The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem*

Thomas Reps,[†] *Thomas Ball,*[‡] *Manuvir Das,*[†] *and James Larus*[†]

Abstract

This paper describes new techniques to help with testing and debugging, using information obtained from path profiling. A path profiler instruments a program so that the number of times each different loop-free path executes is accumulated during an execution run. With such an instrumented program, each run of the program generates a path spectrum for the execution—a distribution of the paths that were executed during that run. A path spectrum is a finite, easily obtainable characterization of a program's execution on a dataset, and provides a behavior signature for a run of the program.

Our techniques are based on the idea of comparing path spectra from different runs of the program. When different runs produce different spectra, the spectral differences can be used to identify paths in the program along which control diverges in the two runs. By choosing input datasets to hold all factors constant except one, the divergence can be attributed to this factor. The point of divergence itself may not be the cause of the underlying problem, but provides a starting place for a programmer to begin his exploration.

One application of this technique is in the "Year 2000 Problem" (*i.e.*, the problem of fixing computer systems that use only 2-digit year fields in date-valued data). In this context, path-spectrum comparison provides a heuristic for identifying paths in a program that are good candidates for being date-dependent computations. The application of path-spectrum comparison to a number of other software-maintenance issues is also discussed.

---

*This work was supported in part by the National Science Foundation under grants CCR-9625667, MIP-9625558, and NYI Award CCR-9357779 (with support from Hewlett Packard and Sun Microsystems), and by the Defense Advanced Research Projects Agency (monitored by the Office of Naval Research under contracts N00014-92-J-1937 and N00014-97-1-0114, and by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525).

[†]Computer Sciences Department, University of Wisconsin–Madison, 1210 W. Dayton St., Madison, WI 53706.
E-mail: {reps, manuvir, larus}@cs.wisc.edu.

[‡]Lucent Technologies, 1000 E. Warrenville Road, P.O. Box 3013, Naperville, IL 60566-7013.
E-mail: tball@research.bell-labs.com.

1. Introduction

The world faces cataclysmic breakdown at the turn of the millennium!

While this alarm may be old news to anyone who was present at the turn of the last millennium, there are significant reasons for residents of the (first) world to be concerned this time around: Because many computer programs use only two digits to record year values in date-valued data, they may process a year value of 00 as 1900 in cases where 2000 was intended. If the intended value is 2000—such as when 00 represents the value of the current year in a computation performed after the calendar rolls over on January 1, 2000—then a faulty computation may be carried out. Because computations can involve dates in the future, the phenomenon can occur well before the calendar rolls over on January 1, 2000. For example, if the (approximate) age of someone born in 1956 were calculated for January 1, 2000, he would appear to be $00 - 56 = -56$ years old! If the program tries to use the value -56 to index into a life-expectancy table, the program will either fetch a bogus life-expectancy value or quit with an error (depending on whether the run-time system catches "index-out-of-bounds" errors). In both cases, the system functions improperly. In general, such behavior can have serious—even life-threatening—consequences. This problem and a variety of other date-related problems that will show up with increasing frequency around January 1, 2000 are known collectively as the "Year 2000 Problem" (Y2K problem).

This past summer, the first author was asked by the Defense Advanced Research Projects Agency (DARPA) to help them plan a project aimed at reducing the impact of the Y2K problem on the Department of Defense. DARPA was particularly interested in whether there were "any techniques in the research community that could be applied to the Y2K problem and have impact beyond present commercial Y2K products and services".

The most exciting of the ideas that turned up concerns a method for using path profiling as a heuristic to locate some of the sites in a program where there are problematic date manipulations. It works as follows:

> In path profiling, a program is instrumented so that the number of times each different loop-free path executes is accumulated during an execution run. With such an instrumented program, each run (or set of runs) of the program generates a path spectrum for the execution—a distribution of the paths that were executed. Path spectra can be used to identify paths in a program that are good candidates for being date-dependent computations by finding differences between path spectra from execution runs on pre-2000 data and post-2000 data. By choosing input datasets to hold all factors constant except the way dates are used in the program, any differences in the spectra obtained from different execution runs can be attributed to date-dependent computations in the program. Differences in the spectra reveal paths along which the program performed a new sort of computation during the post-2000 run, as well as paths—and hence computations—that were no longer executed during the post-2000 run.

With some further analysis of the spectra, for each such path that shows up in the spectral difference, it is possible to identify the shortest prefix that distinguishes it from all of the paths in the other path set.

Of course, the path-spectrum-comparison technique is not guaranteed to uncover all sites of date manipulations. No technique can do this; all one can hope for are good heuristics. However, because path-spectrum comparison involves a different principle from the principles that lie behind the heuristics used in commercial Y2K tools, it should be a good complement to current techniques.

The path-spectrum-comparison technique is actually applicable to a much wider range of software-maintenance problems than just the Y2K problem. In particular, the problem of how to carry out adequate execution tests is a huge problem for software developers, and will still be with us long past the year 2000. As discussed in Section 6, the path-spectrum-comparison technique offers new perspectives on testing, on the task of creating test data, and on what tools can be created to support program testing.

Note that the idea of comparing path spectra to identify possible execution errors is a completely different use of path profiling in program testing from another use that has been proposed for path profiles in program testing, namely as a criterion for evaluating the coverage of a test suite [22,14,8,16].

The remainder of the paper is organized into seven sections: Section 2 provides background on the Y2K problem. Section 3 describes the use of path profiling to locate date-dependent paths and their shortest distinguishing prefixes. Section 4 summarizes the key insights behind recent work that makes it possible to carry out path profiling in an efficient manner, as well as an alternative technique for locating shortest distinguishing prefixes of path-spectrum differences. Section 5 describes our implementation of a tool based on these ideas, as well as the results of our preliminary experience with the tool. Section 6 discusses other applications of the technique to a broader range of software-maintenance problems. Section 7 discusses related work. Section 8 presents a few final remarks.

2. The Year 2000 Problem

In addition to the rollover problem with two-digit year fields, the phrase "Year 2000 Problem" has come to mean a whole host of date-related problems that will eventually crop up, many of which strike around the turn of the millennium. For example, leap years come every four years, except for centuries, except for centuries divisible by 400. Thus, the year 2000 is, in fact, a leap year. However, some programs implement the exception, but not the exception to the exception. Such a bug could cause havoc in financial transactions (*e.g.*, by causing failures in computer-driven trading) and military maneuvers (*e.g.*, by causing logistical planning failures). UNIX systems are also subject to date-representation rollover problems, most of which occur later in the 21st century.[1]

For both date-representation rollover problems and leap-year bugs, it is necessary to find the code that declares and manipulates date-valued variables, rewrite it, and test the modifications. Unfortunately, dates are hidden in programs. "Date" is not a data-type in most programming languages, and so heuristics must be developed for identifying the locations where date-valued data is manipulated. Even when a language does have a "date" data-type, there is nothing to forbid programmers from creating or encoding "raw" dates that are embedded in data of other data types, such as character strings.

Much of the problem is in administrative computing: purchasing and billing records, maintenance and inventory records, payrolls, and the like. However, all of the world-wide infrastructure that incorporates automated components could conceivably be affected, including telephone and electrical power systems, industrial plants, nuclear power plants, defense early-warning systems, logistics and planning systems, and

[1]Overflow in the ANSI C library occurs on 1/1/2010; in the UNIX *time* function on 9/30/2034; and in UNIX systems on 1/19/2038 [2].

weapons systems. Cost estimates for correcting the various date problems run as high as $600 billion world-wide [9], $300 billion in the U.S., $30 billion for the Federal government, and $10 billion for the Department of Defense—not to mention an estimated $1 trillion in legal fees in the aftermath.

The Y2K problem is in large part a management problem: There are enormous difficulties that must be addressed by any organization that faces the Y2K problem, including battling for adequate resources (*e.g.*, financial, equipment, and staffing), inventorying an organization's custom programs and COTS ("commercial off-the-shelf") programs, and coordinating the deployment of "renovated" systems (which may have to interoperate with systems, including those of other companies, that have not yet been renovated). However, there are serious technical problems as well, including program-analysis methods for determining the sites at which date-manipulation code occurs, code- and data-transformation algorithms, post-renovation testing, and the technical challenges of coping with interoperating renovated and unrenovated systems.

The techniques described in this paper are relevant to two of these problems: (i) determining the sites at which date-manipulation code occurs, and (ii) post-renovation testing.

Because the leverage that tools for the Y2K problem can provide is limited by their accuracy for locating the places in a piece of code where dates are employed, the date-location issue is crucial to the creation of effective tools for correcting date-manipulation problems. Two techniques for locating dates are used in present commercial products:

(1) Some "date seeds" can be identified by the places where a program makes certain calls to the operating system, for example, to retrieve the current date. This method is accurate, but does not identify all the date-manipulation sites in the program. For instance, the variable into which the current date has been placed can be manipulated elsewhere in the program, or its contents can be assigned to another variable. In addition, other date values can be read in from files, from across the network, or from interactive user input.

(2) Some date-manipulation sites can be identified by exploiting any conventions that programmers may have used for naming the variables in the program. Automatic string-searching tools are used to search the source code—or alternatively, just the identifiers in a tokenized version of the source code—with respect to patterns that reflect such conventions, for example, "*date*", "*gmt*", "*yy*", etc. (where "*" is a wild-card symbol that means "match any substring").

After these techniques have been used to identify candidate sites at which dates are manipulated, this information can be "amplified", via searching and slicing [21,13,10,15] operations, to find other potential locations of problems.

3. Path Profiling and the Year 2000 Problem

In path profiling, a program is instrumented so that the number of times different paths of the program execute is accumulated during an execution run. Typically, the paths of interest are loop-free intraprocedural paths. The distribution of paths from an execution of the program is called a *path profile* or a *path spectrum*. We are sometimes just interested in Boolean information (which paths were executed? which were not?), but other times we are interested in the frequencies with which paths were executed. This corresponds to considering a path spectrum as either a set of paths or a multi-set of paths, respectively.

The observation underlying our technique for applying program profiling to the Y2K problem is that differences between path spectra obtained from different runs of a program can be used to identify paths that are good candidates for being date-dependent computations. By choosing input datasets to hold all factors constant except the way dates are used in the program, any differences in the path spectra from different execution runs can be attributed to date-dependent computations in the program. In particular, one would obtain path spectra from execution runs of the program in which the program is run on pre-2000 data and post-2000 data (or data that is likely to bring to light whatever "date vulnerability" we are trying to test). By comparing the two path spectra, paths along which the program performed a new sort of computation during the post-2000 run can be identified, as well as paths—and hence computations—that were no longer executed during the post-2000 run.

Our thesis is that this technique provides a good heuristic for identifying date-dependent computations. The basis for this belief is that a path spectrum provides an approximate characterization of the program's behavior, in the following sense:

> The program's execution paths serve as representatives for a set of execution states: Consider the set of all possible execution states of the form (pt, σ), where σ is a store value and pt is not an arbitrary program point, but one occurring at the beginning of a path p that the profiler is prepared to tabulate. In terms of characterizing the program's execution behavior, two execution states (pt, σ_1) and (pt, σ_2) are "similar" if they both cause the program to proceed from pt along execution path p. Path p serves as a representative of this equivalence class of similar execution states.

Differences in the path spectra obtained during two runs of a program on different inputs indicate differences in the (equivalence classes of) execution states encountered, and hence are a reflection of differences in the program's behavior due to the differences in the input. In the case of runs using pre- and post-2000 data, differences in the path spectra must therefore reflect changed behavior due to date-dependent computations.

Of course, this only holds in one direction: Not all differences in behavior due to date-dependent computations will necessarily show up as differences in the (equivalence classes of) execution states encountered.

Example. Consider the program fragment shown in Figure 1, which reads and processes data from a database of customer information. (This fragment does not contain any cycles, but might appear as part of a loop in a larger program. Path profiling in programs with loops is typically carried out by considering loop-free segments of the program. See Section 4.1 or reference [5] for more discussion of this issue.)

For purposes of this example, assume that years are represented with only two digits and that no person recorded in the database who is younger than fifteen years old possesses a college degree. Because of the latter assumption, no path from a pre-2000 run can begin with the prefix $[a, b, d]$.

Now consider a post-2000 run (*e.g.*, a simulated post-2000 run in which the system clock has been set ahead so that *current_year*() returns a value representing a year in the future, say 00, representing the year 2000), and suppose that the program reads in data about someone born in 1956 who possesses a college degree: The initialization code in region a would set *age* to $00 - 56 = -56$; because the test $-56 < 15$ evaluates to true, region b would be executed; because the person possesses a college degree, region d would be executed; finally, either region f or g would be executed. In either case, the program performs a faulty

```
a: birth_year := read()
   has_college_degree := read()
   purchases := read()
   age := current_year() − birth_year
if age < 15 then
   b: . . .
else
   c: . . .
fi
if has_college_degree = true then
   d: . . .
else
   e: . . .
fi
if purchases > 3 then
   f: . . .
else
   g: . . .
fi
```

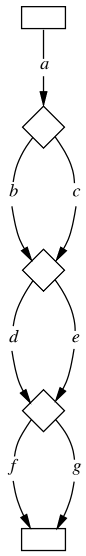


Figure 1. A program fragment that reads and processes data from a database of customer information, and its control-flow graph.

computation: The path executed is a path that should only be executed when a record is encountered for a person younger than fifteen who possesses a college degree. Because no such paths are ever executed during the pre-2000 run, the path-spectrum-comparison technique would detect the fact that the program performed a new sort of computation during the post-2000 run.

In addition, other anomalies may be detected: The pre-2000 run could very well execute paths with the prefix $[a, c]$. Because in the post-2000 run the value of *age* is always negative, the post-2000 run would never execute such paths.

The following table shows path spectra that might be accumulated during pre-2000 and post-2000 execution runs (assuming that the fragment occurs in a loop, so that it is executed multiple times):

Run	Paths Executed							
	$[a, b, d, f]$	$[a, b, d, g]$	$[a, b, e, f]$	$[a, b, e, g]$	$[a, c, d, f]$	$[a, c, d, g]$	$[a, c, e, f]$	$[a, c, e, g]$
pre-2000			•	•	•	•	•	•
post-2000	•	•	•	•				

These spectra show clearly that the pre-2000 and post-2000 behavior of the program is not the same: Paths $[a, b, d, f]$ and $[a, b, d, g]$ occur in the post-2000 run, but do not occur in the pre-2000 run; paths $[a, c, d, f]$, $[a, c, d, g]$, $[a, c, e, f]$, and $[a, c, e, g]$ occur in the pre-2000 run, but do not occur in the post-2000 run. □

Each path in a path spectrum represents a sequence of edges in the program's control-flow graph. From two path spectra, *new_spectrum* and *old_spectrum*, the path-spectrum-comparison technique reveals paths of *new_spectrum* that are not found in *old_spectrum*, and vice versa. Given a path of *new_spectrum* (resp., *old_spectrum*) that does not occur in *old_spectrum* (*new_spectrum*), we can determine the shortest prefix of the path that distinguishes it from all of the paths in *old_spectrum* (*new_spectrum*). For the Y2K problem, such path prefixes furnish a programmer with even more precise information about what contributes to the differences in behavior between the pre-2000 and post-2000 runs.

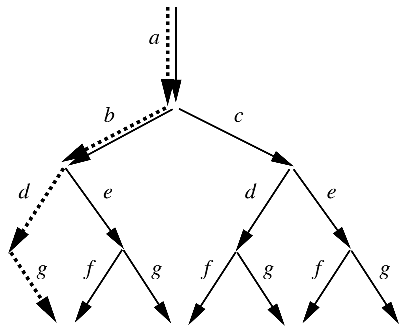
- Let p be an execution path that was executed during the post-2000 run but not during the pre-2000 run. By finding the shortest prefix of p that is not a prefix of any path executed during the pre-2000 run, we identify the critical portion of p that represents a new sort of computation (or state-transformation pattern) performed during the post-2000 run. The programmer can focus on this prefix of p to locate the date-dependent code, which very likely needs to be rewritten.
- Similarly, let q be an execution path that was executed during the pre-2000 run but not during the post-2000 run. The shortest prefix of q that is not a prefix of any path executed during the post-2000 run identifies the critical portion of q that represents a computation (state-transformation pattern) no longer performed during the post-2000 run. Again, the programmer can focus on this prefix of q to locate the date-dependent code.

Example. In the example program discussed earlier, paths $[a, b, d, f]$ and $[a, b, d, g]$ of the post-2000 run do not occur in the pre-2000 run. For both paths, the shortest prefix that is not a prefix of any path executed during the pre-2000 run is $[a, b, d]$. In asking the question "Why is the path $[a, b, d]$ executed during the post-2000 run?", the programmer would be led to ask the question "How can it be that *age* is less than 15 and *has_college_degree* is true?", which would in turn lead him to the statement that computes *age* as a function of *current_year*().

Conversely, paths $[a, c, d, f]$, $[a, c, d, g]$, $[a, c, e, f]$, and $[a, c, e, g]$ of the pre-2000 run do not occur in the post-2000 run. For all of these paths, the shortest prefix that is not a prefix of any path executed during the post-2000 run is $[a, c]$. In this case, the programmer would be led to ask the question "Why is the path $[a, c]$ never executed during the post-2000 run? That is, why is the value of *age* always less than 15 during the post-2000 run?" Again, the programmer is led to the statement that computes *age* as a function of *current_year*(). □

One can find the shortest prefix of a path p that is not a prefix of any executed path in a spectrum S using a trie structure on S [17]: The first edge of p that "deviates from the trie" identifies the edge at which p veers into "unknown territory", and the prefix of p, up to and including this edge, is the shortest prefix of p that distinguishes p from S.

Example. The solid arrows in the diagram below show the trie for the pre-2000 spectrum.



The dotted edges show path $[a, b, d, g]$ (which occurs during the post-2000 run). The shortest prefix of $[a, b, d, g]$ that is not a prefix of any path executed during the pre-2000 run is $[a, b, d]$. □

3.1. Thresholding

Rather than concentrating on paths p that are executed in *new_spectrum* but not in *old_spectrum* (or vice versa), we may wish to gather information from a path p' that is executed a different number of times in the two spectra. Usually, we would be interested in a path p' that is executed frequently in *new_spectrum* but not in *old_spectrum*, or vice versa. Perhaps some threshold ratio, say 100 to 1, would be used to identify "interesting paths". For instance, in the example from Section 3, suppose the database did contain a few records for people younger than fifteen years old in possession of a college degree. In this case, the differences between the pre- and post-2000 runs would show up as the post-2000 run appearing to process a large multiple of the number of such records processed by the pre-2000 run.

In this situation, we would again be interested in understanding which prefix distinguishes path p' from the paths in *old_spectrum*. To do this, we merely remove p' (temporarily) from the *old_spectrum* path set, and then perform the normal path-comparison operation on p' with respect to *old_spectrum* (*e.g.*, via a trie on *old_spectrum* or by the alternative technique described in Section 4.2).

It is important that the over-threshold paths be removed from *old_spectrum* only one at a time. The reason is that the over-threshold paths in *old_spectrum* may share prefixes in common. If all of the over-threshold paths were removed from *old_spectrum* simultaneously, and the path comparison carried out against the resulting spectrum, an incorrect set of shortest distinguishing prefixes could be reported.

3.2. Other Uses of Path Profiling for the Year 2000 Problem

In addition to its utility for "date prospecting" in the Y2K problem, the path-spectrum-comparison technique also has the potential to help out with two other important issues that are part of the Y2K problem: (i) determining whether COTS components (*i.e.*, libraries) or COTS tools have date problems, and (ii) testing renovated code:

(i) COTS software is usually distributed without source code, as an object-code file or as an executable file. (Executables are usually distributed without symbol-table or relocation information, as well.) Because it is possible to perform the instrumentation necessary for obtaining path spectra on object-code files and executable files [11,20,12], the path-spectrum-comparison technique is one of the few methods we are aware of that can be used to identify date-manipulation problems in programs for which source code is not available: Differences between pre-2000 and post-2000 spectra are an indication that a piece of COTS software may have a Y2K problem.

Of course, in this scenario the lack of access to the source code prevents one from actually fixing the Y2K problem. However, the manufacturer can presumably make use of the information that the path-spectrum-comparison technique brings to light about suspicious paths through the object code.

(ii) A correctly renovated system should have similar path spectra from execution runs on pre-2000 data and post-2000 data. Remaining path-spectrum differences could indicate that a Y2K problem still exists in the renovated system.

3.3. Prioritization of Spectral Differences

Not all path-spectrum differences necessarily deserve equal consideration by the user. For this reason, it is useful to augment the path-spectrum-comparison technique with a prioritization method for establishing an order in which the spectral differences should be brought to the attention of the user. One method is to rank them by the order in which paths were executed (in one of the two execution runs). For instance, suppose that p is a path in *new_spectrum* that does not occur in *old_spectrum*. Relative to all of the other paths in this category, p's rank would be established according in the order in which an instance of p was executed for the first time. The reasoning behind this heuristic is that the early instances of behavioral differences between the two runs may be more likely to point to the cause of the underlying problem.

To track the order in which paths are first executed, the instrumented code could use a global counter: Each time the end of a never-before-executed path is encountered (*i.e.*, each time a path count is set from 0 to 1), the counter's value would be recorded with the path, and the counter incremented.

3.4. Why Not Node Profiling or Edge Profiling?

A comparison process similar to that described above could be carried out using pairs of spectra created using node profiling or edge profiling. However, in general, these variations on the idea are not likely to produce as good results as when spectra from path profiles are used.

By considering what happens during different post-2000 execution runs, the example from Figure 1 can be used to illustrate that path-spectrum comparison is able to distinguish more behavioral differences than either node-spectrum comparison or edge-spectrum comparison. First, note that a pre-2000 run can exercise all edges of the example program's control-flow graph (*i.e.*, regions a, b, c, d, e, f, and g). This is not the case for some post-2000 runs. For instance, for runs during which the system clock is set so that *current_year*() returns a value in the range 00 to 14 (representing a year in the range 2000 to 2014), the value of *age* will always be less than 15, and thus region c will never be executed. For these runs, node-spectrum comparison and edge-spectrum comparison would both detect a behavioral difference between the pre- and post-2000 runs (as would path-spectrum comparison).

In contrast, if the system clock is set so that *current_year*() returns a value greater than or equal to 15 (representing 2015 or later), we again have a situation in which all nodes and edges are able to be executed. In particular, when a record for a person born in the year 2000 is processed during a year-2015 run, the initialization code in region a will set *age* to $15 - 00 = 15$, the test *age* < 15 will evaluate to false, and region c will be executed. Thus, pre-2000 and post-2015 runs can exercise all edges of the example program's control-flow graph, and hence neither node-spectrum comparison nor edge-spectrum comparison would detect any differences in behavior between these runs. However, as in the 2000 to 2014 runs, if the program reads in data about someone born in 1956 who possesses a college degree, the program will follow a path that should only be executed when a record is encountered for a person younger than fifteen who possesses a college degree: The initialization code in region *a* would set *age* to $15 - 56 = -41$; because the test $-41 < 15$ evaluates to true, region *b* would be executed; because the person possesses a college degree, region *d* would be executed; finally, either region *f* or *g* would be executed. Because no such paths are ever executed during the pre-2000 run, the path-spectrum-comparison technique *would* detect the fact that the program performed a new sort of computation during the post-2015 run. This example shows that, in general, path-spectrum comparison is able to distinguish more behavioral differences than either node-spectrum comparison or edge-spectrum comparison.

What is the significance of this for the Y2K problem, in general? For node-spectrum comparison and edge-spectrum comparison to detect behavioral differences between execution runs on pre-2000 data and post-2000 data, the post-2000 run either has to exercise a completely new part of the program, or completely fail to exercise some part of the program that was exercised during the pre-2000 run. In contrast, with the path-spectrum-comparison technique, it is possible to detect behavioral differences even if exactly the same nodes and edges are exercised during the two runs (as long as different paths are exercised). Execution of the same nodes and edges can give rise to different sets of paths if the *correlations between branches* are different in the different runs. Consequently, of the three techniques, the path-spectrum-comparison technique provides the highest-fidelity test for identifying date-dependent computations.[2]

4. Efficient Path Profiling

The path-spectrum-comparison technique is not tied to any particular path-profiling method. Furthermore, there are a wide variety of options in how one performs the instrumentation required to gather information about what paths execute. Instrumentation can be performed at any one of a number of levels:

- At the source-code level, as a source-to-source transformation.
- As part of compilation, by extending a compiler to use its intermediate representations for the purpose of determining where to introduce instrumentation instructions.
- As an object-code-level transformation, by modifying object-code files (such as UNIX ".o" files).
- As a post-loader transformation, by modifying executable files (such as UNIX "a.out" files) [11,20,12].

One could even use different instrumentation methods on different parts of the system.

Although any method for generating path profiles could be used, it is only recently that methods have been devised for obtaining path profiles with acceptable overheads [5,3]. In particular, Ball and Larus

[2]Path-spectrum comparison subsumes node-spectrum comparison and edge-spectrum comparison in the sense that all behavioral differences identified by node-spectrum comparison and edge-spectrum comparison will also be identified by path-spectrum comparison, but not vice versa. Path-spectrum comparison subsumes node-spectrum comparison and edge-spectrum comparison in a second sense, as well. Node profiling and edge profiling can be considered to be degenerate cases of path profiling: edge profiling is the case where the paths tabulated are all of length 1; node profiling is the case where the paths tabulated are all of length 0.

report that execution-time overheads on the order of only 30–40% can be achieved with their method for collecting path profiles [5]. Their work relies on a particular method for numbering the paths in the program, the main points of which are described in Section 4.1. When the paths in path profiles are reported using this numbering scheme, an alternative technique for interpreting path spectra can be used to identify the shortest prefix of a path in *new_spectrum* that is not a prefix of any executed path in *old_spectrum*. This is described in Section 4.2.

4.1. The Ball-Larus Scheme for Numbering Paths

The Ball-Larus path-numbering scheme applies to an acyclic control-flow graph with a unique source node *Start* and a sink node *Exit*. Control-flow graphs that contain cycles are modified by a preprocessing step to turn them into acyclic graphs:

> Every cycle must contain one backedge, which can be identified using depth-first search. For each backedge $w \to v$, add edges $Start \to v$ and $w \to Exit$ to the graph. Then remove all of the backedges from the graph.

The resulting graph is acyclic. In terms of the ultimate effect of this transformation on profiling, the result is that we go from having an infinite number of unbounded-length paths in the control-flow graph to having a finite number of bounded-length paths. A path p in the original graph that proceeds several times around a loop will, in the profile, contribute "execution counts" to several smaller acyclic paths whose concatenation makes up p. In particular, the paths from *Start* to *Exit* in the modified graph correspond to acyclic paths in the original graph (where following the edge $Start \to v$ that was added to the modified graph corresponds to following backedge $w \to v$ in the original graph and beginning a new path at v, and following the edge $w \to Exit$ that was added to the modified graph corresponds to ending the path in the original graph at w).

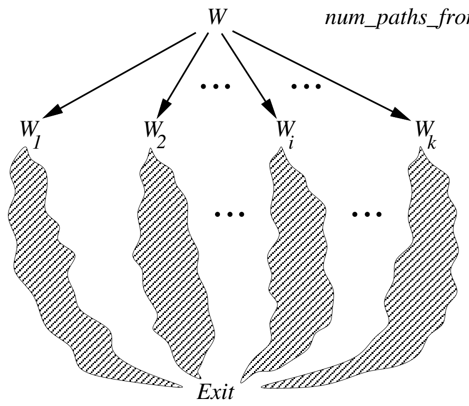
In the discussion below, when we refer to the "control-flow graph", we mean the transformed (*i.e.*, acyclic) version of the graph.

The Ball-Larus numbering scheme labels the control-flow graph with two quantities:

(1) Each *node* V in the control-flow graph is labeled with a value, $num\_paths\_from(V)$, which indicates the number of paths from V to the control-flow graph's *Exit* node.
(2) Each *edge* in the control-flow graph is labeled with a value derived from the *num_paths_from* quantities.

For expository convenience, we will describe these two aspects of the numbering scheme as if they are generated during two separate passes over the graph. In practice, the two labeling passes are combined into a single pass.

In the first labeling pass, nodes are considered in reverse topological order. The base case involves the *Exit* node: It is labeled with 1, which accounts for the path of length 0 from *Exit* to itself. In general, a node W is labeled only after all of its successors $W_1, W_2, \cdots, W_k$ are labeled. When W is considered, $num\_paths\_from(W)$ is set to the value $num\_paths\_from(W_1) + \cdots + num\_paths\_from(W_k)$, as indicated in the diagram below:



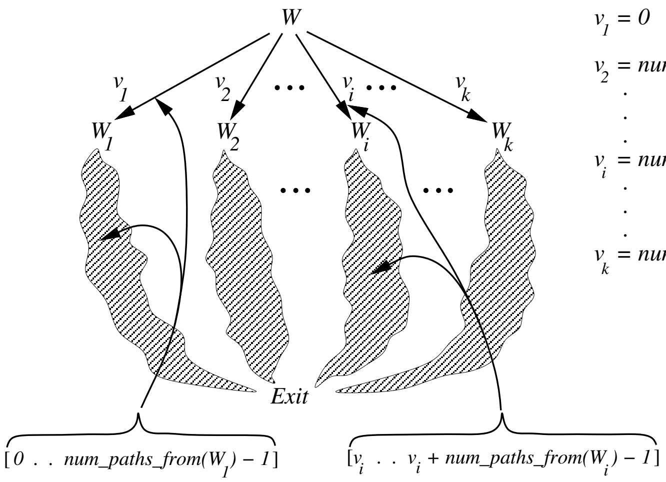
num_paths_from(W) = num_paths_from(W$_1$) + . . . + num_paths_from(W$_k$)

The goal of the second labeling pass is to arrive at a numbering scheme for which, for every path from *Start* to *Exit*, the sum of the edge labels along the path corresponds to a *unique* number in the range [0 .. *num_paths_from*(*Start*) − 1]. That is, we want the following properties to hold:

(1) Every path from *Start* to *Exit* is to correspond to a number in the range [0 .. *num_paths_from*(*Start*) − 1].

(2) Every number in the range [0 .. *num_paths_from*(*Start*) − 1] is to correspond to some path from *Start* to *Exit*.

Again, the graph is considered in reverse topological order. The general situation is shown below:



$v_1 = 0$

$v_2 = num\_paths\_from(W_1)$

.
.
.

$v_i = num\_paths\_from(W_1) + \ldots + num\_paths\_from(W_{i-1})$

.
.
.

$v_k = num\_paths\_from(W_1) + \ldots + num\_paths\_from(W_{k-1})$

At this stage, we may assume that all edges along paths from each successor of W, say W_i, to *Exit* have been labeled with values so that the sum of the edge labels along each path corresponds to a unique number in the range [0 .. *num_paths_from*(W_i) − 1]. Therefore, our goal is to attach a number v_i on edge $W \to W_i$ that, when added to numbers in the range [0 .. *num_paths_from*(W_i) − 1], distinguishes the paths of the form $W \to W_i \to \cdots \to$ *Exit* from all paths from W to *Exit* that begin with a different edge out of W.

This goal can be achieved by generating numbers $v_1, v_2, \cdots, v_k$ in the manner indicated in the above diagram: The number v_i is set to the sum of the number of paths to *Exit* from all successors of W that are to the left of W_i:

$$v_i = \sum_{j=1}^{i-1} num\_paths\_from(W_j).$$

This "reserves" the range $[v_i \,..\, v_i + num\_paths\_from(W_i) - 1]$ for the paths of the form $W \rightarrow W_i \rightarrow \cdots \rightarrow Exit$. The sum of the edge labels along each path from W to $Exit$ that begins with an edge $W \rightarrow W_j$, where $j < i$, will be a number strictly less than v_i. The sum of the edge labels along each path from W to $Exit$ that begins with an edge $W \rightarrow W_m$, where $m > i$, will be a number strictly greater than $v_i + num\_paths\_from(W_i) - 1$.

Example. Returning to the example used in Section 3, Figure 2 shows how the control-flow graph of the program fragment that reads and processes data from a database of customer information would be annotated. Each box is annotated with the number of paths from that node to the final node of the fragment; each edge is annotated with the number that would be assigned by the edge-numbering scheme described above.

```
a: r := 0
   birth_year := read()
   has_college_degree := read()
   purchases := read()
   age := current_year() − birth_year
if age < 15 then
    b: . . .
else
    c:  r := r + 4
          . . .
fi
if has_college_degree = true then
    d: . . .
else
    e:  r := r + 2
          . . .
fi
if purchases > 3 then
    f: . . .
else
    g:  r := r + 1
          . . .
fi
h: profile[r] := profile[r] + 1
```

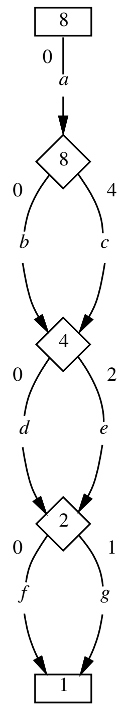


Figure 2. The instrumented version of the program fragment that reads and processes data from a database of customer information, and the program's annotated control-flow graph.

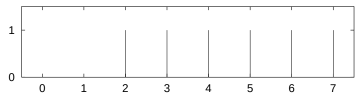
Note that the sum of the edge labels along each path from the beginning to the end of the graph falls in the range [0 .. 7], and that each number in the range [0 .. 7] corresponds to exactly one such path. □

The final step is to instrument the program, which involves introducing a counter variable and appropriate increment statements to accumulate the sum of the edge labels as the program executes along a path.

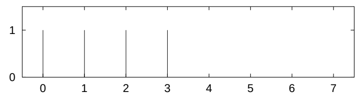
Example. The instrumented version of the program's source code is shown on the left in Figure 2. Statements that increment counter r have been introduced so that at the end of the fragment its value indicates which path through the fragment was executed. This value is then used to increment the appropriate element of array *profile*, which maintains the frequency distribution of paths executed. (Alternatively, *profile* could maintain just a Boolean indicator of whether the path is ever executed.) □

Several additional techniques are employed to reduce the runtime overheads incurred. These exploit the fact that there is actually a certain amount of flexibility in the placement of the increment statements [4,5].

Profiles obtained from the instrumented program can be displayed in the fashion shown below, where paths are arranged on the x-axis according to the path number, and the y-axis is used to indicate either the execution frequency or just a Boolean indicator of whether the path was executed at all. The spectra discussed in Section 3 would be displayed as follows:



Pre-2000 spectrum

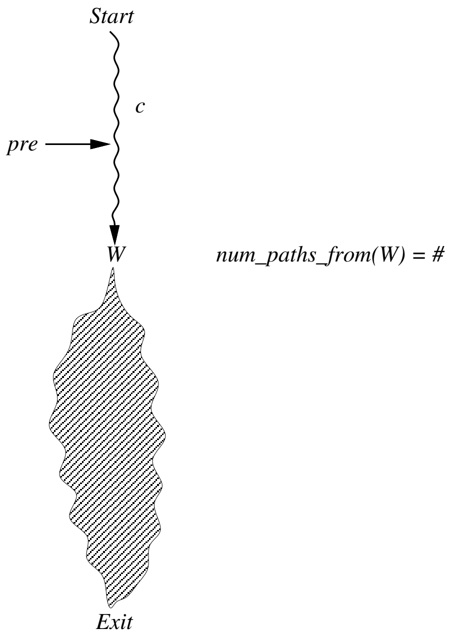


Post-2000 spectrum

4.2. An Alternative Technique for Identifying Problematic Path Prefixes

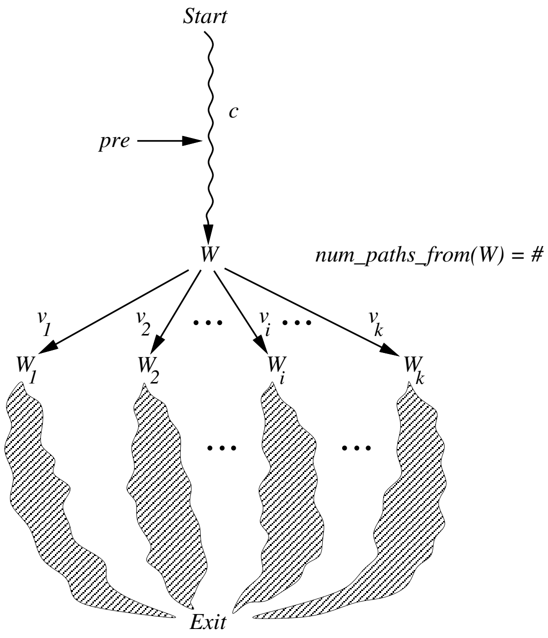
Suppose that p is a path in *new_spectrum* that does not occur in *old_spectrum*. This section describes how to exploit the Ball-Larus path-numbering scheme for the purpose of finding the shortest prefix of p that is not a prefix of any executed path in *old_spectrum*. (If p is a path in *old_spectrum* that does not occur in *new_spectrum*, then flip the roles of *old_spectrum* and *new_spectrum* in what follows.) Instead of using a trie structure on *old_spectrum*, an index structure that supports range queries is built on *old_spectrum*, and a sequence of queries is issued to determine whether certain ranges are empty or not. Let *IsRangeEmpty*(S, a, b) be an operation that returns true if S does not contain any values in the range $[a \,..\, b]$, inclusive. (Standard data structures can be used to implement *IsRangeEmpty*(S, a, b) efficiently, *i.e.*, in time logarithmic in the size of S. For instance, see [17], pp. 373-374.)

Now consider a path from *Start* to *Exit* that has prefix *pre*, where *pre* ends at node W, and suppose that the sum of the labels on the edges of *pre* is c, as shown below:



All such paths have numbers in the range $[c \mathrel{..} c + num\_paths\_from(W) - 1]$, and there are precisely $num\_paths\_from(W)$ such paths. Consequently, by the unique-numbering property of paths from *Start* to *Exit*, all paths from *Start* to *Exit* with numbers in the range $[c \mathrel{..} c + num\_paths\_from(W) - 1]$ have prefix *pre*.

The search for the shortest prefix of p that is not a prefix of any executed path in *old_spectrum* is carried out as follows. As above, suppose that p is a path from *Start* to *Exit* that has prefix *pre*, where *pre* ends at node W, and that the sum of the labels on the edges of *pre* is c. Suppose further that we have already searched from *Start* to node W and have not yet found the edge that distinguishes p from the paths of *old_spectrum*:



(When the search is initiated, $W = Start$, *pre* is the empty path, and $c = 0$.)

Assume that path p continues from W along edge $W \rightarrow W_i$, which is labeled with the value v_i (*i.e.*, p has prefix $pre \,\|\, (W \rightarrow W_i)$, where "$\|$" denotes path concatenation). We need to know if any of the paths in *old_spectrum* also have prefix $pre \,\|\, (W \rightarrow W_i)$. Again, by the unique-numbering property of paths from *Start* to *Exit*, the paths in the graph from *Start* to *Exit* that have prefix $pre \,\|\, (W \rightarrow W_i)$ are exactly the paths with numbers in the range $[c + v_i \,..\, c + v_i + num\_paths\_from(W_i) - 1]$. Thus, to determine if any of the paths in *old_spectrum* have prefix $pre \,\|\, (W \rightarrow W_i)$, we need to perform the test

$$IsRangeEmpty(old\_spectrum,\ c + v_i,\ c + v_i + num\_paths\_from(W_i) - 1).$$

If this test is true, then W is the branch statement at which p veers into "unknown territory" (along the edge $W \rightarrow W_i$). Otherwise, we continue the search at node W_i using the path prefix $pre \,\|\, (W \rightarrow W_i)$ and path-prefix value $c + v_i$.

The running time for this method of identifying the shortest prefix of a path in *new_spectrum* that does not occur in *old_spectrum* is not strictly comparable to the time used by the trie method discussed in Section 3. However, the asymptotic worst-case running time for building the range-query structure is better than the worst-case running time needed to build the trie, and the worst-case space usage of the range-query method is also better than the worst-case space usage of the trie method. For both methods, suppose that we are given an unsorted list of the paths executed in *old_spectrum*. Let $|old\_spectrum|$ denote the size of *old_spectrum* (*i.e.*, the number of paths in *old_spectrum*).

- The time required to build a trie structure for *old_spectrum* is proportional to the sum of the number of edges in the paths in *old_spectrum*. (Note that this is potentially much greater than $|old\_spectrum|$, which is simply the number of paths in *old_spectrum*.) In the worst case, storing the trie could require space proportional to the sum of the number of edges in the paths in *old_spectrum*. The time needed to determine the shortest distinguishing prefix *pre* of a path p is proportional to the number of edges in the answer: $|pre|$.

- The time required to build a range-query structure for *old_spectrum* is proportional to $|old\_spectrum| \cdot \log |old\_spectrum|$, and the space needed to hold the range-query structure is proportional to $|old\_spectrum|$. The time needed to determine the shortest distinguishing prefix *pre* of a path p is proportional to $|pre| \cdot \log |old\_spectrum|$.

5. Implementation and Preliminary Results

We built a prototype system that implements the path-spectrum-comparison technique and carried out two preliminary tests with it. The system runs under Solaris on Sun SPARCstations. It uses Tcl/Tk to implement a graphical user interface, and Larus's implementation of the Ball-Larus path-profiling algorithm as the underlying machinery for generating path spectra. The path profiler instruments executable files, so programs can be written in any language (as long as the compiler for the language obeys certain calling conventions) or even in a mixture of languages.

The goal of the user interface is to allow one to collect up, and perform difference operations on, collections of path profiles. The user can display path profiles as spectra (as shown in Section 4.1 and Figure 4). (At present, we are not using the thresholding technique described in Section 3.1, and the system treats each path profile as merely a set of paths; that is, the frequency counts of the number of times each path executed is ignored. Thus, an executed path in a spectrum is displayed as a stick of height 1.) Spectra have links back to the source code: Clicking on the stick that represents a path brings up an *emacs* window with the elements of the path displayed in a special color.

The system is organized around the notions of *profiles* and *workspaces*: Collections of profiles can be selected and placed in named workspaces. Because we are interested in path-spectrum differences, when path profiles from a workspace are displayed as spectra, each spectrum shows only paths that were executed in at least one of the profiles of the workspace but not in all of the profiles.

As part of calling up spectrum differences, the user forms sub-partitions of the profiles in a workspace. The profiles in a workspace are partitioned into three groups, which we will call A, B, and *Other*. (That is, A, B, and *Other* are each sets of profiles.) Spectrum differences are displayed by showing path sticks for paths that are executed by all profiles in A, but not by some profile in B, and vice versa. Clicking on one of the path sticks brings up an *emacs* window with the statements of the last edge of the shortest distinguishing prefix of the path displayed in one special color, the rest of the shortest distinguishing prefix displayed in a second special color, and the rest of the elements of the path displayed in a third special color.

One experiment that we carried out with the system was aimed at testing the ability of path-spectrum comparison to identify leap-year calculations. This experiment involved the UNIX *cal* utility, which, given a month and a year as input, prints the calendar for that month. The *cal* program does not actually have a leap-year problem: It calculates correctly that the year 2000 is a leap year. However, because our goal was merely to determine whether path-spectrum comparison would be able to identify leap-year calculations, this did not matter—we tested the method's sensitivity to leap-year calculations by comparing spectra from leap years and non-leap years. Path spectra obtained from runs that we expected would involve leap-year calculations (*e.g.*, from inputs like "*cal 2 1992*", "*cal 2 1996*", etc.) were compared against spectra obtained from runs that we expected not to involve leap-year calculations (*e.g.*, "*cal 2 1997*", "*cal 2 1998*", etc.).

```
cal(m, y, p, w)
char *p;
{
    register d, i;
    register char *s;
    int foo = 0;

    s = p;
    d = jan1(y);
    mon[2] = 29;
    mon[9] = 30;

    switch((jan1(y+1)+7-d)%7) {

    /*
     *    non-leap year
     */
    case 1:
        mon[2] = 28;
        break;

    /*
     *    1752
     */
    default:
        mon[9] = 19;
        break;

    /*
     *    leap year
     */
    case 2:
        foo = foo + 1;        /* Statement added so that something in the leap-year case could be highlighted */
        break;
    }
    for(i=1; i<m; i++)
        d += mon[i];
    d %= 7;
    s += 3*d;
    . . .
```

Figure 3. The code displayed in ***Times-BoldItalic***, **Helvetica-Bold**, and **Times-Bold** indicates a path that was executed during a run with input "*cal 2 1996*", but not during a run with input "*cal 2 1997*". The code shown in ***Times-BoldItalic*** and **Helvetica-Bold** indicates the shortest prefix of the path that distinguishes it from all paths of the "*cal 2 1997*" run. The code shown in **Helvetica-Bold** indicates the last edge of the shortest distinguishing prefix (*i.e.*, **switch((jan1(y+1)+7-d)%7)** $\rightarrow$ **foo = foo + 1;**).

For example, in a trial with workspace-partition *A* consisting of the profile from a run with input "*cal 2 1996*" and *B* consisting of the profile from a run with input "*cal 2 1997*", there was

- One path that was executed during the run with input "*cal 2 1996*", but not during the run with input "*cal 2 1997*".

- One path that was executed during the run with input "*cal 2 1997*", but not during the run with input "*cal 2 1996*".

Figure 3 shows the path that was executed during the run with input "*cal 2 1996*", but not during the run with input "*cal 2 1997*", as well as the shortest prefix of the path that distinguishes it from all paths of the "*cal 2 1997*" run. To understand the code shown in Figure 3, it helps to know that the routine "jan1" receives a year value as its parameter, and returns a number in the range [0 .. 6] that represents the day of the week on which January 1 falls that year. The values 0 through 6 correspond to Sunday through Saturday, respectively. The switch statement chooses one of three cases, depending on the difference (in terms of number of days of the week) between jan1(y) and jan1(y+1). The switch value is 1 in the case of an ordinary, non-leap year; 2 in the case of a leap year; and 5, represented by the default case, in 1752, the year that England and the Colonies shifted from the Julian to the Gregorian calendar. The default case is used to make a minor adjustment to one of the program's internal tables, which has an effect elsewhere on how the calendar for September 1752 is created.

Figure 3 also illustrates a small glitch due to the fact that the path profiler we used instruments executable files. The program shown in Figure 3 has an additional statement, "foo = foo + 1;" that we added in "case 2" of the switch statement. With the original program, in which "case 2" was empty, we were initially confused by the path that the system highlighted. No part of "case 2" was highlighted, and we did not at first recognize that the path actually did go into that branch of the switch statement. The reason for this was that the prototype path-spectrum comparator uses information generated by the compiler to map from addresses in executable files to lines in the source code. Our confusion was caused by the fact that the compiler had not generated any instructions for the empty case, and so the path-spectrum comparator did not have the information it needed to highlight "case 2". In Figure 3, the statement "foo = foo + 1;" was added so that something existed in the body of "case 2" that could be highlighted. (A path profiler that performs source-code instrumentation would not have this problem.)

A second experiment that we carried out was aimed at testing the ability of path-spectrum comparison to identify date-rollover problems. The test involved a version of *ncftp*, a file-transfer utility. As in the first experiment, the standard version of the program does not, in fact, have a Y2K problem—so we introduced one, by arranging for all year values that the program manipulates to be in the range [00 .. 99]! The results are presented in Figure 4, which shows the path spectra obtained from six runs of the date-sensitive version of the program. The six runs processed input data associated with different years. Note how the path spectra change as we cross the year 2000 boundary, but are almost completely stable on either side of it.

6. Other Applications of Path-Spectrum Comparison in Software Maintenance

The path-spectrum-comparison technique is actually applicable to a much wider range of problems that arise in software maintenance than just the Y2K problem. A number of other ways to enlist path-spectrum comparison in the cause of providing better help for software-maintenance problems are described below.

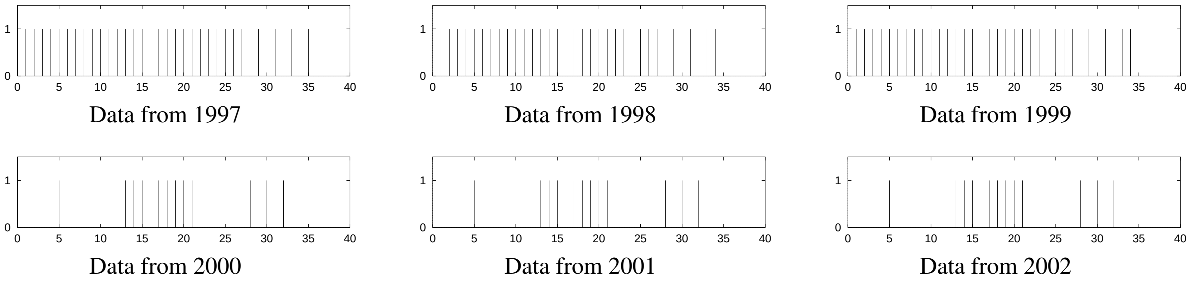


Figure 4. Path spectra from six runs of a date-sensitive version of *ncftp*.

Testing

The application of path-spectrum comparison to the Y2K problem involves comparing path spectra from different execution runs. The principle is that "information about possible date-dependent computations can be obtained by comparing path spectra from execution runs on pre-2000 data and post-2000 data". This is essentially a testing strategy, although one of a novel kind. The Y2K problem is also just one example of a problem to which this kind of testing strategy can be applied.

In broadest terms, the general principle can be stated as follows:

> A path spectrum is a finite, easily obtainable characterization of a program's execution on a dataset, and provides a behavior signature for a run of the program. When different runs of a program produce different path spectra, the spectral differences can be used to identify paths in the program along which control diverges in the two runs. By choosing input datasets to hold all factors constant except one, any such divergence can be attributed to this factor. The point of divergence itself may not be the cause of the underlying problem, but provides a starting place for a programmer to begin his exploration.

This principle offers new perspectives on testing, on the task of creating test data, and on what tools can be created to support program testing.

This approach to testing is a new variant of white-box testing, which we propose to call "I/B testing", for "Input/Behavior" testing, by analogy with I/O testing. In contrast to I/O testing, I/B testing can reveal possible problems—by finding path-spectrum differences—even when the output of an execution run is correct.

The effectiveness of path-spectrum comparison for uncovering errors depends on how good the two input datasets are at eliciting different behaviors during the different runs. For instance, the results in the Y2K problem depend on how well the input data stimulates different behaviors during the pre-2000 and post-2000 runs. This raises a number of questions. Two of them—analogs of well-known issues that arise with conventional testing methods, and left open here for future research—are: "How does one design pairs of input sets that are likely to cause errors to be revealed via spectral differences?" and "How does one evaluate the quality of a suite of input-set pairs?"

Testing is a huge problem for software developers, and will be with us long past the year 2000. We believe that the path-spectrum-comparison technique holds the promise of providing a useful adjunct to conventional methods for testing whether programs are functioning properly (and debugging them when they are malfunctioning).

Systems that Warn of Possible Errors Within Themselves

As described thus far, the spectra that are compared come from different runs of a program. However, the underlying principle is simply that "information about possible execution problems can be obtained by comparing two spectra". The spectra do not necessarily have to be from different runs of the program. All we care about is that there are two spectra to be compared (and that the spectra provide some sort of behavior signature). The spectra could be obtained from two or more runs (as in the application of the technique to the Y2K problem); however, there are situations in which it would be meaningful to compare spectra obtained during a single run.

Two situations in which this would be useful are: (i) when a system is being tested, and (ii) in a system that warns of possible errors within itself. In both cases, the idea is to have the system compare each path executed by the program with the paths executed so far. When a new path is discovered (*i.e.*, when the path is executed for the first time) the program would signal that a possibly erroneous computation has just occurred—*i.e.*, to warn the user or system tester that the program has just gone down a possibly bad path. (The system could issue the warning directly to the user, to a dialog box, to the console window, or to a log file.)

Such information (*e.g.*, perhaps the last few such paths reported) could provide important clues that would help in tracking down a bug once a symptom comes to the attention of the user. Of course, one would want to wait until the program had run for a while before starting to issue such warnings, but after a break-in or warm-up period it would begin to be useful to gather such information.

Testing Which Parts of a System are Affected by a Modification

Another variation on path-spectrum comparison could be used to support the testing of bug fixes and other small changes to a system. The goal here would be to understand whether the only behavioral changes introduced by the modification were to the intended parts of the system. The idea is to use path-spectrum comparison as a heuristic method for understanding the magnitude of behavioral changes between two versions of a program.

In this context, the comparison that needs to be carried out is somewhat different from what has been discussed earlier: Instead of comparing spectra from two runs of the *same* program on *different* data, one would compare spectra from two runs of a (slightly) *different* program on the *same* data. As before, the premise that "states are similar if they proceed down the same path" provides the justification for why it makes sense to be comparing path spectra (even though they now come from execution runs of *different* programs).

Of course, one expects there to be differences between the two spectra obtained from the two versions of the program. For example, one would expect to see differences on the input that elicits the bug in the original program. The purpose of comparing the path spectra would be to obtain information about the extent of actual changes in behavior. One wants to make sure that a small change in the program text does not lead to radical changes in the behavior. The behavior of most of the unmodified parts of the system should be unaffected by a modification. The programmer can use the information obtained from path-spectrum comparison to develop an understanding of the actual magnitude of behavioral differences that a bug fix introduces.

In order to carry out comparisons between paths from two different programs, a concordance between paths in the old program and paths in the new program would be needed. The instrumentation strategy used affects how difficult it is to provide such a concordance: It would not be too hard to establish a correspondence between paths in the old and new programs when source-code instrumentation is used, but would be much more difficult when instrumentation is carried out on object-code files or executable files.

Testing for Inconsistent Data

Another potential application of path-spectrum comparison is to the "data hygiene" problem. The goal here is to identify data in a database or file that is contaminated, or inconsistent with the assumptions about the data that the program relies on. Our hypothesis is that some contaminated data items will cause the program to take unusual paths through the code (but ones that do not actually crash the system). Presumably the percentage of contaminated data is low; thus, the idea behind using path-spectrum comparison is to use information about infrequently executed paths to identify possibly contaminated data in the database. Any peculiar paths (*i.e.*, paths with count 1 or low relative frequency in the path spectrum) when the program is run against the database would be taken as a signal that the program was processing possibly contaminated data. To actually identify the contaminated data, one would need the instrumented program to gather some additional information in order to link the low-frequency paths back to the inputs that were most recently read in at the times the path was executed.

7. Related Work

This paper has described new techniques to help with testing and debugging, using information obtained from path profiling. Our work is based on the idea of comparing path spectra from different runs of the program to identify paths in the program along which control diverges in the different runs. The path-spectrum-comparison technique is a completely different way of using path profiling in the context of program testing from another use that has been proposed in the past, namely as a criterion for evaluating the coverage of a test suite [22,14,8,16]. The question of whether there is any hope of using the path-coverage criterion in practice has often been raised. The published results of Ball and Larus suggest that the answer to this question is "no". They report that some of the SPEC benchmarks had approximately $10^9 - 10^{11}$ paths, of which only 10^4 were ever executed on a given run [5]. Although not all of the possible paths are necessarily feasible, it could be necessary to run $10^5 - 10^7$ tests (and probably far more) to achieve a high degree of coverage.

Because our goal is different—our aim is to use spectral differences to identify paths in the program that represent changed behavior in the different runs—our use of path profiling to support program testing does not run afoul of the "high-number-of-paths/low-coverage-per-run" issue. This is not to imply that our use of path profiling does not come equipped with its own set of problems. On the contrary: The effectiveness of the path-spectrum comparison for testing depends on how good two test sets are at eliciting different behaviors during execution, and the question of how one designs pairs of input sets that are likely to cause errors to be revealed has been left open for future research.

The Docket project has explored ways to use information obtained from testing and dynamic analysis, including information about paths traversed during execution, in tools to support program comprehension [6]. One application of the Docket toolset addressed the problem of extracting "business rules" from programs [18]—*i.e.*, high-level requirements on how input data is to be processed, expressed in terms of the application domain (*e.g.*, "to be billed after delivery the customer must have a credit rating of at least satisfactory, otherwise, the customer must pay on delivery" [19]). Information about an input/output value pair, the types of the input and output values, and the path through the program that was executed is used to generate several candidate assertions (*viz.* possible "business rules") that characterize the I/O transformation.

There is a distant relationship between some of the techniques proposed in Section 6 and previous work on testing and debugging:

- Relative debugging allows programmers to compare the execution behaviors of multiple instances of the same program [1]. The setting for relative debugging is the porting of code (usually Fortran) from one platform (hardware/OS) to another. Because of differences in hardware and/or numerical libraries, the *same* program may exhibit different behaviors on different platforms. With relative debugging, the programmer places assertions in the source code, which are then checked against one another as the two programs execute in parallel on the different platforms. The debugger takes care of the details of inserting breakpoints and comparing data structures across the two executions. When a substantial difference in behavior is found (*i.e.*, an assertion is violated), the programmer is notified. Relative debugging also supports runtime comparison of a modified program to an older reference program.
- Dependences between tests and program entities have been used to implement selective regression testing in the TestTube system [7]. In this case, there are two different versions of a program, and dependence information gathered from previous tests is used to determine whether a test needs to be rerun on the new version.

8. Final Remarks

Throughout the paper, we have referred to a path spectrum as a "behavior signature for a run of the program", and have advanced the general principle that "information about differences in execution behavior can be obtained by comparing two behavior signatures". One has to be a bit careful about the notion of a behavior signature, however. For instance, a trace of the program counter could also be considered to be a behavior signature for a run of the program.

This leads to the conclusion that not all behavior signatures are equally valuable. Consider again the Y2K problem: The comparison of two traces of the program counter, generated during separate runs of the program, would yield only a small amount of useful information that could be used by a programmer trying to determine where the program performs problematic date manipulations. For example, the first point at which the traces diverge could be attributed to a date-dependent computation, but extracting additional information about other problematic date manipulations in the program would be difficult. However, when the two traces are transformed into path spectra, they take on a form that does provide utility for this task. This suggests the following analogy:

> In physics, it is often more appropriate to work with the Fourier transform of a function rather than with the function itself. The reason is that the Fourier transform reveals the fundamental excitation modes of a system. Manipulations of Fourier transforms operate directly in terms of these fundamental modes.
>
> Similarly, we have identified situations in which it is more appropriate to work with path spectra rather than with traces of the program counter. The reason is that a path spectrum reveals the "fundamental modes" of the program that were "excited" during an execution run. Manipulations of path spectra operate directly in terms of the program's fundamental excitation modes.

(The fact that a Fourier transform is information preserving, whereas a path profile loses information available in an execution trace, does not invalidate the analogy. The loss of information is tied up with a different issue, namely that of abstraction. That is, the creation of a path spectrum involves both a transformation that reveals underlying structure, as well as an abstraction of information available in the original trace.)

One reason for casting things in these terms is that it suggests an avenue for further research: Are there other "transforms" to be discovered that would tease out interesting information (similar to our "fundamental modes of a program excited during an execution run") from other kinds of data about programs, execution runs, etc.?

Acknowledgements

We are grateful for the helpful comments of K. Baxter, B. Carlson, J. Field, S. Horwitz, T. Taft, and D. Weise.

References

1. Abramson, D., Foster, I., Michalakes, J., and Sosic, R., "Relative debugging: A new methodology for debugging scientific applications," *Commun. of the ACM* **39**(11) pp. 68-77 (November 1996).
2. Backman, T., *Summary of the MITRE assessment on the effects of two-digit years for the year 2000,* The MITRE Corporation, McLean, VA (January 1996). (Available on the WWW at URL http://www.mitre.org:80/research/y2k/docs/BRIEF.html.)
3. Bala, V., "Low overhead path profiling," Technical Report, Hewlett Packard Labs (1996).
4. Ball, T., "Efficiently counting program events with support for on-line queries," *ACM Trans. Program. Lang. Syst.* **16**(5) pp. 1399-1410 (September 1994).
5. Ball, T. and Larus, J., "Efficient path profiling," in *Proceedings of MICRO-29*, (December 1996).
6. Benedusi, P., Benvenuto, V., and Tomacelli, L., "The role of testing and dynamic analysis in program comprehension supports," pp. 149-158 in *Proceedings of the Second IEEE Workshop on Program Comprehension,* (July 8-9, 1993, Capri, Italy), ed. B. Fadini and V. Rajlich, IEEE Computer Society Press, Washington, DC (July 1993).
7. Chen, Y.-F., Rosenblum, D.S., and Vo, K.-P., "TestTube: A system for selective regression testing," in *Proceedings of the Sixteenth International Conference on Software Engineering,* (May 16-21, 1994, Sorrento, Italy), IEEE Computer Society Press, Washington, DC (1994).
8. Clarke, L.A., Podgurski, A., Richardson, D.J., and Zeil, S.J., "A comparison of data flow path selection criteria," pp. 244-251 in *Proceedings of the Eightth International Conference on Software Engineering*, IEEE Computer Society Press, Washington, DC (1985).
9. Gartner Group, *Year 2000 Problem Gains National Attention,* Gartner Group, Stamford, CT (April 1996). (Available on the WWW at URL http://www.gartner.com/aboutgg/pressrel/pry2000.html.)
10. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (January 1990).

11. Johnson, S.C., “Postloading for fun and profit,” pp. 325-330 in *Proceedings of the Winter 1990 USENIX Conference*, (January 1990).
12. Larus, J.R. and Schnarr, E., “EEL: Machine-independent executable editing,” *Proceedings of the ACM SIGPLAN 95 Conference on Programming Language Design and Implementation,* (La Jolla, CA, June 18-21, 1995)*, ACM SIGPLAN Notices* **30**(6) pp. 291-300 (June 1995).
13. Ottenstein, K.J. and Ottenstein, L.M., “The program dependence graph in a software development environment,” *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* (Pittsburgh, PA, Apr. 23-25, 1984)*, ACM SIGPLAN Notices* **19**(5) pp. 177-184 (May 1984).
14. Rapps, S. and Weyuker, E.J., “Selecting software test data using data flow information,” *IEEE Transactions on Software Engineering* **SE-11**(4) pp. 367-375 (April 1985).
15. Reps, T., Horwitz, S., Sagiv, M., and Rosay, G., “Speeding up slicing,” *SIGSOFT 94: Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering,* (New Orleans, LA, December 7-9, 1994)*, ACM SIGSOFT Software Engineering Notes* **19**(5) pp. 11-20 (December 1994).
16. Roper, M., *Software Testing,* McGraw-Hill, New York, NY (1994).
17. Sedgewick, R., *Algorithms,* Addison-Wesley, Reading, MA (1983).
18. Sneed, H.M. and Ritsch, H., “Reverse engineering programs via dynamic analysis,” pp. 192-201 in *Proceedings of the IEEE Working Conference on Reverse Engineering,* (May 21-23, 1993, Baltimore, MD), IEEE Computer Society Press, Washington, DC (May 1993).
19. Sneed, H.M. and Erdos, K., “Extracting business rules from source code,” pp. 240-247 in *Proceedings of the Fourth IEEE Workshop on Program Comprehension,* (March 29-31, 1996, Berlin, Germany), ed. V. Rajlich, A. Cimitile, and H.A. Mueller, IEEE Computer Society Press, Washington, DC (March 1996).
20. Srivastava, A. and Eustace, A., “ATOM: A system for building customized program analysis tools,” *Proceedings of the ACM SIGPLAN 94 Conference on Programming Language Design and Implementation,* (Orlando, FL, June 22-24, 1994)*, ACM SIGPLAN Notices* **29**(6) pp. 196-205 (June 1994).
21. Weiser, M., “Program slicing,” *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).
22. Woodward, M.R., Hedley, D., and Hennell, M.A., “Experience with path analysis and testing of programs,” *IEEE Transactions on Software Engineering* **SE-6**(3) pp. 278-286 (May 1980).