Efficient Discovery of Loop Nests in Execution Traces

Qiang Xu
CGGVeritas Inc.

Email: Qiang.Xu@cggveritas.com

Jaspal Subhlok and Nathaniel Hammen
Department of Computer Science
University of Houston
Email: jaspal@uh.edu

***Abstract*—Execution and communication traces are central to performance modeling and analysis. Since the traces can be very long, meaningful compression and extraction of representative behavior is important. Commonly used compression procedures identify repeating patterns in sections of the input string and replace each instance with a representative symbol. This can prevent the identification of long repeating sequences corresponding to outer loops in a trace. This paper introduces and analyzes a framework for identifying the maximal loop nest from a trace. The discovery of loop nests makes construction of compressed representative traces straightforward. The paper also introduces a greedy algorithm for fast "near optimal" loop nest discovery with well defined bounds. Results of compressing MPI communication traces of NAS parallel benchmarks show that both algorithms identified the basic loop structures correctly. The greedy algorithm was also very efficient with an average processing time of 16.5 seconds for an average trace length of 71695 MPI events.**

***Index Terms*—Trace compression, loop discovery, performance modeling**

I. Introduction

Execution and communication traces are central to performance analysis and performance modeling. However, trace processing is a challenge as the trace length can be large even for traces of relatively coarse grain events. Fortunately execution traces often contain repeating sequences that can be identified to capture representative behavior. The goal of the research presented in this paper is to develop effective and efficient procedures to identify the representative sections of an execution traces by discovering the loop nest structure inherent in the trace. There are, of course, several well known algorithms and tools for string compression. Most compression procedures apply heuristics to selectively reduce sets of repeating substrings. Examples include *gzip* [1] that constructs a dictionary of frequently occurring substrings and replaces each occurrence with a representative symbol, and *Sequitur* [2], [3], [4] that infers the hierarchical structure in a string by automatically constructing and applying grammar rules for reduction of substrings. These approaches are efficient and procedures can be designed to have execution time that is nearly linear in trace length. The key reasons for developing a new approach are as follows:

1) These compression procedures are not guaranteed to identify long range loop patterns because of heuristic early reductions.
2) An important objective of this research is to identify the representative sections of a trace. These are clearly defined in a loop nest but not if the compressed representation is in the form of string tables (gzip) or a grammar (Sequitur).

In Table I, we illustrate these points by showing the compressed form of LU benchmark trace with Sequitur and our loop discovery procedure. The compressed representation is a grammar for Sequitur and a loop nest for our procedure. While the compression achieved is good in both cases, it is easy to see that the representative sections of the trace are easily identified in the loop nest form as the elements of the loops with large numbers of iterations.

The loop nest in an execution trace can be derived in a straightforward way by repeatedly identifying the longest matching substring first. However, commonly used algorithms to achieve this are quadratic for a given match length and hence cubic in trace length, and therefore impractical for long traces. A practical approach is to limit the window size for substring matching, which again risks missing long span repeats [5].

Before presenting our approach, we introduce the basic terminology for distinguishing various types of repeating patterns. Repeating substrings (or *repeats*) in a string can be *tandem* repeats where successive repeat substrings immediately follow each other, *overlapping* repeats where repeat substrings overlap, and *split* repeats where repeat substrings are separated by other symbols. Since we seek to identify the loop structure in a trace, we are only interested in tandem repeats. A tandem repeat is *primitive* if it is itself not composed of tandem repeats of another substring. A set of tandem repeats is *maximal* if there is no identical substring immediately preceding or succeeding the sequence of tandem repeats. We will refer to the primitive and maximal tandem repeats in a string as *PM-repeats*. ***In the rest of the paper, "loops" technically refer to PM-repeats in the execution trace, which (presumably) exist because of the execution of program loops.*** Our objective is find and reduce the PM-repeats of different spans in an execution trace, which is the same as discovering the inherent loop nest structure in the execution trace.

To illustrate the properties of PM repeats, consider the string *abababab*. The PM-repeats corresponding to this string are represented as $(ab)^4$ which is the most compact representation.

Compressed trace with Sequitur algorithm:
S0 → V V V 1 X 1 W 2 3 4 Y R U 5 6 7 8 9 9 9 10 11 12 13 R 14 R S R P Y A O Y F T T T
1 → W W 2 → Q 4 3 → Y Q 4 → J 3 H 3 E 3 C 5 → 15 15 6 → 5 15
7 → 16 16 8 → 7 16 9 → 10 10 10 → 11 11 11 → 12 12 12 → 17 17
13 → 14 6 5 8 7 14 → 2 Y 15 → 18 18 16 → 19 19 17 → 20 20 18 → 21 21
19 → 22 22 20 → 13 13 21 → 23 23 22 → 24 24 23 → 25 25 24 → 26 26
25 → 27 27 26 → 28 28 27 → N K I D 28 → M L G B
Compressed trace with loop nest discovery algorithm:
$(V)^3(W)^2X(W)^3(QJYQHYQEYQCY)^2RU((NKID)^{160}(MLGB)^{160}QJYQHYQEYQCY)^{249}$ $(NKID)^{160}(MLGB)^{160}RQJYQHYQEYQCYRSRPYAOYF(T)^3$

TABLE I
COMPRESSED TRACE WITH SEQUITUR AND THE LOOP NEST DISCOVERY ALGORITHM PRESENTED IN THIS PAPER. EACH SYMBOL REPRESENTS A UNIQUE MPI OPERATION. THE INPUT TRACE IS FOR CLASS C LU BENCHMARK WITH 323048 TRACE SYMBOLS

The string can also be represented as tandem repeats $(abab)^2$ but this would not be primitive, since the repeating substring itself is a tandem repeat of another string ab. The string can also be represented as $(ab)^3ab$ but this would not be a maximal repeat. Hence, $(ab)^4$ represents the only PM-repeats sequence, or optimal loop, for this string.

Our approach to identifying the loop structure in a trace is derived from Crochemore's algorithm [6], which can identify all repeats in a string, including tandem, split, and overlapping repeats, in $O(nlogn)$ time. A framework was developed in this research to discover the loop nest structure by recursively identifying the longest span PM repeat in a trace. Intuitively, this procedure discovers loop nests by repeatedly identifying and reducing the outermost loop in a trace. For the terminology of this paper we will refer to such a loop nest as "optimal". The procedure was applied to identify the loop nests in the MPI communication traces of NAS benchmarks. The compression results were very good, but the execution time was unacceptable for long traces; processing of a trace consisting of approximately 320K MPI calls took over 31 hours.

The results motivated us to develop a greedy procedure for loop structure discovery, which is a key contribution of this paper. The greedy procedure intuitively works bottom up - it identifies and reduces the shorter span inner loops and replaces them with a single symbol, before discovering the longer span outer loops. In this respect, it appears similar to other approaches that apply heuristics to identify repeating substrings and replace them with symbols to enable efficient processing. However, the key characteristic of our algorithm is that only primitive and maximal tandem repeats (PM-repeats) representing a section of the trace that corresponds to loop execution, are reduced to a single symbol. No other repeating substrings are reduced. The intuition is that reduction of trace sections corresponding to complete inner loop execution will not interfere with the discovery of outer loops. An important contribution of this work is to establish that the loop structure discovered by the greedy algorithm is provably *near optimal*. Intuitively, the greedy approach still guarantees the discovery of long span loops or PM-repeats, but with up to 2 less iterations. The result is formally described and refined in the paper.

The greedy loop nest discovery procedure was also implemented and employed to discover the loop nests in the MPI traces of NAS benchmarks. The loop nests always satisfied the criteria above, and were, in fact, identical to the optimal loop nests in all but one case. However, the time for loop discovery was dramatically lower than the optimal algorithm, with the compression time reduced to approximately 62 seconds from 31 hours for one trace.

To the best of our knowledge, this is the first effort toward extracting complete loop nests from execution traces. The paper presents detailed results on the effectiveness of these algorithms in discovering loop nests and achieving compression. The performance and scalability of the greedy and optimal algorithms are also presented and analyzed. Of particular interest are the insights into the theoretical complexity of the algorithms and the empirical measurements of performance. The methodology developed is applicable to any sequence that is likely to contain a loop structure even though the experimental results presented in this paper are limited to message passing communication traces.

II. MOTIVATION AND CONTEXT

This research was motivated by performance estimation in foreign environments based on performance skeletons. A *performance skeleton* of an application is defined to be a short running program whose execution time in any scenario reflects the execution time of the application it represents; thus an estimate of the application execution time in a new environment is obtained by simply executing the performance skeleton and appropriately scaling the measured skeleton execution time. The key steps in the construction of a performance skeleton from the MPI level process traces of an application are the following:

1) *Trace logicalization* For parallel scientific applications, the traces for different processes are typically similar to each other and the communication is associated with a well defined global communication pattern. *Logicalization* converts a family of processor traces into a single *logical program trace* that represents the aggregate execution of the program. Logicalization is a complex and potentially lossy procedure that is orthogonal to the subject of this paper but is detailed in [7].

2) *Trace compression* The focus of this paper is trace compression by discovering the implicit loop structure, which is applied to a single logicalized trace for skeleton construction. Trace logicalization and trace compression are complementary procedures. Although the results presented in this paper are in the context of compression of logical MPI traces, the framework for compression can be applied to any trace to identify any loop structure that may exist.
3) *Performance skeleton generation* The final step is generation of an executable performance skeleton program from the compressed logical trace consisting of MPI communication and computation sections. Synthetic computation and communication calls are generated to recreate the execution behavior captured by the traces.

This discussion provides a brief context for this paper. Skeleton construction and the prediction power of skeletons in different execution scenarios are detailed in [8].

III. Related work

Compression is a basic operation in a wide variety of scenarios. Many algorithms have been developed for text compression and employed in utilities like *gzip* [1]. The basic approach in such algorithms is to identify recurring short strings and replace them with identifiers. *Sequitur* [2], [3], [4] is a well-known algorithm that was developed to discover the natural hierarchy in text and other data. The insight is that repeating substrings are replaced by a grammar rule that generates that substring and the process is continued recursively, resulting in a hierarchical representation of the structure of the string. In order to improve the processing time and quality of compression, PGTC (path grammar guided trace compression) [9] is proposed as an enhanced approach that employs program static analysis to build a grammar and guide compression. Noeth et al. [5] have developed an online method for identifying loops in a message passing trace. However, the algorithm is not guaranteed to capture long range loops as matching is limited to a maximum sliding window to avoid $O(n^2)$ time complexity in the length of the trace.

The goal of this work is to identify complete loop nests from the repeating substrings discovered in a string. There are two well known approaches to identifying all repeats in a string systematically - one based on suffix trees and the other based on Crochemore's algorithm. We have employed Crochemore's algorithm as the basis of our approach and that is discussed in detail in this paper. We briefly discuss suffix trees here. Suffix trees are a fundamental data structure supporting a wide variety of efficient string searching algorithms. In particular, suffix trees are well known to allow efficient and simple solutions to problems concerning the identification and location of repeated substrings. Several algorithms [10], [11], [12] can build a suffix tree in linear time. Stoye and Gusfield have developed an $O(n \log n)$ time method [13] to find all occurrences of primitive tandem repeats in a string with suffix trees. They also proposed a novel method [14] to collect only the primitive tandem repeat $types$ in $O(n)$ time and find occurrences of all primitive tandem repeats in $O(n + z)$ time, where z is the number of occurrences of primitive tandem repeats in a string. In [15], the repeating substrings in a string and their statistics are inferred from suffix trees, and used for compression through greedy off-line textual substitution.

We have based our loop nest identification procedure on Crochemore's algorithm instead of suffix trees for two main reasons. First, we are not aware of a straightforward approach to finding all $primitive$ and $maximal$ tandem repeats with suffix trees. Second, the process of building and processing suffix trees is significantly more complex than that based on Crochemore's algorithm.

IV. Optimal trace compression

The main contribution of this paper is a framework to compress execution traces by discovering the loop structure inherent in the trace. All repeating substrings in a trace are identified by employing the well known Crochemore's algorithm, but these repeats are implicit in a complex data structure. The total number of repeats can be combinatorial in the size of a string and very large in practice. The contribution of this work in this context is the development of a framework to efficiently construct the loop structure in the trace by selectively filtering and reducing the repeats. The procedure consists of the following steps for discovery and reduction of outermost loops:

1) **Repeats discovery:** Discovery of all types of repeats (overlapping, split, and tandem) of all sizes by Crochemore's algorithm.
2) **Loop identification:** Identification of all PM-repeats (primal and maximal tandem repeats) corresponding to loops.
3) **Loop filtering:** Discovery of outermost loops and their replacement with loop symbols.

The above process is repeated recursively inside each discovered loop. For a string with n symbols, the repeats discovery takes $O(n \log n)$ time while loop identification and loop filtering take $O(n^2)$ time. Hence the overall complexity is $O(n^2)$. We discuss each of the above steps and the overall loop identification and compression procedure.

A. Repeats discovery

Before we can discuss the *loop identification*, and *loop filtering* that are our contributions, we have to explain how *repeats discovery* is done with Crochemore's algorithm. As an optimized ***successive refinement method***, Crochemore's algorithm [6] computes all repeating substrings (tandem, overlapping, and split) in a finite string S of length n in $O\left(n \log n\right)$ time. The successive refinement begins with grouping all positions in the string that have the same symbol/character into a single class. Each class is then refined into new subclasses that contain starting positions of repeating substrings of length two. The process is continued to find the starting position of all repeating substrings of length, 3,4,5....until a size is reached for which no repeating substrings exist.

Before describing the details of Crochemore's algorithm, we introduce some basic string definitions.

Definition: A *string* $S = s_1 s_2 s_3 ... s_n$ is an ordered list of characters/symbols written contiguously from left to right. The *length* of *S* is $|S|$. $S[i..j]$ is the *substring* of *S* that starts at position *i* and ends at position *j*.

Definition: E_k is an *equivalence relation* over a string S defined as follows: iE_kj if and only if substrings $S[i..i+k]$ and $S[j..j+k]$ are identical. E_k partitions the positions of string S into equivalence classes; if iE_kj then i and j will be in the same E_k class. We also use E_k to denote the set of those equivalence classes.

The simple successive refinement is based on the fact that for string S, if iE_kj and $S[i+k]$=$S[j+k]$, then $iE_{k+1}j$. For example, consider the string

S =	a	b	a	a	b	a	b	a	a	b	a	a	b	$
	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Initially, we construct three E_1 classes containing repeating substrings of length one. Note that the unique character "$" is appended as the end of string symbol.

$$E_1 : \underset{a}{\{1,3,4,6,8,9,11,12\}} \ \underset{b}{\{2,5,7,10,13\}} \ \underset{\$}{\{14\}}$$

The first step of refinement splits each class of E_1 into classes that contain starting positions of substrings of length two. We check the character following each position in that class. For class a–{1,3,4,6,8,9,11,12}, we need to check positions {2,4,5,7,9,10,12,13} in S. Since $S[4]$=$S[9]$=$S[12]$=a and $S[2]$=$S[5]$=$S[7]$=$S[10]$=$S[13]$=b, class a is split into subclass aa–{3,8,11} and subclass ab–{1,4,6,9,12}. Similarly, class b–{2,5,7,10,13} is split into subclass ba–{ 2,5,7,10} and subclass $b\$$–{13}. The class $–{14} has no substring of length two starting from it, so it is discarded.

The successive refinement continues with checking of the kth character following the positions in each class of E_k to construct E_{k+1}. Any singleton classes are discarded. Eventually a value of k is reached for which there are no classes of E_k and the process is terminated. The entire process of refinement of string $S = abaababaabaab\$$ is shown in Table II.

The total running time for the algorithm as described above is $O\left(n^2\right)$, since there can be $O\left(n\right)$ levels and each level can take $O\left(n\right)$ time. But two techniques proposed in [6] optimize the successive refinement and reduce the running time to $O\left(n \log n\right)$. We mention them very briefly here.

The first technique is "***indirect refinement***". This is based on the observation that any class of E_{k+1} is a subset of some E_k class. Because, $iE_{k+1}j$, if and only if iE_kj and $i+1E_kj+1$, we can use classes at the same level to carry out successive refinement instead of referring back to the original string S. This helps in reducing the complexity when used with another technique called " ***small classes***", which can be outlined as follows. The indirect refinement process for an E_k class into E_{k+1} classes requires matching against several, but not all, other E_k classes. The small classes technique prescribes that the classes be selected in a specific manner that favors matching against smaller classes, and thereby avoiding some matching against larger classes. This description only gives a flavor of these technique and the interested reader is referred to [6], [16] for details. Our implementation includes the indirect refinement and small classes techniques.

B. Loop identification

We formally define and illustrate the various types of repeats that are central to the discussion and analytical results in this paper. The repeats discovered by Crochemore's algorithm include tandem, overlapping, and split repeats. Their definitions are as follows:

Definition: For positions i and j of string S, that belong to the same E_k class, if $|j-i| = k$, then repeating substrings $S[i..i+k-1]$ and $S[j..j+k-1]$ are $tandem$ repeats; if $|j-i| < k$ they are $overlapping$ repeats; and if $|j-i| > k$ they are $split$ repeats.

For example, the substring aba repeats four times in string S at positions 1,4,6, and 9 in Table II. The second aba is right behind the first one, so they constitute **tandem** repeats. The second and third aba are **overlapping** repeats, while the first and third aba are **split** repeats.

Our goal here is to find loop structures, so we need to identify and report only the tandem repeats. Tandem repeats in a string can be represented by a triple (i, β, l), where i is the starting position, β is the repeated substring, and l is the number of iterations. But a substring may be represented by multiple tandem repeats. For example, the string $abababababababab$, could be described as $(1, ab, 8)$, or $(1, abab, 4)$, or $(1, abababab, 2)$. Clearly the loop that we would like to identify corresponds to $(1, ab, 8)$. To generalize, we define **PM-repeats** and a corresponding **PM-triple**, where P and M stand for $primitive$ and $maximal$. A triple (i, β, l) corresponds to a primitive tandem repeats sequence if and only if β is not periodic. The triple corresponds to a maximal tandem repeats sequence if and only if there is no β right before or after the repeats. So, the above string can be represented by a unique PM-triple, $(1, ab, 8)$. (A PM-triple is a representation of a PM-repeats sequence and we will use the terms interchangeably.)

For each E_k class refined in Crochemore's algorithm, a PM-triple can be identified by the following Lemma, which is also mentioned in [16]:

Lemma 4.1: Triple (i,β, l) is a PM-triple, where β is a k-length substring, if and only if some single class of E_k contains a maximal series of numbers i, i+k, i+2k, ..., i+lk, such that each consecutive pair of numbers differs by k.

In order to identify loops, PM-triples must be identified at each level during the execution of Crochemore's algorithm. Since the total number of members in all classes at a level k is bounded by string length n, the process takes $O\left(n\right)$ time. Since the maximal possible size of β, the loop element, is half the length of the string n, we need to report PM-triples after discovering the repeats by Crochemore's algorithm till level $n/2$. The running time for identifying loops represented by PM-triples is $O\left(n^2\right)$.

$Level1-$	$E_1:$	{1,3,4,6,8,9,11,12} a	{2,5,7,10,13} b	~~{14}~~ $		
$Level2-$	$E_2:$	{1,4,6,9,12} ab	{3,8,11} aa	{2,5,7,10} ba	~~{13}~~ $b$$	
$Level3-$	$E_3:$	{1,4,6,9} aba	~~{12}~~ $ab$$	{3,8,11} aab	{2,7,10} baa	~~{5}~~ bab
$Level4-$	$E_4:$	{1,6,9} $abaa$	~~{4}~~ $abab$	{3,8} $aaba$	~~{11}~~ $aab$$	{2,7,10} $baab$
$Level5-$	$E_5:$	{1,6,9} $abaab$	~~{3}~~ $aabab$	~~{8}~~ $aabaa$	{2,7} $baaba$	~~{10}~~ $baab$$
$level6-$	$E_6:$	{1,6} $abaaba$	~~{9}~~ $abaab$$	~~{2}~~ $baabab$	~~{7}~~ $baabaa$	
$Level7-$	$E_7:$	~~{1}~~ $abaabab$	~~{6}~~ $abaabaa$$			

TABLE II
SUCCESSIVE REFINEMENT OF STRING $abaababaabaab$$

C. Loop filtering

The previous steps provide a list of PM-triples which represent all the loops in the trace. Our interest is in finding all the outermost or longest span loops. These are represented by the PM-triples at the highest level. (The inner loops are discovered by running the entire process recursively). In case of multiple overlapping loops of equal span at the same level, we select the one that starts earliest in the string.

As an example, for the string $abcdabcdabcdabcda$, 4 PM-triples will be identified. These PM-triples are $(1, abcd, 4)$, $(2, bcda, 4)$, $(3, cdab, 3)$ and $(4, dabc, 3)$. The first two PM-triples both have a span of 4 versus a span of 3 for the remaining two. Based on the earliest starting point, the selected loop will be $(1, abcd, 4)$.

As another example, consider the string $EababababFEababababFEababababF$, which contains a loop nest containing loops at two levels. The PM-triple $(1, EababababF, 3)$ represents the selected outer loop. The inner loops represented by triples $(2, ab, 4)$, $(12, ab, 4)$, and $(22, ab, 4)$ are ignored at this stage.

The loop filtering step repeatedly finds the PM-triple corresponding to the longest span loop, until no PM-repeats are left. Since the loop element can theoretically be as small as 2 symbols, the theoretical upper bound of this step with a simple implementation is $O(n^2)$. An $O(nlogn)$implementation is possible. However, in practice this is a very quick step as the number of loops is normally very small, and $O(n^2)$ is a very loose upper bound.

D. Compression framework summary

The procedure discussed in this section identifies all outermost loops represented in a trace. The algorithm runs recursively on the substring that constitute the loop element (or body) for identification of the inner loops. The recursive steps are important to get a high degree of compression. It is theoretically possible to reuse some of the information from discovery of outer loops to identify inner loops. In practice, this is likely to make little difference in performance since the processing time is dominated by the time to discover outer loops. In order to develop a compressed representation, the loop spans in the trace are replaced with tuples (LE, l), where LE represents the loop element, and l is the number of loop iterations. A separate table is constructed, which maps a loop element symbol to the substring that constitutes the loop element. The overall complexity of the steps in this procedure is $O(n^2)$ and is dominated by the loop identification step.

V. GREEDY TRACE COMPRESSION

The scheme discussed in Section IV discovers the outermost loop. However, the execution time for loop discovery can be high for long traces. We will discuss experimental results in detail later in this paper, but we jump ahead to Table IV to motivate the case for a faster approach. As an example consider the class C SP benchmark in the table. The total time to run the algorithm is around 747 seconds, although all program loops had already been discovered in just 5.8 seconds. The reason is that the largest loop consists of only 67 symbols while the trace size is 26888 symbols. Our approach builds equivalence classes and looks for loops in increasing order from 1 to half the trace length. Even though all loops were discovered by the time the equivalence class of 67 was constructed, (and these loops spanned over 99% of the trace) there is no way for the algorithm to be certain that a larger loop does not exist, and hence refinement continues until the equivalence class of 13444 that corresponds to half the original string size. If the loops already discovered were replaced by a single symbol at the equivalence class of 67, the trace size to be processed would be less than 1% of the original trace size, and the remaining processing would be much faster. This motivates greedy compression.

The key idea of greedy compression is early compression as PM-triples are discovered. The entire span of the corresponding loop is replaced by a single symbol, and compression continues on the newly formed (shorter) string. The procedure continues until half the current string size is reached, but

the string size decreases dynamically as loops are discovered, which is the key reason for improved performance. Figure 1 outlines the greedy compression procedure.

S = string corresponding to the original trace
$Current\_S = S$; $Level = 1$;
Step 1:
if $Level > |Current\_S|/2$ **then**
  Goto Step 3
else
  i) Find all repeats of size $Level$ in $Current\_S$ by successive refinement with Crochemore's algorithm.
  ii) Identify all PM-triples with repeating substring of size *Level*.
  if any PM-triples with repeating substring of size $Level$ are discovered **then**
    Goto Step 2
  else
    $Level = Level + 1$; Goto Step 1
  end if
end if
Step 2:
Update $Current\_S$ by reducing all PM-triples with repeating substrings of size $Level$ in decreasing order of loop span, employing filtering (discussed in section IV-C) for overlapping triples. The symbols replacing the PM-triples are stored in a mapping table.
$Level = 1$; Goto Step 1.
Step 3:
Stop. $Current\_S$ along with the mapping table for symbols is the compressed trace that captures the loop nests.

Fig. 1. Greedy compression procedure

The worst case time complexity for this greedy algorithm is the same as the optimal algorithm discussed in Section IV. In fact, the two algorithms will run identically if there were no PM-repeats in a trace. However, the greedy algorithm is much more efficient in practice for programs with a loop structure.

A. Risk of greedy compression

For most traces that we have analyzed, the greedy and optimal procedures yield identical results. Here we illustrate with carefully selected examples how the results of greedy compression can be suboptimal.

Consider the string $abaababaabaab$. The greedy compression proceeds as follows:

a b a a b a b a a b a a b $
a b L_1 b a b L_1 b L_1 b $ $L_1 = (a)^2$
a b L_1 b a L_2 b $ $L_2 = (bL_1)^2 = (baa)^2$

The loop structured discovered by the greedy procedure is $ab(a)^2ba(b(a)^2)^2b$ whereas the optimal loop structure is $(ab(a)^2b)^2(a)^2b$. A 2 iteration loop with the largest element, $(abaab)^2$, is completely missed.

Now consider the string $abaababaababaabaab$. The greedy compression proceeds as follows:

a b a a b a b a a b a b a a b a a b $
a b L_1 b a b L_1 b a b L_1 b L_1 b $ $L_1 = (a)^2$
a b L_1 b a b L_1 b a L_2 b $ $L_2 = (bL_1)^2 = (baa)^2$
L_3 a L_2 b $ $L_3 = (abL_1b)^2 = (abaab)^2$

The loop structured discovered by the greedy procedure is $(ab(a)^2b)^2a(b(a)^2)^2b$ whereas the optimal loop structure is $(ab(a)^2b)^3(a)^2b$. The loop with the largest loop element $(abaab)$ is captured, but with one less iteration than optimal.

In general, the greedy algorithm will discover any loop, except that the discovered loop may have up to 2 less iterations than the maximal loops (first and/or last iteration may not be discovered) and the elements in the loop body may be a rotation of the elments in the maximal loop. In practice, the difference between the results of the two algorithms are identical or trivially different from compression standpoint. The "near optimality" of the greedy algorithm is discussed further in Section VII.

VI. Experimental results

A framework for loop discovery and compression discussed in Section IV and a framework for greedy compression as discussed in Section V were implemented. The goal was to validate loop discovery and compression achieved by these algorithms and study the tradeoffs between the execution performance and the degree of compression.

This research was motivated in the context of analysis of execution traces of MPI applications to build representative executable "performance skeleton" programs. All results presented in this paper are for compression of MPI communication traces for Class B/C NAS Parallel Benchmarks running on 16 nodes. The traces were collected with the PMPI library and converted to strings of symbols. Each symbol in the trace that was input to the compression procedure corresponds to a specific MPI operation with a specific set of parameters.

A. Results and discussion

Table III shows the results of the optimal compression procedure. We observe that the length of the traces ranged from 8909 to 323048 (average 71695) and the length of the compressed traces ranged from 38 to 1118 (average 281) with the degree of compression ranging from 8 to 5384 (average 1173). The structures of the major loop nests discovered are also described in the table. Most of the trace was covered by loops for all benchmarks, around 98% on average. The conclusion is that many MPI traces have a loop structure that can be discovered automatically by this approach. The degree of compression is excellent and the length of the compressed trace is generally relatively small. However, we caution that the degree of compression is naturally dependent on the loop structure implicit in a trace and good compression results are not universal. In particular, compression will be poor if the control flow of a program leads to dynamic execution behavior with little repetition in the execution trace.

Table IV focuses on the execution time for optimal compression. The total time for loop discovery, which includes repeats discovery, loop identification and loop filtering, is

Name	Trace Length	Major Loop Structure	Trace Span Covered by Loops	Compressed Trace Length	Compression Ratio
BT B/C	17106	$(85)^{200} = (13 + (4)^3 + ... + (4)^3)^{200}$	99.38%	85	201.25
SP B/C	26888	$(67)^{400}$	99.67%	162	165.97
CG B/C	41954	$(552)^{75} = ((21)^{26} + 6)^{75}$	98.68%	38	1104.05
MG B	8909	$(416)^{20}$	93.39%	1006	8.86
MG C	10047	$(470)^{20}$	93.56%	1118	8.99
LU B	203048	$(812)^{249} = ((4)^{100} + (4)^{100} + 12)^{249}$	99.58%	60	3384.13
LU C	323048	$(1292)^{249} = ((4)^{160} + (4)^{160} + 12)^{249}$	99.58%	60	5384.13
Average	71695		98.16%	281	1172.86

TABLE III
RESULTS OF OPTIMAL COMPRESSION

Name	Trace Length	Repeats Discovery Time (sec)			Loops Discovery Time (sec)				Loop Element Size	
		Total	Up to Smallest Repeat	Up to Largest Repeat	Total	Up to Smallest Element	Up to Largest Element	Per Trace Record	Smallest Size	Largest Size
BT B/C	17106	12.85	0.47	0.87	311.18	0.63	4.84	.013	4	85
SP B/C	26888	15.88	0.98	0.98	747.73	5.81	5.81	.014	67	67
CG B/C	41954	239.29	1.46	5.78	2021.78	3.73	67.77	.018	21	552
MG B	8909	35.85	0.00	3.95	113.48	0.00	13.74	.027	1	416
MG C	10047	45.96	0.00	4.97	144.54	0.00	17.41	.048	1	470
LU B	203048	2565.73	4.93	24.31	44204.82	6.51	463.18	.218	4	812
LU C	323048	8028.83	7.83	59.72	113890.21	10.18	1172.63	.352	4	1292

TABLE IV
PERFORMANCE AND EXECUTION TIME BREAKUP FOR OPTIMAL COMPRESSION. THE LOOPS DISCOVERY TIME INCLUDES THE TIME FOR REPEATS DISCOVERY THAT IS ALSO LISTED SEPARATELY AS WELL AS THE TIME FOR LOOP IDENTIFICATION AND LOOP FILTERING.

reported along with the time per trace record. The largest and the smallest loop element sizes are also noted. One observation is that the repeats discovery time is a relatively small component of the total loop discovery time, which is dominated by the loop identification time. The loop filtering time was consistently very small in comparison and is not reported separately in the table.

The times for repeats discovery and loop discovery increase as the trace size increases. Further, the time per trace record for loop discovery increases from .013 seconds for the shortest trace to .352 seconds for the longest trace, suggesting a low polynomial relationship between trace length and processing time. The LU class C benchmark takes 8028.83 seconds (2.23 hours) to finish discovering all possible repeats, and identification of loops from those repeats takes 113890.21 seconds (31.64 hours). As the experiments are for a modest input data size running on only 16 nodes, the execution time is a major concern for realistic longer running applications on larger clusters.

Table IV also shows the times at which the smallest loop and the largest loop was discovered during the compression of each benchmark trace. For all benchmarks, the largest loop was discovered within a small fraction of time as compared to the total execution time. A similar pattern is observed for the largest repeats. Loop discovery with Crochemore's algorithm employs successive refinement from 1 up to half the trace size. However, the largest loop element in all cases was a small fraction of the trace size. Hence, the bulk of the time spent by the algorithm was *after* all the loops had already been discovered. Of course, in hindsight, the process could have been terminated earlier with the same results. However, there is no definitive way to be certain that optimal compression has been achieved, although heuristics can be developed based on the degree of compression already achieved.

The observations from the results of this optimal algorithm and the fact that it spent much of the time in processing that did not contribute to final compression was the motivation for us to develop a greedy compression algorithm. Greedy compression reduces the running time by reducing the length of the original string during compression as loops are discovered. Consider the trace of class C LU benchmark. Table IV shows that the smallest loop contains 4 symbols, so the reduction in trace size starts at level 4 by replacing those loops with new loop symbols. The largest loop has 1292 symbols, which contains two inner loops with 4 symbols iterating 160 times and another 12 symbols. The loops span 99.58% of the trace. Hence, after the next reduction, which happens at level 14, the trace size will be approximately (100-99.58) = 0.42% of the original trace size, and compression will be virtually over. In contrast, optimal loop discovery procedure will have to execute all levels with the the full trace size of 320348 until a level equal to half that size.

The results from greedy compression are presented and compared with the optimal compression results in Table V. The reduction in the execution time with the greedy approach is dramatic. The maximum compression time with the greedy

NPB Name	Trace Length	Greedy Compression Time (secs)	Optimal Compression Time (secs)	Major Loop Structure Discovered by Greedy Compression	Compression Ratio Optimal Algorithm	Compression Ratio Greedy Algorithm
BT B/C	17106	8.91	311.18	$(85)^{200} = (13 + (4)^3 + ... + (4)^3)^{200}$	201.25	201.25
SP B/C	26888	7.61	747.73	$(67)^{400}$	165.97	165.97
CG B/C	41954	8.48	2021.78	$(552)^{75} = (5 + (21)^{25} + 22)^{75}$	1104.05	524.43
MG B	8909	8.64	113.48	$(416)^{20}$	8.86	8.86
MG C	10047	10.88	144.54	$(470)^{20}$	8.99	8.99
LU B	203048	33.16	44204.82	$(812)^{249} = ((4)^{100} + (4)^{100} + 12)^{249}$	3384.13	3384.13
LU C	323048	61.9	113890.21	$(1292)^{249} = ((4)^{160} + (4)^{160} + 12)^{249}$	5384.13	5384.13

TABLE V
RESULTS OF GREEDY COMPRESSION

procedure ranges from 7.6 seconds to 61.9 seconds, versus the range from 113 seconds to 113,000 seconds for the optimal procedure. Clearly, this is a much more promising approach for large traces. Since the greedy approach is not guaranteed to always identify the outer loops correctly, we report the loop nests discovered and the compression achieved for each benchmark. For 6 of the 7 benchmarks, the loop nests discovered and the compression achieved were identical with the optimal and greedy approaches. One exception was the CG benchmark as noted in Table V. In this case the optimal procedure yielded a compressed trace size of 38, while the greedy procedure yielded a compressed trace size of 80. Clearly this is not a practical concern even though the degree of compression reported varies by a factor of 2. The reason for the difference is clear when the loop nest discovered for the CG benchmark as shown in Table V is compared with the loop nest discovered as shown in Table III. One discovered loop is offset due to the greedy procedure such that it has one less iteration as compared to the optimal loop nest, and as a result, there are more symbols outside of loop nests with greedy compression.

VII. ALGORITHMIC RESULTS

We have observed that the algorithms that we have developed generally return the best loop nest that is possible, or something very close. An important contribution of this work is theoretical results that provide specific bounds and more insight. Because of lack of space and complexity of the problem, the discussion here is abbreviated.

Our "optimal" algorithm is guaranteed to discover the longest span loop in a string at every step, but not necessarily the most compact loop nest overall. The two definitions of optimality are different although they lead to the same result except in in pathological cases. Further discussion is beyond the scope of this paper. More interesting to us is how the loop nest discovered by the greedy algorithm may differ from this optimal algorithm. In general, the following result holds:

The early reduction of inner loops corresponding to a fixed loop body can impact the identification of a longer span outer loop only as follows: the body of the discovered loop may be a reordering of the body of the original loop, and the number of iterations in the discovered loop may be up to 2 fewer than the number of iterations in the original longest span outer loop.

We have established this is indeed the case when one inner loop is reduced. The result is formally stated as follows:

Consider PM-triple L represented as (j, α, m) with $|\alpha| > 2$. Let β be another substring with $|\alpha| > |\beta|$. Suppose every PM-triple with β as the repeating substring is identified and reduced to a symbol. As a result of these reductions, if L is not identified as a PM-triple, then another PM-triple L' (j', α', m') will be identified where,

j' is between j and $j + |\alpha - 1|$,

α' and α are identical strings or one is a rotation of the elements of the other,

m' is between m and $m - 2$.

The proof is detailed in Appendix A.

The above result places a bound on the impact of reduction of one inner loop, on the discovery of an outer loop. *The result does not specifically address the impact of multiple loop reductions on the discovery of an outer loop.* In practice, it is still the case that the outer loop is generally discovered, with up to 2 fewer iterations. However, there are pathological cases where a long span outer loop is not discovered at all, even when there are more than 2 iterations. An example is the string $abcabcabcdbcdbcdbcdacdacdacdabdabd$ repeated 5 times with an optimal loop nest $((abc)^3(dbc)^3(dac)^3(dab)^2d)^5$. The greedy algorithm proceeds to reduce it as follows:

$(abc)^3\ (dbc)^3\ (dac)^3\ (dab)^3\ (cab)^2$ ....

The result is that the long outer loop with 5 iterations is not detected at all.

Additional machinery is necessary to ensure that results within the bounds discussed are always achieved when a series of loops is reduced with the greedy approach. Specifically, the greedy algorithm is modified as follows:

- loops with 3 or fewer iterations are not reduced.
- the loop element α of a newly discovered loop is compared against the loop elements of all previously discovered loops. In case the loop element α is a rotation of the loop element of an existing loop, say β, then a modified version of the new loop with loop element β is reduced, possibly with one fewer loop iteration.

With these changes the near optimality of the greedy algo-

rithm can be retained in general. For the above example, the final loop nest discovered is:
$(abc)^3\ ((dbc)^3(dac)^3(dab)^3c(abc)^2)^4\ (dbc)^3\ (dac)^3\ (dab)^2\ d$
The discovered outer loop has one less iteration than optimal.

A version of the algorithm with these additions is not implemented yet, and detailed analysis is omitted due to lack of space. However, we have not seen any instances in actual traces where this machinery is necessary to keep the compressed greedy traces withing the stated bounds.

VIII. Conclusion

This paper has presented an efficient and effective framework to identify complete loop nests in execution traces. The methodology constructs a loop nest from the repeating patterns implicitly identified by Crochemore's algorithm. A fast greedy approach is also developed. Experimental results demonstrate that the approach is effective in identifying loop nests in communication traces. Both the optimal and greedy approaches discover similar loop nests and deliver similar compression results. However, the processing time rises to hours with the optimal procedure for modest length traces of 1000s of MPI calls. The greedy approach provides virtually identical compression at a fraction of the execution time of the optimal method. The maximum compression time for our test suite was around one minute with the greedy procedure. Most importantly, unlike most compression heuristics, the greedy approach developed is theoretically proven to yield "near optimal" results.

While many compression algorithms exist, the efficient discovery of long range repeating patterns due to outer loops in an execution trace remains a significant challenge. Further, an important objective in trace analysis is to identify the parts of the trace corresponding to execution loops, and not simply to compress the trace. To the best of our knowledge, this is the first effort to discover the optimal loop nest in execution traces. The algorithms developed in this work have already been applied as a module in generation of executable performance skeletons from parallel application traces by identifying the dominant execution and communication patterns. However, the procedures developed are general and can be applied to trace compression and similar problems in a variety of scenarios. We believe this paper makes an important fundamental algorithmic contribution and represents a concrete step forward in analyzing execution traces for performance modeling and performance prediction.

IX. Acknowledgments

Support for this work was provided by the National Science Foundation under Award No. CNS-0410797 and CNS-0834750. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression." *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[2] C. Nevill-Manning and I. Witten, "Sequitur. http://SEQUITUR.info." [Online]. Available: http://SEQUITUR.info

[3] ——, "Identifying hierarchical structure in sequences: A linear-time algorithm," *Journal of Artificial Intelligence Research*, vol. 7, pp. 67–82, 1997.

[4] C. Nevill-Manning, I. Witten, and D. Maulsby, "Compression by induction of hierarchical grammars," in *Data Compression Conference*, Snowbird, UT, Mar 1994, pp. 244–253. [Online]. Available: citeseer.ist.psu.edu/129141.html

[5] M. Noeth, F. Mueller, M. Schulz, and B. Supinski, "Scalable compression and replay of communication traces in massively parallel environments," in *21th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, Long Beach, CA, April 2007.

[6] M. Crochemore, "An optimal algorithm for computing the repetitions in a word." *Inf. Process. Lett.*, vol. 12, no. 5, pp. 244–250, 1981.

[7] Q. Xu, J. Subhlok, S. Voss, and R. Zheng, "Logicalization of communication traces from parallel execution," in *2009 IEEE International Symposium on Workload Characterization*, Austin, TX, Oct 2009.

[8] Q. Xu and J. Subhlok, "Construction and evaluation of coordinated performance skeletons," in *The 15th annual IEEE International Conference on High Performance Computing (HiPC 2008)*, Bangalore, India, Dec 2008.

[9] X. Gao, A. Snavely, and L. Carter, "Path grammar guided trace compression and trace approximation," in *15th IEEE International Symposium on High Performance Distributed Computing (HPDC-15)*, Paris, France, June 2006.

[10] P. Weiner, "Linear pattern matching algorithms," in *FOCS*, October 1973, pp. 1–11.

[11] E. McCreight, "A space-economical suffix tree construction algorithm." *J. ACM*, vol. 23, no. 2, pp. 262–272, 1976.

[12] E. Ukkonen, "On-line construction of suffix trees." *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995.

[13] J. Stoye and D. Gusfield, "Simple and flexible detection of contiguous repeats using a suffix tree," *Theoretical Computer Science*, vol. 270, no. 1-2, pp. 843–856, 2002.

[14] D. Gusfield and J. Stoye, "Linear time algorithms for finding and representing all the tandem repeats in a string." *J. Comput. Syst. Sci.*, vol. 69, no. 4, pp. 525–546, 2004.

[15] A. Apostolico and S. Lonardi, "Some theory and practice of greedy off-line textual substitution." in *Data Compression Conference*, Snowbird, UT, Mar 1998, pp. 119–128.

[16] D. Gusfield, *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

Appendix

Impact of inner loop reduction on discovery of an outer loop

We present a set of results that will be employed to prove that the early reduction of an inner loop can impact the identification of a longer span outer loop only as follows: the body of the discovered loop may be a reordering of the body of the original loop, and the number of iterations in the discovered loop may be up to 2 fewer than the number of iterations in the original longest span outer loop. Recall that the notation (j, α, m) representing a PM-triple means that the corresponding PM-repeats start at location j in the string, the repeating substring is α and the number of repeats is m.

Lemma A.1: Suppose PM-triple L represented as (j, α, m) is *leftmost* meaning that there are no PM-repeats of length $|\alpha|$ starting left of L, from location $i-1$. Let $A = |\alpha| - 1$ and assume $m > 2$. Then there exist PM-triples $(i+1, \beta_1, k_1)$, $(i+2, \beta_2, k_2)$, ...$(i+A, \beta_A, k_A)$ such that β_i is a rotation of α and k_i is either m or $m-1$.

We refer to this group as the family of PM-triples corresponding to leftmost PM-triple L.

Proof: This result is stating the direct observation that, for every repeating sequence, starting with a forward offset smaller than the size of the repeating string yields another repeating sequence with at most one fewer number of repeats and with a rotated repeating string. ■

Lemma A.2: Let S be a string of symbols. Suppose there exist strings A and B such that: $S = AB = BA$. Then there must be another string C such that $S = [C]^k$. One implication is that S cannot be the repeating substring in a PM-repeat as it is not primal.

Proof:

1) If $|A| = 1$, then it can be easily shown that $S = [A]^k$, where $k = |S|$. Same holds if $|B| = 1$.
2) If $|A| = |B|$, then S is of the form $[A]^2$.
   If either of the above cases holds, the result holds.
3) Otherwise, without loss of generality, let $|A| < |B|$. Then we can define a string T such that $B = TA$. Now, we have $S = AB = ATA$, and $S = BA = TAA$. This implies that $S = ATA = TAA$.
   We define $S = S'A$, with $S' = AT = TA$. We again have:
   a) If $|T| = 1$, then $S' = [T]^{k'}$, where k' is $|S'|$. Hence, $S = [T]^k$, where k is $|S|$. Hence the result holds.
   b) If $|T| = |A|$, then $S' = [T]^2 = [A]^2$, and $S = S'A = [A]^3$. Hence, the result holds.
   c) Otherwise, S' can again be split as S was split in the previous level.

   The size of the string decreases by at least 1 in every new level. The result is proved based on the principle of induction.

■

Lemma A.3: Given two PM-triples L and S, with repeating substrings E_L and E_S, respectively, where $|E_L| > |E_S|$. If there is an overlap between any interior instance (i.e. all instances except the first and the last) of E_L and the span of S, then the length of the span of S cannot equal or exceed $|E_L|$.

Proof:

Without loss of generality, we assume that overlapping instances of E_L, and S are aligned at the left boundary as shown in Figure 2. If that is not the case, then the proof is generated with an aligned member of the family of PM-triples corresponding to PM-triple L as discussed in Lemma A.1.

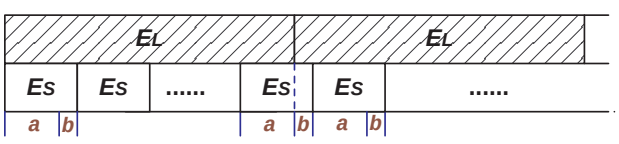

Fig. 2. Overlapping PM repeats

If span of $S = E_L$, then clearly PM-triple S is not maximal as repeats of E_S continue in the next instance of E_L. Hence that cannot be true.

Suppose the length of span of $S > E_L$ in contradiction to the result to be proved. Then one instance of E_S will cross the boundary between instances of E_L as shown in Figure 2. (Since the first instance of E_L is an interior instance, a following instance must exist.) We split this instance of E_S in two substrings a and b at the boundary as illustrated in Figure 2, i.e., $E_S = ab$. Now the left instance of E_L starts with E_S = ab, while the right instance of E_L starts with ba. But since they are repeats, they must be identical.

Hence we have $E_S = ab = ba$.

From Lemma A.2, $E_S = [x]^k$ for some x and k, which means that S is not a *primitive* repeat. Hence, by contradiction, the span of S cannot exceed E_L. Therefore, the span of S must be strictly smaller than $|E_L|$. ■

We now present the main result formally.

Theorem A.4: Consider PM-triple L represented as (j, α, m) with $|\alpha| > 2$. Let β be another substring with $|\alpha| > |\beta|$. Suppose every PM-triple with β as the repeating substring is identified and reduced to a symbol. As a result of these reductions, if L is not identified as a PM-triple, then another PM-triple L' (j', α', m') will be identified where,

j' is between j and $j + |\alpha - 1|$,

α' and α are identical strings or one is a rotation of the elements of the other,

m' is between m and $m - 2$.

Proof:

Let S be a PM-triple with repeating substring β that overlaps with an interior instance of α corresponding to L. We initially assume that no other PM-triple with repeating substring β overlaps with this instance of α. From Lemma A.3 we know that the entire span of S must be smaller than $|\alpha|$. Further we assume for now that the entire span of S is contained within a single instance of α.

Under the above scenarios, every instance of α in L will contain S as part of the substring at the same location, which will be replaced by the same symbol. Hence the identification of the PM-triple corresponding to L will be unaffected by reductions of S.

Now suppose the span of S is *not* contained within a single instance of α and crosses two instances. In that case the above result can be proved for another member of the family of PM-triples corresponding to PM-triple L as discussed in Lemma A.1, although the number of repeats (or iterations) may be reduced by 1.

Finally, there can be multiple PM-triples with repeating substring β that overlap with the same instance of α. However these instances themselves cannot be overlapping - otherwise it can be shown that they are not PM-repeats based on Lemma A.2. The impact of non-overlapping PM-triples can be serialized leading to the same result as for a single overlapping PM-triple above.

The final result is that the number of repeats in the recognized PM-triples can be up to 2 less than L since none of the results applies to the first or the last instance of α in L. ■