The Design and Evaluation of Path Matching Schemes on Compressed Control Flow Traces

Yongjing Lin [a], Youtao Zhang [b,*], and Rajiv Gupta [c]

[a] *Computer Science Department, University of Texas at Dallas, Richardson, TX 75083*

[b] *Computer Science Department, University of Pittsburgh, Pittsburgh, PA 15260*

[c] *Computer Science Department, University of Arizona, Tucson AZ 85721*

Abstract

A control flow trace captures the complete sequence of dynamically executed basic blocks and function calls. It is usually of very large size and therefore commonly stored in compressed format. On the other hand, control flow traces are frequently queried to assist program analysis and optimization, e.g. finding frequently executed subpaths that may be optimized. In this paper, we identify path interruption and path context problems in querying an intraprocedural path over control flow traces. While algorithms that perform pattern matching on compressed strings have been proposed, solving new challenges requires the extension of traditional algorithms. We design and evaluate four path matching schemes including those that match in the compressed data directly and those that match after decompression. In addition, simple indices are also designed to improve matching performance. Our experimental results show that these schemes are practical and can be adapted to environments with different hardware settings and path matching requests.

Key words: Path Matching, Sequitur Compression, Control Flow Trace

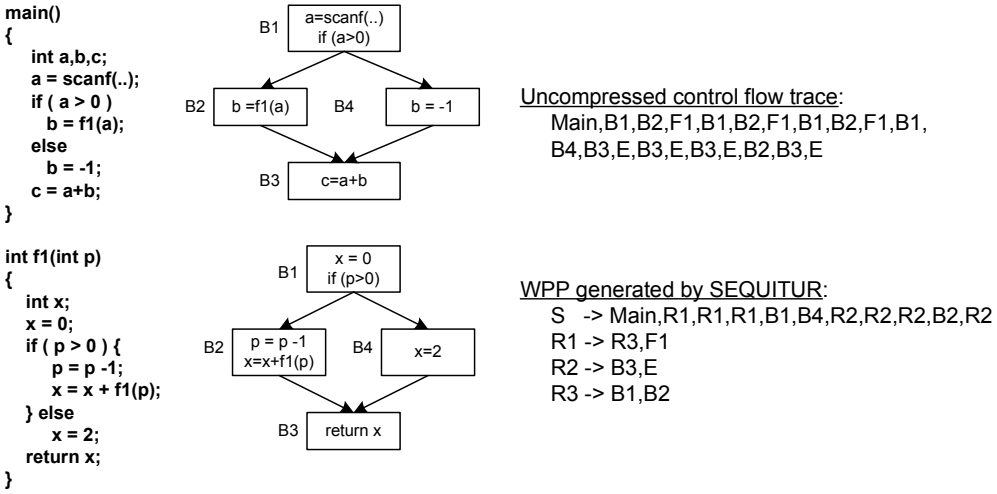
1 Introduction

Data streams with large amounts of data are often stored in compressed form using common compression algorithms such as Lempel-Ziv family (Ziv and Lempel, 1977; Ziv and Lempel, 1978) and SEQUITUR (Nevill-Manning and Witten, 1997). When the need to search for a pattern in the data stream arises, it is highly desirable to avoid uncompressing the data. Therefore researchers have been developing algorithms for pattern matching that operate directly on compressed data (Amir and Benson, 1992; Amir et al., 1996; Kida et al., 1999; Mitarai, 2001; Navarro et al., 2001).

In the program analysis and optimization community, program execution traces especially control flow traces are often collected to assist analysis, debugging and optimization (Ball and Larus, 1996; Larus 1999; Zhang and Gupta, 2001). Since a control flow trace captures the complete sequence of all executed basic blocks and function calls, it is usually very large ranging from hundreds of megabytes for a moderate to several gigabytes for a long run. Recently Larus (Larus 1999) proposed to compress a control flow trace using SEQUITUR algorithm which is proven effective in reducing the size. The compressed form of a control flow trace produced by SEQUITUR is referred to as a *whole program path* (WPP). In addition, control flow traces are frequently mined for subpaths that are commonly followed by the program. This information is useful for both program understanding and optimization because it has been observed that while large programs contain millions of paths, only a few thousand are observed to be taken by the program in practice.

Searching for an occurrence of a path in the WPP poses unique challenges. The applications of WPPs require searching for an *intraprocedural path*. However, if an intraprocedural path contains procedure calls, it may be *interrupted* by paths belonging to called procedures. Therefore a match in the WPP may be separated by items from its nested function calls. Moreover, the same path may be generated in different *contexts*, that is, during the execution of different procedures. This is because each procedure reuses the same basic blocks identifiers. As a result, a literal match may not be counted if it is from a different function call. Due to the *path interruption* and *path context* problems, existing pattern matching algorithms, such as the one in (Mitarai, 2001), are not directly applicable for finding a path in a WPP. In this paper we develop a set of *path matching* algorithms that operates on WPPs. We analyze their complexity and present experimental data that shows that our algorithms are effi-

* Corresponding author.
Email address: zhangyt@cs.pitt.edu (Youtao Zhang).



(a) Sample Code and its CFG. (b) Control Flow Trace and WPP.

Fig. 1. An Example.

cient in practice.

For the rest of the paper, Section 2 discusses the related work. Section 3 defines the problem. Section 4 presents the design and analysis of four path matching algorithms. Experimental results are collected and discussed in Section 5. Section 6 concludes the paper.

2 Related Work

The traditional string matching problem is to find the first or all occurrence of a pattern P in some text string T. The widely used Knuth-Morris-Pratt algorithm (Knuth et al., 1977) requires O(n+m) operations in the worst case where m is the length of P and n is the length of T. The Boyer-Moore algorithm (Boyer and Moore, 1977) improves the performance to O(n/m) in some cases while it shares the same worst case performance. The Aho-Corasick algorithm can find any one from a list of pattern strings in O(mx+n) where mx is the sum of lengths of all pattern strings by constructing a failure function similar to the KMP algorithm.

Data streams of large sizes are usually compressed using different compression algorithms e.g. Lempel-ZIP family (Ziv and Lempel, 1977; Ziv and Lempel, 1978) and Sequitur (Nevill-Manning and Witten, 1997). Matching in the compressed text without decompression has been introduced (Amir and Benson, 1992) and throughly studied later on (Amir et al., 1996; Farach and Thorup, 1998; Navarro and Raffinot, 1999; Kida et al., 1999; Mitarai, 2001). Simulating the KMP algorithm over LZ78 compressed text can solve the occurrence problem in $O(m^2+n)$ (Amir et al., 1996). A randomized algorithm proposed later on (Farach and Thorup, 1998) solves the same problem over LZ77 in $O(m+n\log^2(u/n))$. It was then generalized to search over Ziv-Lempel compressed texts for simple and extended patterns (Navarro and Raffinot, 1999). By introducing a collage system as a unifying framework, (Kida et al., 1999) designed a generalized compressed pattern matching algorithm on the collage system. Practical timing study of compressed pattern matching algorithms was reported in (Mitarai, 2001). The results show that matching directly over Sequitur-compressed texts is 1.27 times faster than a decompression followed by an ordinal search.

SEQUITUR algorithm was proposed to compress text strings using context free grammars (Nevill-Manning and Witten, 1997). It is proven effective in compressing strings with small alphabet and usually achieves better compression ratio in these cases. Larus proposed whole program path (WPP) (Larus 1999) which compresses a control flow trace using SEQUITUR. While the algorithm to identify most hot paths is given in (Larus 1999), the problem to find if a given path appears in the WPP has not been solved.

String matching has been extended to other areas as well. Quantified inexact matching approaches have been recently proposed for model specification (Zhuge, 2003). The goal is to achieve better repository-based model reuse.

3 Problem Definition

Of all different kinds of program traces that exist, a control flow trace is the most commonly used form because of the ease with which it can be collected and the variety of ways in which it can be used. Therefore we consider the form of the control flow trace in this paper.

Consider a program consisting of the `main` function and several other functions. Each function is represented in the form of a directed graph called the *control flow graph* (CFG). Each node in a CFG represents a *basic block* which is a straight line sequence of statements that can be entered only from the beginning and exited only from the end. The edges

in a CFG capture the flow of control among basic blocks. The control flow trace consists of a sequence of basic block ids that are executed during one program run from the start to the end. In addition, it also contains indicators that identify entry and exit to a function. An entry is indicated by the appearance of the function name in the control flow trace while the exit point is universally identified by E.

3.1 The SEQUITUR Algorithm

The SEQUITUR algorithm forms a grammar from a sequence based on the repeated phrases in that sequence (Nevill-Manning and Witten, 1997). It creates grammar rules for these repetitions and replaces the repetitions with non-terminal symbols, producing a more concise representation of the overall sequence (Nevill-Manning and Witten, 1997). Using SEQUITUR a control flow trace is compressed into a whole program path (WPP) (Larus 1999). The grammar of the WPP generates a single string which is the uncompressed control flow trace. Figure 1 shows an example program, its CFGs, a sample uncompressed control flow trace and the corresponding WPP.

The SEQUITUR algorithm constructs the grammar based on the following two properties.

- Diagram uniqueness. A diagram is defined as a pair of adjacent symbols. This property ensures that every diagram in the grammar be unique. For example, the diagram of B_3 and `E` in Figure 1 appears once in the rule `R2`. If it occurs elsewhere in the sequence, a new rule will be formed and replace the diagram with its new left-hand-side non-terminal symbol at both places.
- Rule utility. This property states that every rule is used more than once in the grammar. In constructing the grammar, it is possible that after creating a new rule, the appearance of another non-terminal symbol is reduced to one. For example, if we have
  `S->AbA, A->aa`
  and the next symbol is `b` (where `S, A` are non-terminal symbols, and `a, b` are terminal symbols), we create a new rule `B->Ab` and get
  `S->BB, B->Ab, A->aa`.
  Since `A` is used only once, it is eliminated such that we have
  `S->BB, B->aab`.

3.2 New Problems

A WPP may be queried for different kinds of information during program analysis. Here are some examples of typical queries: *Does the path $B_1B_2B_3$ of F_1 appear in this trace?*; and *How many times does path $B_1B_2B_3$ of F_1 appear in the trace?*. In this paper we will present a path matching algorithm which can serve as the basis for answering different forms of queries. Specifically the algorithm that we present solves the following path matching problem:

Find the first occurrence of path P generated by function F in a given WPP.

The solution to the above path matching must tackle two main problems that are discussed next. These problems distinguish path matching from other pattern matching algorithms in the literature.

- *Path Interruption Problem.* Even if an intraprocedural path is executed, the sequence of block ids on the path may not appear in sequence in the control flow trace. In particular, if the blocks contain function calls, then the sequence will be interrupted by appearance of block ids corresponding to the called functions. For example, the path $B_1B_2B_3$ of function F_1 was executed 2 times, but no such sequence appears in the control flow trace.
- *Path Context Problem.* While globally unique block ids could be assigned to the basic blocks, this approach is not taken because the total number of basic blocks is very large and therefore globally unique ids will require a large number of bits. Instead the same block ids are reused within each function of the program. However, this also means that the same sequence of block ids that form a path can be generated by different functions. Therefore in addition to find an appearance of the block ids in a path we must also ensure that this sequence appears in the *context* of the function of interest. For example, the path $B_1B_2B_3$ can be generated by `main` or function F_1.

While it may appear that path interruption problem can be avoided by unwinding the traces, this is not a practical solution. Since unwinding must be performed before compression, the full online nature of SEQUITUR is lost. Moreover, we still must solve the path context problem. Therefore, in this paper, we do not alter the form of the control flow trace; but rather we develop solutions to the above problems.

4 Path Matching Algorithms

In this section, we design and analyze four different path matching algorithms. We begin by considering path matching in context of an uncompressed control flow trace and show some insights in how to handle the *path interruption* and *path context* problems. We then discuss path matching directly in a compressed trace and path matching with additional indices.

4.1 Scheme 1: Path Matching on Uncompressed Control Flow Traces

While a trace is stored in compressed format, it is always possible to uncompress the trace followed by a path matching in the uncompressed format.

It is useful to view the complete control flow trace as being composed of control flow subtraces corresponding to individual function invocations. The complexity of path matching arises from the fact that while we are searching for the

appearance of a path belonging to a specific function in the control flow subtrace corresponding to a function invocation, the control flow subtraces corresponding to other invocations of the same or different function may be encountered. In fact, at any given point in the complete control flow trace, multiple function invocations can be active. We refer to the control flow subtraces of these active invocations as *active subtraces*. At any point in the complete control flow trace, the trace preceding the point contains the *prefixes* of the active subtraces and the trace following the point contains the corresponding *suffixes* of these active subtraces.

To facilitate the discussion of active subtraces, we associate a *nesting level* with each entry in the control flow trace. The nesting levels of all members of a subtrace (i.e., the function name, block ids, and the end marker) are the same. Moreover this nesting level is the same as the nesting level of the corresponding function invocation in the dynamic call graph for the program run. Therefore while scanning a part of the control flow trace we can compute the relative nesting levels of the block ids encountered by incrementing the nesting level when a function name is encountered and decrementing it when an end marker is encountered. It should be noted that at any given point in the complete control flow trace the nesting levels of all active traces are distinct.

The Algorithm Based upon the above notion of nesting level we can state that a *control flow subtrace* of a function F in the complete control flow trace T has the following properties:

- A subtrace begins with an F and ends at the first $E \in T$ that appears after F such that the nesting levels of F and E are the same (say nl).
- A block id B between the above F and E belongs to the subtrace iff its nesting level is also nl.
- All occurrences of function names, end markers, and block ids that do not belong to the subtrace corresponding to F and E and appear between F and E in T have nesting levels greater than nl.

Traditional pattern matching algorithms (e.g., KMP algorithm) are based upon finite state automatons whose states indicate the matching status, that is, how much of the pattern has been seen so far. Path matching will also be based upon a finite state automaton. However, the form the automaton required is different since at each point we need to track the *matching status* of multiple *active subtraces*. Next we describe the form of the automaton appropriate for *path matching*.

We will first discuss how the previous discussed problems are solved in our automata. Instead of one integer value in the KMP algorithm, each state is a vector consisting of varying-numbered integer values. The number of entries in the tuple corresponds to the number of active traces. Each item in this tuple indicates the matching status of the corresponding active trace. If a function is called, we generate a new state value for matching these nested items. We can resume the matching after the function call as we remove the corresponding state value at the end of function call. In this way, nesting items may not disturb the matching of an intraprocedural path. Thus we solve the *path interruption problem*. We then introduce a special state value "-1" which is used to indicate that the corresponding function invocation is not interested, i.e. its function name does not match the one that we are searching for. Therefore the elements of the subtraces need not be matched with P. Once a state value is changed to "-1", it never changes till the end of the current invocation end. In this way we solve the *path context* problem.

Formally, given an uncompressed control flow trace T, a path $P_{1..m}$, a function F_i, and the value of MNL which is the *maximum nesting level* that a control flow trace can reach, the path matching automaton for $P_{1..m}$ is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where

- Q is a finite set of states such that

  $Q = \{(s_1, s_2, ..., s_l) \mid 0 \leq l \leq MNL \; and$
  $\forall i \in [1, l], -1 \leq s_i \leq m\}$.
- $q_0 \in Q$ is the start state and $q_0 = (\epsilon)$
- $A \subseteq Q$ is a distinguished set of accepting states.
- Σ is a finite input alphabet, $\Sigma = \{B_1, ..., B_{max}, F_1, ..., F_{max}, E\}$ and B_j, F_j, E denote different basic blocks, function entry points and function return points.
- δ is a function from $Q \times \Sigma \rightarrow Q$, called the transition function of M. The state transition function has the following form:
  $\delta((s_1, s_2, ..., s_l), B) = (s_1, s_2, ..., \delta((s_l), B))$
  $\delta((s_1, s_2, ..., s_l), F_i) = (s_1, s_2, ..., s_l, 0)$
  $\delta((s_1, s_2, ..., s_l), F_j) = (s_1, s_2, ..., s_l, -1)$, where $F_i \neq F_j$
  $\delta((s_1, s_2, ..., s_l), E) = (s_1, s_2, ..., s_{l-1})$
  $\delta((-1), B) = (-1)$.

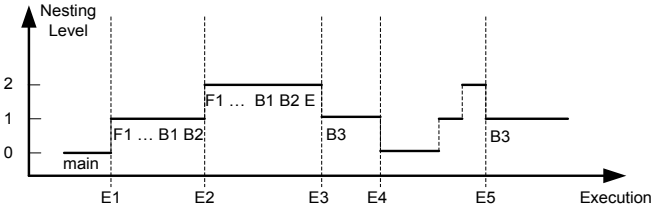
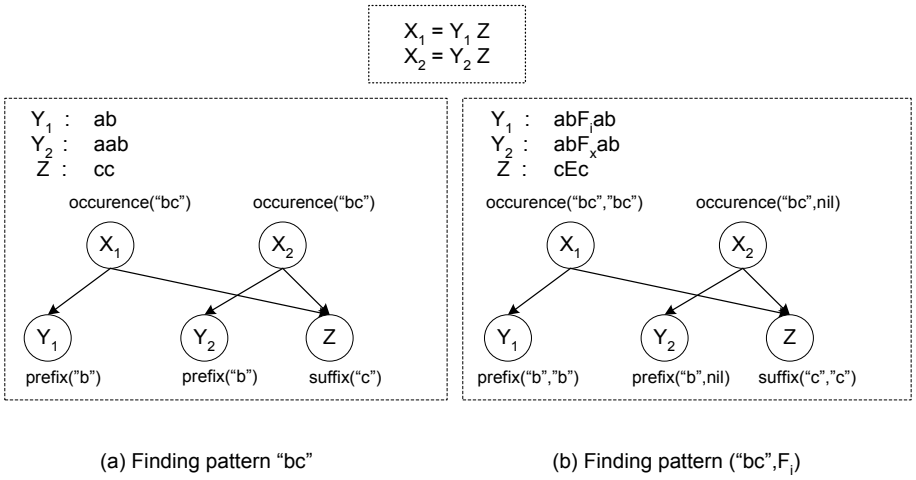


Fig. 2. An Example for Scheme 1.

An Example In Figure 2 an example execution is shown with the line to show items at the same level. The x axis shows the execution over the time while the y axis indicates the current nesting level, e.g. the execution starts with function `main` at level 0 and calls function F_1 at time E_1. It changes to level 1 at E_1. In the basic block B_2, it calls F_1 again and return to B_3 when the second invocation finishes. Assume we are searching for path $B_1B_2B_3$ in F_1. It is clear that B_3 at E_3 can be combined with B_1,B_2 before E_2 to form an instance of path $B_1B_2B_3$ in function F_1. However, it may not be combined with B_1,B_2 before E_3 as they belong to different call instances. Similarly, B_3 at E_5 should not be combined with B_1,B_2 either before E_2 or E_3.



(a) Finding pattern "bc" (b) Finding pattern ("bc",F_i)

Fig. 3. Comparing with traditional compressed matching algorithms.

Let us use Q_{2i-1} and Q_{2i} to indicate the state before and after each E_i for $1 <= i <= 5$. The state Q_3 before E_2 is (-1, 2). "-1" at level 0 means that all basic blocks from "main" are not interested as we are searching for a path in F_1. "2" at level 1 means that we have partially matched two items, i.e. B_1B_2. After E_2, the state is changed to Q_4 = δ(Q_3, F_1) = (-1, 2, 0). A new state value is added and initialized for the new function invocation. It is initialized to "0" as the function name matches F_1.

The states at E_3 before "E" is Q_x = (-1, 2, 2). With E indicating the end of the second invocation of F_1, we have Q_5 = δ(Q_x, E) = (-1, 2). With B3 after E3, we get Q_6 = δ(Q_5, B_3) = (-1, 3) which indicates a match. In addition, when we reach E_4, we have Q_8 = (-1) such that B_3 at E_5 will not match B_1B_2 at E_3. In summary, by remembering multiple active subtraces, we get the correct match as desired.

4.2 Scheme 2: Compressed Path Matching Algorithm

A WPP obtained by compressing the control flow trace using SEQUITUR (Nevill-Manning and Witten, 1997) can be represented as a set of rules denoted by R with the following form:

$$S \leftarrow X_{i_1}X_{i_2}...X_{i_n}$$
$$X_1 \leftarrow a_1; \quad X_2 \leftarrow a_2; \quad ..... \quad X_k \leftarrow a_k;$$
$$X_{k+1} \leftarrow X_{l(1)}X_{r(1)}; \quad X_{k+2} \leftarrow X_{l(2)}X_{r(2)}; \quad .....$$
$$X_{k+s} \leftarrow X_{l(s)}X_{r(s)};$$

where $\Sigma = \{a_i | 1 \leq i \leq k\}$, a_i can be one of B_j (block id j), F_j (entry to function F_j), and E (a function exit point).

In dictionary-based compressed matching algorithms (e.g., (Amir et al., 1996; Kida et al., 1999)) each non-terminal symbol in the dictionary has an associated *prefix flag*, a *suffix flag* and an *internal flag*. When combining two non-terminal symbols, their combination will decide the internal, prefix, and suffix flags of the combined node. Assume we are to find the occurrence of a subsequence "bc" in the compressed representation in Figure 3a. In the example, the suffix flag "c" at node Z indicates that when matching Z from the beginning with "bc" from the end, the longest common subsequence is "c". Similarly when matching Y_1 from the end with "bc" from the beginning, the longest common subsequence is "b" – the prefix flag of Y_1. When these two nodes combine together to form X_1, the prefix flag "b" and the suffix flag "c" can determine one occurrence of "bc" in X_1. It is then recorded in the internal flag.

However, this is not sufficient for path matching. As Figure 3b shows we can find that both "c" of Z may or may not be a part of a path occurrence. Since there are function calls (F_i) and ending items (E) involved in each rule, the previous nested calling context must be considered. Instead of a single prefix flag, we therefore associate a list of prefix flags to indicate the prefix at each different nesting level. Similarly we also must provide lists of internal and suffix flags. The size of the list is the nesting level of the associated node and it can be at most equal to the maximum nesting level MNL. Note that "nil" is different from the null string ϵ and they represent "-1" and "0" in our state automaton respectively.

The Algorithm Our algorithm has two main steps: a *preprocessing* step and a *path matching* step. The preprocessing is done in two parts. First several data structures extended from those defined in (Amir et al., 1996; Kida et al., 1999) are created and second using these data structures certain *flags* associated with the non-terminals are computed. The path matching step searches through the right hand side of the starting rule and continues the search till an occurrence of the path is found.

Let us first discuss the preprocessing algorithm. As mentioned above, flags are associated with each non-terminal whose values are computed during preprocessing based upon the path P being considered. Consider a non-terminal X in

```
Preprocessing (P, WPP) {
   Preprocess_2 P to enable flag updates
   for each non-terminal X from R in bottom-up order do
      case (X ← α) of
         α == B_i:
            if B_i ∈ P then
               prefix_x = (B_i); suffix_x = (B_i); internal_x = (B_i);
               fentry_x = 1; fexit_x = 1; freturn_x = -1;
            else
               prefix_x = (0); suffix_x = (0); internal_x = (0);
               fentry_x = 1; fexit_x = 1; freturn_x = -1;
            endif
            if (B_i == P) then foccur_x = 1;
            else foccur_x = -1 endif
         α == F_i:
            prefix_x = (ϵ,ϵ); suffix_x = (ϵ,ϵ); internal_x = (ϵ,ϵ);
            fentry_x = 1; fexit_x = 2; freturn_x = -1; foccur_x = -1;
         α == F_x, where x ≠ i:
            prefix_x = (ϵ,nil); suffix_x = (ϵ,nil); internal_x = (ϵ,nil);
            fentry_x = 1; fexit_x = 2; freturn_x = -1; foccur_x = -1;
         α == E:
            prefix_x = (ϵ,ϵ); suffix_x = (ϵ,ϵ); internal_x = (ϵ,ϵ);
            fentry_x = 2; fexit_x = 1; freturn_x = 2; foccur_x = -1;
         α == YZ:
            Compute X's level vector from Y and Z's level vectors
            Y's vector: (1, ..., y_fexit, ..., y_max); Z's vector: (1, ..., z_fentry, ..., z_max)
            Therefore X's vector is: (1, ..., x_fx, ..., x_max), where
            x_fx = max ( y_fexit, z_fentry ), x_max − x_fx = max (y_max − y_fexit, z_max − z_fentry)
            Create mapping functions fmapY/fmapZ which map levels of Y/Z to levels in X
            x_fentry = fmapY(y_fentry); x_fexit = fmapZ(z_fexit);
            if (fmapY(y_freturn) ≤ fmapZ(z_freturn) then
               x_freturn = fmapY(y_freturn);
               for each level l < x_freturn do
                  Update X's level l flags from corresponding level flags of Y and Z
               for each level l >= fmapY(y_freturn) and < fmapZ(z_freturn) do
                  Update X's suffix flag using suffix flag of Y
                  Update X's prefix flag suing prefix flag from Y and all flags from Z
                  Update X's internal flag with internal flags from Y and Z
               for each level l >= fmapZ(z_freturn) do
                  Update X's prefix flag using prefix flag for Z
                  Update X's suffix flag using suffix flag for Y
                  Update X's internal flag to nil
               if foccur_y > 0 then foccur_x = foccur_y
               else
                  Examine prefix/suffix flags of Y/Z to see if P is in YZ.
                  if occurrence of P found in YZ then set foccur_x
                  elseif foccur_z > 0 then foccur_x = |Y| + foccur_z
                  else foccur_x = -1 endif
            else /* (fmapY(y_freturn) > fmapZ(z_freturn) */
               processing is as above with minor modifications
            endif
      endcase
   endfor
}
```

Fig. 4. Preprocessing Algorithm.

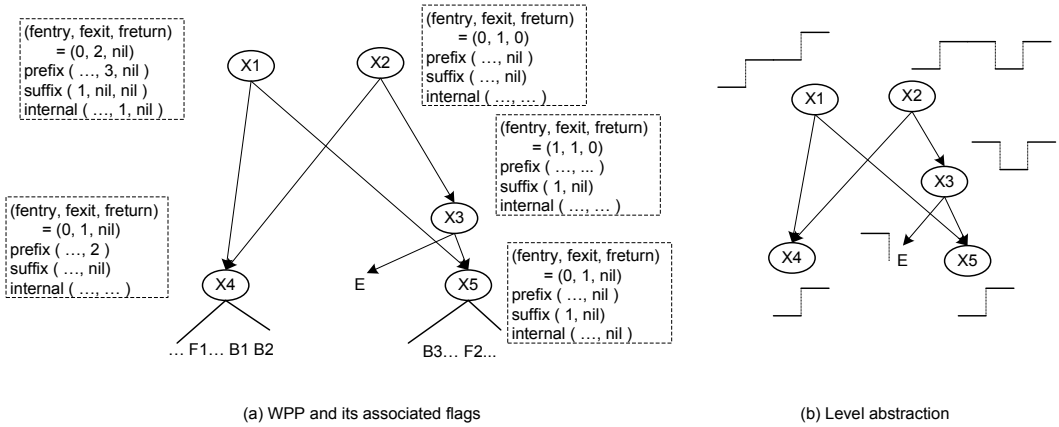


Fig. 6. An Example for Scheme 2.

```
PathMatching_2 {
  Let S ← X_{i_1} X_{i_r 2} ... X_{i_n}.
  Initialize: Z_old ← ϵ;
  for (k = 1; k ≤ n; k++) {
      Consider the rule Z_new ← Z_old X_{i_k}
      Compute the flags for Z_new from flags of Z_old and X_{i_k}
      if foccur flag of Z_new is ≠ −1 then
          PRINT(occurrence found at foccur); terminate;
      Z_old ← Z_new;
  }
  PRINT(no occurrence found);
}
```

Fig. 5. Compressed control flow trace matching algorithm

the rule set R. We denote the fully expanded string corresponding to X by $XStr$. The following flags are maintained for X.

- *fnl flag:* $XStr$ may contain terminals corresponding to multiple nesting levels. These nesting levels represented in $XStr$ are always consecutive and have the following form: $(nl+1, nl+2, \cdots, nl+fnl)$. We associate a ***fnl flag*** with X which remembers the value of fnl. When X is being processed, we associate a nesting *level vector* with it which is of the form $(1, 2, \cdots, nl)$. In other words, the level vector only represents the *relative nesting levels* of the terminals in $XStr$.
- *Level flags:* There are three level flags associated with X – *fentry*, *fexit*, and *freturn*. $fentry$ and $fexit$ are the relative nesting levels of the first and last symbols of $XStr$. $freturn$ is the minimum relative nesting level of all E's in $XStr$. For example, if X represents $aEbEcFdFgFh$ then $fentry_X = 3$, $fexit_X = 4$, and $freturn_X = 2$. When the strings of two non-terminals are combined, the $fentry$ and $fexit$ level flags of the non-terminals are used to compute the relative nesting levels of symbols in the combined string. The $freturn$ flag is used to update the $internal$, $prefix$, and $suffix$ flags which are described next.
- *Prefix flags:* This is a list of flags with *fnl* items. Let $XStr(nl)$ denote the sequence of terminals from $XStr$ that have the relative nesting level nl. The prefix flag for level nl identifies the longest prefix of path P that is also the suffix of $XStr(nl)$.
- *Suffix flags:* This is also a list of flags. A suffix flag for level nl identifies the longest suffix of path P that is also the prefix of $XStr(nl)$.
- *Internal flags:* This is also a list of flags. An internal flag for level nl identifies the substring of path P from position i to j that is identical to $XStr(nl)$.
- *Position flag:* The position flag *foccur* gives the index in $XStr$ where the first occurrence of path P ends. If P does not appear in $XStr$, then the value of $foccur_X$ is -1.

The preprocessing algorithm for a given R and P is presented in Figure 4. R can be represented by a directed acyclic graph (DAG). By reversing the edges we obtain R-DAG. The algorithm will update the flags of the nodes in the topological order generated from R-DAG. There are two types of nodes: leaves which are of the form $X \leftarrow a$; and internal nodes of the form $X \leftarrow YZ$. For leaf nodes the flags are initialized based upon a which is B_j, F_j or E. For an internal node $X \leftarrow YZ$, we first match level vectors of Y and Z, using $fexit$ flag of Y and $fentry$ flag of Z, and generate a new level vector for X. The lists of internal, prefix, and suffix flags of X are then updated using the flags from corresponding levels of Y and Z. The $foccur$ flag of X takes the value of $foccur$ flag of Y, if the latter is not -1. Otherwise we check if the concatenation of flags of Y and Z can form a new occurrence of P. If this is not the case, then $foccur$ flag of X is computed from $foccur$ flag of Z.

The path matching algorithm is given in Figure 5. It searches through the right hand side of the starting rule and continues the search till an occurrence of the path is found. The previously computed flags of R are used as well as some additional flag computations are performed as shown in Figure 5.

An Example Let us study the same example in discussing scheme 1. Assume we compress part of the WPP to the graph shown in Figure 6. X_1-X_5 represent non-terminal nodes generated by Sequitur. For example, X_4 represents a

```
PathMatch_3 (P) {
    for(i=1; i<=MAX_CHUNK_NUM; i++)
        {
            containFunc[i] = load_from_the_index();
            exitLevel[i] = load_from_the_index();
            smallesEndLevel[i] = load_from_the_index();
        }

    for(i=1; i<=MAX_CHUNK_NUM; i++) {
        /* current state is (s1, s2, ..., sn); */
        if (containFunc[i] == 0)
            { /* this block doesn't contain blocks from the given F */
                if (smallestEndLevel[i] == 0) /* no end in this Chunk */
                    ni = entryLevel[i];
                else
                    ni = smallestEndLevel[i]-1;
                Change state to (s'1, s'2, ..., s'exitLevel[i]) = (s1, ..., sni, nil, ..., nil);
            } else {
                decompress(Chunk[i]);
                for each Fi/Bi/E in Chunki
                    ... /* follow state transition algorithm in scheme 1 */
            }
```

Fig. 7. Path Matching Algorithm with Partial Decompression.

substring containing of F_1, B_1, B_2 and some other basic block ids. We abstract their nesting levels in Figure 6(b). For example, due to the function invocation (F_1), basic blocks in X_4 are divided into two levels with a jump to indicate this invocation.

In the control flow trace, only function calls and returns can change the nesting level. While splitting two consecutive basic blocks into two different grammar rules does not change their nesting levels, the relative level in each grammar rule may be different. To seamlessly recover the level information, when combining two symbols together, we match the exit level of the first symbol with the entry level of the second symbol. Depending on the entry and exit levels of each symbol, the upper level non-terminal symbol (e.g X_1 or X_2) may have different nesting levels. For example, X_4 and X_5 have 2 levels each while X_1 (which is combined from X_4 and X_5) has 3 levels. This is because the relative exit level of X_4 is at level 1 and has to match the relative entry level of X_5 at level 0. Now the relative level 1 of X_5 changes to level 2 in X_1. In general each node may have several levels with prefix, suffix and internal flags at each level.

Complexity Analysis Lets first consider the complexity of the preprocessing algorithm. First we perform the preprocessing needed to answer the queries used in updating the flags as described in (Amir et al., 1996; Kida et al., 1999; Mitarai, 2001). As shown in prior work, the time complexity of this step is $O(m^2)$ where m is the length of path P. The preprocessing carried out to update the flags processes each rule (or the corresponding node in the WPP DAG) once and during the processing all the flags for the rule are updated. The update of each flag takes constant time as that in (Kida et al., 1999); however flags are associated with each relevant nesting level. Since the number of entries in the level vector is at most MNL and the number of rules is $|R|$, the total time spent on updating the flags is $O(|R| \times MNL)$. Therefore the total preprocessing time is $O(m^2 + |R| \times MNL)$.

During path matching each symbol on the right hand side of the start production is processed and during the processing constant number of flag updates are performed. Thus the time spent on path matching is $O(|S| \times MNL)$. Combining the results of preprocessing step and path matching step we obtain the overall time complexity as $O((|R| + |S|) \times MNL + m^2)$ where MNL is maximum nesting level, $|S|$ is the length of the right hand side of the start rule, $|R|$ is the number of non-terminal symbols in R, and m is the length of path P. It is also quite straightforward to show that the space complexity of the algorithm is also similar.

4.3 Scheme 3: Path Matching with Partial Decompression

In this scheme, we first divide the trace into chunks each of which is then separately compressed. In performing the path matching, a chunk is loaded and decompressed only if it contains basic blocks from the desired function. This information is kept in a small index which is attached to head of the compressed file.

We still use gzip compression algorithm to compress the trace. Except the last chunk, all chunks are of same predefined sizes, e.g. 4M bytes. That is, in collecting the traces, a buffer of 4M bytes is used, when the buffer is full, the gzip algorithm is invoked to compress it and store the result to the disk. For each function, we maintain a bit vector whose number of bits is the same as the number of total chunks. Each bit is used to indicate if the corresponding chunk contains basic blocks from that function. For a trace of 512M bytes and 100 different functions, we divide it into 128 chunks and therefore need 128 bits for each function. The total size

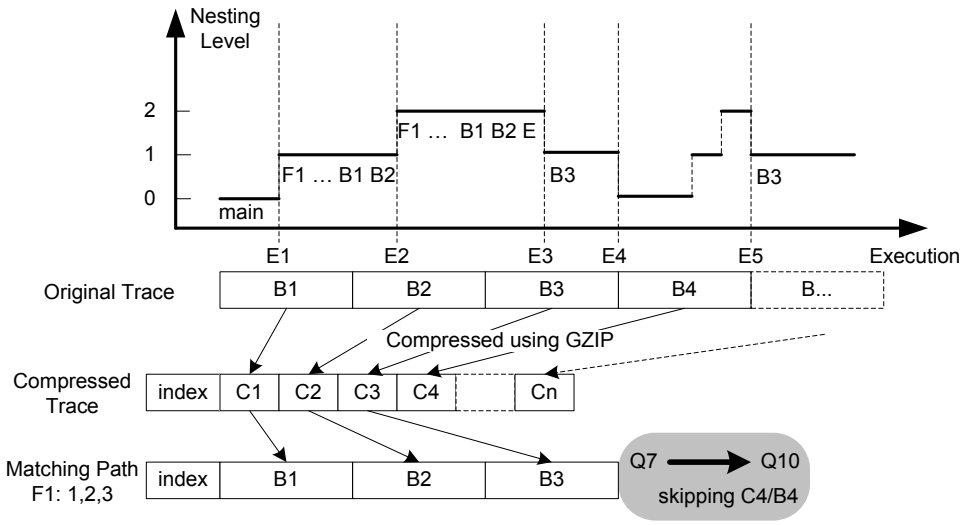


Fig. 8. Path Matching after Partial Decompression.

of the bit vector is 1600 bytes (=128 bits/function $\times$ 100 functions). In addition, each block needs to identify the exit level of its last basic block id, and the smallest level of appeared E appeared in the chunk. The exit level is useful as it helps to decide, after jumping several blocks, at which level the matching should continue. The smallest E level helps to terminate those partial subtraces. With 128 blocks and 2 bytes to identify a nesting level, this cost sums up to 128 $\times$ 2 $\times$ 2 = 512 bytes. Both overheads are very modest and can be stored in the head of the compressed trace.

The Algorithm The algorithm of performing partial path matching over uncompressed control flow trace is shown in Figure 7. The algorithm is divided into two phases. At the initialization phase, the index is pre-loaded into the memory. ContainFunc[i] indicates if Chunk$_i$ contains basic blocks from the corresponding function. exitLevel[i] indicates the exit levels of Chunk$_i$ while smallesEndLevel[i] indicates the the smallest nesting level of "E" in Chunk$_i$.

We then decompress and match sequentially on the demand. According to which function the path belongs, only a subset of all compressed chunks are loaded and decompressed. If a chunk contains interesting basic blocks, path matching within the chunk is the same as that in scheme 1. Otherwise, we directly update the current state vector without iterate over each item in the chunk. There are two impacts. First the largest level of the state vector is adjusted to the exit level of the skipped chunk. Second all state value at the level bigger than the smallest end level is set to "-1" indicating the start of an irrelevant function invocation.

An Example In the example shown in Figure 8, the original trace is divided into blocks B_1, B_2, ... etc. They are compressed to C_1, C_2, ... using GZIP algorithm. At the step to match a path in function F (which appears only in chunk 1 to chunk 3), we just need to load C_1-C_3 and decompress them accordingly. The matching is then performed in B_1-B_3 and reports the occurrences of this path if found. In skipping the resting chunks, e.g. chunk 4, we update the state vector directly using the nesting level information. As we have containFunc[4] = 0, exitLevel[4] = 2, smallestEndLevel = 0 (we assume the end item E at E4 is contained in chunk 3), The state after skipping the chunk is $Q_{10} = \delta(Q_7, Chunk_4) = \delta((-1),Chunk_4) = (-1, -1, -1)$.

4.4 Scheme 4: Compressed Path Matching with Partial Matching

The property that only partial trace needs to be searched can also be exploited by compressed matching algorithms. We next illustrate how to design the index information for fast processing without decompressing the trace first.

Using SEQUITUR algorithm, the trace is compressed into a set of rules. It is possible to design a bit vector (index) for each rule which records if the corresponding rule contains basic blocks from each appeared function or not. However it is not desirable in practice as the overhead in keeping the index information per rule based for a large number of rules is large and significantly worsen the performance (compression ratio). On the other hand, we observed that the rule base using the SEQUITUR algorithm exhibits the following property. Its starting rule usually contains a large of right hand side (RHS) items while other rules have small number of RHS items. For example, in a typical compressed trace – the compressed trace of `gcc` program, the former is in range of the number of rules while the latter is less than ten. We therefore propose to divide the RHS of the first rule into chucks. Index information is kept for each chunk of the first rule. In the experiments, we divide the first rule up to 128 segments. Similar to scheme 3, each function is associated with a bit vector where each bit is used to represent if the corresponding block in the first rule can derive basic blocks from that particular function. In addition we also need to remember the exit block level and the smallest end item level

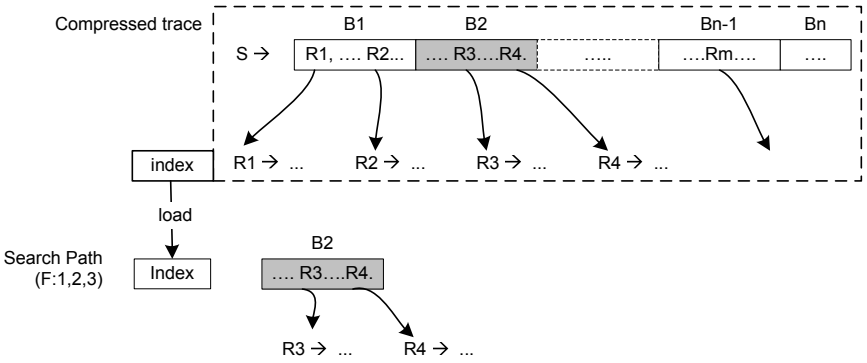


Fig. 9. Path Matching of Partial Compressed Trace.

```
PathMatch_4 (P) {
    for(i=1; i<=MAX_CHUNK_NUM; i++)
        {
            containFunc[i] = load_from_the_index();
            exitLevel[i] = load_from_the_index();
            smallestEndLevel[i] = load_from_the_index();
        }

    for(i=1; i<=MAX_CHUNK_NUM; i++) {
        /* current state is (s1, s2, ..., sn); */
        if (containFunc[i] == 0)
            { /* this block doesn't contain blocks from the given F */
                if (smallestEndLevel[i] == 0) /* no end in this Chunk */
                    ni = exitLevel[i-1]; /* exitLevel[0] is initialized to 0 */
                else
                    ni = smallestEndLevel[i]-1;
                Change state to (s'_1, s'_2, ..., s'_exitLevel[i]) = (s_1, ..., s_ni, nil, ..., nil);
                Update prefix, suffix, internal flags directly for CX_i.
            } else {
                PreProcessing_2(P, CX_i); /* Process the chunk using scheme 2 */
            }
            Update the current state using Z_new ← Z_old CX_i
```

Fig. 10. Path Matching Algorithm with Partial Matching.

in each chunk. The index size is therefore the same as that in scheme 3.

An Example In the example shown in Figure 9, let us assume only R3 can derive items from function F which results in that only chunk 2 is marked. This speeds up matching process as we just need to search rules in or derived from this chunk. While some extra (and thus useless) rules such as R4 may be processed as well, we still gain a lot as R1, R2, and many other rules are skipped.

The Algorithm The algorithm of performing partial path matching over compressed trace is shown in Figure 10. Conceptually, the algorithm replace the first rule from

$$S \leftarrow X_{i_1} X_{i_2} ... X_{i_n}$$

to

$$S \leftarrow CX_1 CX_2 ... CX_n$$
$$CX_1 \leftarrow X_{i_1} ... X_{i_{50}} \qquad CX_2 \leftarrow X_{i_{51}} ... X_{i_{100}}$$
...

where we assume each chunk contain 50 RHS items of the first rule in this case. In processing the rewritten first rule, if a chunk does not contain basic block from the given function, using the index information we generate the prefix, suffix, internal and nesting level flags without scanning the items in the corresponding chunk. Otherwise, we follow the algorithm given in scheme 2 to update these flags.

5 Experiments

To study the effectiveness of our proposed path matching schemes, we implemented and evaluated them with several programs from SPECint92, SPECint95 and SPECint2k benchmark suites (SPEC, 1995). The control flow traces are collected using Trimaran compiler infrastructure (Trimaran). As shown in Fig. 11, they cover a range of traces with different characteristics – the trace for 126.gcc contains a large number of different functions; the trace for 130.li has deep nested function calls; the trace for 026.compress has small numbers of functions and nesting levels; 197.parser has a long trace. The experiments are done on a Pentium IV

Benchmark	Gzip (MB)	Gzip with Blocks (MB)			Gzip Index (KB)	Sequitur (MB)	Sequitur Index (KB)
		80M Block	40M Block	4M Block			
008.espresso	3.24	3.25	3.23	3.23	2.6	3.74	2.6
026.compress	1.02	1.06	1.03	1.02	0.1	1.72	0.1
126.gcc	35.67	35.68	35.78	36.32	17.4	11.43	17.4
130.li	1.96	1.97	1.97	1.99	1.9	0.59	1.9
164.gzip	16.41	16.35	16.35	16.33	0.83	17.74	0.83
175.vpr	17.24	17.26	17.27	17.25	2.55	18.31	2.55
197.parser	19.11	19.13	19.14	19.14	4.22	7.52	4.22
256.bzip2	4.71	4.77	4.73	4.73	0.44	3.34	0.44

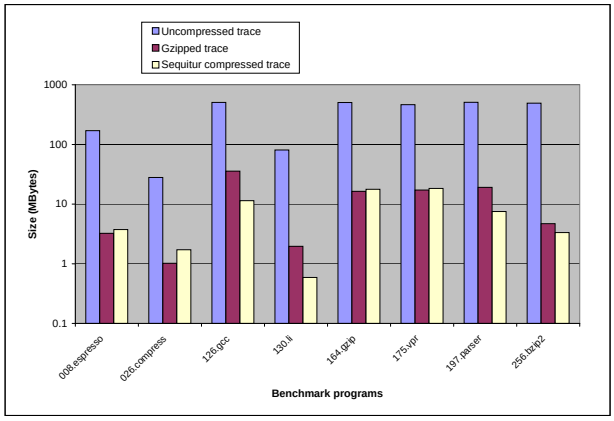


Fig. 12. The Size of Compressed Traces.

Benchmark	Control Flow Trace Size (MB)	Max Nesting Level	Function Number
008.espresso	170.3	32	164
026.compress	27.9	4	7
126.gcc	507.0	91	1085
130.li	80.9	424	121
164.gzip	504.4	9	49
175.vpr	463.6	9	150
197.parser	509.2	61	248
256.bzip2	492.7	7	26

Fig. 11. Benchmark Characteristics.

2.0GHz machine with 256MB memory with Linux Redhat 9.0 installed.

5.1 The Size of Compressed Traces

We first studied the compression sizes using different schemes. In general, GZIP and SEQUITUR achieved similar compression ratios in compressing control flow traces. When the trace exhibits more regularity, such as 130.li and 197.parser, SEQUITUR performs better. This is due to the fact that more rules can be reused and thus the size increase is slow in SEQUITUR result.

We also collected the results for scheme 3 which divides the trace into blocks and then individually compress each block using GZIP. We can see that the total size is about the same as the one without blocks.

Figure 12 also shows the index size stored in the compressed traces. As discussed in section 3, the trace is divided up to 128 chunks with a bit vector of a 16-byte vector stored for each function in the benchmark. Each bit indicates if the corresponding block contains basic blocks from the corresponding function. In addition, the exit and smallest end levels of each chunk is also kept in the index. As shown in the figure, compared to the total size, the index size is very small and has a negligible increase in size. As discussed, the indices for scheme 3 and 4 are the same.

5.2 Path Matching Performance

For all four schemes discussed in the paper, we evaluated path matching performance with three types of paths. The searching time results are summarized in Fig. 13 and compared in Fig. 14. In the figure, Y axis is searching time in

Benchmark	Paths	Scheme 1 (sec)	Scheme 2 (sec)	Block Coverage	Scheme 3 (sec)	Scheme 4 (sec)
008.espresso	Type 1	0.900 + 0.366	0.373	1/44	0.021 + 0.008	0.001
	Type 2	0.900 + 0.392	0.348	20/44	0.430 + 0.187	0.155
	Type 3	0.900 + 0.336	0.360	41/44	0.839 + 0.313	0.312
026.compress	Type 1	0.183 + 0.060	0.086	1/7	0.026 + 0.009	0.001
	Type 2	0.183 + 0.055	0.083	4/7	0.105 + 0.039	0.056
	Type 3	0.183 + 0.054	0.077	7/7	0.183 + 0.056	0.078
126.gcc	Type 1	4.134 + 1.141	1.305	1/127	0.033 + 0.009	0.003
	Type 2	4.134 + 1.149	1.432	63/127	2.051 + 0.570	0.937
	Type 3	4.134 + 1.136	1.323	126/127	4.130 + 1.130	1.135
130.li	Type 1	0.497 + 0.194	0.347	1/21	0.024 + 0.009	0.001
	Type 2	0.497 + 0.205	0.352	4/21	0.095 + 0.039	0.155
	Type 3	0.497 + 0.174	0.341	21/21	0.497 + 0.174	0.341
164.gzip	Type 1	4.112 + 0.917	1.397	1/127	0.032 + 0.007	0.013
	Type 2	4.112 + 1.610	1.395	68/127	2.202 + 0.862	0.952
	Type 3	4.112 + 0.975	1.397	126/127	4.080 + 0.967	1.392
175.vpr	Type 1	3.778 + 0.922	1.867	1/116	0.032 + 0.008	0.014
	Type 2	3.778 + 0.855	1.680	80/116	2.606 + 0.590	1.501
	Type 3	3.778 + 0.894	1.708	93/116	3.029 + 0.717	1.655
197.parser	Type 1	4.152 + 1.295	2.363	1/128	0.032 + 0.010	0.062
	Type 2	4.152 + 1.281	2.353	82/128	2.660 + 0.821	0.338
	Type 3	4.152 + 1.348	2.351	116/128	3.763 + 1.221	1.826
256.bzip2	Type 1	3.784 + 0.892	0.266	1/124	0.031 + 0.007	0.040
	Type 2	3.784 + 1.162	0.269	18/124	0.549 + 0.169	0.112
	Type 3	3.784 + 2.107	0.267	97/124	2.960 + 1.649	0.194

Fig. 13. The Performance of Path Matching in Compressed Traces.

shown in logarithmic scale while X axis list different benchmark programs. The four bars show the results of different path matching schemes respectively.

Path type 1 has low frequency and coverage. When we divide the whole trace into 4M blocks, it appears in only 1 block. Path type 2 has modest frequency and coverage. It appears in roughly half of all blocks. Path type 3 has high frequency and coverage. It appears in nearly all blocks. The searching time for scheme 1 and 3 includes both the decompression (the first number) and the matching time (the second number). For scheme 3 and 4, the block size is 4MB. In performing the experiment, we keep roughly about the same amount of main memory across different schemes.

From the table, we observed that path matching on compressed traces performs better than that on uncompressed traces. For the latter, the major overhead comes from the need to decompress the trace. The decompression time is the major fraction for schemes using either fully decompression or partial decompression (scheme 1 or 3). As an example, in matching type 3 of 008.espresso, scheme 2 spends 360ms while scheme 1 has to spend 900ms to decompress the trace and then 336ms to perform the matching. The time to decompress a large trace is substantial when the size is more than 500MB. A full scan takes around 1 second while the decompression takes around 4 seconds.

The actual matching time is comparable in searching either compressed or uncompressed traces. Matching in the compressed traces sometimes is slower due to the complicated control structure that it has to maintain during searching. For example, in search path 2 of 126.gcc, scheme 2 has to spend 1432ms while scheme 1 spends only 1149ms.

We also observed that for different path types, scheme 1 and scheme 2 have almost constant path matching performance. This is due to the fact that matching has to be performed from the start to the end for these two schemes. With small indices, scheme 3 and scheme 4 gain large performance

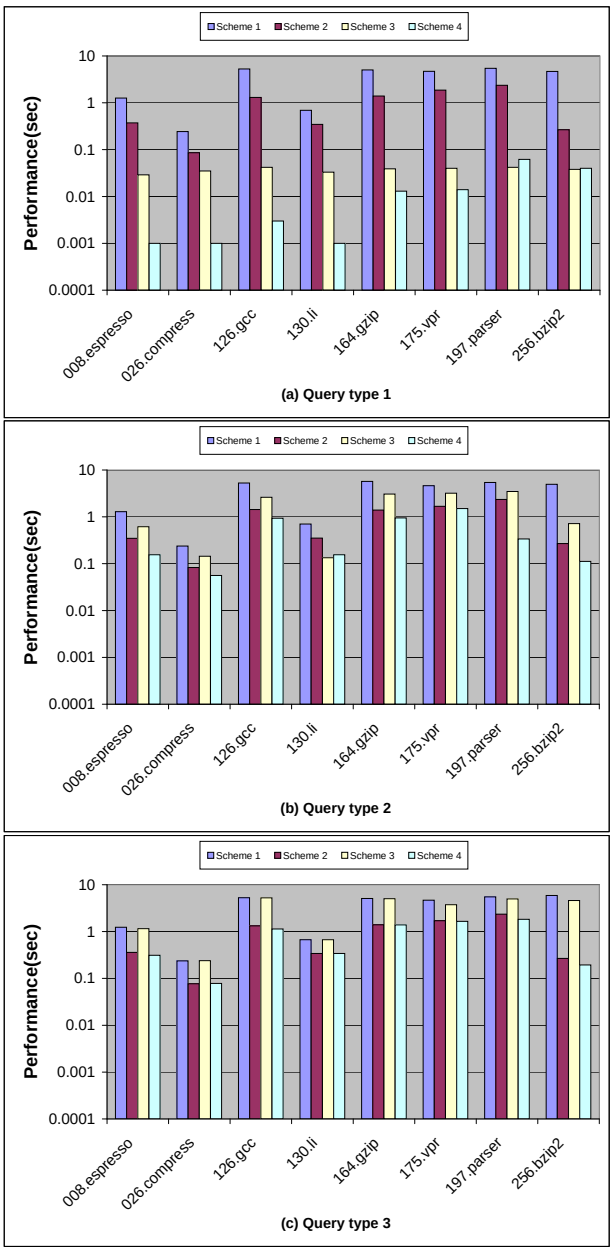


Fig. 14. The Performance Comparison of Path Matching Schemes

benefits for paths that have low to modest coverage. For example, for all benchmarks, path type 1 can be extremely quickly identified. Only one block needs to be decompressed and searched using scheme 3. Scheme 4 also proportionally reduces the number of rules to be searched.

From the results, we conclude that both compressed path matching and matching after decompression algorithms are useful in practice. In the case that the main memory size is large and there are a lot of path matching requests, we can cache these decompressed blocks in the memory. Once the decompression cost can be amortized from consecutive searches, the simple structure in the uncompressed trace has better in-memory search time. On the other hand, if the main memory size restricts the caching of decompressed blocks, or the path matching requests are random, we may choose a compressed search scheme instead.

6 Conclusions

In this paper, we designed and evaluated four different path matching schemes over compressed and uncompressed control flow traces. We not only identified the challenges but also exploited the opportunities in matching intraprocedural paths in control flow traces. We evaluated the proposed schemes with real control flow traces. Our experimental results show that small indices are very effective in improving matching performance of paths that are of small to modest coverage. In particular, the path matching scheme based on Sequitur-compressed traces with small indices achieves large performance benefits over other path matching schemes.

References

Amir, A., Benson, G., 1992. Efficient Two-dimensional Compressed Matching. In: Proceedings of Data Compression Conference. Snowbird UT, pp. 279-288.

Amir, A. Benson, G., Farach, M., 1996. Let Sleeping Files Lie: Pattern Matching in Z-compressed Files. In: Proceedings of the fifth ACM-SIAM symposium on Discrete Algorithm, pp705-714.

Boyer, R., Moore, J., 1977. A Fast String Searching Algorithm. Communication of ACM 20(10) pp. 762-772.

Farach, M., Thorup, M., 1998. String Matching in Lempel-Ziv Compressed Strings. Algorithmica 20(4), pp.388-404.

Kida, T., Shibata, Y., Takeda, M., Shinohara, A. , Arikawa, S., 1999. A Unifying Framework for Compressed Pattern Matching. In: Proceedings of 6th International Symposium on String Processing and Information Retrieval, IEEE Computer Society, pp.89-96.

Ball, T., Larus, J., 1996. Efficient Path Profiling. In: Proceedings of 29th IEEE/ACM International Symposium on Microarchitecture, pp.46-57.

Knuth, D.E., Morris, J.H., Pratt, V.R., 1977. Fast Pattern Matching in Strings. SIAM Journal on Computing 6(1) pp.323-350.

Larus, J., 1999. Whole Program Paths. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, pp.259-269.

Nevill-Manning C., Witten, I., 1997. Identifying Hierarchical Structure in Sequences: A Linear-time Algorithm. Journal of Artificial Intelligence Research 7, pp. 67-82.

Navarro, G., Raffinot, M., 1999. A General Practical Approach to Pattern Matching over Ziv-Lempel Compressed Text. In: Proceedings of CPM, LNCS 1645, pp.14-36.

Mitarai, S. Hirao, M., Matsumoto, T. Shinohara, A. Takeda, M., Arikawa, S., 2001. Compressed Pattern Matching for SEQUITUR. In: Proceedings of Data Compression Conference.

Trimaran Compiler Infrastructure. http://www.trimaran.org/.
Zhang Y., Gupta, R., 2001. Timestamped Whole Program Path Representation and its Applications. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, pp.180-190.
SPEC. http://www.spec.org/osg/cpu2k/.
Navarro, G., Kida, T., Takeda, M., Shinohara, A., Arikawa, S., 2001. Faster Approximate String Matching Over Compressed Text. In: Proceedings of Data Compression Conference, pp. 459-467.
Zhuge, H., 2003. An Inexact Model Matching Approach and Its Applications. Journal of Systems and Software 67(3), pp.201-212.
Ziv, J., Lempel, A., 1977. A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory 23(3), pp.337-343.
Ziv, J., Lempel, A., 1978. Compression of Individual Sequences via Variable Length Coding. IEEE Transactions on Information Theory 24(5), pp.530-536.

Yongjing Lin is a PhD candidate in the Department of Computer Science at University of Texas at Dallas. Her research interests include data compression, profiling and code optimization, text mining. Yongjing Lin received her Master's degree from Louisiana Technology University in 2000, and a Bachelor's degree from the Xiamen University, China in 1997.

Youtao Zhang is an assistant professor in the Computer Science Department, University of Pittsburgh. He completed his PhD in Computer Science at the University of Arizona in 2002. His research interests are in the area of the computer security, program analysis and compiler optimization, and computer architecture. He is the recipient of US NSF Career Award in 2005, the distinguished paper award of the IEEE/ACM International Conference on Software Engineering (ICSE) conference in 2003, the most original paper award of the International Conference on Parallel Processing (ICPP) conference in 2003. He is a member of the ACM and the IEEE.

Rajiv Gupta is a Professor of Computer Science at The University of Arizona. His areas of research interest include Compiler and Architectural Support for Optimization of Performance, Power, Memory, and Security in Embedded Systems; Program Analysis: Static and Profile-based; and Software Tools for Profiling, Slicing, and Debugging. Rajiv has published over 190 articles in refereed conferences and journals, he holds 8 US patents, and has supervised 12 PhD dissertations. Papers coauthored by him have been selected for: inclusion in 20 Years of PLDI (1979-1999), distinguished paper award in ICSE 2003, most original paper award in ICPP 2003, and outstanding paper award in ICECCS 1996. His research has been funded by NSF, DARPA, Intel, IBM, Microsoft, and HP.

Rajiv received the National Science Foundation's Presidential Young Investigator Award in 1991 and served as an IEEE Distinguished Visitor for the period of 2000-2002. He served as the Program Chair for PLDI'03, HPCA'03, and LCTES'05 conferences and Co-General Chair for CGO'05 conference. Rajiv has served on over 85 program committees including those of the PLDI, POPL, CGO, LCTES, CASES, MICRO, HPCA, ICS, PACT, PEPM, PASTE, and ICCL conferences. He serves as an Associate Editor for ACM Transactions on Architecture and Code Optimization, Parallel Computing journal, and Journal of Embedded Computing. Rajiv is a member of the ACM and a senior member of IEEE.