Measuring the Difference of Large Program Traces Based-on Levenshtein Distance

Abstract—Program traces play an important role in analyzing and understanding program behavior. However, large instruction traces are composed of millions of instructions, which in turn hinders their effective analysis. In this paper, we present an efficient algorithm based-on Levenshtein distance to measure the difference of large program traces. This algorithm calculates the similarity of two traces and reconstructs edit sequences, so that engineers know how similar the two traces are and where the difference is. We present a joint trace grouping, filtering and partition technology that significantly reduces the complexity of Levenshtein distance computation. We apply our algorithm to real program traces collected by a Pin-tool and the full system simulator GEM5. Experimental results show that our traces comparison algorithm is precise, efficient and scalable.

Index Terms—Edit distance, program trace, pin-tool.

1 Introduction

Instruction trace characterizes a program's dynamic behavior and is widely used for program optimization, debugging and new architecture evaluation. The comparison of program traces reveals important information about the behavior of two executions. However, low level program traces, such as instruction sequences, tend to be considerably large and usually contain millions of instructions, which in turn hinders their effective analysis. Some existing trace analysis tools work on high level program traces, such as control flow graph, call graph and internal event sequences, and rely on visualization techniques to help software engineers make sense of trace content [?] [?] [?] [?] [?]. Existing instruction trace analyzing tools suffer from a variety of limitations including complexity, inaccuracy and short length. In this paper, we propose a new algorithm to measure the differences of large traces based-on Levenshtein distance. This algorithm is precise, efficient and scalable. It can find out how similar the two traces are and where the difference is.

2 Measuring Trace Differences using Levenshtein Distance

Automatic trace comparison is an effective method for identifying the difference between two traces. However, as the scale of parallel benchmark and the number of processors in an individual system are continuously growing, the traditional approach of simply one-by-one instruction comparing becomes increasingly constrained by the large number of instructions. In this paper, we propose an efficient approach to measure the differences between two traces using Levenshtein distance [?]. We firstly describe the problems we met when analyzing large traces using Levenshtein distance, then we explain how to preprocess and partition the instruction sequences to reduce time and space complexity of difference measurement.

2.1 Levenshtein Distance and Its Problem in Trace Comparing

The similarity relation between two traces is defined in following aspects:

- Positional similarity: the degree to which matched instructions are in the same respective positions;
- Ordinal similarity: whether instructions are executed in the same order;
- Material similarity: the degree to which they consist of the same instructions.

This criteria of similarity is first given by Faulk to classify similarity relations between strings [?]. Program trace is a kind of string with "instruction" letters, thus similar issues should be addressed in traces comparison. One possible approach to solve this matching problem known from textual pattern recognition is the edit distance, which is one of the string metrics for measuring the amount of difference between two sequences. The edit distance between two strings is defined as the minimum number of edits needed to transform one string into another, with the allowable edit operations being insertion, deletion, or substitution of a single character. It is commonly known as Levenshtein distance [?] introduced by V. I. Levenshtein. For example, the Levenshtein distance between "traces" and "trabce" is 2, as a deletion of "b" after "a" and a insertion of "s" after "e" are required to change "trabce" into "traces".

Mathematically, the Levenshtein distance between two strings a, b is given by $lev_{a,b}(|a|, |b|)$, where

$$lev_{a,b}(i,j) = \begin{cases} 0, & i = j = 0 \\ i, & i > 0 \wedge j = 0 \\ j, & i = 0 \wedge j > 0 \\ min, & else \end{cases} \tag{1}$$

and

$$min = \begin{cases} lev_{a,b}(i-1,j) + 1 \\ lev_{a,b}(i,j-1) + 1 \\ lev_{a,b}(i-1,j-1) + [a_i \neq b_j] \end{cases} \tag{2}$$

Note that the first element in equation 2 corresponds to insertion, the second to deletion and the third to match or mismatch, depending on whether the respective symbols are the same.

We compare the code space of a program to an alphabet, in which each **letter** $\langle PC, INST \rangle$ denotes an instruction $INST$ at address PC in the space. Program traces are accordingly considered as long strings that are made up of millions of such **letter**s. From this, the similarity of two traces t_1, t_2 is given by

$$S_{t_1,t_2} = \frac{|comm(t_1,t_2)|}{\max(|t_1|,|t_2|)} \tag{3}$$

where $comm(t_1,t_2)$ denotes the piecewise best-matching of the two input traces. If t_1 and t_2 are a same trace, then $comm(t_1,t_2) = |t_1| = |t_2|$. Consequently, the larger the $comm(t_1,t_2)$ is, the more similar the two traces is.

Levenshtein distance is widely used in approximate string matching to find matches for short strings in many longer texts. It can be calculated using a $(|a|+1)\times(|b|+1)$ matrix to hold the distances between all prefixes of the first string and all prefixes of the second. In the end of the computation, each cell $D[i,j]$ will hold the value $Lev_{a,b}(i,j)$. The algorithm begins from the trivially known boundary values $D[i,0] = i$ and $D[0,j] = j$, and arrives at the value $D[|a|,|b|] = Lev_{a,b}(|a|,|b|)$ by recursively computing the value $D[i][j]$ from the previously computed values $D[i-1,j-1]$, $D[i,j-1]$, and $D[i-1,j]$ [**?**]. The straightforward implementation of this algorithm runs in $O(|a| \times |b|)$ time with the space complexity of $O(|a| \times |b|)$. If we do not need to reconstruct the edit sequence, the space complexity can be reduced, $O|a|$ instead of $O(|a| \times |b|)$, since it only requires that the previous row and current row be stored at any one time.

However, we need to reconstruct the edit sequence in trace comparison, so that we know where the difference is and how one execution diverges from another. In order to do this, we need to keep the whole matrix in the memory so that we can track back and reconstruct the edit sequence. The traces generated from one execution are very long and it is almost impossible to allocate such a large matrix to hold all the intermediate values. A large number of data dependencies during the computation renders the overall algorithm hard to efficiently parallelize. Hence, it takes much longer time to calculate the distances of two traces even on nowadays powerful machines.

2.2 Trace Preprocessing

The computation complexity of the matrix-based Levenshtein distance calculation depends on the length of input strings. Since instructions are generated from structural programs with inherent relations and orders, it provides the opportunity to reduce the length of input traces according to these features. In this section, we show a joint trace grouping, filtering and partition technology that significantly reduces the complexity of Levenshtein distance computation.

2.2.1 Trace Grouping

A basic block is a sequence of instructions with only one entry point and one exit point. Whenever the first instruction is executed, the rest of instructions in a same block are certainly executed once in given order. Based on this observation, program traces are partitioned into basic blocks, and instructions in a same basic block are grouped together. Each basic block is considered as a MACRO instruction and is represented by the last instruction of the same block. Existing statistics results show that each basic block contains 7 instructions in some benchmarks [0]. The number of MACRO instructions is about $1/7$ of the number of instructions in original trace. Instruction grouping does not change the result of Levenshtein distance calculation, but it significantly reduces the length of input traces.

2.2.2 Trace Intersecting

Two executions may execute different instructions in different code regions. This provides the opportunity to reduce the length of traces by removing basic blocks that only appear in one trace. Let B_E be the set of all basic blocks that visited by execution E, Then set $B_{E_1,E_2} = B_{E_1} \cap B_{E_2}$ contains all the basic blocks visited by both E_1 and E_2. $B^{'}_{E_1} = B_{E_1} - B_{E_1,E_2}$ and $B^{'}_{E_2} = B_{E_2} - B_{E_1,E_2}$ contains all the unique basic blocks visited by E_1 or E_2 respectively. Removing all the basic blocks that appear in one trace also does not change the result of Levenshtein distance calculation, and it helps to reduce the length of the two input traces.

2.3 Call Traces Comparison

In matrix-based Levenshtein distance computation, if both string a and b can be split into a number of substrings and Levenshtein distances are calculated for each substring, then both time and space complexity could be reduced. In general, this is difficult for text matching problems, because we do not know how to chop the strings into pieces so that the final distance can be calculated based-on the distances of substrings. Fortunately, program traces are generated from structural programs and we know some of the instructions have special semantic in the sequences. For example, the "call" and "return" instructions appear in pair at most time, and these pairs are usually structurally and hierarchically nested. If the two traces are the same, then the two execution have a same function invocation paths, on which instructions are executed with "call" and "return"s. Therefore, if we can rebuild the dynamic call paths from the two traces, we can firstly compare the two traces in a higher abstract level to check if the two traces diverge at some points on their call paths. In order to do that, we pick out the "call"s and "return"s from the two traces and generate another two shorter traces, which are called "call traces" of the two instruction traces. Then, we compute the Levenshtein distance for these two call traces and reconstruct the edit sequence. After that, we know a number of synchronization points(labeled by matched "call" and "return" instructions) in the traces, from where the two traces are chopped into small pieces.

Figure 1 shows an example of two call traces(c_1 and c_2) generated from two instruction traces(t_1 and t_2). Given that $pc_i = pc'_i (1 \leq i \leq 6)$ holds for each "call" pair and "return" pair, the two call traces match each other and therefore the Levenshtein distance for these two call traces is 0. Then we split the two instruction traces apart into several small fragments from the point of "call"s and "return"s. Next, Levenshtein distances are calculated for each pair of the fragments(t_{1i} and t_{2i}, $1 \leq i \leq 7$). The final distance for the two traces is given by

$$Lev_{t_1,t_2}(|t_1|, |t_2|) = \sum_{i=1}^{7} lev_{t_{1i},t_{2i}}(|t_{1i}|, |t_{2i}|) \quad (4)$$

The call trace, which is the frame of instruction trace with other instructions casted out, provides a higher and global view about the expatiatory instruction trace. This could be very helpful for the engineers who are dragged into the sea of instructions and cannot find their way out.

2.4 Instruction Traces Comparison

Algorithm 1 shows how the similarity S_{t_1,t_2} of two traces t_1 and t_2 is calculated. At first, two call traces c_1 and c_2 are generated from t_1 and t_2 respectively. Then we compute the edit distance between c_1 and c_2. The reconstructed edit distance e for t_2 contains all the operation of deletion, insertion and substitute that are performed on symbols in c_2 to transform c_2 into c_1. We remove all the "call"s and "return"s that are

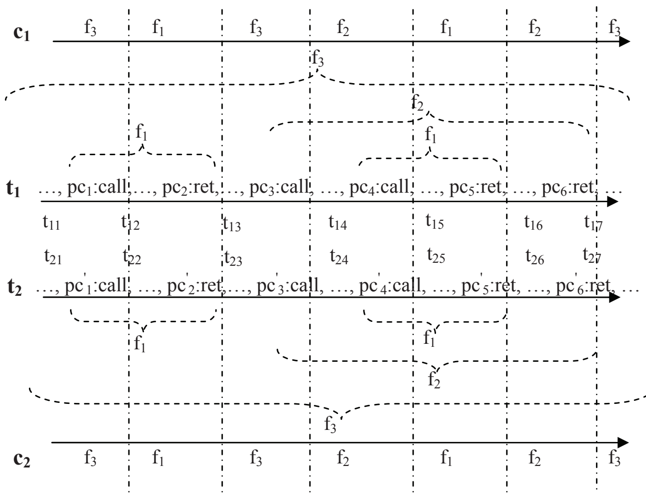
c1
f3 f1 f3 f2 f1 f2 f3
f3
f2
f1
f1
t1 …, pc1:call, …, pc2:ret, …, pc3:call, …, pc4:call, …, pc5:ret, …, pc6:ret, …
t11 t12 t13 t14 t15 t16 t17
t21 t22 t23 t24 t25 t26 t27
t2 …, pc'1:call, …, pc'2:ret, …, pc'3:call, …, pc'4:call, …, pc'5:ret, …, pc'6:ret, …
f1
f1
f2
f3
c2
f3 f1 f3 f2 f1 f2 f3

Fig. 1. Traces snippets generation and Traces partition

involved in e from c_1. The new call traces c'_1 generated from c_1 now only includes "call"s and "return"s that appear in both t_1 and t_2 with positional similarity and ordinal similarity. After that, we split t_1 and t_2 into a number of subsequences according to the "call"s and "return"s included in c'_1. We compare each pair of the subsequences and calculate the final similarity of the two traces using equation 3.

Algorithm 1: Trace similarity calculation

```
   Input: traces $t_1$ and $t_2$
   Output: Trace similarity $s$ and call trace edit
           distance $e$
1  Generate call traces $c_1$ and $c_2$ from $t_1$ and $t_2$
   respectively;
2  $S_c = Lev_{c_1,c_2}(|c_1|, |c_2|)$;
3  Reconstruct the edit sequence $e$ for $c_1$ and $c_2$;
4  Build the best-matched common sequence $c$
   based on $c_1$, $c_2$ and $e$;
5  Split $t_1$ into a collection of subsequences $S_1$
   based on $c$, where $S_1 = \{t_{1i} | 1 \leq i \leq |c'_1|\}$;
6  Split $t_2$ into a collection of subsequences $S_2$
   based on $c$, where $S_2 = \{t_{2i} | 1 \leq i \leq |c'_1|\}$ ;
7  $comm(t_1, t_2) = \emptyset$;
8  for $i = 1$ to $|c|$ do
9      $comm(t_1, t_2) = comm(t_1, t_2) \cup comm(t_{1i}, t_{2i})$;
10 end
11 $s = |comm(t_1, t_2|)/\max(|t_1|, |t_2|)$;
```

The reconstructed edit sequence for c_2 is also useful for understanding the differences of the two traces. In our implementation, the function names are also extracted from the output of objdump, so that we know where one call path diverges from another.

2.5 Parallel Instruction Trace Comparison

The matrix-based Levenshtein distance calculation algorithm is difficult to be parallelized for data dependencies. However, algorithm 1 in 2.4 is easy to be

TABLE 1
System configuration

	GEM5	Pin-tool
CPU	X86/2.0 GHz/4-core	Xeon E5606/2.13GHz/4-core
Memory	128 MB	24 GB
OS	Linux 2.6.22.9.smp	Linux ubuntu 2.6.38
Compiler	gcc 4.5.2	gcc 4.5.2

TABLE 2
Benchmarks

Benchmark	GEM5 trace length	Pin trace length
fft		
lu-non		
radix		
lu-con		
FMM		
Ocean		
Ocean-non		3

parallelized, as it has an independent loop (line 8-10) and all iterations can be computed concurrently without synchronization. In our implementation, each iteration is wrapped into a task and all the tasks are assigned to different working threads based on the working-stealing scheduling algorithm.

3 Evaluation

3.1 Evaluation Method

We extracted two different execution traces of a same binary independently using a Pin [?] tool and the full system simulator GEM5. The configuration of the host machine for Pin-tool and GEM5 simulator is given in table 1. Even though the simulated x86-Linux is different from the host machine in many aspects, we expect highly similar traces for a same executable binary. This is because the difference in micro-architecture dose not changes the order of instructions in one thread; however, in effect, they are not exactly the same due to Pin instrumentation, thread scheduling and synchronization. The operating system images and compilers are all provided by the GEM5 team. All the benchmarks are compiled with the "-static" options to make sure that no differences will be introduced into the traces by using different libraries. The features of the benchmarks we used are listed in table 2.

Figure 2 shows the calculated similarity(given by equation 3) between two traces collected by hacked GEM5 and a Pin tool respectively. It shows that the traces collected using GEM5 are much similar to the ones collected by the Pin-tool. Though the the similarity of call traces is as high as up to 98% for most benchmarks, the instruction traces exhibit a relative low similarity around 85%. The reason for this is that the instructions executed at the startup phase before the main function are not included in the GEM5 traces, which add up to about 10 thousands in total. Meanwhile, we found that the difference between call traces is slightly enlarged as the number of threads increases. According to the restructured edit sequence, we found that the target program spends much longer time at synchronization points than their sequential versions. That is why we observed high similarities for call traces on one side and low similarities for instruction traces on the other side.

Figure xx shows the scalability of parallelized algorithm 1 with the working-stealing scheduling policy. The baseline for speed measurement is the sequential version. It shows that our algorithm scales up well as the increasing of thread number.

4 Conclusion

We present an precise, efficient and scalable algorithm to measure the differences of two traces. This algorithm computes edit distances for traces and reconstruct edit sequences, so that we know how similar the two traces are and where the difference is. It will be very helpful for the users who need to analyze and understanding very large traces composed of millions of instructions. Experimental results show that our approaches are effective and efficient to compare large program traces.

Acknowledgments

References

Quantifying Behavioral Difference Between C and C++ programs

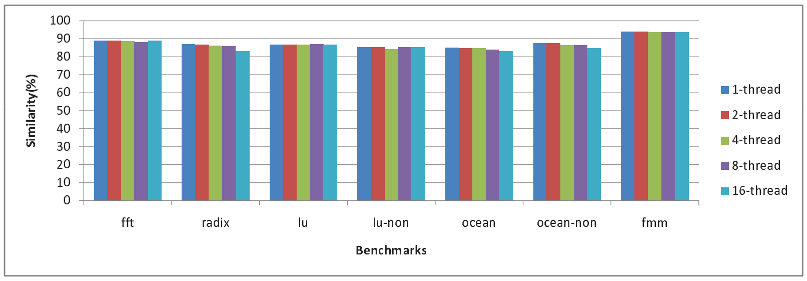

Fig. 2. Instruction trace similarity for I-X-S and I-X-P