Trace-Based Memory Aliasing Across Program Versions

Murali Krishna Ramanathan, Suresh Jagannathan, and Ananth Grama

Department of Computer Science,
Purdue University,
West Lafayette, IN 47907
{rmk, suresh, ayg}@cs.purdue.edu

Abstract. One of the major costs of software development is associated with testing and validation of successive versions of software systems. An important problem encountered in testing and validation is memory aliasing, which involves correlation of variables across program versions. This is useful to ensure that existing invariants are preserved in newer versions and to match program execution histories. Recent work in this area has focused on trace-based techniques to better isolate affected regions. A variation of this general approach considers memory operations to generate more refined impact sets. The utility of such an approach eventually relies on the ability to effectively recognize aliases.

In this paper, we address the general memory aliasing problem and present a probabilistic trace-based technique for correlating memory locations across execution traces, and associated variables in program versions. Our approach is based on computing the *log-odds ratio*, which defines the affinity of locations based on observed patterns. As part of the aliasing process, the traces for initial test inputs are aligned without considering aliasing. From the aligned traces, the log-odds ratio of the memory locations is computed. Subsequently, aliasing is used for alignment of successive traces. Our technique can easily be extended to other applications where detecting aliasing is necessary. As a case study, we implement and use our approach in dynamic impact analysis for detecting variations across program versions. Using detailed experiments on real versions of software systems, we observe significant improvements in detection of affected regions when aliasing occurs.

1 Introduction

Identifying memory aliasing involves correlating memory locations that exhibit similar behavior across two versions of a program. Memory aliasing occurs in many software engineering applications, including impact analysis [1, 15, 17], detecting invariants [7] and correlating program properties to ensure that properties are preserved in the newer version or matching program execution histories [19]. Devising a scalable robust solution to this problem has proven to be challenging. Zhang and Gupta [19] present a relative offset based matching approach to solving the problem of matching program execution histories. While

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 381–395, 2006.

their approach performs well in the presence of simple variable renaming, it is not clear how their technique could be generalized to deal with more complex program behavior. In this paper, we present a scalable and general solution to the memory aliasing problem based on available execution traces.

Memory aliasing is the problem of identifying whether two pointers refer to the same memory location. In this paper, we address the problem of memory aliasing across program versions, i.e., given two sets `X` and `Y` of memory locations corresponding to two different versions, identify whether `x` and `y` are aliases (refer to the same memory location relatively), where $\texttt{x} \in \texttt{X}$ and $\texttt{y} \in \texttt{Y}$. We present a solution for the memory aliasing problem in the context of identifying variations across program versions. We use test results on older versions to automatically identify regions in newer versions that are affected by the changes that characterize their differences. As a first step towards detecting and isolating variations in program versions, we abstract a program as a sequence of memory reads and writes. Test inputs are fed into two versions and traces of memory operations are collected by instrumenting the binary executables. A trace is a sequence of $<$*Operation, Value*$>$ tuples, where *Operation* is either a read or write to memory and *Value* is the value read from, or written into memory. The trace is analogous to a string and the tuple analogous to an alphabet. Comparing two functions that exist in two program versions is equivalent to comparing the subsequence of the trace corresponding to the two functions under comparison.

Based on a user-defined cost function, the Levenstein [12] distance is calculated using dynamic programming and the gaps [2] in the comparison recorded. (The Levenstein distance between two strings is defined as the shortest sequence of edit operations that lead from one string to the other.) By repeating the process for multiple test inputs, cumulative information on the gaps present in the older version relative to the newer version is obtained. By projecting the tuples back to the corresponding regions in the source, information on the affected locations within an impacted function is obtained. If the Levenstein distance between the two functions is zero, then we regard the function in the newer version as unaffected by changes in the older version.

Ignoring memory locations associated with each operation in the trace and using only the operation type and associated value in calculating the Levenstein distance presents a potential problem as shown in the following example:

```
void  old() {                        void  new() {
    for(i = 0; i < 100; i++) {           for(i = 0; i < 100;i++) {
        a = i;                                   b[i] = i;
    }                                    }
}                                     }
```

Using just operation types and associated values in traces, it is easy to conclude that the functions `old` and `new` are identical, since the Levenstein distance associated with strings generated by traces on any test input is zero. However, it is obvious by examination that the functions have clearly distinct behavior.

In function `old`, values from 1...100 are written into a single memory location whereas in function `new`, the same values are written into consecutive memory locations.

To alias memory locations across versions, we rely on a model [6] used to solve a similar problem in computational biology while matching amino-acid (protein) sequences. Here, one amino acid needs to be aliased with another amino acid from which it potentially has mutated. The solution is based on computing the probability of one amino acid aligning with another amino acid in valid (known) alignments. In our approach, we execute the two versions on a test input and align the traces with respect to the operation types and the associated values (ignoring memory locations). The alignment of the memory locations for the alignments of the strings obtained using the previous step is used in computing the probability that a memory location can be aliased with any other location. This process is performed for a few test inputs and a comprehensive map of the probability that a memory location in the older version is aligned with any other location in the newer version is obtained. From this map, the memory location that has the highest probability is used as an alias. In subsequent alignment of the traces, apart from computing the equality of the operation type and values associated across versions, memory alias information computed earlier is used.

We implement our approach and evaluate its performance across a range of open-source benchmarks. In these benchmarks, the majority (approximately 90%) of the memory locations in the older version are uniquely associated with locations in the newer version. The remaining few memory locations correspond to multiple locations and these are the locations for which our approach has been found useful. When our technique is used in detecting variations between program versions, we find a significant change in the size of the impacted regions within an affected function. Furthermore, in some cases, we also find improved precision in the impact sets computed.

2 Aliasing Technique: Log-Odds Ratio

We present our aliasing technique by initially discussing a related problem in biology. Amino acid sequences of an organism's protein mutate gradually from one generation to another in the process of evolution. An important application is to determine whether two sequences are homologous or have the same ancestor. The general technique is to construct a substitution matrix where entry `(i,j)` is equal to the probability of the amino acid `i` being altered into amino acid `j` within a bounded time. There are two popular techniques to construct such a matrix in the literature: PAM [6] and BLOSSOM [9]. In this paper, we build a substitution matrix based on the technique used for PAM [6]. For our problem, we need to determine the probability that the memory location `i` in one version being the memory location `j` in another version.

We start with a discussion of the probabilistic technique used to correlate memory locations. We initially present an abstract problem in strings and then relate it to the problem of correlating memory locations that arise in software

engineering and testing environments. We are given two sets of alphabets A and B, where A ⋂ B = {}. Consider the following problem: "find a mapping from alphabets in A to alphabets in B, for all alphabets in A, such that an alphabet in A or B can have no more than a single mapping and for two strings x and y composed of alphabets in A and B respectively, the Levenstein distance of the strings x and *sub(y,x)* is minimum. Here, x is a string composed of alphabets from A, y is a string composed of alphabets from B, *sub(y,x)* is the string obtained by converting the string y by substituting each character in y by the alphabet mapped into A." For example, if we have A = {*a, b, c*}, B = {*d, e, f*}, x = *abcb* and y = *eddd*, we find that a mapping from b to d, a to e and c to f results in the smallest Levenstein distance of the strings x and *sub(y,x)*. We are unaware of a polynomial-time optimal solution to the above problem.

A relaxation of the above problem incorporates the presence of a *history*, i.e., there is a set of pairs of strings x and y, and alignment $R(x, y)$ associated with each pair. Any new pair of strings that form an input to the problem follows the historical pattern. Given this scenario, a probabilistic approach to mapping alphabets from A and B is given below. This technique has been applied in generating PAM matrices for amino acid substitutions in computational biology. For each pair of strings x and y in the history:

1. Calculate $p(a_i, b_j)$ by dividing the number of times a_i aligns opposite b_j in $R(x,y)$ by the total number of aligned pairs.
2. Calculate $p(a_i)$ by dividing the the number of times a_i appears in x by the length of x. Similarly calculate $p(b_j)$.
3. The log-odds ratio $\mathtt{LR}(a_i, b_j)$ is defined as $\log(\frac{p(a_i,b_j)}{p(a_i)p(b_j)})$

x: a1 a2 a3 a1 a1 a2 a3 a1 a1 a2 y: b4 b2 b1 b4 b3 b2 b3 b4 b4 b2

	b1	b2	b3	b4
a1	0	0	0	1
a2	0	1.74	0	0
a3	2.32	0	1.32	0

p(a1) = 0.5
p(a2) = 0.3
p(a3) = 0.2
p(b1) = 0.1
p(b2) = 0.3
p(b3) = 0.2
p(b4) = 0.4

p(a1, b4) = 0.4
p(a1, b3) = 0.1
p(a2, b2) = 0.3
p(a3, b1) = 0.1
p(a3, b3) = 0.1

Fig. 1. An example illustrating the computation of log-odds ratio

We present an illustrative example in Figure 1. A contains alphabets {a1, a2, a3} and B contains alphabets {b1, b2, b3, b4}. Strings x and y are given and are shown in the figure. The probability of occurrence of each alphabet and the joint probability of two alphabets aligning is also shown. The log-odds ratio is

calculated as specified by the formula $\frac{p(a_i,b_j)}{p(a_i)p(b_j)}$ and is specified in the table. We observe that there is perfect aliasing (when $p(a_i, a_j) = p(a_i) = p(a_j)$) between `a2` and `b2`. In contrast, there is no such aliasing between `a1` and `b4`. However, since `a1` aligns more frequently with `b4`, these two alphabets are aliased. Even though `a3` aligns with two different alphabets once, we alias it with `b1`, since it occurs only once in the string y. However, the history of pairs of alignments need to be taken into consideration to specify the alias with high confidence. We give the details below.

Compute the log-odds ratio for all possible pairs of alphabets in `A` and `B`. Iterate the process for each pair of strings in the history and obtain the cumulative log-odds ratio. On completion, for every alphabet a_i, find the alphabet b_j in `B` for which the log-odds ratio is the highest. Map the alphabet a_i to b_j, if b_j is not already mapped. Notice that $\texttt{LR}(a_i, b_j)$ can become zero, in the absence of even a single alignment between the alphabets. In practice, we observe many such pairs and these pairs can be removed immediately from consideration.

In the context of memory aliasing, the problem can be now described as follows: `A` and `B` correspond to the set of memory locations in the older version and newer version, respectively. In other words, a memory location is simply an alphabet in the appropriate set. Sequences x and y represent sequences of memory locations operated in each of the versions. To obtain the history as mentioned in the relaxed version of this problem, a *learning* process is executed. In this process, the memory locations are totally ignored and the traces, obtained by executing the versions on the same test input, are aligned using the tuple $<$*Operation*, *Value*$>$. This alignment is performed using dynamic programming. After obtaining the alignment of the trace, an alignment of the memory locations (corresponding to the tuple) across the two versions is obtained. The log-odds ratio for the locations is subsequently determined. On completion, we obtain a probabilistic aliasing of the memory locations. We elaborate on this process using the following example.

2.1 Example

We show two program fragments in Figure 2, one labeled `old`, and the other `new`. There is one memory location (`a`) associated with function `old` (local variable `i` is not considered). There are two memory locations (`b[0],b[1]`) associated with function `new`. We denote the address of any variable `v` as `mem(v)`.

Using the algorithm presented above, memory traces associated with the invocation of these functions on the same test input (`j` = 5) are first obtained. The memory trace thus generated is:

```
old: <mem(a),W,0>, <mem(a),W,1>, <mem(a),W,5>
new: <mem(b[0]),W,0>, <mem(b[1]),W,1>, <mem(b[0]),W,5>, <mem(b[1]),W,6>

Trace Element: <Location, Operation, Value>
Location: Memory location associated with the operation
```

```
void old(int j){                    void new(int j){
  for(i = 0; i < 2; i++) {            for(i = 0; i < 2; i++) {
      a = i;                              b[i] = i;
  }                                     }
  a = j;                                b[0] = j;
                                        b[1] = j + 1;
}                                   }
```

Fig. 2. Example of functions from two program versions

```
Op: Read(R),Write(W)
Value : 32 bit value
```

Initially, we ignore all memory locations and align the above strings. The gaps are represented by a hyphen(-). We obtain the following alignment:

```
old: <W, 0>, <W, 1>, <W, 5>, -
new: <W, 0>, <W, 1>, <W, 5>, <W, 6>
```

We retrieve the corresponding memory locations and get the following alignment:

```
old: mem(a), mem(a), mem(a), -
new: mem(b[0]), mem(b[1]), mem(b[0]), mem(b[1])
```

The logs-odd ratio for the memory locations are obtained using the above alignment. We get $LR(\mathtt{a}, \mathtt{b}[0])$ equals $log(4/3)$ and $LR(\mathtt{a}, \mathtt{b}[1])$ equals $log(2/3)$. By performing the above operations on other test inputs, we can correlate the memory locations of `a` and `b[0]` with high confidence. In subsequent trace alignments, we use the alias in the dynamic programming to ensure that the alphabets under comparison are identical.

3 Case Study

We discuss an implementation for discovering variations across program versions. Our analysis tool consists of two components – an instrumentation module and a comparison module. Both components operate over program binaries. The binaries, representing a program and its revision, are instrumented to record read and write operations, and execute on the same test input. The effect of the instrumentation yields memory traces on selective operations. These traces are then compared using dynamic programming, and optimally aligned (with or without memory aliasing) based on a user defined cost function. A block diagram of this process is shown in Figure 3.

Gaps in the alignment help detect operations performed by the newer version that are absent in the older version, and vice versa. Accumulating this information over all test inputs provides the set of affected regions in the newer version.

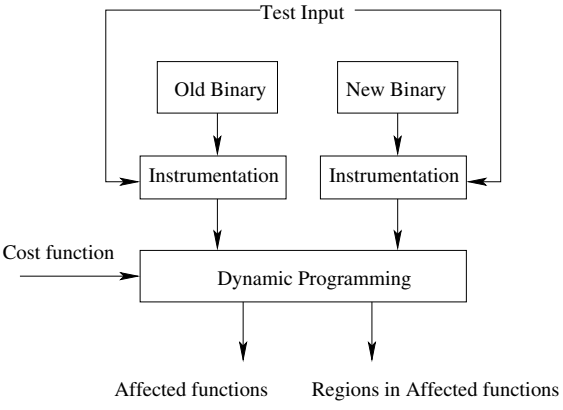

Fig. 3. Block Diagram for detecting variations across two program versions

If there are no gaps present in such a comparison over all test inputs, the functions are declared to be unaffected. Otherwise, the affected regions (in the form of line numbers) in the newer version are identified.

3.1 Instrumentation Tool Using PIN

We use PIN [16], a dynamic binary instrumentation tool, for instrumentation purposes. PIN supports a rich set of abstract operations that can be used to analyze applications at the instruction level without detailed knowledge of the underlying instruction set. Instrumentation code can be inserted at desired locations in the binary. For our current implementation, we track all heap related operations ignoring other instructions, including reads or writes to the stack.

The instrumentation module takes as input the binary and the list of functions in the binary that need to be instrumented. When the binary is executed on a given test input with dynamic instrumentation, a list of tuples is generated. The elements in the tuple include the type of operation (read or write), its 32 bit value (read or written), the corresponding memory location, the line number and the function in which the instruction was generated.

3.2 Comparison Tool Using Dynamic Programming

We provide a simple example for aligning two strings optimally. Given two strings `abacbd` and `aabcabcd`, one of the longest common subsequences is `abacd`. Table 1 presents the dynamic programming table d that gives the edit distance between the two strings. The cost at any box of the dynamic programming table d, $d_{i,j}$ is calculated as follows. If alphabets i and j are equal, then the cost $d_{i,j}$ is the minimum of $d_{i-1,j-1}$, $d_{i-1,j} + 1$ and $d_{i,j-1} + 1$. Otherwise, the cost $d_{i,j}$ is the minimum of $d_{i-1,j} + 1$ and $d_{i,j-1} + 1$. The alignment table is correspondingly update to reflect whether the diagonal, left or top element is chosen in d. The edit distance in this case is four assuming unit cost for insertions and deletions. After filling up all the values in table d, a traversal from the end of the alignment table

Table 1. Dynamic programming table d representing the gap costs

		a	a	b	c	a	b	c	d
	0	1	2	3	4	5	6	7	8
a	1	0	1	2	3	4	5	6	7
b	2	1	2	1	2	3	4	5	6
a	3	2	1	2	3	2	3	4	5
c	4	3	2	3	2	3	4	3	4
b	5	4	3	2	3	4	3	4	5
d	6	5	4	3	4	5	4	5	4

Table 2. Alignment table. ℓ: left, t: top, c: diagonal.

		a	a	b	c	a	b	c	d
		ℓ	ℓ	ℓ	ℓ	ℓ	ℓ	ℓ	ℓ
a	t	c	ℓ	ℓ	ℓ	ℓ	ℓ	ℓ	ℓ
b	t	t	t	c	ℓ	ℓ	ℓ	ℓ	ℓ
a	t	t	c	t	t	c	ℓ	ℓ	ℓ
c	t	t	t	t	c	t	t	c	ℓ
b	t	t	t	c	t	t	c	t	t
d	t	t	t	t	t	t	t	t	c

(the last row and last column) gives the alignment of the two strings. Table 2 presents the table that is used to calculate the optimal alignment for the pair of strings. One possible alignment for the strings is as follows: `a-b-a-cbd` and `aabcabc-d`. We refer the reader to [5] for a more elaborate discussion.

Our comparison module operates over a pair of traces generated by instrumenting the binaries to be compared as they execute on the same input. To provide an analogy, if the trace is viewed as a string, the equivalence of an alphabet in the string here is a tuple <*Memory Location*, *Operation*, *Value*>. We use the dynamic programming technique to compute an alignment between the pair of traces. The optimality of an alignment is dependent on the cost function used which can be defined in many ways. In this paper, we consider a simple notion of optimality. Gaps in an alignment have unit cost, while all other alphabets have zero cost. Thus an optimal alignment is one that has the smallest number of gaps; observe that for any pair of strings, there maybe many such optimal alignments.

Given memory traces of length m and n for two versions, the time complexity of dynamic programming is $O(mn)$. Thus, even traces of modest length (approximately 15K) can considerably slow down the comparison process. Indeed, for some applications, there are a several million reads or write operations to memory. To make our approach scalable, we employ a heuristic that performs dynamic programming piece-wise on smaller substrings. The heuristic is based on the observation that there is likely to be sufficient locality to apply dynamic programming on the strings yielded by subtraces to yield a good, if not necessarily optimal, alignment. Our heuristic works as follows:

1. Obtain a prefix of fixed length r from both traces.
2. Apply dynamic programming on the prefixes obtained.
3. Find the farthest location in each prefix respectively after which there is no alignment between the prefixes.
4. Obtain a prefix of r starting from these locations respectively from each trace and repeat the process from Step 2.

We use the example discussed above to explain the heuristic. Fix r to be three. In the first step, prefixes `aba` and `aab` are extracted. Aligning these prefixes,

we get `-aba` and `aab-`. In the next step, we extract `acb` from the first string and `cab` from the second string. Aligning the prefixes, we get `ac-b` and `-cab`. Subsequently, we extract `d` and `cd` and align them as `-d` and `cd` respectively. The final alignment is `-abac-b-d` and `aab-cabcd`.

3.3 Extract from Bzip2

We present an extract from `bzip2` to show that ignoring memory locations can indeed lead to reduced precision in detecting variations across program versions. The following piece of code is extracted from the function `generateMTFValues` in `bzip2`, version 0.9.5d.

```
163 void generateMTFValues ( EState* s )
213       ll_i = s->unseqToSeq[block[j] >> 8];
225             j++;
233             zPend--;
240                   s->mtfFreq[BZ_RUNA]++;
247          mtfv[wr] = j+1; wr++; s->mtfFreq[j+1]++;
```

The following piece of code is extracted from the same function in bzip2, version 1.0.2

```
164 void generateMTFValues ( EState* s )
211       ll_i = s->unseqToSeq[block[j]];
219             zPend--;
226                   s->mtfFreq[BZ_RUNA]++;
250             mtfv[wr] = j+1; wr++; s->mtfFreq[j+1]++;
```

In the *learning* process for aliasing, it was observed that memory location of variable in `line 225` in the older version is associated with `lines 211,219, 226, 250`. However none of the lines have a high log-odds ratio with `line 225`. All of them have higher log-odds ratio with some other line in the older version (`lines 213, 233, 240, 247,` respectively). By examining the source code of both the versions, it is obvious that the loop associated with variable `j` has been rewritten and a similar construct is not available in the newer version. By ignoring memory locations, `line 225` was aligning with other unrelated lines, leading to imprecise alignments and therefore reduced precision in the impact analysis. As our experimental results show, memory aliasing can reduce the number of realignments that are observed in affected functions.

4 Evaluation

4.1 Experimental Setup

We provide a comparison of detecting variations across program versions with and without memory aliasing using two versions of the following software packages: `bzip2` [4], `bunzip2` [4], `gawk` [8], `htmldoc` [13] and `wget` [18]. All of these programs are written in C. The details on the versions used for the benchmarks,

the lines of code, the number of functions and other parameters are given in Table 3. We explain the significance of the other columns below. The test cases used for the benchmarks are either randomly generated or are from standard test suites available for them.

Table 3. Benchmark Information and Results (Time in seconds)

	Old Version	New Version	LoC (in K)	Total Fns.	Longest Trace (10^3)	Total Tests	Instr. Time	Analysis Time		Memory (in MB)	affected %
								No Alias	Alias		
`bzip2`	0.9.5d	1.0.2	9	107	6099	107	2631	991	3121	351	25.4
`bunzip2`	0.9.5d	1.0.2	9	107	1839	101	1297	351	2637	89	26.6
`gawk`	3.1.3	3.1.4	41	522	3598	133	1390	450	368	670	41.7
`htmldoc`	1.8.23	1.8.24	65	246	1399	101	3451	970	996	84	48.4
`wget`	1.6	1.7	28	313	158	105	1025	263	232	16	44.4

We perform our tests on Linux 2.6.11.10 (Gentoo release 3.3.4-r1) system running on a Intel(R) Pentium(R) 4 CPU 3.00GHz with 1GB memory. The version of the PIN [16] tool used was a special release 1819 (2005-04-15) for Gentoo Linux. The sources were compiled using GCC version 3.3.4.

4.2 Results

In our current implementation, a list of functions that need to be instrumented and the pair of functions to be considered for comparison are also provided. The number of memory reads and writes, the associated values yielded, the memory locations used, and the line in the source responsible for such an action is presented as output of the instrumented program executed under PIN. By performing this process for both versions, we have two traces of heap reads and writes, and corresponding information that is provided as input to the comparison module. The instrumentation time given in Table 3 includes the time taken to insert instrumentation code and time taken to execute the instrumented version of the binary for all the test cases.

When no aliasing is employed, the operation values and types are compared and aligned. We use a block size of 50 based on the heuristic given in the previous section for a piece-wise alignment of the traces. Furthermore, related memory locations are aligned and the log-odds ratio computed as mentioned earlier in the paper. The comprehensive log-odds ratio is computed by totalling the individual ratios. It is appropriate to note here that memory locations in any two different runs of the same program may be different. Therefore, to ensure consistency, we associate the memory location with the line number in the source. Notice that this approximation may lead to a loss of precision in the presence of two or more memory locations in the same line. Determining a better technique to ensure consistency across test runs is part of our ongoing research. The idea behind the current approximation of memory locations to line numbers is that even though the locations may be different across two runs of the same program, the line numbers are still consistent. More importantly, such an approach can

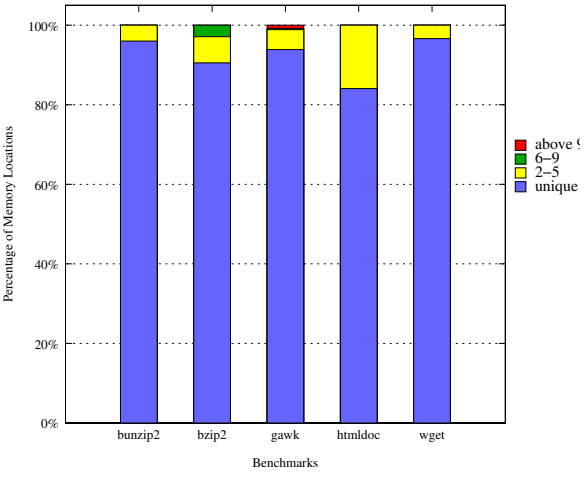

Fig. 4. The percentage of memory locations and the mappings associated

be deployed across program versions as the line numbers need not correlate. On performing the entire process as mentioned above for each test case, we obtain the regions (in the form of line numbers) in the newer version that differ from the older version.

In Figure 4, we present a histogram, which shows the distribution of the memory locations with respect to the number of mappings for each of the benchmarks. We expect that the majority of the locations have unique correspondence, since generally newer versions do not deviate widely from the existing version. Our intuitions are confirmed by the results shown in the figure. Most memory locations (approximately 90%) in the older version have unique correspondence (perfect mapping) with memory locations in the newer version. The remaining locations have multiple mappings and these are the locations that can reduce the effectiveness of impact analysis. We observe that among these memory locations, most of them correspond to two or three other memory locations in the newer version. In the case of `gawk` [8], we observe as many as 20 different locations in the newer version that can be mapped to one memory location. However such occurrences are rare. Such increased correspondence signifies that the memory location cannot be aliased with any other memory location and the observed alignments are accidental. Our extract from `bzip2` [4] in the previous section corroborates this observation.

When aliasing is used in our comparison tool, we ensure that the memory locations associated with each memory operation are compared and two tuples are aligned if and only if the operation value and type are equal and the associated memory locations are aliases. In Figure 5, we present the results of the improvement using this approach. The figure shows the distribution of functions with respect to the change (in percentage) of the number of lines within an impacted function when aliasing is used, as opposed to no aliasing. We observe that there is a significant change in the number of lines affected, even though only a few

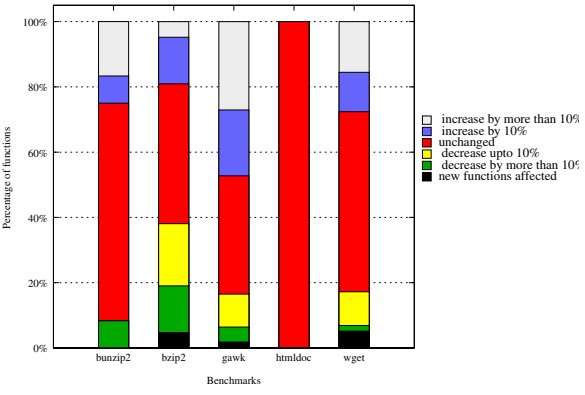

Fig. 5. Change in size of impacted regions when aliasing detection is used

memory locations had multiple mappings (from Figure 4). Approximately 50-60% of the functions do not change as a result of using aliasing. In some cases, for example in `htmldoc` [13], we observe that even though the uniquely mapped memory locations is the least over all the benchmarks, there is, nonetheless, no change in the affected regions.

In Table 3, we provide the specifics of our benchmarks and the results obtained using our technique. The number of lines of code varies from 9K to 65K with the number of functions varying from 100 to 500, approximately. The length of the trace represents the number of reads and writes to the heap in thousands of instructions. The longest trace observed is approximately 6 million for `bzip2`. The average memory used, while significant, is not problematic. This is expected for many dynamic analysis scenarios because precise information on heap operations is being gathered. No significant difference in memory utilization was observed when aliasing was used and is therefore not shown as a separate column in the table. The percentage of affected regions is also provided in the table. The affected percentage reveals that a sizeable fraction of the newer version of a benchmark program is impacted by changes to the older.

The time taken for our technique is composed of the instrumentation time of the binary and execution time of comparison module. There are two reasons for the inefficiency of the instrumentation process. The first is the use of a dynamic binary instrumentation tool as opposed to static instrumentation. Therefore for each test case, we insert appropriate instrumentation code. We believe the time taken for this can be significantly reduced using alternative instrumentation strategies. The current impact analysis tool currently tracks all heap related operations. This can also play an important role in increasing the instrumentation time. The length of the trace and instrumentation time are directly correlated. For example, `wget` has a shorter trace and thus significantly smaller instrumentation time compared to `bzip2`. One way to reduce the number of heap operations tracked is to discard operations found in regions already known to have been

affected from previous test runs. The difference in the analysis time of the two approaches is not significant except for `bzip2` and `bunzip2`, where we observe that the approach using aliasing requires more time. Note that in the aliasing approach, since memory locations are compared for every alphabet mapping, the increased time can be attributed to the number of memory locations occurring in the benchmark. Since we expect that our approach will be generally used offline, we believe the improved precision resulting from taking aliasing information into account outweighs the added costs for performing location comparisons.

5 Related Work

Our approach is motivated by similar techniques employed for problems in bioinformatics. PAM [6] and BLOSSOM [9] are two commonly used substitution matrices for detecting amino acid substitution. Amino acids can mutate with time and it is necessary that two different amino acids be aliased to give a better sequence alignment. Analogously, we have multiple versions (the newer version can be considered as the mutation of the older version) and to align these newer versions, it is necessary that the memory locations in the two versions be aliased. We address this problem in the paper and use the log-odds ratio to alias memory locations. We also show by way of experiments on many open source systems that the memory aliasing approach employed here enhances the precision of the impact analysis.

Zhang and Gupta [19] present a novel method for matching dynamic execution histories across program versions for detecting bugs and pirated software. They use an offset-based aliasing technique to correlate stack locations across the two programs. While this is applicable in many cases, it is not clear whether this approach is generalizable. In this paper, we abstract the problem of memory aliasing into one of finding longest common subsequence of two strings composed of different alphabets. We also present a probabilistic solution to this problem.

There has been much prior work in impact analysis for improving testing efficiency in the presence of program changes [1, 15, 17]. In these approaches, functions that follow a modified function in some execution path are added to the impact set. We share obvious similarities with these efforts, but differ both in the mechanisms used to identify impacted functions, and the ability to identify localized regions of change within these functions. Furthermore, since our technique operates over binary executables, we are not reliant on program analysis of input sources or programmer annotations.

Our approach can also be extended to correlate variables across program versions to check whether the invariants are preserved across these versions. Ernst *et. al.* provide a technique for automatically detecting invariants within a single program version [7]. However, it is not clear how invariants across program versions can be correlated, in the presence of variable renaming, deletions, additions, and general changes in the program's data- and control-flow . By proposing a

new formulation for generating the initial history of alignments and applying the log-odds ratio, we believe variables can be correlated even under these circumstances.

Pointer analysis is a well studied problem and a number of techniques in the literature is discussed by Hind and Pioli [11]. "A pointer alias analysis attempts to determine when two pointer expressions refer to the same storage location" [10]. However, many of the techniques discussed in the literature is for alias analysis within a single program. In this paper, we address an entirely different problem of 'must' aliasing [14] of pointers across two program versions. The technique presented in the paper provides a solution based on statistical correlation.

Dynamic programming, more specifically longest common subsequence (LCS), is used in many applications. One such application in software engineering is described in [3]. The foundation of their approach that for similar bugs, the call stack also shares similarities. Therefore, by pruning unnecessary information from the call stack, and comparing the resulting string representation with an existing signature, a score can be computed to the match using a longest common subsequence algorithm. The similarity between their approach and ours is restricted to the underlying technique and its applicability in a software engineering context, but does not extend to impact analysis or variation detection across program revisions.

6 Conclusions

This paper describes a technique for identifying memory aliases across program versions. Our approach works by collecting traces of program executions, in which each element of the trace contains an operation, value, and a memory location; trace results applied to the same test input on the versions being compared are aligned without considering the memory locations used by the program. By computing the log-odds ratio between memory locations based on the above alignments on a few test inputs, we obtain an aliasing of the locations across versions. To validate our approach, we compare program versions based on traces with and without aliasing detection on a number of open-source programs. Experimental results show that our technique improves the precision of identifying impacted regions significantly. Our approach can be easily extended to other applications where memory aliasing is required. For example, as part of ongoing work, we are investigating the use of this approach for correlating variables across program versions to test preservation of invariants and program matching. Another avenue for future work is to evaluate the applicability of our approach to more heap-centric languages such as Java.

Acknowledgements. This work is supported by National Science Foundation Grants CCF-0444285, CNS-0334141 and STI 501-1398-1078. We also like to thank the anonymous reviewers for their constructive feedback.

References

[1] T. Apiwattanapong, A. Orso, and M. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 432–441, 2005.

[2] http://www.ncbi.nlm.nih.gov/education/ blastinfo/information3.html.

[3] M. Brodie, S. Ma, G. Lohman, T. Syeda-Mahmood, L. Mignet, N. Modani, M. Wilding, J. Champlin, and P. Sohn. An architecture for quickly detecting known software problems. In *ICAC 2005: Proceedings of the International Conference on Autonomic Computing*, 2005.

[4] http://www.bzip.org.

[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 2nd edition, 2001.

[6] M.O. Dayhoff, R.M. Schwartz, and B.C.Orcutt. A model of evolutionary change in proteins. matrices for detecting distant relationships. *M. O. Dayhoff, (ed.), Atlas of protein sequence and structure, volume 5, pp. 345–358 National biomedical research foundation Washington DC.*, 1979.

[7] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001.

[8] http://www.gnu.org/software/gawk/gawk.html.

[9] S. Henikoff and J. Henikoff. Amino acid substitution matrices from protein blocks. In *Proc. National Academy of Sciences, USA*, 1992.

[10] M. Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.

[11] M. Hind and A. Pioli. Which pointer analysis should i use? In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[12] D. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of ACM, 24(4)*, pages 664–675, 1977.

[13] http://www.htmldoc.org/.

[14] S. Jagannathan, P. Thiemann, S. Weeks, and A.K. Wright. Single and loving it: Must-alias analysis for higher-order languages. In *Symposium on Principles of Programming Languages*, pages 329–341, 1998.

[15] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 308–318, 2003.

[16] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[17] A. Orso, T. Apiwattanapong, and M. Harrold. Leveraging field data for impact analysis and regression testing. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 128–137, 2003.

[18] http://www.gnu.org/software/wget/.

[19] X. Zhang and R. Gupta. Matching execution histories of program versions. In *Proceedings of the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE)*, pages 197–206, Sep, 2005.