each of the sets are written to a trace file. After the trace file has been updated, the fuzzy value is reset to zero. This process of computing a sample's bias continues until the program terminates. For a summary of the process see Figure 4.

4. THE CASE STUDY

To gauge the amount of phased branch behavior, we constructed a testsuite of C++ programs. We collected programs that represented a variety of applications. Each of the programs served as benchmarks for the simulator. The benchmarks include an incremental dataflow constraint solver called *deltablue*[?], an operating simulator called *richards*, an optimizing compiler for a hardware description language called *lcom*, a sample back-end called *idl* from the Object Management Group for the Interface Definition Language, an object oriented ray tracer called *eon*, and an object oriented graphics editor called *idraw* built using the InterViews framework[?]. Note idraw, idl, and deltablue are designed in an object oriented fashion[?]. Also, we included a program named eon that was not included in previous studies[?; ?].

Each of these benchmarks was built using GCC 2.8.1 except lcom which was built using GCC 2.7.3. All of the benchmarks were executed on a *SPARC Ultra 10* using our shade analyzer to create a trace file consisting of sampled bias data for each branch. A C++ program was written to examine the trace file and determine the number of phases for each individual branch in a benchmark. Adjacent samples with similar bias were chained together to form a single phase(see Figure 6). After adjacent samples with similar bias were combined, a cumulative distribution plot was generated to compare the number of phases versus the percentage of branches that had contained at least those number of phases.

5. RESULTS

We used the Shade simulator to analyze each of the benchmarks to evaluate the amount, consistencies, and variations in individual branch behavior for the benchmark programs. We evaluated the effects of different granularities on the phased behavior in each of the programs. We used a cumulative distribution graph to view the number of phases each branch has within a program. Also, the input for each program was held constant, except for the interactive GUI, *idraw*. At the time of the study we were not aware of a scripting engine for GUI's but we have since discovered *Android*, an excellent scripting engine for GUI testing and evaluation[?].

5.1 Methodology

Each of the graphs in Figures 7a through 7f illustrate the cumulative distribution of the phases in a conditional branch(x-axis) as compared to the percentage of branches(y-axis) that have at least one phase for a given granularity. Three different granularities were used for this study: 1000, 500, and 100. The different granularities allowed us to evaluate the variation in phased behavior in conditional branches with respect to the length of the branch. For each graph, the point $x = 1$ represents the amount of branches with just one phase and no phased behavior. This point was used to determine the amount of individual phased branch behavior present within each benchmark.

5.2 Overall Phased Behavior in Conditional Branches

The graphs in Figures 7a - f seems to show that a significant portion of the branches for the benchmarks exhibit phased behavior which may give evidence that path profiling could be useful in exploiting optimization opportunities present in phased branches. The mean percentage of branches with phased behavior is 13% at a granularity of 1000, 13.7% at a granularity of 500, and 17% at a granularity of 100. The mean values seem to increase as the granularity gets smaller which may indicate that the majority of phased branch behavior in the benchmarks(except for eon and deltablue) seems to occur mostly in branches with a relatively small number of executions provided the phases are a significant portion of a branch's execution.

The reason why we feel that most phased behavior occurs in the branches with a smaller number of executions is due to the relatively small number of phases occurring within each branch exhibiting phased behavior as seen in Figures 7b, d, and f and due to the increase in the percentage of branches showing phased behavior as the granularity is decreased. As the granularity is decreased, the number of phases per branch still seems low. If phased behavior was mainly occurring in branches with a larger number of executions, we would expect the number of phases per branch to increase and the phased branches to be more evenly distributed over the number of phases per branch as the percentage of phased branches in a benchmark increases with decreasing granularity.

It could also be the case that the phases within each of the benchmarks are an insignificant portions of a branch's execution so when the granularity is decreased more interleaving phases are detected. This would also explain the small amount of phases occurring per branch within the branches with phases. We do not think this is likely, but we can not be certain unless have length information about the branch phases.

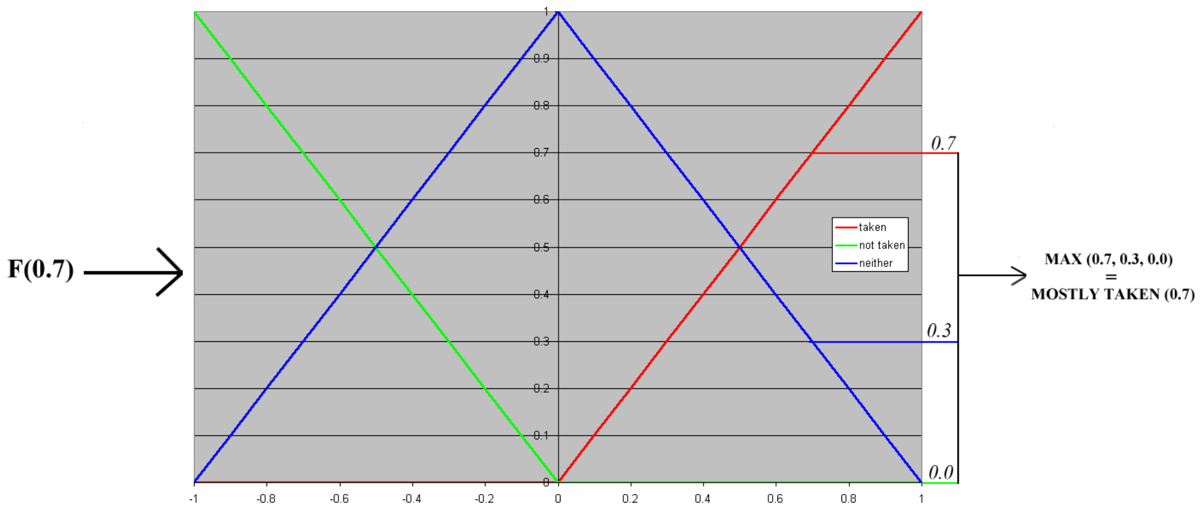


Figure 5: *Determining Branch Bias.* This figure demonstrates how a branch's bias is determined. The fuzzy value is applied to each of the membership functions. The membership set that has the highest membership grade is chosen as the bias for the branch.

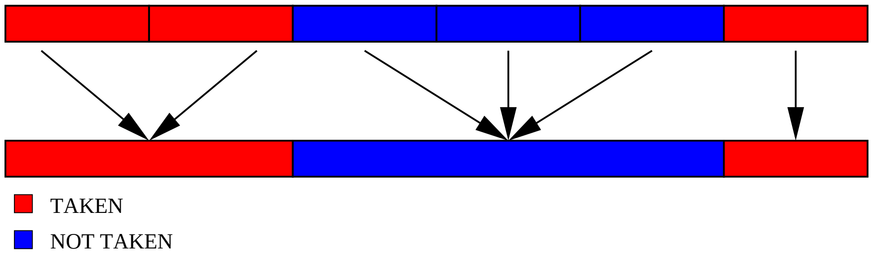


Figure 6: *Phase Combining.* Adjacent samples with similar bias are combined. In the figure the first two samples with taken bias are combined. Also the next three samples are combined since all three samples are biased towards not taken.

5.3 Variations in the Number of Phases Per Branch

The phased branches within the benchmarks except for eon spend most of their executions in a small number of phases. For example at a granularity level of 1000, 99% of the conditional branches in deltablue contain 7 phases or less. In contrast, the phased branches within eon seems to be more evenly distributed over the possible amount of phases per branch that they can contain. Looking at Figure 7b, the phased branches within the eon benchmark look much more evenly distributed over the number phases it can contain per branch than the other benchmarks.

5.4 Changes in the Percentage of Branches with Phases Due to Changes in Granularity

As the granularity was decreased, the percentage of branches with phased behavior within the idraw, lcom, richards, and idl benchmarks increased. This increase was most pronounced in the lcom benchmark. As the granularity was lowered from 500 to 100, the percentage of branches exposed with phased increased from 20% to 31%. Also , for the richards benchmark initially did not show any branches with phases at a granularity of 1000, but at a granularity of 100 the percentage of branches exposed with phases increased to 8%.

Unlike the idraw, lcom, richards, and idl benchmarks decreasing the phase granularity for the deltablue benchmark does not expose more phased branch behavior. In fact, the percentage of branches with phased behavior decreases from 19% to 16% when the granularity is decreased from 500 to 100. This decrease might indicate that branches that have more executions tend to exhibit more phased behavior than branches that have a few executions within the deltablue benchmark.

The eon benchmark seems to behave quite differently than the other benchmarks. The changes in granularity do not seem to have much of an effect on the benchmark. The branches that have a small number of executions seem to have the same percentage of branches with phases as the branches that have a large number of executions. Also, the phased branches within eon seems to be more evenly distributed over the number of phases per branch that they can contain; whereas, the phased branches within the remaining benchmarks spend most of their execution in only a few phases.

6. DISCUSSION OF RESULTS

In this section, we attempt to evaluate the experiments and results reported in the previous section. We discuss our approach and our management of the test suite.

6.1 Phase Length

The results in the previous section did not relate the amount of phased behavior to phase length present in individual branches. In terms of individual branches, we define *phase length* as the number of executions an individual branches stays biased towards taken, not taken, or stays not biased towards either direction. Since phase length is not considered for the results obtained in the previous section, we can not tell whether the phases described in the graphs in Figure 7 take up a significant portion of a branch's execution. The only thing we can say is that each phase within a branch has a length of at least the granularity size. For example, the graph in Figure 7a shows that about 20% of the branches in the deltablue benchmark have more than one phase. Even though 20% of the branches have phases, the phases contained in these phased branches might take up less than 1% of each of these branches' executions. Also, discrepancies between the phase length of branches that execute very little versus branches that execute for long periods of time would help determine which branches of a program a compiler could best spend its time analyzing for phased behavior.

6.2 Changes in Input

In general the input for each program was held constant. We looked for changes in individual branch phases for a given input. Our results could be specific to the given input we use. Recent research indicates that for certain programs the paths within the program could vary dramatically due to small changes in the input[?] which may suggest that our results may not say much about the general behavior of the programs we studied. Looking at the graph in Figure 8 we can see that the percentage of branches with phases and the amount of phases per branch seems to vary based on changes in the input. More conclusive results possibly could be obtained by looking at changes in phased behavior due to differences in input. Maybe this can be achieved by comparing profiles for different inputs and mapping back the differences in paths to the source programs[?; ?].

6.3 More Realistic Programs

The programs we used to analyze phased branch behavior are relatively small programs. These programs may not be representative of conventional desktop programs. More applicable results might be obtained using Etch[?] to obtain conditional branch traces of Microsoft Office and Netscape on Win32 machines.

Also, each benchmark ran for a relatively small amount of time. We may not have adequately tested each benchmark. We used rather small input data sets. Larger input data may needed for accurate results.

6.4 Program Phases Versus Individual Branch Phases

We analyzed each branch in isolation from other branches and then tried to reason about the overall behavior of C++ programs. We did not look at how the phases of one branch affects the phases of another branch, so our results may not say much about overall program behavior. Our results are limited to describing the amount of phases present in each C++ benchmark we used. It would be interesting to see whether a series of branches are biased towards taken for a period of time and not taken for another period similar to work being done by Sherwood and Calder[?].

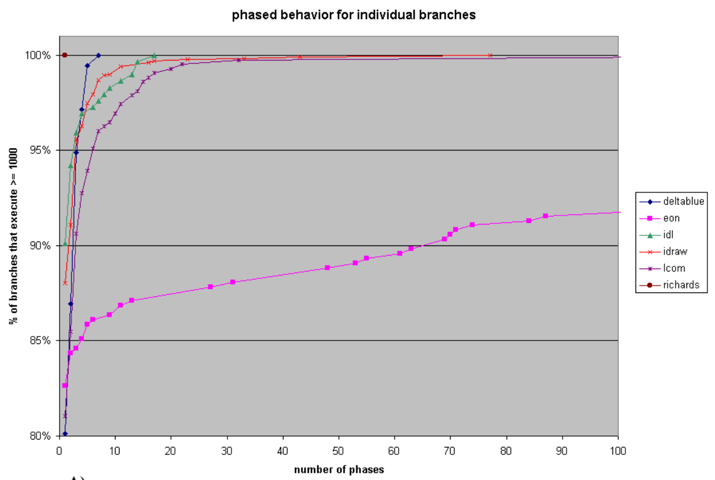
7. RELATED WORK

In this section, we overview the research that relates to program optimizations, phases and the use of profiling.

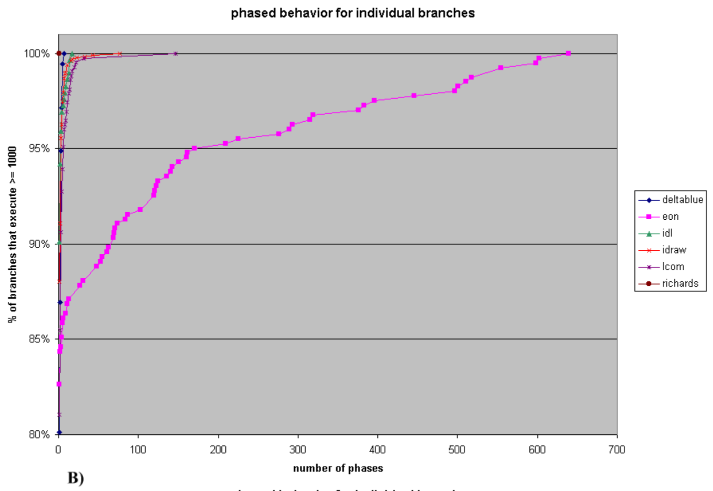
7.1 The Deco Project at Harvard and Hewlett Packard's Dynamo

The Deco project[?] at Harvard University and Hewlett Packard's Dynamo[?] are feedback-direct optimization systems which attempt to create executables which can adapt to their run-time environment. Each system uses a different mechanism for detecting changes in phase. The current prototype of Deco records Ball and Larus acyclic paths[?] and stores them in a hash table(profile table). An execution count is maintained for each entry in the hash table. Deco uses an interrupt mechanism to examine the hash table and determine which path to optimize based on a threshold execution count. Each optimized piece of code is put into a fixed sized optimization cache. After the hash table is examined, each entry's execution count is zeroed. If the cache is full, a random entry is evicted in order to deal with program phases.

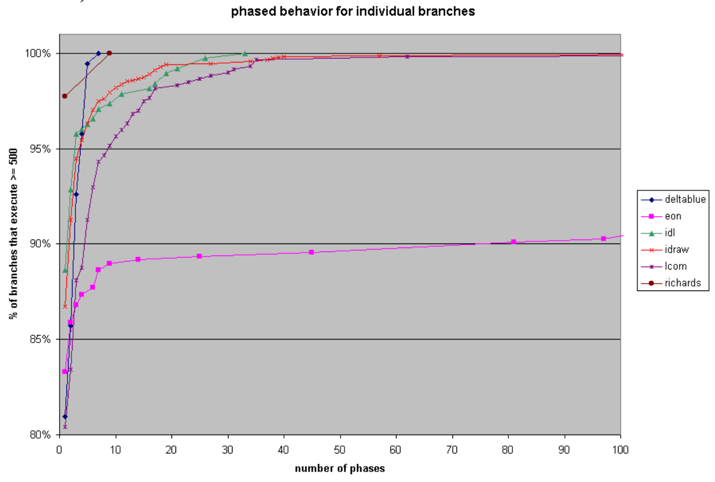
Dynamo uses an heuristic called MRET(Most Recently Executed Tail)[?] for identifying hot traces. Within Dynamo backward taken branches represent the end of a trace. The target addresses of these branches identify the start of new traces. An execution count is associated with each of these target addresses. If this execution count exceeds some threshold value, dynamo begins to record the instruction stream until another backwards taken branch is encountered. The



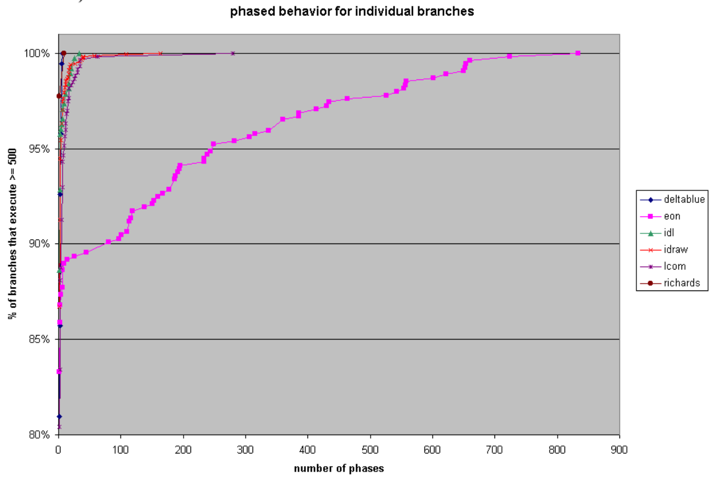
A)



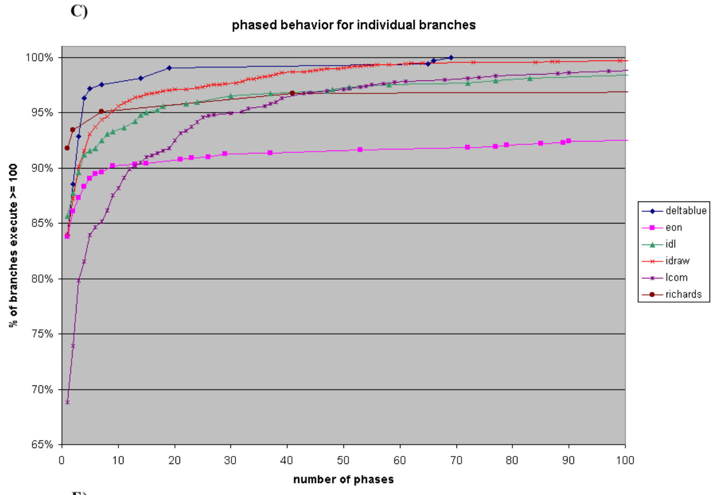
B)



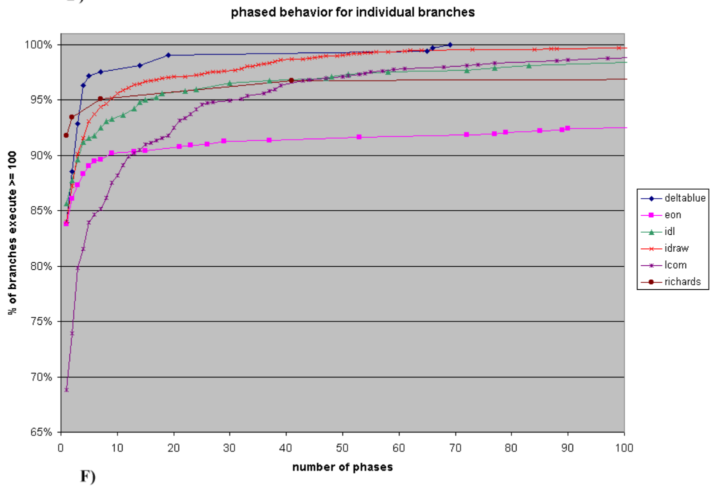
C)



D)



E)



F)

Figure 7: *Results.* Each of these cumulative distribution graphs plots the number of phases versus the percentage of branches that execute at least as much as the sample size. The plots on the left zoom in on the corresponding graph on the right in order to view the differences in phased behavior at x=1. A-B) Graph at a granularity of 1000. C-D) Graph at a granularity of 500. E-F) Graph at a granularity of 100.

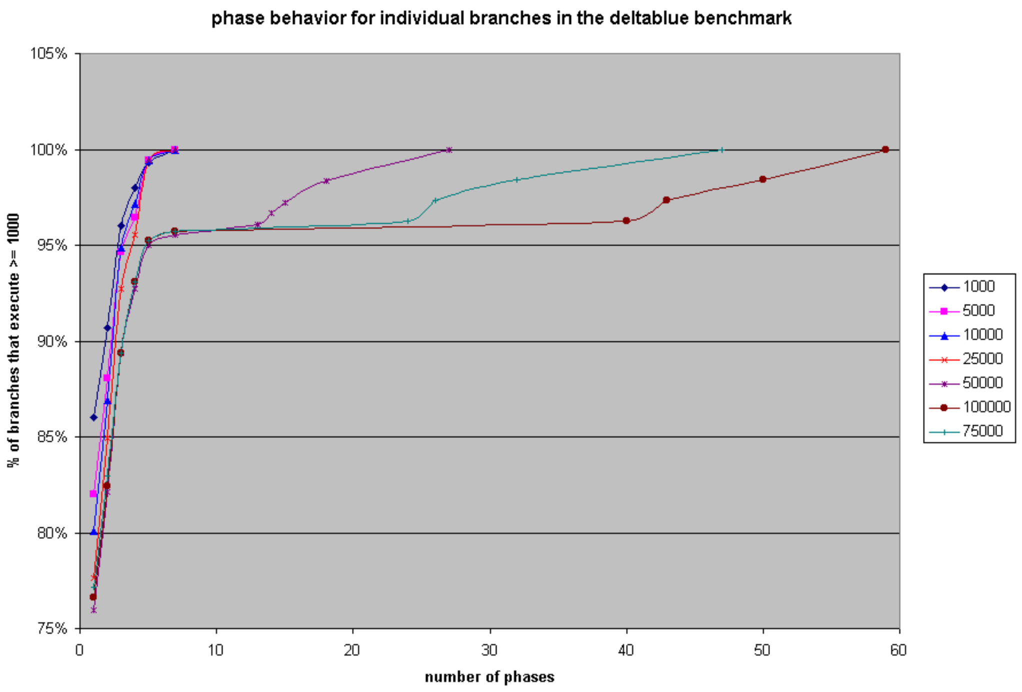


Figure 8: *Changes in Phased Behavior Due to Input Changes.* The graph shows the changes in the percentage of branches that exhibit phased behavior due to changes in input. Each of the inputs represent the number of constraints the deltablue program must solve. The percentage of branches that demonstrate phased behavior ranges from 14% to 24specified. Also the amount of phases present for each branch ranges from 7 to 57.

reason for using the target of backwards branches as identifies for hot traces is if the target of a backwards branch is hot than it is also likely that the instructions following the branch target are also hot. The trace is optimized and placed into a cache indexed by the target address that starts the trace. Subsequent encounters of the target address result in hits to the cache. The target address is replaced by the address of the optimized trace. Once the trace ends control is returned to the Dynamo interpreter which starts the tracing process again. To adapt to changes in phase, Dynamo flushes the trace cache whenever the rate of trace creation surpasses some threshold value. This flushing strategy attempts to readjust optimizations to changes in branch phase.

7.2 Time Varying Behavior

Sherwood and Calder looked at phases of program behavior that vary over time[?]. They looked at how the instructions executed per cycle, branch prediction rates, address prediction rates, cache miss rates, and value prediction rates influence each other and change over the lifetime of the SPEC95 benchmark suite. They used the SimpleScalar[?] simulator to record statistics for every 100 million committed instructions. Each point (for each 100 million instructions committed) obtained was graphed to view trends over time and to look for cyclic behavior. The cyclic behavior of the SPEC95 benchmarks was used to reduce the amount of time needed to get an accurate picture of program behavior. The graph was used to determine the length of a program cycle.

7.3 Adaptive Parallelism

Even though adaptive loop transformations for parallel computations can provide significant performance speedups, existing adaptive techniques waste processor resources. For a particular parallel computation speedups gained by adding more processors could level off or possibly decrease. Hall and Martonosi showed that the behavior of some loops from the Specfp95 and NAS benchmark suites may go through phases where there may be insufficient levels of parallelism so additional processors may not help improve performance[?]. In some cases a serialized version of a loop might perform better.

Hall and Martonosi developed an extension for the SUIF parallelizing compiler which dynamically adjusts the number of threads allocated to a particular program based the program making effective use of the number of threads allocated to it[?]. They were able to improve workload performance by up to 33% over non-adaptive techniques.

8. FUTURE WORK

In this section we describe directions for future work and evaluate our results in the light of our original goals.

8.1 Relationships Between Branch Phases and Branch Prediction

One of the original aims of this work was to examine the relationships between branch miss-prediction rates and branch phase. We wanted to see the affect different levels of bias within a branch phase has on the miss-prediction rate of a simple two bit predictor versus more complicated predictors such as correlated or bimodal predictors. For example if a branch is highly biased towards taken or not taken, a simple two bit predictor may be sufficient. Also if a branch is not biased towards taken or not taken, a complicated predictor may not provide a sufficient correct prediction rate to justify its use over a simpler predictor. We wanted to look at phases for individual branches and try to understand their relationship to phases that branch predictors go through. Our conjecture was individual branches would exhibit a high miss-prediction rate while in a phases where the branch is not biased towards taken or not taken. Looking at Figure 9, it seems that the miss-prediction rate increases when a branch changes phases, not necessarily when a branch is not highly biased in the taken or not taken direction.

8.2 Mapping the Phases to Their Corresponding Branches

At the present, we just record the amount of phases for a given granularity size that a particular branch has. We don't really know what constructs or portions within the C++ benchmarks are responsible for the branches with a significant amount of phased behavior. Shade is distributed with the SpixTools[?] which can be used to map traced instructions back to the instruction's source code. We can use these tools to map the phased found using the analyzer to the actual programming constructs within each of the benchmarks. The information obtained could be useful to compiler writers and computer architects for determining what constructs are important to monitor in order to detect changes in branch phase.

8.3 Address and Value Prediction

Researchers have also shown that phased behavior also occurs with instructions that access memory[?; ?]. A particular load or store operation might use the same address or value for one period of time and switch to a different address or value for another period of time. It would be interesting to analyze the amount of phased address or value behavior present in the C++ benchmarks. Also, looking at the relationships between load/store and branch behavior might help in understanding C++ program behavior.

9. CONCLUSIONS

We investigated the impact of phased behavior on various C++ benchmarks. We show that the conditional branches within the C++ benchmarks exhibit a significant amount of phased behavior. We observed a mean percentage of phased behavior of 13% at a granularity of 1000, 13.7% at a granularity of 500, and 17% at a granularity of 100. We believe this indicates a great deal of opportunity for an optimizer that can exploit path profiling. We have described techniques to improve the accuracy of our work, as well as approaches to extend the work for analyzing phased behavior in conditional branches.

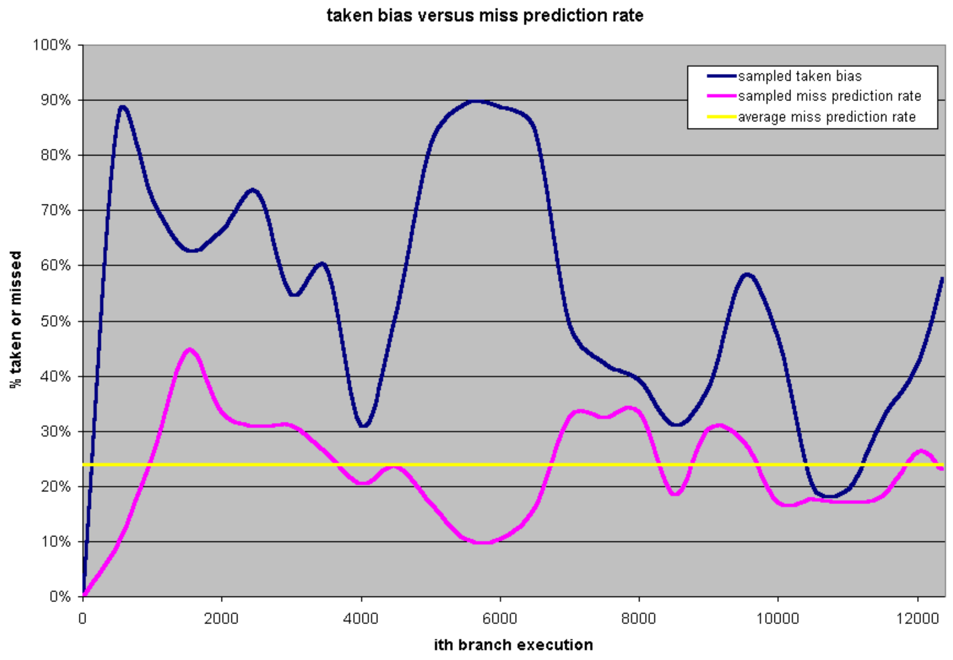


Figure 9: *Phases in Branch Predictors.* This is a branch(same branch used in Figure 1) taken from the lcom benchmark. This graph shows changes in phase of a branch relative to changes in the miss prediction rate of a simple binary 2 bit predictor.