Programs Follow Paths

Thomas Ball
(Bell Laboratories, Lucent Technologies)
James R. Larus
(Microsoft Research)

January 6, 1999

Technical Report
MSR-TR-99-01

Bell Laboratories, Lucent Technologies Technical Report
BL0113590-990106-01

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Programs Follow Paths

Thomas Ball
tball@research.bell-labs.com
Bell Labs, Lucent Technologies
263 Shuman Blvd, Room 2A-314
Naperville, IL 60566

James R. Larus
larus@microsoft.com
Microsoft Research
One Microsoft Way
Redmond, WA 98052

Program paths—sequences of executed basic blocks—have proven to be an effective way to capture a program's elusive dynamic behavior. This paper shows how paths and path spectra compactly and precisely record many aspects of programs' execution-time control flow behavior and explores applications of these paths in computer architecture, compilers, debugging, program testing, and software maintenance.

1 Introduction

What happens when a program runs? This simple question underlies work in many areas of computer science, ranging from processor design to software development. At a basic level, a computer steps through a sequence of instructions. As the machine works its way through the program, it executes instructions, which modify the machine's current state. The process is inherently unidirectional, for as soon as an instruction completes execution, it and the previous state are lost.

A program's relentless forward progress makes it difficult to understand its dynamic behavior. Examining a program's state—by stopping it, for instance—provides only clues about the sequence of events leading up to that point in execution. The need to go further, to understand the sequence of events that actually occur when a program executes, unifies disparate areas of computer science. The insight to design new computers, write better compilers, or even debug programs arises from understanding patterns and causality among program events.

Instruction traces provide a complete description of a program's control-flow behavior. However, the large size of traces and the high cost of obtaining them makes traces impractical for everyday use. Traditional tools for measuring program behavior, such as profiling and test coverage tools, summarize the execution history of statements or procedures. These profiles are inexpensive to collect and invaluable in finding heavily executed code, but provide little insight into the dynamic sequencing of operations.

Recent work in many areas of computer science and engineering has shown that *program paths* provide a practical approach to capturing many important aspects of a program's dynamic behavior. Paths have been used to improve computer hardware, compilers, and software, in general. A program path records how a program's control transfers through a sequence of consecutively executed basic blocks (see Sidebar 1). Although a program's execution traces a single path, practicality demands this path be broken into shorter, more manageable path segments. The difficulty is that the number of potential paths through a program with loops is unbounded, which makes individual paths difficult to identify and name. On the other hand, a complete path can be assembled from shorter subpaths, such as Ball and Larus's acyclic paths [2]. Because the number of acyclic paths in a program is finite, they can be identified and named (see Sidebar 2). In this paper, the term "path" will refer to these acyclic paths.

Paths are a useful for two reasons. First, they concisely capture the execution history of many instructions, and so record a program's dynamic control flow. The set of (acyclic) paths executed by a program, its *path spectra*, compactly describes much of the program's dynamic behavior.

Second, the *control locality* of paths is even more pronounced than code locality in a program as a whole. Code locality is typified by the 80-20 rule—the observation that 80% of a program's execution occurs in only 20% of its code. If programs are viewed as collections of paths, the 80-20 rule becomes the 100-0 rule, since programs execute only a miniscule fraction of the possible paths through the nearly infinite maze of their flow graph. The 80-20 rule reappears, however, within the domain of executed paths, as programs spend most of their time in a small number of "hot paths."

These hot paths, which can account for 90% of the executed instructions—and similar fractions of instruction stalls, cache misses, etc.—are a natural focus for processor and compiler enhancements. These two observable facts—that programs execute few paths and only a small fraction of executed paths account for most of a program's execution cost—mean that simple hardware devices, such as branch predictors, can use the history of previously executed paths to make highly accurate predictions about a program's future behavior. Similarly, hot paths enable compilers to identify a program's typical behavior, which provides a quantitative basis for making the trade-offs necessary to optimize.

On the other hand, this control locality also frustrates debugging and testing, as it is difficult to force programs to exercise a significant fraction of their paths, many of which may contain errors. Moreover, many, if not most, of these unexecuted paths are computationally infeasible. Since separating the two groups is an undecidable problem, defining an adequate level of testing is impossible. In practice, programs' natural locality often makes achieving even minimal path coverage difficult. However, one promising approach is a software

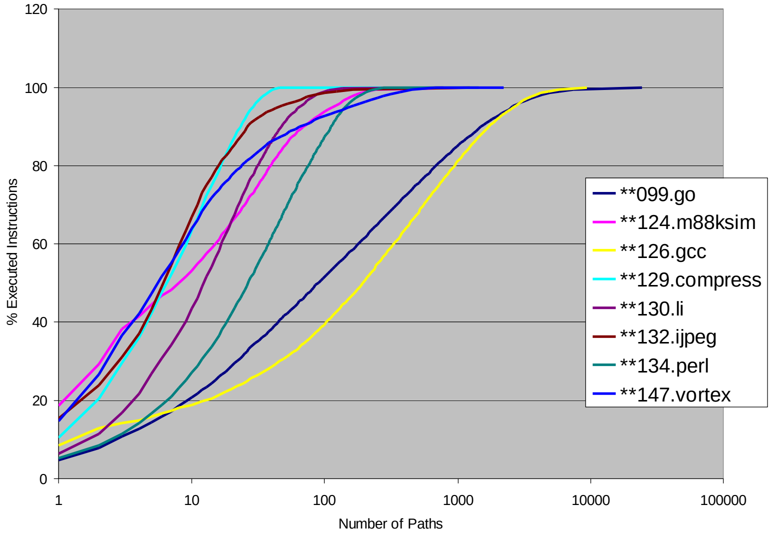


Figure 1. Cumulative distribution of instructions along paths in SPEC95 integer benchmarks. The chart shows the smallest number of paths along which a program executes a given percentage of its instructions.

equivalent of "design for testability," in which code is rewritten to increase the ratio of tested to potential paths. This can both simplify a program's code and improve test coverage.

2 Path Measurements

Ammons, Ball, and Larus developed an efficient technique to record the intraprocedural acyclic paths executed by a program and to capture cost metrics along these paths (see Sidebar 2) [1, 2]. To demonstrate their tool, they measured the SPEC95 benchmarks and found unexpected and striking program behavior. For example, Figure 1 shows the cumulative distribution of executed instructions along paths in the integer SPEC95 benchmarks (the floating-point benchmarks have similar distributions, but far fewer paths [2]). Programs cluster into two distinct groups. The first, which includes programs other than *go* and *gcc*, execute 90% of their instructions along 10–100 distinct paths. The other group (*go* and *gcc*) executes only 40–50% of their instructions along the top 100 paths and requires approximately 1000 paths to reach the 90th percentile. The behavior of these two programs, which are the largest and most computational interesting SPEC benchmarks, is closer to commercial software. Figure 2 compares the path distribution of these two programs against two runs of Microsoft Word.

Even a thousand paths are an insignificant, and quite manageable, fraction of the potential paths in these programs. It is difficult to accurately compute the number of potential acyclic paths in a program, as this value quickly exceeds the capacity of 32, or even 64, bit integers. All SPEC benchmarks (including library code on Sun Microsystems Solaris 2.5 operating system) contain more than 2^{32} paths. Microsoft Word (without library code)

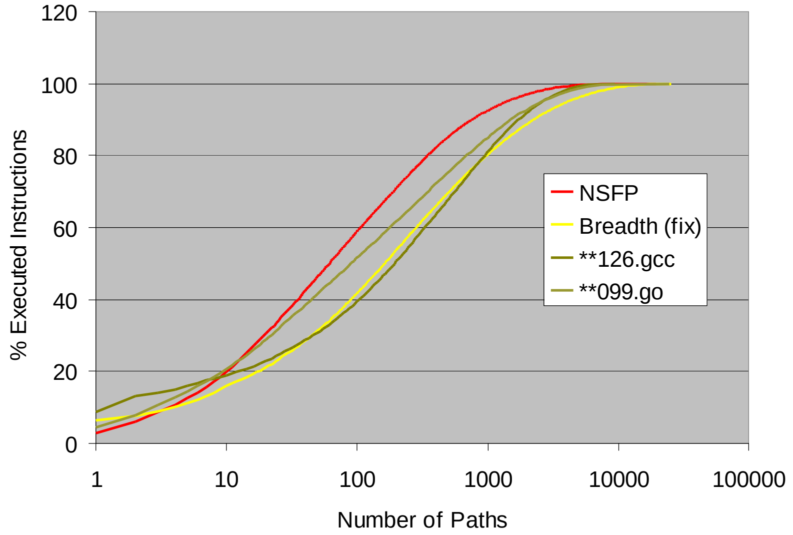


Figure 2. Cumulative distribution of instructions along paths in two runs of Microsoft Word (NSFP and Breadth) and two SPEC benchmarks.

contains more than 2^{64} paths.[1] Many, or perhaps even most, of these potential paths may be computationally infeasible. Unfortunately, determining a path's feasibility is, in general, an undecidable question.

This amazing amount of control locality is an interesting phenomenon in its own right. However, it also provides a practical basis for improving program performance. The next two sections explore how computer architects and compiler writers have exploited control locality to make programs run faster.

3 Computer Architecture

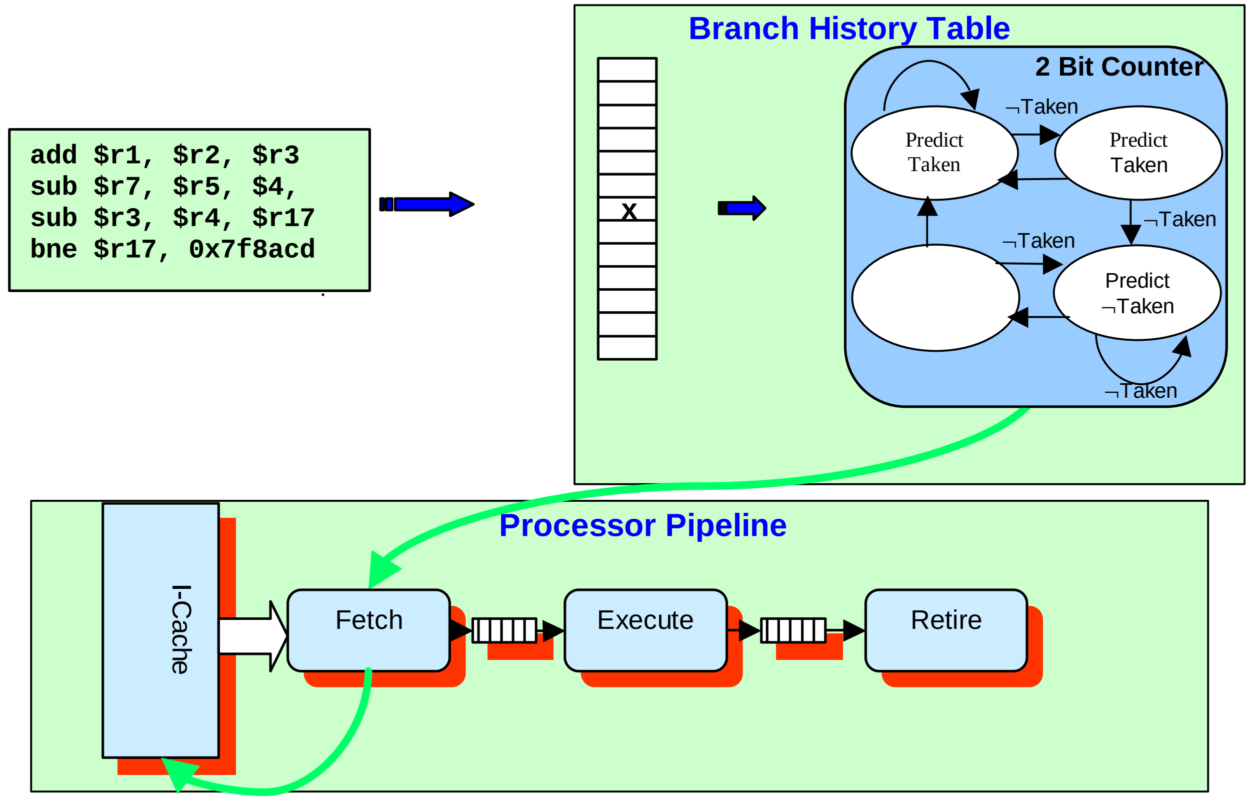
Since programs execute few distinct paths, knowledge of which path a program is executing aids computer hardware in predicting a program's future behavior. To make these predictions, the hardware must track previously executed paths and recognize enough of the current path prefix to predict the remainder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

3.1 Branch Prediction

As a concrete example, consider hardware branch prediction, which attempts to predict the target of a conditional branch, before the branch instruction executes, so that target instructions can be fetched quickly enough to avoid stalling a processor's pipeline [3].

Early branch predictors treated each branch in isolation. These predictors recorded the outcome of a branch, and used this history to predict whether the branch would be taken at its next execution. Figure 3 shows a commonly used approach, which maintains a branch history table containing two-bit counters to predict branch

[1] As, most likely, do some of the SPEC benchmarks, but the tool applied to them used only 32-bit integers to count the number of paths.

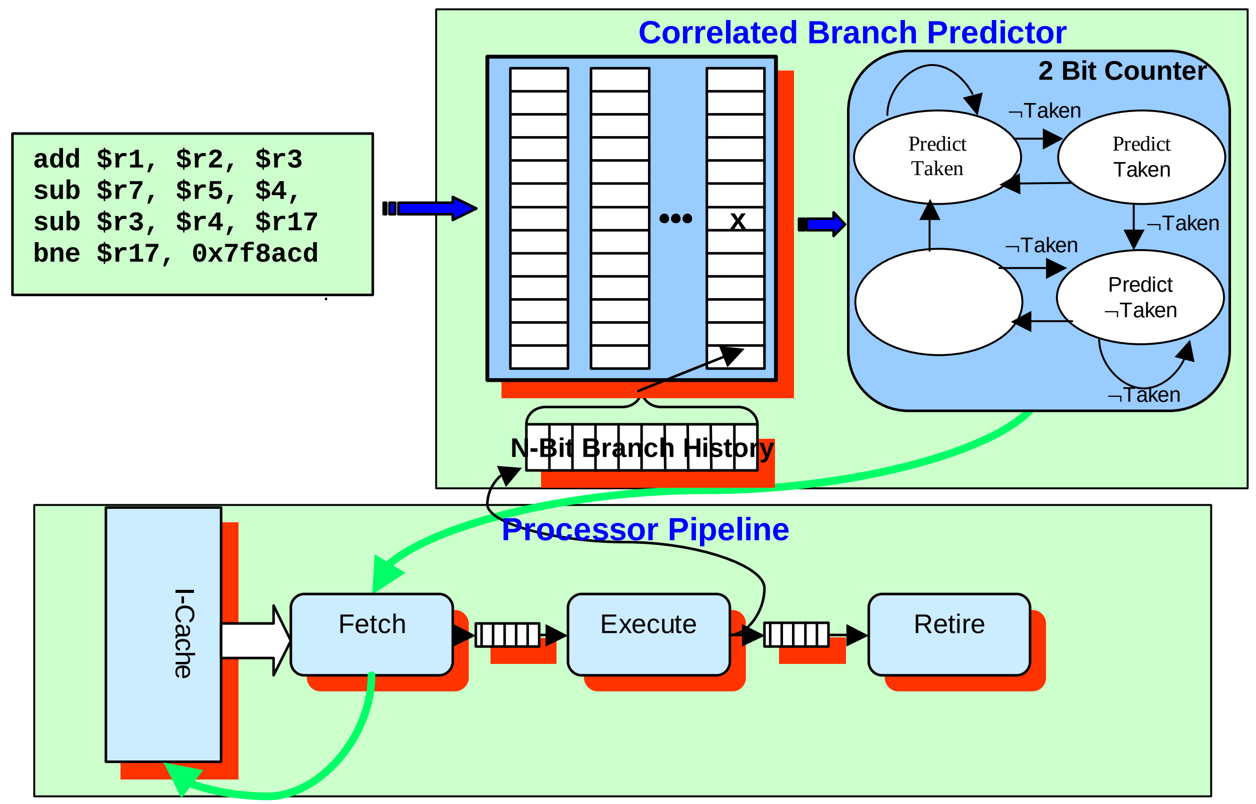


```
add $r1, $r2, $r3
sub $r7, $r5, $4,
sub $r3, $r4, $r17
bne $r17, 0x7f8acd
```

Figure 3. Branch history table. The address of a branch 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.

outcomes. The low-order bits of a branch's memory address index the table. The two-bit saturating counter records the outcome of the previous two executions of the branch. Ignoring addressing collisions, in which several branches unintentionally share a counter, each counters operates as a finite-state automaton, which predicts that a branch will have the same outcome as its previous executions. Although, one bit suffices to record the previous outcome, Smith showed that the hysteresis provided by a second bit significantly reduces mispredictions due to occasional aberrations in a stable sequence of branch outcomes [4].

More recently, correlated or two-level adaptive branch predictors have exploited control locality to improve the accuracy of branch prediction. Correlated predictors also use two-bit counters to predict the outcome of a branch (Figure 4). However, the counter that makes the prediction is selected by a combination of the branch's address and a history of the outcome of the previous few branches, which approximates the path leading to the branch [5, 6]. 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 mispredictions from 10–15% to 5–10% of branches. This



```
add $r1, $r2, $r3
sub $r7, $r5, $4,
sub $r3, $r4, $r17
bne $r17, 0x7f8acd
```

Figure 4. Correlated branch predictor hardware. The N bits of branch history records the path leading to a branch, which allows a correlated branch predictor to distinguish a branch's behavior along different path and to tailor the prediction to the path.

improvement is attributable to the increased predictability of a branch along a single path, as compared to its aggregate behavior along all paths [7].

3.2 Trace Cache

Rotenberg, Bennett, and Smith's Trace Cache makes clearer the connection between program paths and high performance hardware [8]. A trace cache is an instruction cache that stores instructions in the order in which they execute, not the order in which they are stored in memory. In other words, it explicitly stores and fetches program paths. Trace caches both improve cache memory utilization, by only storing executed instructions, and improves instruction fetching, as a single cache access may provide a processor with instructions from several, non-contiguous basic blocks. Trace caches are practical because programs execute relatively few different paths, and heavily execute a small subset of these paths. These caches directly exploit a program's control locality, much as earlier instruction caches exploited code locality.

3.3 Discussion

Computer architects have successfully used program paths (traces) to improve the mechanisms needed to supply modern, multiple-issue processors with instructions for execution. However, paths are not without problems. Complete path information is too voluminous to maintain in hardware, so processors either approximate it with a bit-stream of branch outcomes, or cache recently executed paths. Measurements of programs' high control locality explain why these approximations achieve favorable results. A more fundamental problem is distinguishing two paths that partially overlap. In this case, a limited execution history may prove insufficient to predict the point at which the paths diverge. In many cases, this problem is not apparent, as the system records only one path with a given prefix (typically the most recent), so it never distinguishes these paths. Fortunately, programmers or compilers can compensate for this problem, when it arises, by duplicating code, so that the two paths appear distinct to the hardware.

4 Compilers

Program paths have also proven useful in formulating more 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 the semantics of a program, 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 paths offer far larger opportunities to improve program performance.

4.1 Trace Scheduling

Nowhere is this contrast clearer than in trace scheduling, one of the earliest use of program paths [9, 10]. This compilation technique schedules instructions along a heavily executed path, as if they executed in a single basic block (Figure 5). Larger blocks increases a scheduler's opportunities to move instructions around, both to hide operation and memory latency and to effectively utilize multiple functional units in a processor. 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 reordered instructions in the trace.

In practice, the fix-up code significantly increases the size of programs. Nevertheless, the technique has been used in several high performance compilers, and improvements and extensions of the basic idea underlie many of scheduling techniques for superscalar processors.

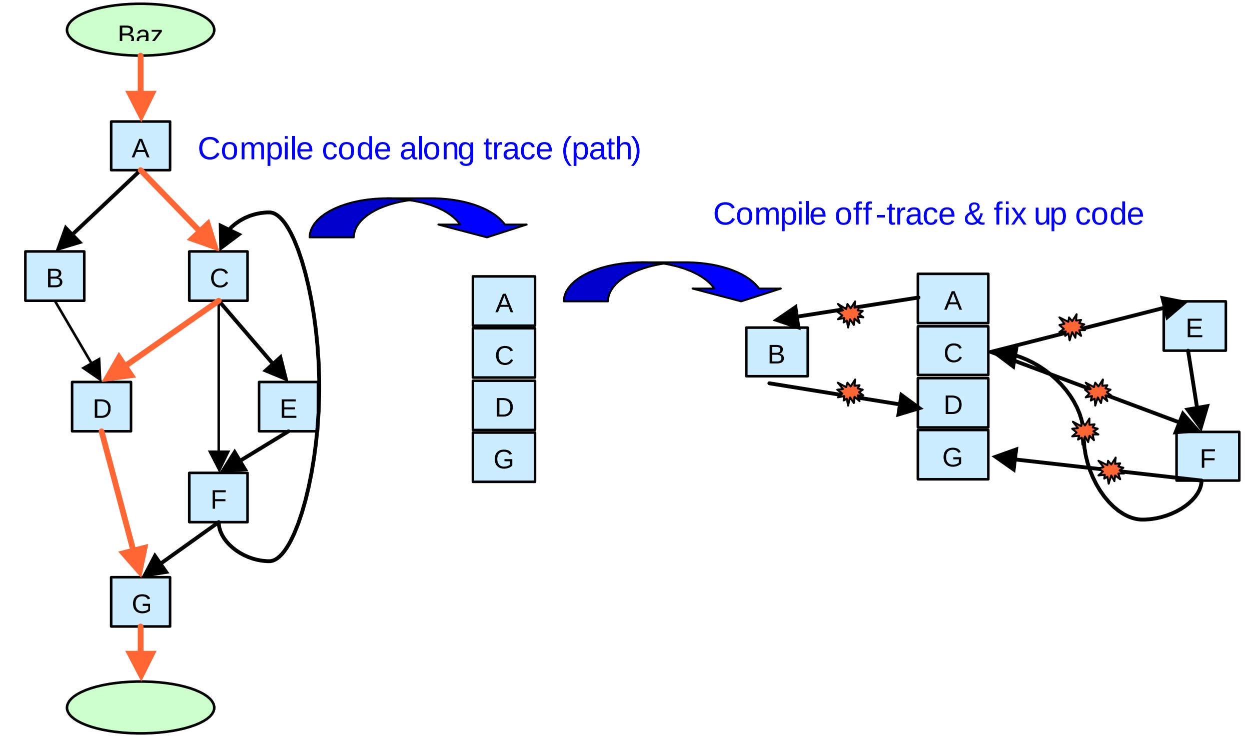

Baz
A
Compile code along trace (path)
B
C
D
E
F
G
A
C
D
G
Compile off-trace & fix up code
B
A
C
D
G
E
F

Figure 5. Trace scheduling optimizes and compiles the code along a hot path (called a trace) first, then goes back and adds compensation code where control flows into or out of the trace.

4.2 Path-Based Optimization

Recent compiler algorithms have looked to paths to provide a method to untangle a program's control flow and to perform optimizations in a localized and profitable manner. For example, Mueller and Whalley show that separating overlapping paths, by duplicating code, can expose redundant operations [11]. To illustrate the idea, consider the contrived 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 (`x != 0`), the outer conditional can be eliminated:

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

for ( ; i < 100; i ++)
```

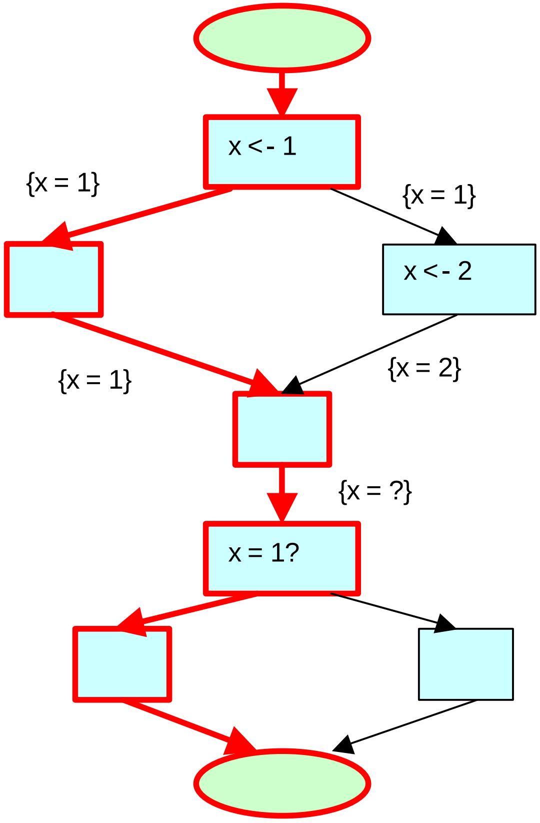


Figure 6. Data flow analysis introduces imprecision. The hot path in this control flow graph is emboldened. Along this path, the variable x has the value 1. However, another path merges into thi s path, introducing the value of 2 for x. As a result, conventional data flow analysis will not detect the constant value along the hot path, which may prevent the compiler from properly optimizing this important portion of the code.

```
     print (i / x);
```

This optimization is difficult to express in conventional compiler terms—without paths—as it depends on recognizing that the original loop's body contains three paths ((1) through the print statement, (2) through the assignment statement, and (3) through the missing alternative of the nested conditional) that execute in a fixed order. The third path executes zero or more times, then the second path executes once, and only then does the first path executes 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.

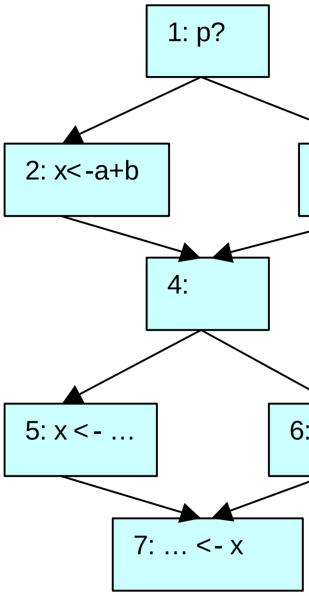
4.3 Program Analysis

Another area in which paths have proven useful is program analysis. Compilers traditionally analyze programs using data flow analysis [12], which emphasizes correctness, rather than precision, as it assumes that all paths are equally likely to execute. Data flow analysis propagates a collection of values (representing relations that hold when a program executes) through all paths in a program's control flow graph and updates these values to reflect the effects of statements along a path. Since the number of potential paths is unbounded, data flow 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, it may not be the most specific or useful result. Figure 6 contains a simple example that shows how flow analysis introduces imprecision. The variable *x* has the value 1 along the hot path in the figure. However, data flow analysis combines other values for *x* (namely 2) that reach blocks along this path, and so a conventional analysis would not detect that the variable is constant along the hot path. As this example illustrates, 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, Ammons and Larus introduced path-enhanced flow analysis, which increases the precision of flow analysis along a program's hot paths [13]. Before applying data flow analysis, this technique duplicates a routine's hot paths, and then performs flow analysis in the conventional manner on the resulting augmented flow graph. The analysis is precise as possible along the hot paths, as no other paths merge with these paths. However, duplicating hot paths increases the program's size. So, as a final step, this technique examines the analytical results for the hot paths, to see if they are more precise than the results for the original, unduplicated paths. If duplication did not sufficiently increase the precision along a hot path, it is folded back into the original path. In practice, this folding step provides a mechanism for trading precision against size.

4.4 Path-Guided Optimization

Path frequencies can also aid a compiler in making tradeoffs among various optimization strategies. For example, Gupta, Berson, and Fang showed how path profiles can guide partial dead code elimination [14]. An expression is dead at a 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 to where it would be more heavily executed. For example, Figure 7 contains two scenarios, which cannot be distinguished with conventional block or edge profiling. In the first scenario, 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.



Beneficial to Move

Path	Frequency
1-2-4-6-7	100
1-3-4-5-7	100

Not Beneficial to Move

Path	Frequency
1-2-4-5-7	100
1-3-4-6-7	100

Figure 7. Path-guided optimization example. The expression (a+b) should be moved from block 2, where it is partially dead, to block 6, if it is not evaluated more in block 6 than it was at block 2. The path profiles illustrate two scenerios, which are indistinguishable with either block or edge profiles, in which moving the partially dead expression is either beneficial or detrimental.

5 Debugging

Program paths are an essential part of debugging programs, though not directly supported by conventional debuggers. Much of the debugging process is spent answering the question, “how did the program get here”—a question that paths are well suited to answer. Programmers use debuggers to stop at a succession of intermediate program states, thereby working their way back along a program’s execution path to find the cause of an error [15]. At each intermediate state, a programmer examines values, looking for clues as to why an error occurred. When the state offers no clues, a programmer sets breakpoints in other places in the program, or adds assert or print statements, to produce more information. The program often must be re-run at each such modification. In many cases, breakpoints mark out the executed path through a program, which a programmer laboriously tracked back to the source of an error.

Many researchers have tried to improve debuggers by making them path sensitive. The discussion below presents three of these approaches: historical debuggers, path expressions, and path spectra.

5.1 Historical Debuggers

Historical debuggers automate the debugging process with a checkpoint/replay facility, which stores intermediate states of a program (checkpoints) and allows execution to restart from saved checkpoints (replay). For example, Tolmach developed a reverse-execution debugger for Standard ML, which allowed programmers to rewind program execution [16]. The main drawback to such approaches is the huge overhead of recording the trace necessary to recover previous execution states. For example, Tolmach's debugger incurred a 300% run-time overhead. Just as data-flow analysis assumes that all paths are equally likely to contain code that modifies a data flow solution, historical debuggers assume that all paths are equally likely to contain the cause of an error. In data-flow analysis, this assumption leads to over-conservative answers. In historical debuggers, this assumption leads to high run-time overhead, as the debugger ends up recording a huge amount of useless information.

5.2 Path Expressions

Some research has investigated mechanisms for reasoning about program executions at the level of paths. Path expressions are regular expressions over an alphabet of control flow entities (such as statements or procedures), which provide a programmer with an explicit path-based query facility for directing the debugging process [17]. For example, consider the following path expression, which captures the allowed sequence of operations on a file:

Open (Read | Write)* Close

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 the strings specified by the regular expression and rejects all others. As a program executes library calls, a debugger can step through the finite state machine, recording the partial path taken so far. Upon reaching a rejecting or accepting state, the machine can instruct the debugger to take further action, such as informing the user of an error or reporting that the execution satisfied the path expression.

The main benefit of path expressions, as compared to historical debuggers, is that no trace or checkpoints are needed. Path expressions provided by a programmer specify exactly which events must be detected. Moreover, the events need not be recorded for future use, but only cause state machine transitions. The drawback of these expressions is that a programmer must have an idea of the cause of an error and must be able write a formal description of the normal behavior of his or her program.

5.3 Path Spectra

Path spectra offer an intermediate point between historical debuggers and path expressions. Rather than record a complete execution trace, path spectra decompose a program into smaller path segments whose execution

```
    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 8. Example showing how path spectra can help to locate date dependent code.

is recorded. Path expressions may be able to follow longer sequences of operations than path spectra, but they require a programmer to describe the likely cause of a problem in advance. Path spectra provide a cheap and automated insight into a program's intermediate states.

Reps et al. describe how path spectra can help locate code that may have a dependency on dates [18]. The basic idea is to compare path spectra from different runs of a program. By choosing input datasets that preserve all values except one, differences in the spectra can be attributed to the changed value. For example, suppose one has input *I* to a program *P*, where input *I* contains dates before the year 2000. Running program *P* on input *I* yields path spectra *S*. Now, imagine perturbing the input *I* by modifying a pre-2000 date to a post-2000 date, to yield input *I2*. Running program *P* on input *I2* yields path spectra *S2*. Differences between path spectra *S* and *S2* point to differences in intermediate states caused by the input perturbation.

Figure 8 contains a small example (from Reps et al.) illustrating this approach. The sample program reads three pieces of data representing a person: *byear*, the person's birth year; *college*, a boolean representing whether or not 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, G*). We assume dates are represented by the last two digits of the year, e.g. 1998 is represented as 98. We will denote each of the eight paths through the program by the actions it covers, so *BDF* represents the path in which each predicate evaluates to true. The table in Figure 8 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 paths *BDF* and *BDG* do 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, both these paths execute because the variable *age* becomes negative.

Why not use block or branch profiles, instead of paths, to compare differences between the executions? In general, path spectra provide a more detailed and precise summary of a program's execution than block or branch spectra. Several executions that cover the same set of blocks and branches may follow different paths. In our example, a 1998 run can execute all blocks and branches. On the other hand, block *C* will never execute for a 2001 execution since *current_date* returns a value (1) that ensures that age will always be less than 15. A branch or block spectrum could reliably find the difference between a 1998 and 2001 execution. However, for years after 2015, when *current_date* returns a value greater than or equal to 15, all blocks and branches can execute. Path spectra alone will still show a difference, because no one under 15 completed college.

6 Testing

Program paths form a key part of the process of testing programs, both in assessing test coverage and in automated test generation.

6.1 Path Coverage

Test coverage evaluates the adequacy of a collection of program test cases by measuring which parts of a program they test. The most widely used coverage metrics are the fraction of statements and branches that execute when the tests run. Path coverage is, of course, a problematic criterion. Programs contain a finite number of statements and branches, but because of loops, have an infinite number of paths. Even considering only acyclic paths, the number of possibilities is huge. Furthermore, many of these paths may be infeasible, in that no inputs could satisfy the predicates along such a path. Nevertheless, path coverage can be useful in localized contexts. For example, consider the following fragment of C code:

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

This fragment contains two statements (*X* and *Y*), four branches (*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 the seven paths achieves full statement coverage. If only two paths are feasible, some boolean subexpressions are unreachable. To improve the quality of testing, branch coverage requires executing four paths. Examining the remaining three paths might reveal unexpected interactions among the predicates, or might be redundant tests that do nothing to improve software quality.

Another approach to path coverage is to focus on paths likely to reveal a fault. For example, consider a path that contains an assignment to variable *x*, but no use of the variable (i.e., *x* is dead in the path). Such a path is less likely to reveal a fault in the assignment than a path that contains the assignment followed by a use of *x*. This observation has motivated a large family of data flow, path-based coverage criteria [19].

6.2 Automated Test Generation

With debugging, a crucial question was, “which path *did* program execution follow to this point?” An analogous question for testing is, “which path *can* program execution follow to this point?” In other words, in order to test a particular piece of code, a tester must find some program input that causes the execution of the code. One approach is to find a path to the code and then determine an input to the program that causes this path to execute. This simple formulation conceals a host of nasty problems. First, the chosen path may be infeasible, so no input will cause the path to execute, no matter how hard a tester, or tool, tries. Second, determining an input that causes a program to execute a given path is an undecidable problem. Despite (or perhaps because of) these considerable difficulties, considerable research has explored the area of automated test generation.

Consider the following example, which counts how many of three variables have positive values and 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. Clearly, only one such path exists—although automatically determining this is not so straightforward—and the precondition for its execution is that the three input variables all have positive value. In the counting example, each of the first three predicates tests an independent variable, so each predicate (branch) is independent of the others. This greatly simplifies the job of an automated testing tool. However, in general, 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, may be necessary to make the inferences [20].

7 Paths, Software Complexity and Program Understanding

Program paths also offer a new view on the complexity of software. Nearly all software complexity metrics rely on counting entities in a program. For example, the popular McCabe cyclomatic complexity measure counts the number of decision points (branches or predicates). However, as the two control flow graphs in Figure 9 show, structural relationships among predicates greatly influence the number of paths. Figure 9(a) contains three predicates (*A*, *B*, and *C*) and 8 paths. Figure 9(b) also contains three predicates, but only four paths. The McCabe

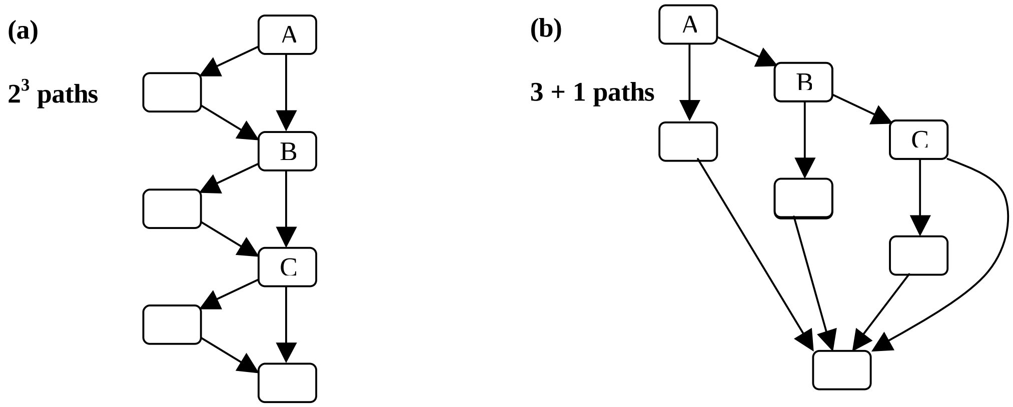


Figure 9. Two procedures with the same number of predicates and branches, but very different numbers of paths.

complexity of both procedures is five, but the complexity of understanding and testing the two procedures differ greatly. In general, given N binary predicates, procedure can have anywhere from N+1 paths, when the predicates are nested N level deep (as in Figure 9(b)), to 2^N paths, when the predicates are strung out in sequence (as in Figure 9(a)).

Each path in a procedure represents a potential execution scenario that a programmer may have to consider when understanding and 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 from Section 6.2. The code contains 16 potential paths. The first three predicates are independent of each another (as each refers to a different variable), so there are eight feasible paths through the first three statements. The predicate in the last statement (which checks if the count is 3) is dependent on the first three predicates, since if any of them evaluates false, *count* will not equal three and the final predicate will evaluate false. This means that there are eight feasible paths through the four statements. Despite the path complexity of the code, it is relatively simple to understand, because of the independence of the first three branches.

Now, consider the following code, which manipulates a tree data structure:

```
y = (!x->left || !x->right) ? x : x->right;
z = (!y->left) ? y->right : y->left;
if (x != y) x->key = y->key;
```

How many feasible paths does this code contain? There are three paths through the first statement, two through the second, and two through the third, for a total of 12 paths. Only four of the paths are feasible. Note that the first statement aliases *y* to *x* if either the left or the right child of *x* is nil. In this case, the value of the predicate in the second statement is dependent on the predicate in the first statement. That is, if *x->left* is nil then *z* will be assigned *x->right*. Otherwise, if *x->right* is nil then *z* will be assigned *x->left*. As a result, there are only four, not six, feasible paths through the first two statements. In two of the four paths, *y* is aliased to *x*, so the last predicate always will evaluate false. In the other two paths, *y* is assigned the right child of *x*, so *y* and *x* cannot be equal, and the last predicate will always evaluate true. Thus, there are only four feasible paths through the entire code.

The ratio of four feasible paths to twelve totals paths is striking. Something seems wrong. The following restructured code captures exactly the same semantics, more efficiently and concisely. The following code has four paths, all feasible:

```
y = x;
if (!x->right)
   z = x->left;
else if (!x->left)
   z = x->right;
else {
   y = x->right;
   z = (!y->left) ? y->right : y->left;
   x->key = y->key;
}
```

Comparing the structure of this code to the original, we see that we have eliminated much of the unnecessary sequencing in the original code, which led to the creation of a large number of infeasible paths.

8 Conclusion

Program paths offer new insight into a program’s dynamic behavior. The control locality exhibited by paths underlies many high performance computer hardware and compiler optimizations. Paths also make clear the complexity of writing, understanding and testing computer programs. In both cases, paths provide a long-missing perspective, which moves beyond point-by-point analysis of profiling tools into a dynamic realm in which operations occur in order. However, paths do not completely represent all aspects of program’s dynamic behavior. For example, the acyclic paths discussed in this paper lose context at loop and procedure boundaries. Moreover, paths say nothing about the other half of a program’s dynamic behavior, its data references. Both deficiencies are likely to be addressed by future work.

Acknowledgements

Daniel Weise and David Tarditi provided many helpful comments on this paper.

```
int AddEvenNumbers(int limit)
{
  int sum = 0;
  for (j = 1; j <= limit; j += 1) {
    if ((j % 2) == 0) {
      sum += j
    }
  }
  return sum;
}
```

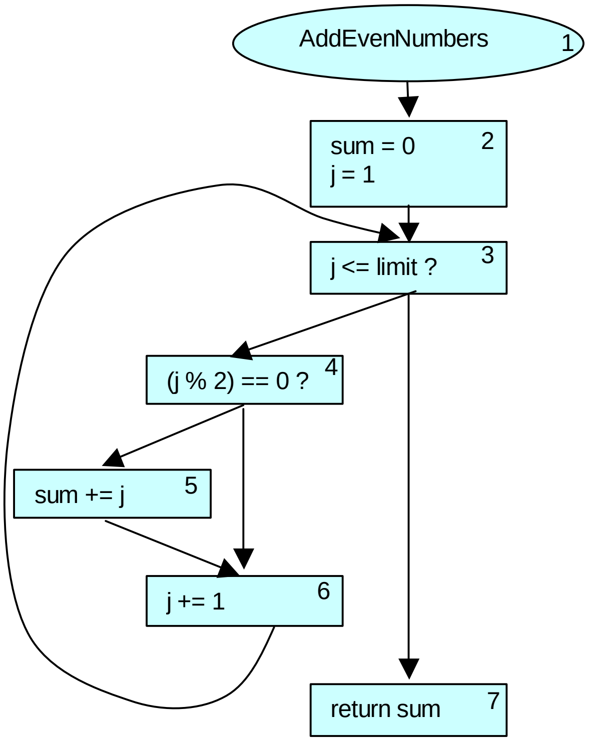


Figure 10. The AddEvenNumbers function and its control flow graph. Each node in the graph is called a basic block and contains straight-line code. Edges in the graph represent transfers of control between blocks.

Sidebar 1: Program Paths

A program path is a consecutively executed sequence of branches and control transfers through a program's instructions. Compilers translate procedures in a program into an internal representation called a *control flow graph (CFG)* [12]. Nodes in a CFG are called *basic blocks*. They contain straight-line code that only transfers control to the next instruction. Edges between the basic blocks represent control transfers, so that a block that ends with a jump will have an outgoing edge leading to the block containing the instruction at the target of the jump. Figure 10 shows a function `AddEvenNumbers` and its control flow graph.

A path in a CFG is the sequence of edges that a program traverses when it executes. Alternatively, if each pair of blocks is connected by at most one edge, a path can also be described as the sequence of blocks executed by a program. For example, if we call `AddEvenNumbers(2)`, the function will executed blocks: 1, 2, 3, 4, 6, 3, 4, 5, 6, 3, 7.

Not all paths that appear in a program's control flow graph can be executed. *Infeasible* paths arise for many reasons, one of which is logical contradictions in the predicates along the path. For example, in:

```
if (x < 10) { A; }
if (x > 100) { B; }
```

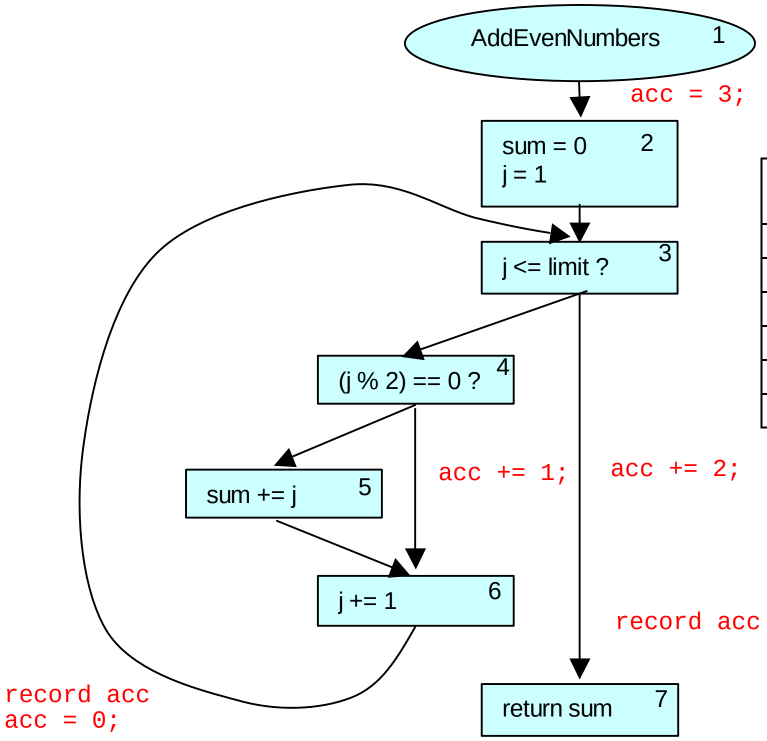
The path containing *A* and *B* is infeasible (assuming that the code in *A* does not modify the variable *x*). Unfortunately, not all examples of infeasibility are as easy to identify as this one. In general, determining that a path is infeasible is uncomputable, as it is equivalent to solving the Turing halting problem.

Sidebar 2: Efficient Path Profiling

Ball and Larus developed a simple method to record a significant portion of the path executed by a program [2]. Their technique records path spectra consisting of intraprocedural, acyclic paths. A path is *intraprocedural* if it is contained entirely within one procedure. When a call instruction occurs along a path, it is treated as if it did not transfer control. 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 recursion). They cause problems because unbounded iteration make the set of potential paths unbounded, as each loop iteration introduces a new path.

Unbounded sets are difficult to represent and manipulate, so Ball and Larus focused on acyclic paths that do not cross a loop back edge. These acyclic paths fall into four categories:

1. A path from the procedure entry to the procedure's exit.
2. A path from the procedure entry to a loop's back edge.
3. A path from the head of a loop to a loop's back edge.
4. A path from the head of a loop to the procedure's exit.



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 11. Path profiling code for AddEvenNumbers function. The code along edges computes unique numbers for each acylic path in the function.

The Ball-Larus profiling technique adds code along some edges in a procedure's CFG. This code adds specially selected values into an accumulator. When control reaches either a loop back edge or the 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 11 contains the annotated CFG and shows the path numbers computed by the instrumentation code. The overhead cost of this form of profiling can be very low, as most of the instrumentation simply increments a counter by a constant value. Most of the expensive of the profiling comes from recording the executed paths. More detailed performance information can be associated with specific paths by using a processor's hardware metric counters to determine the number of events (e.g. cache misses, instruction stalls, etc.) that occurred in the instructions between the beginning and end of each path.

Heavily skewed distributions also occur for hardware metrics other than a simple count of executed instructions. Figure 12 shows the cumulative distribution along paths of 16 hardware metrics in a Sun UltraSPARC processor running the *gcc* benchmark. Many metrics closely track the executed instruction curve (Instr_cnt). The other metrics are relatively infrequent events in this program, such as store buffer, floating-point, or load-use stalls. Figure 13 focuses on one important metric, Level-1 data cache misses. It shows that a small

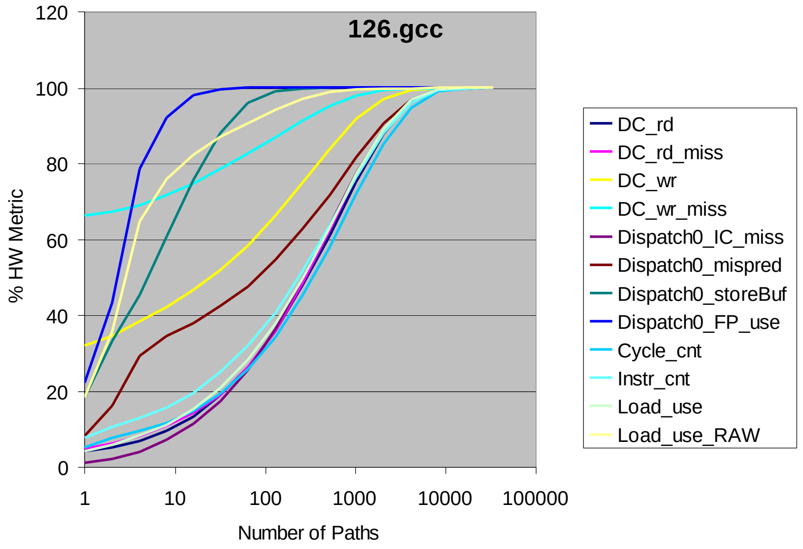


Figure 12. Cumulative distribution of hardware costs along paths in gcc SPEC95 benchmark.

number of hot paths—in this case, paths that contribute 1% or more of cache misses—account for a vast majority of L1 data cache misses. Again, the *gcc* and *go* benchmarks stand out. However, since these programs execute more paths, each hot path contributes less, so it is necessary to redefine a hot path to contribute 0.1% of the cache misses. With this change, these programs exhibit similar behavior, except that the absolute number of hot paths is roughly an order of magnitude higher than the simpler programs.

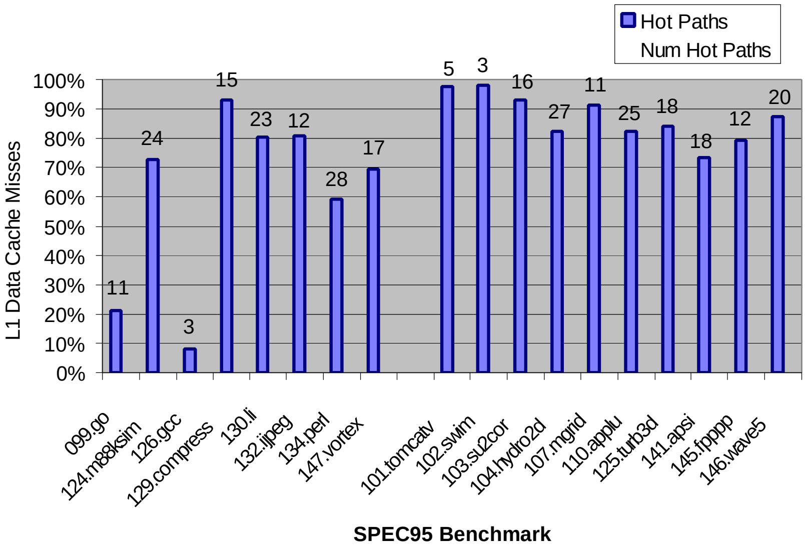


Figure 13. L1 Data cache misses in SPEC95 benchmarks along the hot paths that contributed 1% or more misses.

9 References

[1] G. Ammons, T. Ball, and J. R. Larus, "Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling," in *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, 1997, pp. 85-96.

[2] T. Ball and J. R. Larus, "Efficient Path Profiling," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, Paris, France, 1996, pp. 46-57.

[3] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, 2 ed: Morgan Kaufmann, 1996.

[4] J. E. Smith, "A Study of Branch Prediction Strategies," in *Proceedings of the Eighth Annual International Symposium on Computer Architecture*. Minneapolis, MN, 1981, pp. 135-148.

[5] S.-T. Pan, K. So, and J. T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. Boston, Massachusetts, 1992, pp. 76-84.

[6] T.-Y. Yeh and Y. Patt, "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History," in *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993, pp. 257-265.

[7] C. Young, N. Gloy, and M. D. Smith, "A Comparative Analysis of Schemes for Correlated Branch Prediction," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 276-286.

[8] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*. Paris, France, 1996, pp. 24-34.

[9] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*. Cambridge, MA: MIT Press, 1986.

[10] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, vol. C-30, pp. 478-490, 1981.

[11] F. Mueller and D. B. Whalley, "Avoiding Unconditional Jumps by Code Replication," in *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, 1992, pp. 322-330.

[12] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*: Addison-Wesley, 1985.

[13] G. Ammons and J. R. Larus, "Improving Data-flow Analysis with Path Profiles," in *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Las Vegas, NV, 1998, pp. 72-84.

[14] R. Gupta, D. A. Berson, and J. Z. Fang, "Path Profile Guided Partial Dead Code Elimination Using Predication," in *Proceedings of International Conference on Parallel Architecture and Compilation Techniques (PACT)*. San Francisco, CA, 1997.

[15] E. Shapiro, *Algorithmic Program Debugging*: MIT Press, Cambridge, MA, 1982.

[16] A. Tolmach and A. W. Appel, "A Debugger for Standard ML," *Journal of Functional Programming*, vol. 5, pp. 155-200, 1995.

[17] B. Bruegge and P. Hibbard, "Generalized Path Expressions: A High-level Debugging Mechanism," *The Journal of Systems and Software*, vol. 3, pp. 265-276, 1983.

[18] T. Reps, T. Ball, M. Das, and J. R. Larus, "The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem," in *Proceedings of the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, M. Jazayeri and H. Schauer, Eds.: Springer-Verlag, 1997, pp. 432-449.

[19] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, "A Formal Evaluation of Data Flow Path Selection Criteria," *IEEE Transactions on Software Engineering*, vol. 15, pp. 1318-1332, 1989.

[20] L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Transactions on Software Engineering*, vol. 2, pp. 215-222, 1976.