Using Paths to Measure, Explain, and Enhance Program Behavior

Paths can reveal a program's dynamic behavior and uncover patterns of path locality that can be exploited to increase program performance. The authors explore several methods for doing so.

Thomas Ball
James R. Larus
Microsoft Research

What happens when a computer program runs? The answer can be frustratingly elusive, as anyone who has debugged or tuned a program knows. As it runs, a program overwrites its previous state, which might have provided a clue as to how the program got to the point at which it computed the wrong answer or otherwise failed. This all-too-common experience is symptomatic of a more general problem: the difficulty of accurately and efficiently capturing and analyzing the sequence of events that occur when a program executes.

Program paths offer insight into a program's dynamic behavior that is difficult to achieve any other way. Unlike simpler measures such as program profiles, which aggregate information to reduce the cost of collecting or storing data, paths capture some of the usually invisible dynamic sequencing of statements. Examination of programs' paths has unveiled a striking degree of path locality, which the computer architecture and compiler communities have profitably exploited to increase program performance.

Paths are useful at another level: as a convenient abstraction for reasoning about a program's runtime behavior. The compiler, debugging, and testing research we describe in this article builds on this abstraction. This work exploits the insight that program statements do not execute in isolation, but are typically correlated with the behavior of previously executed code.

HOW PROGRAM PATHS WORK

A program path records a program's executable statements in the order in which they run. For example, consider a simple function to add the even natural numbers from 1 to *N*:

```
int AddEvenNumbers(int N)
{
 int sum = 0;
 /* S1 */
 for (int j = 1; j <= N; j += 1) {
    /* S2 */
    if ((j % 2) == 0) {
       /* S3 */
      sum += j
    }
 }
 /* S4 */
return sum;

}
```

When invoked with an argument of 3, this function executes the path `S1, S2, S1, S2, S3, S1, S2, S1, S4`.

This type of path, which is also known as an instruction or statement trace, is unwieldy and difficult to manipulate for two reasons: first, its length is proportional to how long a program runs and, second, it must be read sequentially, like a magnetic tape. Computer architects use instruction traces to simulate processor designs, but most others found that the cost of collecting and recording a full instruction trace outweighed its utility.

Program profiles

Program profiling, a widely used substitute for paths, captures which statements execute, but not the order in which they run. Since a profile only records statement executions, it can be compactly summarized as a table of execution frequencies. The preceding example's program profile is S1 = 4, S2 = 3, S3 = 1, S4 = 1.

0018-9162/00/$10.00 © 2000 IEEE

Efficient Path Profiling

We developed a simple method to record a significant portion of the path executed by a program.[1] Our technique records path spectra consisting of intraprocedural, acyclic paths. A path is intraprocedural if it is contained entirely within one procedure. A path is acyclic if it does not contain a cycle, in which control returns to a point for a second time. These cycles are introduced by loops or by recursion. They cause problems because unbounded iteration makes the set of potential paths unbounded, as each loop iteration extends a path.

Unbounded sets are difficult to represent and manipulate, so we focused on intraprocedural, acyclic paths that do not cross a loop's back edge. The number of such paths is bounded by the graph's size, although the bound is exponential in the worst case, as in Figure 7a. These acyclic paths fall into four categories: a path from the

- procedure entry to the procedure's exit,
- procedure entry to a loop's back edge,
- head of a loop to a loop's back edge, or
- head of a loop to the procedure's exit.

Our profiling technique adds code along some edges in a procedure's control-flow graph (CFG). This code adds specially selected values into an accumulator. When control reaches either a loop's back edge or a procedure's exit, the value in this accumulator uniquely identifies the acyclic path executed by the procedure.

This value can be recorded, and the accumulator reset to record the next path executed by the procedure. For example, Figure A contains the annotated CFG of the AddEvenNumbers procedure and shows the path numbers computed by the instrumentation code. In general, our algorithm guarantees that every acyclic path will be represented by a unique integer value. Further, the algorithm computes the most compact encoding possible.

In Figure A, the six acyclic paths' values range from 0 to 5. The overhead cost of this form of profiling can be very low because most instrumentation simply increments a counter by a constant value. Most of the profiling expense comes from recording the executed paths in an array or hash table.

Reference

1. T. Ball and J.R. Larus, "Efficient Path Profiling," *Proc. 29th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 46-57.

Path	Value in acc
3, 4, 5, 6	0
3, 4, 6	1
3, 7	2
1, 2, 3, 4, 5, 6	3
1, 2, 3, 4, 6	4
1, 2, 3, 7	5

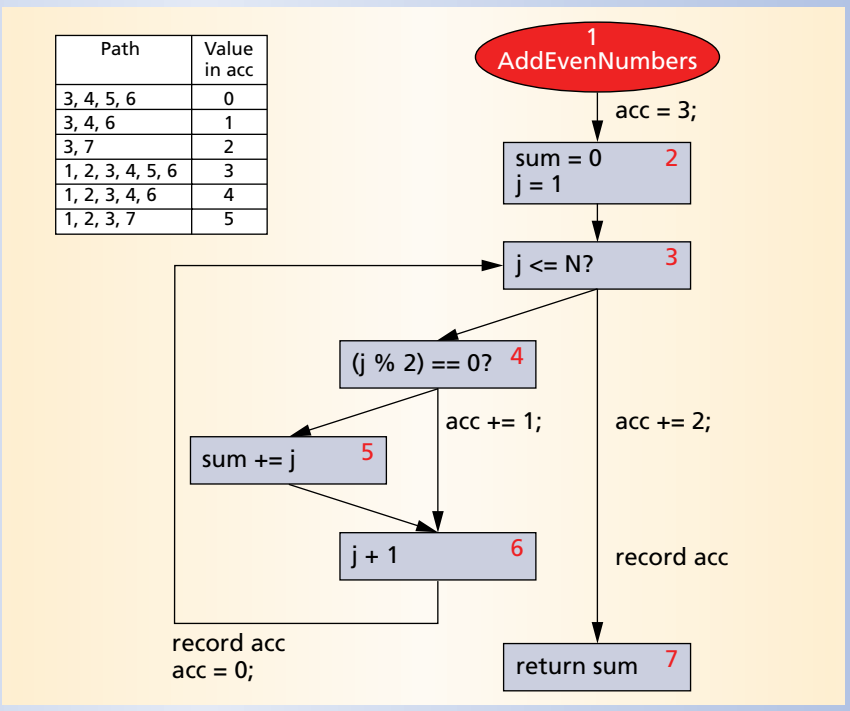

Figure A. Path profiling code for the AddEvenNumbers function. The code along the edges computes unique numbers for each acyclic path in the function.

Compact and inexpensive to collect, profiles help identify heavily executed code but provide little insight into a program's dynamic behavior. The profile of the sample program, for example, does not capture the loop iteration in which statement S3 executes.

Acyclic path segments

A practical compromise between these two extremes breaks a full program path into shorter segments, each of which fully captures the program's control flow for a portion of its execution, then profiles the segments. Because program loops introduce an unbounded number of potential path segments, naming individual paths can be difficult. To overcome this obstacle, we identified a tractable set of segments—intraprocedural, acyclic paths—that consist of the longest paths within a function that do not traverse a loop's back edge. Because programs contain only a finite number of these paths, each one can be identified and named. Moreover, they can be profiled efficiently. The sample program, for example, contains three acyclic paths: P1: S1 → S2, P2: S1 → S2 → S3, and P3: S1 → S4. The profile for these paths is P1 = 3, P2 = 1, and P3=1.

Acyclic path segments offer two benefits. First, they

concisely and efficiently capture the execution history of many instructions, and so record at least part of a program's dynamic control flow. Second, control locality of acyclic paths is even more pronounced than code locality in the program as a whole. Code locality is typified by the 80-20 rule: Eighty percent of a program's execution occurs in only 20 percent of its code. In considering paths, the 80-20 rule becomes the 100-0 rule because a program executes only a miniscule fraction of the potential paths through the nearly infinite maze of its flow graph. The 80-20 rule reappears, however, within the domain of executed paths, as a program spends most of its time in only a few *hot paths*.

Hot paths, which can account for 90 percent of a program's executed instructions—and similar fractions of instruction stalls, cache misses, and so on—are a natural focus of processor and compiler research and the primary way in which paths have helped improve program performance. Programs' high path locality helps simple hardware devices, such as branch predictors, dynamically predict a program's future behavior with high accuracy. Similarly, a path profile helps a compiler statically predict a program's expected behavior, which enables more precise program analysis and facilitates optimization decisions.

On the other hand, paths have not yet led to improvements in software development, although several directions look promising. The information that paths compactly record about a program's dynamic behavior can be useful in finding program bugs. Preliminary work in this area has shown that comparing paths executed in two program runs can isolate code that behaves differently with distinct values, such as dates before and after the year 2000. Unfortunately, paths also point to a major weakness in program testing, as it is difficult to force programs to exercise a significant fraction of their paths, any of which may contain unseen bugs. One promising idea, however, involves the software equivalent of "design for testability," in which code is written to increase the ratio of tested to potential paths.

In this article, we use a series of examples from several computing areas to show the benefits of thinking about program behavior in terms of paths.

PATH MEASUREMENTS

Researchers[1,2] have developed an efficient technique for recording the acyclic paths executed by a program and capturing cost metrics along these paths, as described in the "Efficient Path Profiling" sidebar. To demonstrate their tool, they measured the SPEC95 benchmarks and found unexpected program behavior. For example, Figure 1 shows the cumulative distribution of instructions executed along paths in the integer SPEC95 benchmarks. The floating-point benchmarks have similar distributions but fewer distinct paths. Programs cluster into two distinct groups. The first group, which includes programs other than go and gcc, executes 90 percent of its instructions along 10 to 100 distinct paths. The other group (go and gcc) executes only 40 to 50 percent of its instructions along the top 100 paths and requires approximately 1,000 paths to reach the 90th percentile. The behavior of these two programs—the largest and most computationally interesting SPEC benchmarks—approaches that of commercial software.

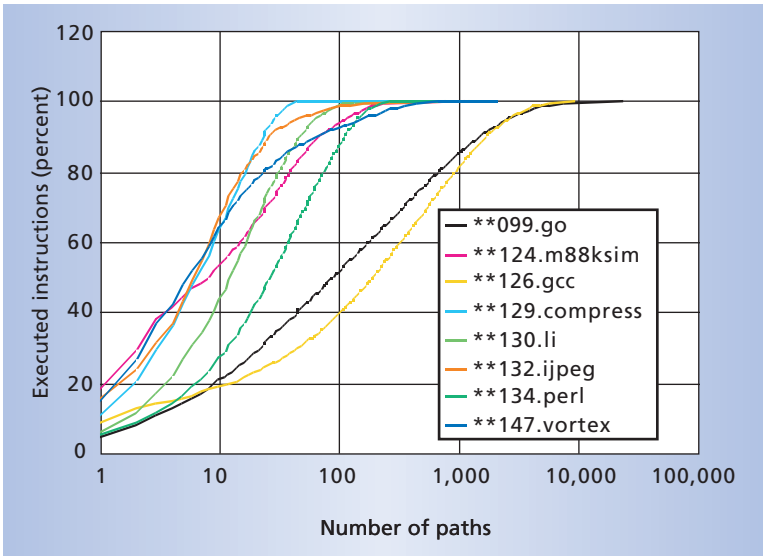

Figure 1. Cumulative distribution of instructions along paths in SPEC95 integer benchmarks. The chart shows the smallest number of paths along which a program executes a given percentage of its instructions.

In commercial programs, even a thousand paths constitute an insignificant and quite manageable fraction of all potential paths. Accurately computing the number of potential acyclic paths in a program is difficult, as this value quickly exceeds the capacity of 32-bit or even 64-bit integers. Many, or perhaps even most, of these potential paths may be computationally infeasible. This amazing amount of control locality provides computer architects with a practical basis for improving program performance.

COMPUTER ARCHITECTURE

Since programs execute few distinct paths, knowledge of which path a program is executing aids computer hardware in predicting a program's future behavior. To make these predictions, the hardware must track previously executed paths and recognize enough of the current path prefix to predict the remainder of the path.

Branch prediction

As a concrete example, consider hardware branch prediction, which attempts to predict the target of a

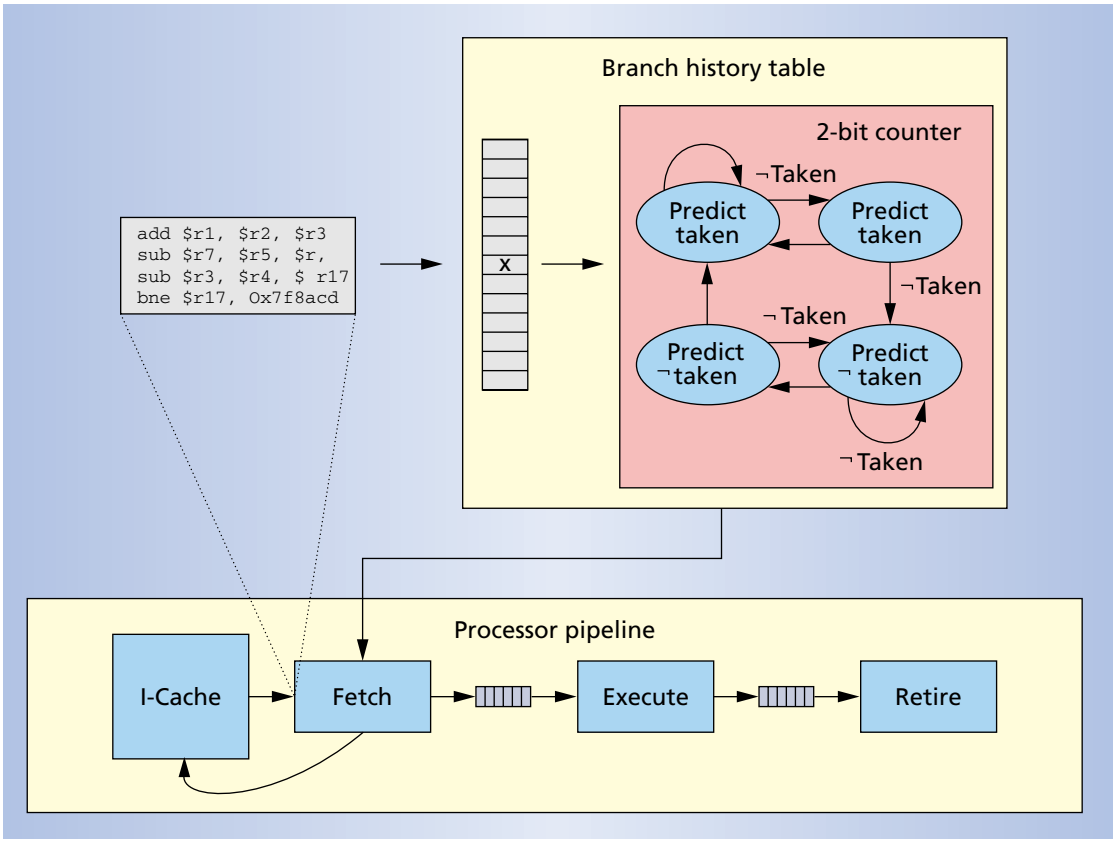

Figure 2. Branch history table. A branch's address is hashed into a table, which contains a 2-bit saturating counter that predicts if the branch will be taken, based on its past few outcomes.

conditional branch before it executes so that target instructions can be fetched early enough to avoid stalling a processor's pipeline.

Early branch predictors treated each branch in isolation. They recorded the outcome of a branch and used its history to predict whether the branch would be taken next time. Figure 2 shows a commonly used approach, which maintains a branch history table that contains two-bit counters to predict branch outcomes. The low-order bits of a branch's memory address index the table. The two-bit saturating counter records the outcome of the branch's previous executions. Ignoring addressing collisions, in which several branches unintentionally share a counter, each counter operates as a finite-state automaton that predicts a branch will have the same outcome as its previous executions.[3]

More recently, correlated or two-level adaptive branch predictors have exploited path locality to improve the accuracy of branch prediction. These branch predictors are widely used in high-performance processors such as the Compaq Alpha and Intel Pentium III. Correlated predictors also use two-bit counters to predict a branch outcome. However, the counter that makes a prediction is selected by a combination of the branch's address and the history of a few previous executed branches, which approximates the path leading to the branch.[4]

This series of branch outcomes may not uniquely identify the executed path, as several paths leading to an instruction can share a tail of identical branch outcomes. Nevertheless, the approximation is good enough to enable these predictors to reduce branch mispredictions from 10 to 15 percent down to 5 to 10 percent. We can attribute this improvement to the increased predictability of a branch along a single path, as compared to its aggregate behavior along all paths.[5]

Trace cache

Eric Rotenberg, Steve Bennett, and James Smith's trace cache makes clearer the connection between program paths and high-performance hardware.[6] A trace cache, by explicitly recording and fetching program paths, stores instructions in the order in which they execute. Trace caches improve cache memory utilization by only storing executed instructions. They also improve instruction fetching because a single cache access may provide a processor with instructions from several, noncontiguous basic blocks. Trace caches are practical because programs execute relatively few different paths and effective because programs heavily execute a small subset of these paths.

COMPILERS

Program paths have also proven useful in formulat-

ing effective compiler algorithms for optimizing programs. Compilers must always strive to balance the twin goals of correctness and efficacy, which leads compiler writers to adopt two contradictory perspectives on program paths. To ensure that an optimization does not change a program's semantics, compiler analysis takes an egalitarian perspective, which treats all potential execution paths equally, even those that rarely or never execute. On the other hand, effective optimization demands a meritocracy in which a compiler's resources are spent identifying and improving heavily executed code. From this perspective, paths are not equally valuable or interchangeable, as some offer far larger opportunities to improve program performance than do others.

Trace scheduling

Nowhere is this contrast clearer than in trace scheduling, one of the earliest uses of program paths.[7] This compilation technique schedules instructions along a heavily executed path as if they executed in a single basic block, as shown in Figure 3. Larger blocks increase a scheduler's opportunities to move instructions around, both to hide operation and memory latency and to effectively utilize a processor's multiple functional units. Program correctness, however, requires fix-up code for paths that partially overlap a trace—by transferring control into and out of the scheduled instructions—to compensate for the side effects of reordering instructions.

In practice, the fix-up code significantly increases program size. Nevertheless, several high-performance compilers use this method, and improvements and extensions of the basic idea underlie many scheduling techniques for superscalar processors.

Path-based optimization

Recent compiler algorithms have looked to paths to provide a method for untangling a program's control flow and for performing localized and profitable optimizations. For example, Frank Mueller and David Whalley show that duplicating code to separate overlapping paths can expose redundant operations.[8] To illustrate this idea, consider the simple example:

```
for (x = 0, i = 1; i < 100; i ++)
  if (x != 0){
    print (i / x);
  }
  else {
    if (f(i)) {
      x = i;
    }
  }
```

By separating the two paths through the innermost conditional, the outer conditional can be eliminated:

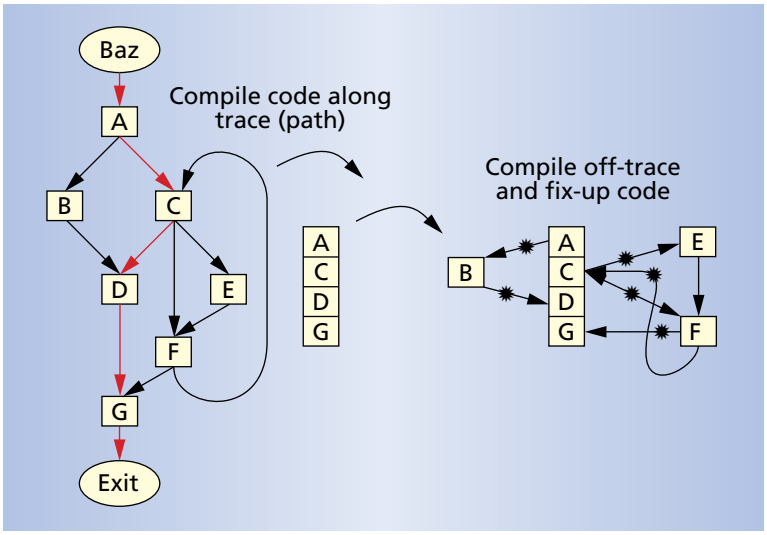

Figure 3. Trace scheduling first optimizes and compiles the code along a hot path (shown in red), called a trace, then goes back and adds compensation code where control flows into or out of the trace.

```
for (x = 0, i = 1; i < 100; i ++)
  if (f(i)) {
    x = i;
    break;
  }

for ( ; i < 100; i ++)
  print (i / x);
```

This optimization is difficult to express in conventional compiler terms—without paths—because it depends on recognizing that the original loop's body contains three paths that execute in a fixed order:

1. through the print statement,
2. through the assignment statement, and
3. through the missing alternative of the nested conditional.

The third path executes zero or more times, then the second path executes once, and only then does the first path execute zero or more times. Conventional program analysis aggregates all paths through the loop to find properties that hold regardless of how execution arrived at a point. From this perspective, little can be done with this loop, as the definition and use of the variable *x* prevents code motion.

Program analysis

Another area in which paths have proven useful is program analysis. Compilers traditionally analyze programs using dataflow analysis, which emphasizes correctness rather than precision, since it assumes that all paths are equally likely to execute. Dataflow analysis propagates a collection of values, which represent relations that hold when a program executes, along all paths in a program's control flow graph. The analysis

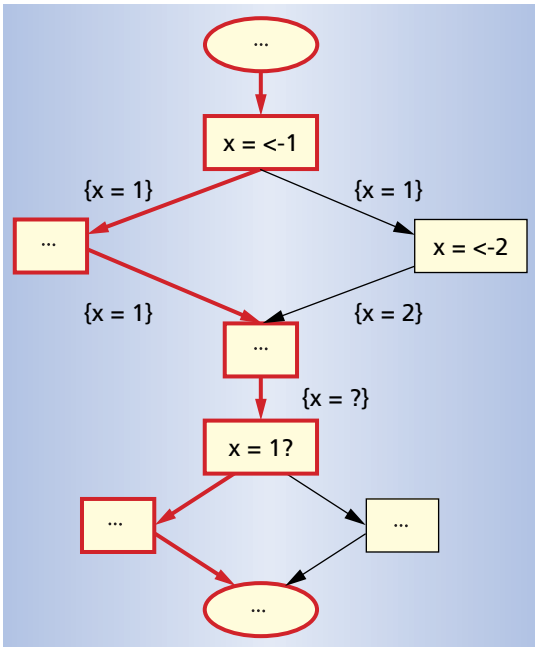

Figure 4. Dataflow analysis introduces imprecision. Along this control flow graph's hot path, shown in red, the variable x has the value 1. Another path merges into this path, however, introducing the value 2 for x. As a result, conventional dataflow analysis does not detect the constant value along the hot path, which may prevent the compiler from properly optimizing this important code section.

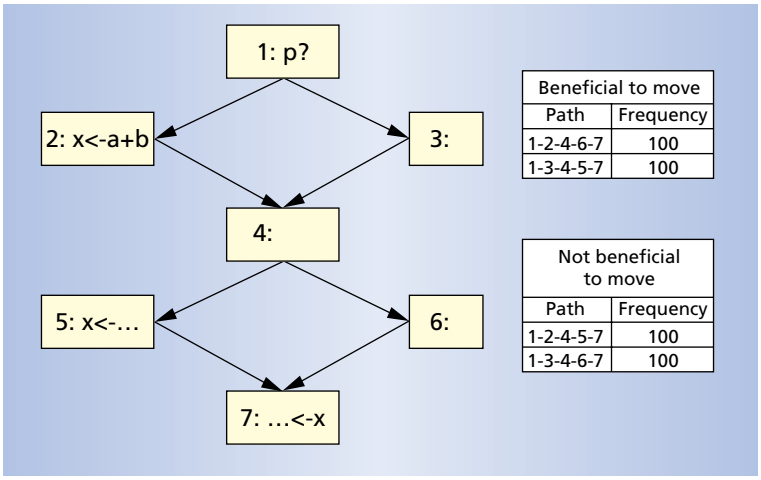

Figure 5. Path-guided optimization. The expression (a + b) *should be moved from block 2, where it is partially dead, to block 6, if it is not evaluated more at block 6 than it was at block 2. The path profiles illustrate two scenarios in which moving the partially dead expression is either beneficial or detrimental.*

updates these values to reflect the effects of statements along a path. Since the number of potential paths is unbounded, dataflow analysis does not maintain individual values along any path. Instead, at every point at which two or more paths come together, flow analysis merges their values into a common result that holds for all paths reaching the merge point. The resulting value is correct for all paths but, like a committee's consensus, may not be the most specific or useful result.

Figure 4 contains a simple example that shows how flow analysis introduces imprecision. The variable x has the value 1 along the hot path in Figure 4. However, dataflow analysis combines other values for x—in this case, 2—that reach blocks along this path. Thus a conventional compiler analysis would not detect that the variable is constant along the hot path. As this example shows, decreased analytical precision can prevent a compiler from optimizing code along a hot path, even if the program rarely or never executes other paths that degrade the analysis.

To address this problem, Glenn Ammons and James Larus introduced path-enhanced flow analysis that increases the precision of flow analysis along a program's hot paths.[9] Before applying dataflow analysis, this technique duplicates a routine's hot paths, then performs flow analysis in the conventional manner on the resulting augmented flow graph. The analysis is as precise as possible along the hot paths, given that no other paths merge with these paths. Duplicating hot paths increases the program's size, however. So, as a final step, this technique examines the analytical results for the hot paths to see if these results are more precise than those for the original, unduplicated paths. If duplication did not sufficiently increase the precision along a hot path, the duplicate is folded back into the original path.

Path-guided optimization

Path frequencies can also aid a compiler in making trade-offs among various optimization strategies. For example, Rajiv Gupta, David Berson, and Jesse Fang showed that path profiles could guide elimination of partially dead code.[10] An expression is dead at some point in a program if its value will not be used subsequently along any path. Dead expressions without side effects should be deleted, both to save code space and prevent wasted computation. An expression is partially dead if it will not be used along some paths leading from a point. A partially dead expression can sometimes be moved so that it executes only along the paths in which its value is needed. Without path profiles, a compiler must be conservative to avoid moving an expression where it would be more heavily executed. For example, Figure 5 contains two scenarios that cannot be distinguished with conventional block or edge profiling. In the first, moving the partially dead expression (a + b) does not increase its execution frequency, while in the second scenario, the expression is evaluated far more often after optimization.

DEBUGGING

Program paths underlie the debugging process, although conventional debuggers do not directly support them. Programmers spend part of the debugging process answering the questions, "How did the pro-

```
byear = read();
college = read();
items = read();
age =  current_date() - byear;
if (age < 15)    { B } else { C }
if (college)     { D } else { E }
if (items) > 3) { F } else { G }
```

Run	Path							
	BDF	BDG	BEF	BEG	CDF	CDG	CEF	CEG
Pre-2000			•	•	•	•	•	•
Post-2000	•	•	•	•				

Figure 6. Example showing how path spectra help expose Y2K problems.

gram get here?" or "What happened when the program ran?" Because paths can help answer such questions, many researchers have tried to improve debugging by making debuggers more path sensitive.

Path expressions

Some work has investigated mechanisms for reasoning about program executions at the path level. Path expressions are regular expressions, over an alphabet of control flow entities such as statements or procedures, that give a programmer a path-based query facility for directing the debugging process.[11] For example, consider the path expression `Open (Read | Write)* Close`, which captures the normal sequence of operations on a file.

The instructions Open, Read, Write, and Close represent calls on an I/O library. This regular expression can be compiled into a finite-state machine that accepts only strings in its language. As a program executes library calls, a debugger can step through the finite-state machine. Upon reaching a rejecting or accepting state, the debugger can take further action, such as raising an error or reporting that the path expression has been satisfied.

Path spectra

A program's path spectrum is the collection of acyclic paths executed in a program run. Although the path segments cannot be stitched back into a single path, comparing spectra from different executions of a program can identify places in which the program's behavior varies with its input. Thomas Reps and colleagues showed that path spectra could help locate code that may be date dependent.[12] By using input data sets that share all values except one, differences in a program's spectra can be attributed to the changed value. For example, suppose program P has input I, which contains a date before the year 2000. Running program P on I yields path spectrum S. Now, perturb input I by modifying the pre-2000 date to a post-2000 date, to yield input I′. Running program P on I′ yields path spectrum S′. The differences between path spectra S and S′ highlight changes in the program's behavior, which might illuminate Y2K errors.

Figure 6 contains a small example that illustrates this approach.[12] The sample program reads three pieces of data about a person: *byear*, the person's birth year; *college*, a Boolean value that indicates the person graduated from college, and *items*, the number of items the person has purchased. The program calculates the age of the person and then performs various actions (B, C, D, E, F and G). Dates are represented by the last two digits of the year—thus, 1998 appears as 98. We denote each of the eight paths by its actions; so BDF represents the path in which each predicate evaluates true.

The table in Figure 6 shows two path spectra, one from a pre-2000 run in which the function *current_date* returns 98, and one from a post-2000 run in which the function *current_date* returns 01. In the pre-2000 run, the path BDF does not execute because the database does not contain any people under the age of 15 who have completed college. However, in the post-2000 run, this path executes because the *age* variable becomes negative.

TESTING

Program paths also form a key part of the program testing process, both in assessing test coverage and in automating test generation.

Path coverage

Test coverage evaluates the adequacy of a program-test-case collection by measuring which parts of a program the collection tests. The most widely used coverage metrics quantify the fraction of statements and branches that execute when the tests run. Path coverage is a far more problematic criterion. Programs contain a finite number of statements and branches, but in general, because of loops, can execute an infinite number of paths. Even if we consider only acyclic paths, the number of possibilities is huge. Further, many paths may be infeasible because no inputs could

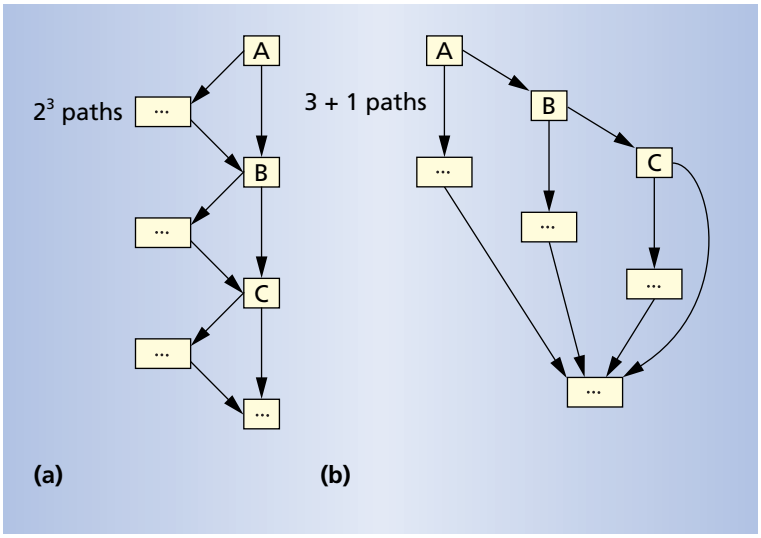

Figure 7. Two procedures with the same number of predicates and branches, but different numbers of paths.

satisfy the predicates along them. Nevertheless, path coverage can be a useful test criterion for small, highly critical code regions. For example, consider the following code fragment:

```
if ((A||B) && (C||D)) {
  X
} else {
  Y
}
```

This fragment contains two statements (X and Y), four tests (A, B, C, and D), and seven paths—four in which the predicate evaluates true and three in which the predicate evaluates false. Executing only two of seven paths achieves full statement coverage. If only these two paths are feasible, some predicate subexpressions are unreachable and the expression could be simplified. Achieving full branch coverage requires executing four paths. The other three paths, if feasible, could possibly reveal unexpected interactions among the tests or might be redundant tests.

Another approach to testing coverage focuses on paths that seem likely to reveal a fault. Consider a path that contains an assignment to variable x, but no use of the variable. Such a path is less likely to reveal a fault in the computation of x's value than a path that assigns and then uses x. This observation motivated a large family of dataflow, path-based coverage criteria.[13]

Automated test-case generation

In debugging, a crucial question arises: "Which path did program execution follow to this point?" An analogous question for testing follows: "Which path can program execution traverse to this point?" To test a particular piece of code, a tester must find a program input that causes the code to execute. One approach identifies a path to the code, then tries to determine an input to the program that will cause the path to execute.

This simple formulation conceals a host of nasty problems. First, the chosen path may be infeasible, so no input can cause the path to execute. Second, finding an input that causes a program to execute a given path is an undecidable problem. Despite—or perhaps because of—these considerable difficulties, much research has explored the area of automated test-case generation.

Consider the following example, which counts how many of three variables have positive values, then prints the count if it is equal to three:

```
x = read(); y = read(); z = read();
count = 0;
if (x > 0) count++;
if (y > 0) count++;
if (z > 0) count++;
if (count == 3) printf("count = 3");
```

Suppose we wish to find a path that causes the call to printf to execute. Only one such path exists, and the precondition for its execution is that the three input variables all have a positive value. In this example, each of the three predicates tests an independent variable, so each predicate branch is independent of the others. In general, however, predicates will be dependent on one another, which greatly complicates the automated generation of input. Techniques such as symbolic execution and theorem proving, both very expensive, are needed to make the inferences necessary to determine if a path is feasible.[14]

SOFTWARE COMPLEXITY AND PROGRAM UNDERSTANDING

Program paths also offer a new perspective on software complexity.[15] Nearly all software complexity metrics count entities immediately apparent in a program. For example, the well-known McCabe cyclomatic complexity measure[16] counts the number of decision points—either branches or predicates.

As the two control flow graphs in Figure 7 demonstrate, however, structural relationships among predicates strongly influence the number of paths through code. Figure 7a contains three predicates—A, B, and C—and eight paths. Figure 7b also contains three predicates, but only four paths. The McCabe complexity of both procedures is 5, but the complexity of understanding and testing the two procedures differs greatly. In general, given N binary predicates, a procedure can contain anywhere from $N + 1$ paths, when the predicates are nested N-level deep, as in Figure 7b, to 2^N paths, when the predicates are strung out in sequence.

Each path represents a potential execution scenario that a programmer may need to consider when under-

standing or testing the code. For this reason, the number of paths through a procedure provides a better metric of a procedure's complexity than a simple count of branches or statements. As usual, feasible and infeasible paths complicate the picture. Recall the example of the counting code. That code contains 16 potential paths. The first three predicates are independent of each other, so eight feasible paths run through the first three statements. The predicate in the last statement, which checks if *count* is 3, is dependent on the first three predicates because if any of them evaluates false, *count* will not be 3 and the final predicate will evaluate false. This dependency means that eight feasible paths run through the four statements. Despite its path complexity, the code is relatively simple to understand because of the first three branches' independence.

In the compiler, debugging, and testing fields, program paths are coming into use and much work remains. Path profiling techniques need to be extended beyond acyclic paths to efficiently capture longer paths that cross procedure and loop boundaries.

These areas, and others, would benefit from a more detailed understanding of infeasible paths. Such work could help develop algorithms for distinguishing those that are easily identified and ignored from the intractable remainder. More generally, further work may help clarify the connection between complex or unreliable code and path complexity. ✸

Acknowledgments

We thank Daniel Weise and David Tarditi for providing many helpful comments.

References

1. G. Ammons, T. Ball, and J.R. Larus, "Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling," *Proc. SIGPLAN 97 Conf. Programming Language Design and Implementation*, ACM Press, New York, 1997, pp. 85-96.
2. T. Ball and J.R. Larus, "Efficient Path Profiling," *Proc. 29th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 46-57.
3. J.E. Smith, "A Study of Branch Prediction Strategies," *Proc. Eighth Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1981, pp. 135-148.
4. T.-Y. Yeh and Y. Patt, "A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History," *Proc. 20th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 257-265.
5. C. Young, N. Gloy, and M.D. Smith, "A Comparative Analysis of Schemes for Correlated Branch Prediction," *Proc. 22nd Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 276-286.
6. E. Rotenberg, S. Bennett, and J.E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," *Proc. 29th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 24-34.
7. J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. Computers*, Vol. C-30, No. 7, 1981, pp. 478-490.
8. F. Mueller and D.B. Whalley, "Avoiding Unconditional Jumps by Code Replication," *Proc. SIGPLAN 92 Conf. Programming Language Design and Implementation*, ACM Press, New York, 1992, pp. 322-330.
9. G. Ammons and J.R. Larus, "Improving Data-Flow Analysis with Path Profiles," *Proc. SIGPLAN 98 Conf. Programming Language Design and Implementation*, ACM Press, New York, 1998, pp. 72-84.
10. R. Gupta, D.A. Berson, and J.Z. Fang, "Path Profile Guided Partial Dead Code Elimination Using Predication," *Proc. Int'l Conf. Parallel Architecture and Compilation Techniques (PACT)*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 102-115.
11. B. Bruegge and P. Hibbard, "Generalized Path Expressions: A High-level Debugging Mechanism," *J. Systems and Software*, Vol. 3, No. 4, 1983, pp. 265-276.
12. T. Reps et al., "The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem," *Proc. Fifth ACM SIGSOFT Symp. Foundations of Software Eng.*, M. Jazayeri and H. Schauer, eds., Springer-Verlag, Berlin, 1997, pp. 432-449.
13. L.A. Clarke et al., "A Formal Evaluation of Data Flow Path Selection Criteria," *IEEE Trans. Software Eng.*, Vol. 15, No. 11, 1989, pp. 1,318-1,332.
14. L.A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Trans. Software Eng.*, Vol. 2, No. 3, 1976, pp. 215-222.
15. B.A. Nejmeh, "NPATH: A Measure of Execution Path Complexity and Its Applications," *Comm. ACM*, Feb. 1988, pp. 188-200.
16. T. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.*, Vol. 2, No. 4, 1976, pp. 308-320.

Thomas Ball *is a researcher at Microsoft. His interests include software tools, model checking, program analysis, and domain-specific programming languages. Ball received a PhD in computer science from the University of Wisconsin-Madison. Contact Ball at tball@microsoft.com.*

James R. Larus *is a senior researcher at Microsoft. His interests include programming languages, compilers, parallel computation, and software tools. Larus received a PhD in computer science from the University of California, Berkeley. Contact Larus at larus@microsoft.com.*