Whole Program Paths

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

Abstract

Whole program paths (WPP) are a new approach to capturing and representing a program's dynamic—actually executed—control flow. Unlike other path profiling techniques, which record intraprocedural or acyclic paths, WPPs produce a single, compact description of a program's entire control flow, including loop iteration and interprocedural paths.

This paper explains how to collect and represent WPPs. It also shows how to use WPPs to find *hot subpaths*, which are the heavily executed sequences of code that should be the focus of performance tuning and compiler optimization.

Keywords

dynamic program measurement, program tracing, path profiling, program control flow, data compression

1. Introduction

A central challenge facing computer architects, compiler writers, and mere mortal programmers is to understand a program's dynamic behavior. Events that occur while a program runs are often elusive, but they provide a basis for understanding the program's behavior and improving its performance. Program paths or traces—sequences of consecutively executed basic blocks—offer one of the few clear windows into a program's dynamic behavior. Paths, unlike other techniques, such as block or edge profiles, capture aspects of a program's dynamic control flow, not just its aggregate behavior.

Paths have long provided a unifying context for performance tuning. Programmers have improved the performance of large, complex systems, such as operating systems and databases, by identifying heavily executed paths and streamlining them into "fast paths" [20, 24]. In compilers as well, trace scheduling and, more recently, path-based compilation demonstrate that program optimization can benefit from a focus on a program's dynamic control flow [2, 8, 11, 12, 14]. Recently designed computer architectures have also directly exploited traces to enhance instruction caching and execution [15, 25, 26].

Paths are often identified by ad-hoc approaches; although recently developed path profiling techniques can inexpensively identify executed path segments and quantify their cost [2, 6, 7]. Previous path profiling algorithms, however, captured acyclic paths, which are short, disjoint snippets of execution that unfortunately end at loop and procedure boundaries—two of the most interesting points in a program's execution [6] (this technique has been extended to handle paths that cross procedure boundaries [19]).

This paper describes a new approach to measuring a program's control flow that captures a complete picture of the program's dynamic behavior. The technique introduces *whole program paths (WPP)*, which are a complete, compact record of a program's entire control flow. A whole program path crosses both loop and procedure boundaries, and so provides a practical basis for interprocedural path profiling. This paper explains how to record a WPP, describes its representation, shows that this representation can be used to analyze program behavior, and demonstrates the technique's practicality on SPEC benchmarks and commercial application programs.

1.1 Overview

Whole program paths are collected in two phases. The first produces a trace of the acyclic paths executed by a program. The second phase transforms the trace into a more compact and usable form by finding its inherent regularity (i.e., repeated code). In practice, compression can run concurrently with the instrumented program, so only the compressed form need be stored. The product of compression is a directed acyclic graph (DAG), which is not only a compact and lossless representation of the program's dynamic control flow, but is also a convenient representation for analysis. This paper describes one such analysis, which identifies heavily executed (hot) subpaths. Figure 1 illustrates the process of recording a whole program path.

Section 2 briefly describes the trace instrumentation and resulting sequence of acyclic paths. One novel contribution of this work is the next phase (compression), which turns a stream of acyclic paths into a context-free grammar. The compression technique is based on Nevill-Manning and Witten's SEQUITUR hierarchical compression algorithm [21, 22]. This linear, on-line algorithm builds a context-free grammar for a string. The resulting grammar reflects its input's hierarchical structure and is typically far more compact than the original sequence. Section 3 describes Nevill-Manning and Witten's algorithm and a modification that enhances its performance. The product of this algorithm is a grammar. The DAG representation of this grammar, called a Whole-Program Path (WPP), compactly and effectively records a program's entire control flow. Section 4 presents another contribution of this work, which is an analysis technique for WPPs. Section 5 contains measurements of the technique on SPEC benchmarks and Microsoft's SQL database and WinWord document processor.

As an example, consider the code in Figure 2. The loop executes nineteen acyclic paths (labeled 1–5). The SEQUITUR algorithm

SIGPLAN '99 (PLDI) 5/99 Atlanta, GA, USA

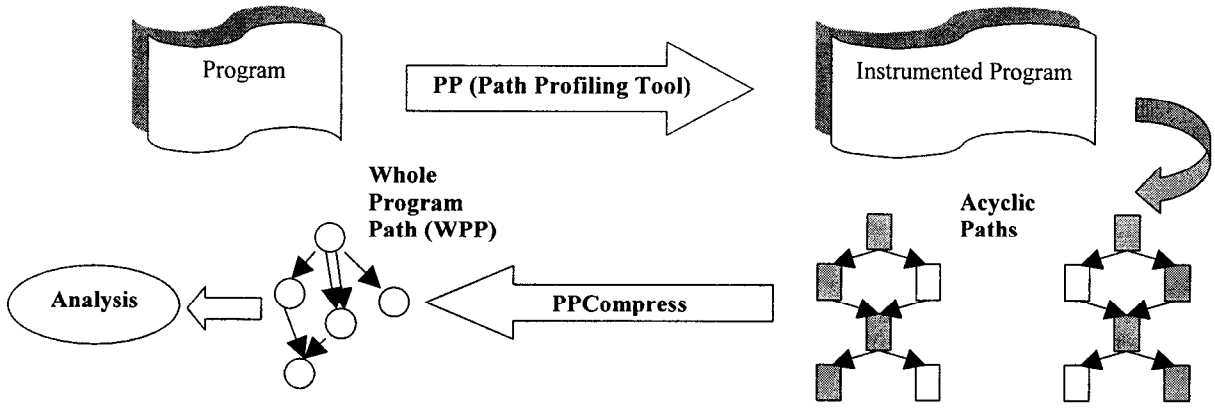

Figure 1. Collecting a Whole Program Path. A path-profiling tool (PP) instruments a program to produce a trace of executed acyclic paths. These paths are processed by another tool (PPCompress) into a Whole Program Path (WPP). Further tools analyze WPPs to find performance bottlenecks or program errors.

processes the path trace and produces the grammar in the figure. The grammar's DAG representation is the WPP data structure.

2. Producing an Acyclic Path Trace

The first step in this process is to instrument a program to record the acyclic paths executed by the program. The instrumentation is a slight variation of a previously published path-profiling algorithm [6]. This algorithm adds code to increment an accumulator by predetermined amounts along a select set of edges in a routine's control-flow graph. At the end of an acyclic path (i.e., at a routine's exit or a loop backedge), the value in this accumulator uniquely identifies the executed path. The original profiling algorithm used this path identifier to index a table of metrics associated with the path. Whole program profiling instead appends the identifier to a trace of executed paths.

Whole program profiling requires a slight redefinition of a path, so edges leading into a basic block containing a procedure call terminate acyclic paths. This change unfortunately reduces the average path length, and so increases the size of a path trace. However, it is necessary to ensure that the path leading to a call site is recorded in the trace before any paths executed by the callee. The path-profiling algorithm truncates paths with the mechanism originally used to terminate paths at loop backedges and to cut paths to limit the size of path identifiers [6].

The path trace consists a sequence of byte code-operand pairs:

OpCode(Operand)	Meaning
`EnterRoutine(ID)`	Subsequent paths execute in routine ID
`LeaveRoutine()`	Leave current routine and return to previous one
`NewPath(ID)`	Path ID executed
`EnterThread(ID)`	Subsequent paths execute in thread ID

The run-time instrumentation tracks non-local returns (setjmp/longjmp and exceptions), to produce the correct number of `LeaveRoutine` operations. Several variants of each opcode—e.g., byte, short, and word—reduce the size of a trace. More aggressive optimization, such as encoding the ID in an opcode did little to reduce the trace, as the range of ID values is larger than that found in instruction bytecodes. Nevertheless, the trace is reasonably compact, as most paths require only three bytes (an opcode and a short path ID). For example, Microsoft's

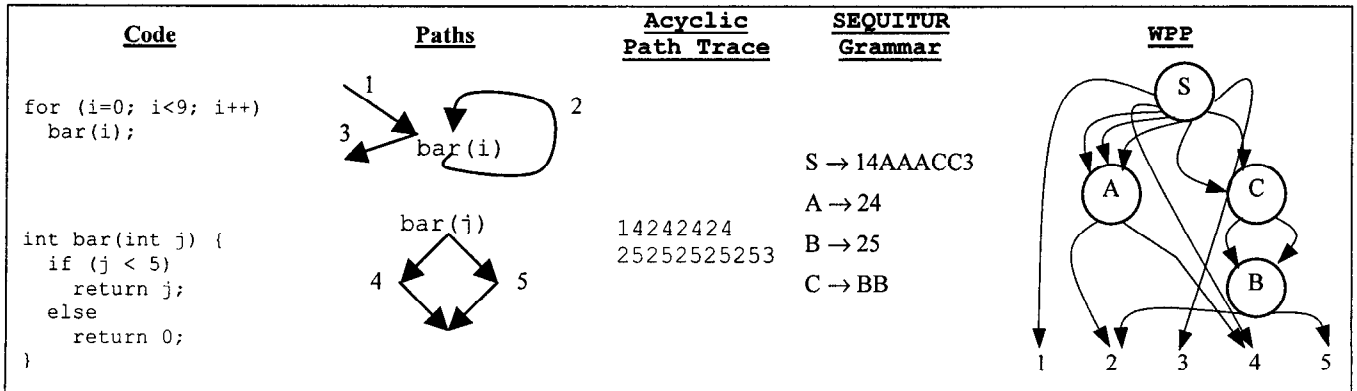

Figure 2. Sample code, the paths through it, and the grammar and WPP produced by PPCompress.

```
while input is not empty do
  c ← next input character;
  append c to start rule S;

  while digram or utility property is violated do
    // Restore digram uniqueness property:
    if digram D occurs twice (no overlap) in any rules then
      if one occurance of D is RHS of rule R then
        replace the other occurance of D with LHS of R;
      else
        create new rule R' with RHS D;
        replace both occurences of D by LHS of R';
      endif

    // Restore rule utility property:
    if rule R is only referenced once then
      replace single use of R by RHS of R;
      delete R;
    endif
  od
od
```

Figure 3. The SEQUITUR algorithm. LHS is the left side (non-terminal) of a grammar production. RHS is the right side of the production.

SQL database system running the TPC-C benchmark for 120 seconds produces 629 MB of trace (and the WPP for this run is only 21 MB).

3. Producing a Whole Program Path

The next stage of whole program profiling employs a modified version of the SEQUITUR algorithm to both compress the path trace and uncover its regular structure. SEQUITUR is a string compression algorithm that constructs a context-free grammar for its input [21, 22]. This algorithm has been used to find hierarchical structures in a variety of sequences, ranging from DNA sequences to genealogical databases. The insight underlying the algorithm is that log N rules can generate N occurrences of a subsequence. For example, the string:

abcabcabcabcabc

is produced by the grammar:

S → AAB
A → BB
B → abc

This grammar requires fewer symbols (11 versus 15), and, equally important for our application, explicitly captures repetitions of the pattern abc. This aspect becomes more apparent when the grammars are represented as DAGs (Section 3.3).

Section 3.1 explains the original SEQUITUR algorithm. Section 3.2 describes a modification to the algorithm to improve its performance. Section 3.3 explains how PPCompress uses the SEQUITUR algorithm to compress an acyclic path trace.

3.1 SEQUITUR Algorithm

The SEQUITUR algorithm (Figure 3) is a linear-time, on-line algorithm for producing a context-free grammar from an input string [22]. The algorithm operates by appending symbols from the input string, in order, to the end of the grammar's start production. After adding each symbol, SEQUITUR manipulates the grammar productions to preserve two invariants:

1. *Digram uniqueness property.* A *digram* is a pair of consecutive symbols on the right side of a grammar production. This property states that a digram occurs at most once in the rules of the grammar. If adding a symbol introduces a duplicate digram, SEQUITUR replaces both (non-overlapping) occurrences of the digram with the non-terminal symbol for a rule (possibly already in existence) that has the digram as its right side. For example, after adding symbol b to a grammar:

   S → abca

   the digram ab now occurs twice. SEQUITUR replaces both occurrences with a new non-terminal symbol A:

   S → AcA
   A → ab

2. *Rule utility property.* The second property is that all non-terminal symbols in a grammar (except the start symbol) must be referenced more than once by (other) rules. SEQUITUR eliminates a rule referenced only once by replacing the reference with the rule's right side. Continuing the example above by adding further symbols leads to:

   S → BCBA
   A → ab
   B → Ac
   C → Ad

   If the next symbol is d, SEQUITUR introduces a new rule:

   S → DD
   A → ab
   B → Ac
   C → Ad
   D → BC

   At this point, non-terminals B and C are only used once and SEQUITUR eliminates them.

Note that applying either rule may introduce new diagrams, which

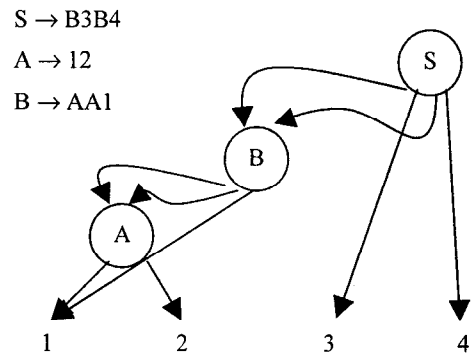

Figure 4. Grammar and WPP for the string 121213121214.

in turn require further transformation before the process converges. Nevill-Manning and Witten proved that SEQUITUR runs in time linear in the length of the input string [22].[1] Note that this time bound is independent of the size of the input alphabet.

The algorithm's space requirement, for a grammar and auxiliary data structures, is linear in the size of the grammar, which is *O(log N)* in the best case, where *N* is the input length. The worst case, in a sequence without repetition, is *O(N)*.

In practice, time and space are reasonable. The largest SPECINT95 benchmark is `099.go`, whose 2GB trace was processed in less than an hour in 300 MB of memory. The grammars themselves are smaller; approximately 100 MB for this benchmark, therefore analysis of a WPP requires less memory.

3.2 SEQUITUR Enhancement

Given that SEQUITUR is an on-line algorithm with tight time and space bounds, it is not surprising that the resulting grammars are not minimal. Although they are quite compact, a small change to the SEQUITUR algorithm improves some grammars by identifying more repetition of a substring. To see the need for these changes, consider the string: `11111211111`. SEQUITUR follows the following steps:

Start Rule	Action
S → 1111	create A → 11
S → AA1211	apply A → 11
S → AA12A1	create B → A1
S → AB2BA	

Although the input contains two occurrences of `11111`, they are represented differently in the grammar, because rules introduced while processing the first occurrence change the sequence of reductions applied to the second occurrence. Fortunately, subsequent occurrences are reduced the second way.

A minor change fixes this problem by looking ahead a single symbol before introducing a new rule to eliminate a duplicate digram. Assume the rightmost symbols of the start rule, which form a duplicate digram, are *x* and *y* and the look-ahead symbol is *l*. If the look-ahead symbol forms a digram with the second symbol of the duplicate digram and this digram, *yl*, is the right side of an existing rule, then do not introduce a new rule to eliminate the duplicate digram. Instead, read the next symbol and apply the existing rule. This algorithm is called SEQUITUR(1). This change does not affect the time bound on the algorithm, and in practice, seems to produce slightly smaller grammars.

Consider the example above. At the step that introduces the rule `B → ...`, the look-ahead character is `1`. Since `11` (the second character of the duplicate digram and the next character) is the right side of `A → 11`, a new rule is not introduced. Instead, SEQUITUR(1) applies the rule `A → 11`, resulting in the string: `S → AA12AA`. The look-ahead character is again `1`, but no digram `A1` is known, so the algorithm introduces a new rule `B → AA`, which leads to the grammar:

```
S → C2C
A → 11
C → AA1
```

3.3 PPCompress

PPCompress uses the SEQUITUR algorithm to compress an acyclic path trace. SEQUITUR operates on a string of symbols. In PPCompress, a symbol is a unique identifier for an *executed* acyclic path. As a previously unknown (routine id, path id) pair appears in the input stream, it is assigned a unique identifier. PPCompress imposes no limits on these symbols, beyond the space needed to maintain a hash table that records this mapping. In practice, programs execute relatively few paths (tens of thousands at most), so the number of symbols and the size of their identifiers remain manageable.

Grammars are typically represented as trees. However, WPPs are directed acyclic graphs (DAGs) since forming a tree would decompress a grammar into a string comparable in size to the path trace. Interior nodes in the DAG represent grammar productions. They are labeled with the non-terminal symbol from the left side of the production. Exterior nodes are terminal symbols (acyclic paths). An edge from node *A* to node *B* represents an occurrence of rule *B* in the right hand side of rule *A*. A node's suc cessors are ordered in the same manner as symbols in corresponding rule's right hand side.

The DAG representation is convenient to analyze. A sequence of executed paths can be recovered by traversing the DAG. Consider for example the string `1212131214` (path 1 might be a loop backedge, and other paths different traces through the loop body). Figure 4 contains its grammar and WPP. Repetitions of an acyclic path or sequence of acyclic paths appear in a WPP as multiple DAG paths from a node containing a non-terminal to

[1] The linearity proof assumes that a digram can be found from a pair of input symbols in constant time by using a hash table. To save space, PPCompress does not use a hash table, which must be sparsely filled to achieve this behavior. Instead, PPCompress associates, with every symbol, a map of the symbols that immediately follow. This map, which is keyed on the second symbol, returns the digram. These maps are implemented as unbalanced binary trees, which typically run in time logarithmic in the number of digrams in which a symbol occurs first. The worst case behavior of this code is $O(N^2)$, which is unlikely to occur in this application, as any path is followed by at most a small number of other paths.

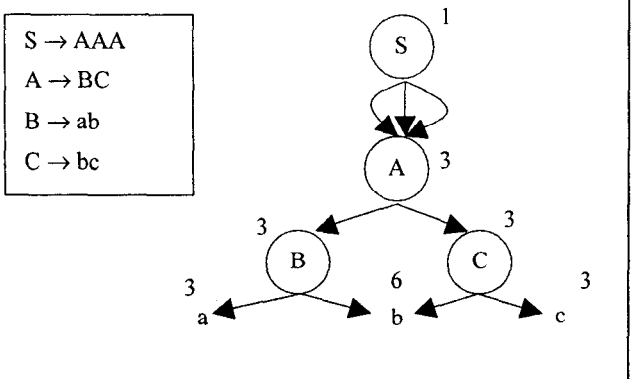

Figure 5. The WPP and grammar for the string abbcabbcabbc. The numbers are execution frequencies.

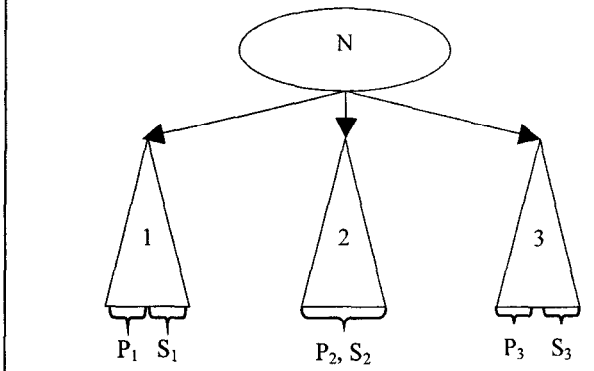

Figure 6. A node in a WPP and its descendents.

another DAG node. For example, node B corresponds to two executions of the path 12 followed by path 1.

Many interesting questions about a program's behavior can be answered directly from a WPP. For example, acyclic path *p* executes before path *q* if there exists a common ancestor of both nodes in which an inorder traversal reaches *p* before *q*. Another useful analysis is the dynamic execution context of code, such as its routine or loop iteration. This context is the paths that execute before and after the code. These paths are neighbors along the fringe of the DAG and are easily found by traversal.

The execution frequency of a sequence of acyclic paths is the number of times that prefix of this sequence is executed immediately before the suffix of the sequence. Sequences, such as ab or abc in Figure 5, that have a least common ancestor (LCA) in the WPP have the same execution frequency as this node. The execution frequency of a node is the number of paths in the DAG from the start symbol to the node (the numbers in the figure). Other sequences, such as ca, do not have a LCA as they arise from the repetition of a subsequence, in this case A (which starts with a and ends with c). Their frequency can be computed from the frequency of consecutive edges leading into the LCA of the subsequence (node A).

Since WPPs represent all executions of an acyclic path as a single terminal node, it is not possible to record distinct metrics for each execution of a path. For example, a path trace could associate metrics from hardware performance counters (e.g., cycles, stalls, cache misses, etc. [1]) with each path. PPCompress cannot directly maintain these metrics from different path executions, but instead must summarize them by aggregating values into a path's terminal symbol. This aggregation is not always disadvantageous, as it helps eliminate "noise" in performance data.

Collective metrics—such as the number of instructions along a path, the average number of cycles executed along the path, or the average number of cache misses along the path—are suitable for aggregation. Individual metrics, such as the number of cache misses in a particular execution, cannot be captured in a WPP. However, context-sensitive metrics similar to those collected by Ammons, Ball, and Larus [1]—for example, the number of caches misses in b after it executes path a—could be handled by associating costs with interior nodes.

4. Analyzing Whole Program Paths

A WPP captures a program's entire dynamic control in its DAG. This structure can be analyzed in many ways. This paper focuses on the important problem of finding hot paths. Previous path profiling work found that a small collection of hot paths typically dominate a program's execution [6]. WPPs provide the opportunity to find longer and more complete paths that cross procedure and loop boundaries. WPPs can also be analyzed find other dynamic program properties.

4.1 Subpaths

A whole program path encompasses a program's entire execution. Performance tuning and compiler optimizations generally focus on heavily executed code, in which small improvements yield large performance gains. Finding this code in a WPP requires the notion of a *subpath*—a consecutively executed sequence of acyclic paths. A string X is a subpath of a WPP grammar G if X is a substring of the string produced by G: $\alpha X \beta = L(G)$ where $\alpha\beta \in T^*$, $L(G)$ is the string produced by G, and T is the set of terminals (acyclic paths).

4.2 Hot Subpaths

A hot path is a path that incurs a substantial fraction of a program's execution cost. Ammons, Ball, and Larus defined hot paths as acyclic paths that contribute more than 0.1–1.0% of some execution metric [1]. They showed that in the SPEC benchmarks, relatively few hot paths (10–200) account for most of a program's execution cost (40–99%). Hot subpaths must be defined differently. Since subpaths lack boundaries, they can grow to encompass an arbitrary fraction of a program's execution.

Intuitively, a hot subpath is a short sequence of acyclic paths that is costly, either because the subpath is frequently executed or because operations along it are disproportionately expensive. Formally, a hot subpath is a sequence of L or fewer consecutively executed acyclic paths that incur a cost of C or more. A subpath's cost is its execution frequency multiplied times the sum of its constituent acyclic paths' costs. A minimal hot subpath is the shortest prefix of a subpath with cost of C or more. Minimal hot subpaths are of interest, since longer hot subpaths are easily found by adding acyclic paths to a minimal subpath.

```
void ReportHotSubPaths(Rule* rule, int mark) {
  if (rule->Mark() != mark) {              // First time visiting rule
    rule->SetMark(mark);

    rule->SetPrefix(new LLimitedString(MaxStringLength)); // Prefix of this rule
    LLimitedString* subPath = new LLimitedString(MaxStringLength); // Subpath thru rule

    // Iterate over successors in DAG (non-terminals on RHS of rule)
    for (Symbol* sym = rule->RHS()->FirstSym(); !rule->RHS()->Done(sym); sym = sym->Next())
    {
      if (sym->IsTerminal())
        appendTerminal(rule, subPath, sym); // Symbols are just appended to subpath
      else {
        Rule* symRule = sym->InRule();
        ReportHotSubPaths(symRule, mark);   // Postorder: find subpaths in successor

        appendTerminalString(rule, subPath, symRule->Prefix());

        if (!symRule->Prefix()->CoversNode())
          *subPath = *symRule->Suffix();    // Node is wider than prefix, so change suffix

        if (symRule->IncrNumPredecessors(-1) == 0) {
          delete symRule->Prefix();         // Free strings after last use
          delete symRule->Suffix();
        }
      }
    }
    rule->SetSuffix(subPath);
  }
}

void appendTerminalString(Rule* rule, LLimitedString* subPath, LLimitedString* string) {
  for (int i = 0; i < string->Length(); i += 1)
    appendTerminal(pps, rule, subPath, (*string)[i]);
}

void appendTerminal(Rule* rule, LLimitedString* subPath, Symbol* sym) {
  appendTerminalToRulePrefix(rule, sym);
  appendTerminalToSubPath(rule, subPath, sym);
}

void appendTerminalToRulePrefix(Rule* rule, Symbol* sym) {
  if (rule->Prefix()->Length() < MaxStringLength)
    rule->Prefix()->Append(sym, this);
  else
    rule->Prefix()->SetCoversNode(false);
}

void appendTerminalToSubPath(Rule* rule, LLimitedString* subPath, Symbol* sym) {
  subPath->Append(sym, this);
  int expense = subPath->Cost()*Frequency(rule);
  if (MinCost <= expense && MinStringLength <= subPath->Length()) {
    print subPath;
    subPath->Clear();
    rule->Prefix()->Freeze(subPath->Length());     // Stop before hot subpath
  }
}
```

Figure 7. Algorithm for finding minimal hot subpaths whose length is between MinStringLength and MaxStringLength and cost greater than MinCost.

Consider the example in Figure 5 (it might be part of a larger WPP, as the rule utility property would otherwise eliminate symbols B and C). Suppose that each acyclic path a, b, and c has a cost of 1 and that we are looking for hot subpaths of length greater than 1 and less than 4 whose cost is 6 or more. The WPP contains four overlapping hot subpaths: ab, bc, bb, and ca. The algorithm in this paper identifies two hot subpaths (ab and bc). The other two can be found by extending these two.

Figure 7 presents an algorithm for finding hot subpaths in a WPP. The algorithm performs a postorder traversal of the DAG, visiting each node once. At each interior node, it examines each subpath formed by concatenating the subpaths produced by two or more of the node's descendents. The algorithm examines only concatenated strings, as the hot subpaths produced solely by a

Table 1. Characteristics of benchmark programs. The first column lists their (uninstrumented) running time. The second column lists the size of the acyclic trace file. The third column is the rate at which this file is produced. The fourth column is the size of a textual representation of the WPP. The fifth column is the rate at which the WPP is produced. The sixth column is the compression ratio. The seventh column lists the number of threads run by each program. The next column is the number of acyclic paths executed by all thread. The following column is the number of rules needed to describe the control flow. The final column is the number of rules per executed acyclic path.

Benchmark	Time (sec)	Trace Size (MB)	Trace/ Sec	WPP Size (MB)	WPP/ Sec	Trace/ WPP	Num Threads	Num Acyclic Paths	Num Rules	Rules/ Path
099.go	90.1	2176.6	24.15	141.1	1.57	15.4	1	17,321	2,760,820	159.4
124.m88ksim	3.0	115.0	38.33	0.3	0.10	392.8	1	1,169	7,927	6.8
126.gcc	9.0	254.3	28.25	23.7	2.64	10.7	1	20,739	489,287	23.6
129.compress (train)	0.0	8.3	22230.90	0.2	632.59	35.1	1	364	5,857	16.1
130.li	4.0	300.4	75.08	2.6	0.64	116.9	1	966	62,076	64.3
132.ijpeg	3.0	47.8	15.94	6.6	2.19	7.3	1	1,637	136,816	83.6
134.perl (jumble)	17.0	605.0	35.59	15.0	0.88	40.3	1	2,115	238,893	113.0
147.vortex	48.0	1598.8	33.31	6.6	0.14	241.9	1	5,310	136,269	25.7
SQL	120.0	628.7	5.24	21.1	0.18	29.9	22	193845	404110	2.6
WinWord	8.0	73.3	9.20	6.8	0.85	10.8	4	54254	139073	2.7

descendent node are found by a recursive call.

Consider a node N (Figure 6). Viewed as a grammar, each of its successors (i.e. rules) produces a string (consisting of acyclic paths). Let P_i be the L-limited prefix of the string derived by successor i and let S_i be the L-limited suffix of the string. An L-limited string is a string containing L or fewer symbols. A node's prefix is the first L symbols that it produces, and its suffix is the last L symbols.

Note that a node's prefix and suffix are independent of its parents. In particular, the algorithm computes only once the prefix and suffix of a node with multiple predecessors—including multiple edges from the same node—which preserves the space and time benefits of the DAG representation.

In the example, the L-limited subpaths for node N are found in the strings: $S_1||P_2$, $S_2||P_3$, and possibly $S_1||P_2||P_3$ (if the string produced by the second successor is shorter than L symbols). The operator || is string concatenation. Similarly, the L-limited prefix and suffix of node N are the first and last L symbols examined when looking for substrings at node N.

This approach finds non-minimal subpaths. For example, if the suffix of a node ends with a hot subpath, it will be extended with symbols from the prefix of the next node. Changing the definition of a suffix corrects this problem. A node's suffix is the maximal suffix of the final L symbols that is not part of a hot subpath. The algorithm clears the subpath string when a hot subpath is found, so that this string (which becomes the node's suffix) only contains symbols encountered after the last hot subpath. Similarly, a node's prefix subpath is maximal prefix of the first L symbols that is not part of a hot subpath. The algorithm freezes a node's prefix string at the first subpath.

Since subpaths are limited to length L or less, the amount of work performed at a node is proportional to the number of its successors. The algorithm in Figure 7 traverses each edge in the WPP once and performs at most L operations per edge, so its running time is $O(EL)$, where E is the number of edges in the WPP. In the worst case, the space used by this algorithm could be $O(NL)$, where N is the number of nodes in the WPP. However, there is no need to retain prefix and suffix strings for nodes whose predecessors have all been visited, and the code frees and reuses this space. In this case, the space requirement is proportional to the number of partially visited nodes in the DAG, which can be far lower than $O(N)$.

5. Performance

This section describes an implementation of whole program profiling that demonstrates that the technique is practical, even for large commercial applications. The application programs were instrumented with a version of the PP path profiler [6] running on Microsoft's Vulcan tool. Vulcan is an executable instrumentation system similar to ATOM and EEL [17, 27]. Traces were processed by PPCompress, which uses the techniques described in this paper to produce and analyze a WPP.

The overhead of path profiling instrumentation and WPP processing overhead are moderate (small integer slowdown and tens of minutes of processing time). To facilitate experimentation, path traces were written to a file, rather than processed on line. Measurements were performed on a dual processor, 200 MHz Pentium-Pro PC with 256 MB of memory running Windows NT 4.0 Server (SP4).

This paper contains measurements of the SPECINT95 benchmarks and two Microsoft application programs. The first is a relational database (Microsoft SQL 7.0) running the TPC-C benchmark. TPC-C is an on-line transaction processing benchmark that involves a mix of five concurrent transactions of different types and complexity executed either on-line or queued for deferred execution [13]. The database is comprised of nine types of records with a wide range of record and population sizes. The benchmark runs for a fixed length of time, in this case a short (non-standard) run of 120 seconds. Note that the instrumented database accomplished far less in this interval than the original code (136 and 2133 transactions, respectively). The second example is a word processing program (Microsoft WinWord 9) running a standard breadth test scenario, which exercises approximately 20% of its code. On the system above, WinWord runs the uninstrumented scenario in approximately 8 seconds.

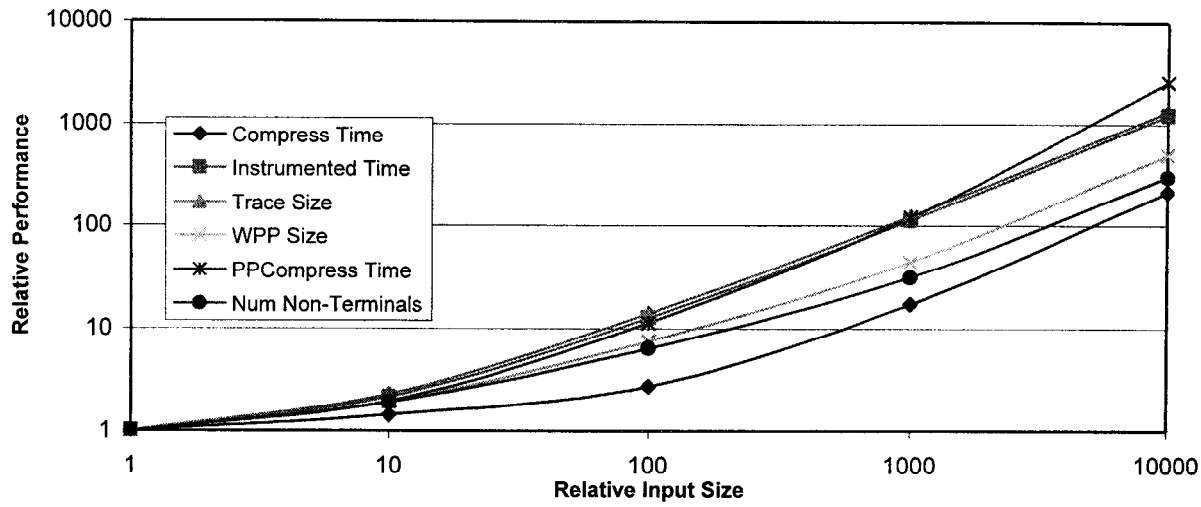

Figure 8. Whole Program Path performance running compress benchmark with various size input files.

5.1 Profile Size

Table 1 reports some overall characteristics of program traces and WPPs. The column labeled *Trace Size* contains the size of the binary file trace of acyclic paths (Section 2). *WPP Size* contains the size of the ASCII grammar produced by *PPCompress* (the binary representation of a WPP can be two times smaller). The ratio of these two files' size is a rough measure of the compression achieved by WPPs.

The SPEC benchmarks were run with their smallest input dataset (*test*), except for 129.compress, which used the more reasonable *train* dataset. 134.perl reports the larger of its two data sets (jumble). In all cases, traces include the standard libraries.

The two commercial applications differ slightly. In both, only the application code—not library code—was measured. WinWord spends a substantial fraction of its time in library (DLL) and kernel code, neither of which was captured in this experiment. SQL, unlike the SPEC benchmarks, performs a substantial amount of IO, which runs in the kernel. Another major difference is that SQL executes many threads, while the SPEC benchmarks are single threaded and WinWord executes almost entirely in one thread. The current system distinguishes control flow in each thread and constructs a separate WPP for each one.

The compression ratio ranged from 7.3–392.8. The highest compression occurred in programs (124.m88ksim, 130.li, 147.vortex) whose control flow is not particularly simple. However, all three programs are highly repetitive, and perform the same task (instruction simulation, chess board search, object-oriented database queries) many times. The programs with the lowest compression (099.go, 126.gcc, 132.ijpeg, and WinWord) have complex, non-iterative control flow. 132.ijpeg differs from the other two SPEC benchmarks, as it executes few (1,637) distinct paths, but requires a relatively large number of rules to capture its control flow.

The application programs (WinWord and SQL) have far fewer rules per acyclic path than the benchmarks. This difference may arise from the structure and behavior of commercial applications, or it may be a measurement artifact due to the absence of library code. Nevertheless, he various compression ratios appear to be a plausible measure of a program's control -flow regularity, which could possibly help isolate and study areas of regular and irregular control flow.

Figure 8 examines the relationship between program running time, file size, and processing time. This experiment used the compress program from the SPEC95 benchmark suite. The size of the file to be compressed ranged between 100–1,000,000 bytes. The figure plots the relative performance of the instrumented and uninstrumented program, the size of the trace and WPP file, and the cost of running PPCompress. Note that the uninstrumented program's execution time does not increase linearly with the input size. Although the compression algorithm is linear, the cost of compressing small files is dominated by writing the program's output, which is independent of the input data. The instrumented program does not share this behavior, since its execution is dominated by writing the trace. Most important, the WPP's size grew at a slightly slower rate than uninstrumented execution time or the trace file size. Unfortunately, the time to produce the WPP grew significantly faster than the size of its input. This may reflect the non-linear components of the algorithm or cache effects.

5.2 Hot Subpaths

Figure 9 reports some hot subpaths found in the SPECINT95 and commercial benchmarks. In this experiment, the cost function for an acyclic path was the number of instructions along the path. The figure graphs the maximum length of a hot subpath (in acyclic paths) against the number of minimal hot subpaths with cost ? 10,000 and 100,000, respectively. Because the commercial benchmarks do not include paths through library code, the absolute number of paths is not comparable between the two sets of programs.

Comparing the two graphs show that the number of hot paths discovered decreases sharply as the threshold increases. The decrease ranges from 1.8 times (129.compress) to 13.9 (126.gcc). As usual, 126.gcc differs from the other SPEC benchmarks, except 132.ijpeg. gcc's decrease, however, was close to the commercial applications (9.5 and 12.8 for WinWord and SQL, respectively).

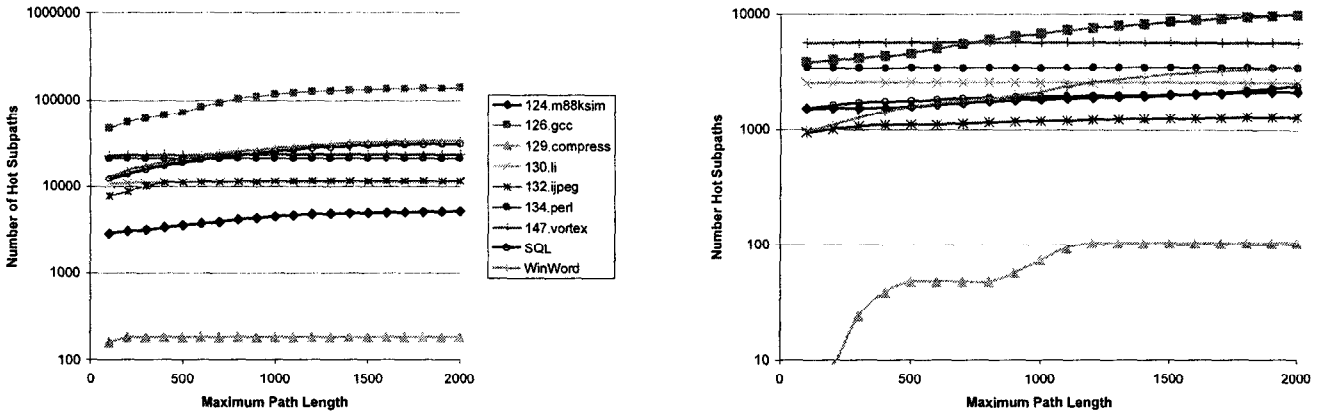

Figure 9. Number of minimal hot subpaths (with cost ≥ 10,000 and 100,000 instructions, respectively) found with different limits on the maximum length of a hot subpath (measured in acyclic paths). Subpaths are at least 10 acyclic paths long.

The shape of the curves is interesting as well. With a few exceptions, most curves are very flat. This means that few new hot subpaths were found by increasing the maximum path length beyond its initial value of 100 acyclic paths. The hot subpaths, in these benchmarks at least, are relatively short (? 100 acyclic paths), heavily executed segments of code. This, of course, is the best situation for compiler optimization, since compilers excel at small improvements, which can produce large benefits in heavily executed code. On the other hand, 126.gcc, 129.compress, and the commercial applications find 2.5–3.0 times as many paths as the length limit increases. This means that a substantial fract ion of the hot subpaths in these programs is 100–1,000 acyclic paths long. This result suggests that compiler optimization with a larger perspective might be useful for commercial applications.

6. Related Work

Ball and Larus's original path profiling algorit hm recorded the execution frequency of intraprocedural, acyclic paths [6]. This paper extends that work to paths that cross both procedure and loop boundaries. Bala's technique captured segments of interprocedural paths by recording a bounded collection of branch outcomes [5]. Unlike Bala's paths, WPPs completely cover a program's execution and do not introduce approximations at path boundaries. Moreover, the WPP representation is more compact and easily analyzable than a collection of branches.

Ammons, Ball, and Larus extended acyclic path profiling in two directions [1]. First, they associated hardware metrics other than execution frequency with paths. Second, they introduced a run-time data structure (the calling context tree) to approximate interprocedural paths by connecting a path at a call site with a path in the callee. In practice, these linkages were imprecise, as more than one path can reach a call site. Moreover, calling context trees do not connect paths across loop iterations. Overall, WPPs are more accurate, compact, and analyzable than calling context trees and capture cyclic paths that span loop boundaries. However, WPPs require more intermediate storage and post-processing.

Melski and Reps describe an interprocedural extension of Ball and Larus's acyclic path profiling technique [18, 19]. Instead of labeling edges in an interprocedural supergraph with integer values, their technique labels edges with functions, whi ch are used to capture the calling context of a procedure. Their approach shares some of the limitations of the original Ball-Larus technique. First, the paths in this technique do not cross loop (or recursive call) boundaries. Second, interprocedural paths are assigned a unique name statically. Since the number of potential paths through a program is huge, a path's run -time representation must be an unbounded integer, or potential paths will need to be truncated to limit the size of path identifiers. In some sense, a WPP is an identifier—though not a minimal one—that uniquely identifies the path that a program took. Finally, their analysis presumes a complete call graph, and introduces ad-hoc techniques to handle exceptions and indirect calls. WPPs, which start with a run-time trace, easily handles cyclic and indirect control flow, as well as complications such as multiple threads. An interesting alternative is to use Melski and Rep's technique in conjunction with the techniques in this paper. Th eir algorithm produces a different vocabulary of longer paths, which might lead to smaller grammars.

Several researchers have investigated techniques to compress program traces. For example, Larus described Abstract Execution, in which a small amount of run-time data guides the re-execution of the address-generating slice of a program [16]. Pleszkun developed a two-pass trace compression scheme, which used a variable-length encoding of a basic block's dynamic control successors and compact representation of linear address patterns to compress address traces to a fraction of bit per reference [23]. These techniques produce impressively small files, but require considerable post-processing to regenerate an address trace, which is a far less compact and analyzable entity than a WPP.

Chen et al. hypothesized that data compression provides an upper limit on the performance of correlated branch prediction [9]. This paper provides evidence to further this connection, as this type of branch prediction performs well because of programs' strong path locality [28], which also underlies the high compression achieved by the SEQUITUR algorithm.

The hot subpath algorithm in Section 4.2 is similar to Baker's technique for finding repeated code in a program [4]. Baker's algorithm uses a suffix tree of a program text. This structure is impractical for program traces, as it uncompresses the trace. WPP's DAG representation is far more compact, yet still

analyzable. Baker's technique, moreover, finds all repetitions, regardless of length. In this application, repetition is only valuable when costly, but Baker's approach does not support a cost metric.

7. Conclusion and Future Work

Whole program paths are a new representation for dynamic program analysis that capture a program's complete control flow in a compact, tractable form. A WPP is a DAG representation of a context-free grammar that generates a program's acyclic path trace. A two-step process produces a WPP. First, the acyclic paths that a program executes are recorded. Next, this trace is processed with the SEQUITUR compression algorithm, which builds a context-free grammar to represent its input string. A grammar's DAG representation is a WPP, which is a compact and easily analyzed representation of a program's control flow. This paper shows how to find hot subpaths in a WPP and demonstrates that the SPEC benchmarks and commercial applications contain a significant number of these paths.

WPPs have many potential uses. This paper concentrated on their application to performance tuning, in which WPPs identify heavily executed code sequences. Programmers or compilers could collect and analyze these WPPs to find hot subpaths to optimize or tune. Because WPPs span procedure and loop boundaries, they expose large-scale optimization opportunities that cross procedure and module abstractions. Without automatic tools to identify expensive interprocedural paths, large-scale performance tuning will remain difficult, costly, and limited to high value software, such as OSs and DBs. Moreover, the long paths identified by WPPs are valuable adjuncts to the global and interprocedural optimization that is becoming necessary to support highly speculative or VLIW microprocessors.

Another, more novel application of WPPs is to detect program errors that do not manifest themselves as erroneous output. Consider the problem of data structure initialization. A program may run correctly when it allocates an uninitialized structure in zeroed memory, but fail when it puts the structure into recycled memory. A similar error is accessing shared structures without acquiring the proper synchronization. This error too may manifest itself only under certain conditions. In some cases, these errors are detectable by examining a program's control flow. The idea has been used in predicate path expressions to specify synchronization constraints [3]. However, temporal logic, as used in model checking [10], offers a better language for expressing control-flow properties to validate.

Moreover, it seems likely that the same compression technique and data representation can be use to capture and analyze programs' data -reference patterns, as well as their control flow.

Acknowledgements

Christopher Fraser pointed out the SEQUITUR algorithm. Julian Burger, Vinod Grover, David Melski, and Tom Reps provided many helpful comments. The anonymous referees also provided unusually detailed and helpful feedback.

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.* Las Vegas, NV, 1997, pp. 85-96.

[2] 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.* Montreal, Canada, 1998, pp. 72-84.

[3] S. Andler, "Predicate Path Expressions," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages.* San Antonio, Texas, 1979, pp. 226-236.

[4] B. S. Baker, "Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance," *SIAM Journal of Computing*, vol. 26, pp. 1343-1362, 1995.

[5] V. Bala, "Low Overhead Path Profiling," Hewlett Packard Labs 1996.

[6] 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.

[7] T. Ball, P. Mataga, and M. Sagiv, "Edge Profiling Versus Path Profiling: the Showdown," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* San Diego, CA, 1998, pp. 134-148.

[8] R. Bodik, R. Gupta, and M. L. Soffa, "Refining Data Flow Information using Infeasible Paths," in *Proceedings of the ACM SIGSOFT Fifth Symposium on the Foundations of Software Engineering.* Zurich, Switzerland, 1997.

[9] I.-C. K. Chen, J. T. Coffey, and T. N. Mudge, "Analysis of Branch Prediction via Data Compression," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems.* Cambridge, MA, 1996, pp. 128-137.

[10] E. N. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, pp. 244-263, 1986.

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

[12] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau, "Parallel Processing: A Smart Compiler and a Dumb Machine," in *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.* Montreal, Canada, 1984, pp. 37-47.

[13] J. Gray, "The Benchmark Handbook for Database and Transaction Processing Systems," in *The Morgan Kaufmann Series in Data Management Systems*, J. Gray, Ed., second ed. San Francisco: Morgan Kaufmann, 1993.

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

[15] Q. Jacobson, E. Rotenberg, and J. E. Smith, "Path-Based Next Trace Prediction," in *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture.* Research Triangle Park, NC, 1997.

[16] J. R. Larus, "Abstract Execution: A Technique for Efficiently Tracing Programs," *Software--Practice and Experience*, vol. 20, pp. 1241-1258, 1990.

[17] J. R. Larus and E. Schnarr, "EEL: Machine-Independent Executable Editing," in *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*. La Jolla, CA, 1995, pp. 291-300.

[18] D. Melski and T. Reps, "Interprocedural Path Profiling," Computer Sciences Department, University of Wisconsin-Madison, Technical Report TR-1382, September 1998.

[19] D. Melski and T. Reps, "Interprocedural Path Profiling," in *Proceedings of CC '99: 8th International Conference on Compiler Construction*. Amsterdam, The Netherlands, 1999.

[20] D. Mosberger and L. L. Peterson, "Making Paths Explicit in the Scout Operating System," in *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, WA, 1996, pp. 153-167.

[21] C. G. Nevill-Manning and I. H. Witten, "Compression and explanation using hierarchical grammars," *The Computer Journal*, vol. 40, pp. 103-116, 1997.

[22] C. G. Nevill-Manning and I. H. Witten, "Linear-time, incremental hierarchy inference for compression," in *Proceedings of the Data Compression Conference (DCC '97)*. Snowbird, UT: IEEE Computer Society, 1997, pp. 3-11.

[23] A. R. Pleszkun, "Techniques for Compressing Program Address Traces," in *Proceedings of the 27th Annual IEEE/ACM International Symposium on Microarchitecture*, 1994, pp. 32-40.

[24] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang, "Optimistic Incremental Specialization: Streamlining a Commercial Operating System," in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*. Copper Mountaint Resort, CO, 1995, pp. 314-324.

[25] 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.

[26] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace Processors," in *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*. Research Triangle Park, NC, 1997, pp. 138-148.

[27] A. Srivastava and A. Eustace, "ATOM A System for Building Customized Program Analysis Tools," in *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*. Orlando, FL, 1994, pp. 196-205.

[28] 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.