Matching Execution Histories of Program Versions [*]

Xiangyu Zhang Rajiv Gupta

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721
{xyzhang,gupta}@cs.arizona.edu

ABSTRACT

We develop a method for matching dynamic histories of program executions of two program versions. The matches produced can be useful in many applications including software piracy detection and several debugging scenarios. Unlike some static approaches for matching program versions, our approach does not require access to source code of the two program versions because dynamic histories can be collected by running instrumented versions of program binaries. We base our matching algorithm on comparison of rich program execution histories which include: control flow taken, values produced, addresses referenced, as well as data dependences exercised. In developing a matching algorithm we had two goals: producing an *accurate match* and producing it *quickly*. By using rich execution history, we are able to compare the program versions across many behavioral dimensions. The result is a fast and highly precise matching algorithm. Our algorithm first uses individual histories of instructions to identify multiple potential matches and then it refines the set of matches by matching the data dependence structure established by the matching instructions. To test our algorithm we attempted matching of execution histories of unoptimized and optimized program versions. Our results show that our algorithm produces highly accurate matches which are highly effective when used in comparison checking approach to debugging optimized code.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Debuggers*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Tracing*

General Terms

Algorithms, Measurement, Reliability

Keywords

execution traces, dynamic analysis, debugging, piracy detection

[*]Supported by grants from IBM, Intel, Microsoft, and NSF grants CCR-0324969, CCR-0220262, CCR-0208756, and EIA-0080123 to the Univ. of Arizona.

1. INTRODUCTION

In many application areas, including the areas of software debugging, maintenance and piracy detection, situations arise in which there is a need for comparing *two versions* of a program. An existing class of algorithms that compare two program versions are *static differencing* algorithms [1, 3, 8, 9]. While these algorithms report static differences between code sequences, in situations where the two program versions correspond to original and transformed versions of a program, it is desirable to match code sequences that dynamically behave the same even though they statically appear to be different.

In this paper we describe the design and implementation of an algorithm for *matching binaries of two versions of a program using their dynamic execution histories*. Let us consider the applications such as software piracy detection and debugging of optimized code. In these two applications one program version is created by transforming the other version. In the first application code obfuscation transformations may have been performed to hide piracy [2, 13]. In the second application transformations are applied to generate an optimized version from the unoptimized version. By performing matching based upon execution histories, we can match corresponding computations by comparing the results produced by them. Static differencing approaches, on the other hand are aimed at finding apparent differences and thus are not suitable for above applications. In case of software piracy detection we will not have access to the source or intermediate code of the pirated version and thus matching should be performed using binaries. Our general solution to matching execution histories relies only on the availability of binaries as dynamic execution histories can be collected by running instrumented program binaries [7, 11].

A General Matching Algorithm. In developing an algorithm for matching execution histories of two versions of a program we have two goals: developing an algorithm that is *fast*, i.e. presence or absence of matches can be determined quickly without excessive search; and *accurate* matches are found, i.e. true matches are reported and very few matches are missed. To address these goals we use the following approach. We begin by conservatively identifying potential matches so that real matches are not missed. By using rich execution history, we are able to compare the program versions across many behavioral dimensions. This leads to mostly true matches being found. Moreover, the number of possible matches that we need to consider greatly reduces when multiple kinds of information is used thereby resulting in a fast matching algorithm. Thus, by basing our algorithm on performing many different simple matches, instead of fewer very complicated matches, we obtain an execution history matching algorithm that is both fast and precise.

Matching for Comparison Checking. While matching has many applications, in this paper we demonstrate the benefits of our matching algorithm in context of *comparison checking* [4, 5], a technique that determines whether erroneous behavior of optimized version is being caused by a bug in the original unoptimized version that was unmasked by optimizing transformations or whether it is due to a bug that was introduced due to an error in the optimizer. Comparison checking works by comparing the values computed by corresponding execution instances of statements in the unoptimized and optimized program versions. If no mismatch in values is found, then the error must be present in the original unoptimized program. A mismatch of values on the other hand indicates that the error must have been introduced by the optimizer. In order to perform *comparison checking*, the compiler writer must write additional code that generates *mapping* between execution instances of statements in the unoptimized and optimized versions.

Modifying the compiler to generate the mappings is a tedious and time consuming task and in fact in some situations this may not be a viable option. In particular, if a commercial optimizing compiler is being used, we will not be able to modify it to generate such mappings. *In absence of mappings and/or compiler source, the algorithm we present in this paper can be used to match the executions of unoptimized and optimized versions*. If a *complete match* is found then the bug must have been present in the unoptimized program. If a *complete match* is not found, then the parts of the execution histories that cannot be matched can be examined by the programmer to identify the likely source of the bug. It is important to note that the static approaches for comparing program versions will not be effective for this application where we want to compare specific executions.

We tested our matching algorithm by matching the execution histories of unoptimized and optimized program versions. Our results show that our algorithm is very effective as it produces *accurate* matches that are *nearly complete* (i.e., there are few false or missed matches).

The remainder of the paper is organized as follows. In section 2 we describe the form of dynamic execution histories used in this work. Our matching algorithm is described in detail in section 3. The results of our experiments are presented in section 4. Related work is discussed in section 5. Conclusions are given in section 6.

2. PROGRAM EXECUTION TRACES

We recently proposed the *Whole Execution Trace* (WET) [16] representation that stores comprehensive execution history of a program in a highly compacted form as annotations to the static representation of executable code. We demonstrated that execution histories of realistically long program runs (few billion instructions) can be kept in memory. Below we briefly describe the information contained in the execution history and how it is useful in matching program versions. The dynamic information can be categorized broadly into two categories: *local* - values and addresses involved in execution of a specific instruction; and *global* - relative execution order and dependences between different instructions. Only the summary of some of the above information is needed by our matching algorithm. Therefore, execution histories of even longer runs can be easily collected and stored for matching.

Values produced (Local). A stream of results produced by an instruction during a program execution are captured. If two program versions are expected to produce the same results on a program input, the streams of values for vast majority of instructions are expected to be the same in the two versions. Therefore these streams of values can be compared to find matching instructions.

Addresses referenced (Local). For a memory referencing instruction a stream of addresses referenced by the instruction during a program execution are captured. While the actual addresses encountered during executions of corresponding instructions of the two program versions may vary, the relative address offsets exhibit similar patterns. Therefore these patterns provide yet another means for matching instructions.

Control flow (Global). The whole program path followed during an execution is captured. This history enables temporal ordering of all interesting events (e.g., I/O operations). These orderings provide a good starting point for search when events of a given type from the execution histories of two program versions are being matched with each other.

Dependences exercised (Global). For each execution of an instruction its data and control dependence history is captured and used in conjunction with the value and address histories to produce accurate matches. Note that value and address histories essentially perform matching of individual instructions and there is a chance that some coincidental or false matches will be produced. By matching the dynamic data dependence graphs, we can confirm with much greater certainty that the instructions truly correspond to each other. Matching of dynamic data dependence graphs confirms the collective correctness of the matches for a group of instructions. Thus, it effectively reduces false matches.

Other. System calls, I/O operations, and memory allocation deallocation operations are examples of special events that are captured. Since these events are likely to change little from one program version to another, matching them using the temporal ordering is quite useful.

From the discussion it should be clear that matching of program versions is based upon comparison of program behavior along many behavioral dimensions. This is key to the success of our approach as it prevents inaccurate matches from occurring.

3. MATCHING EXECUTION HISTORIES

The goal of the matching process is to automatically establish a correspondence between *executed instructions* from the two program versions. We assume that the two versions of the program resulted after one version was transformed into another through semantics preserving program transformations. In finding a matching it is our goal to produce highly *accurate* matches. Accuracy has two dimensions: *completeness* - we would like to discover as many true matches as possible; and *correctness* - we would like to discover only true matches.

Given a pair of execution histories for two versions of a program, we establish a correspondence between the executed instructions by examining the *dynamic data dependence graphs* (dDDGs) of the two versions as well as the histories of individual instructions in the dDDGs. A dDDG contains a node for each *instruction that is executed at least once* and an edge for each *data dependence that is exercised at least once* during the program execution. The matching process consists of three main algorithms:

Signatures. Based upon the local execution history of a given instruction, a signature is computed. The signatures of instructions are compared to determine potential matches or exclude matches throughout the two algorithms identified next.

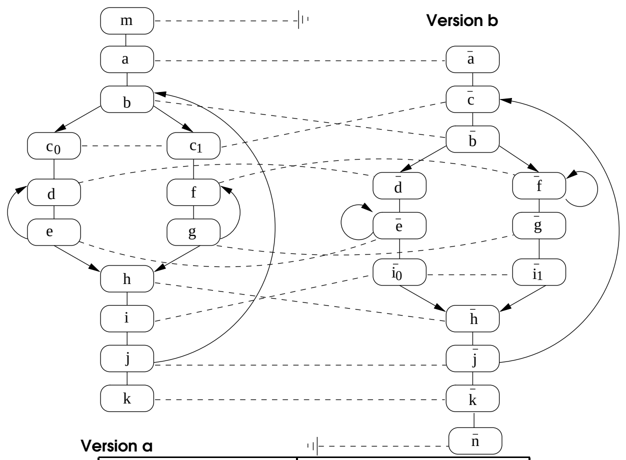
Root Nodes. We begin by matching the roots of the dynamic data dependence graphs of two program versions. The matching is assisted by global information in form of the temporal order in which the root nodes are executed and local information in form of signatures of individual instructions.

Dependant Nodes. Given the matching of the roots, an iterative algorithm is used to match the remaining nodes in the dynamic data dependence graphs. For each instruction in one dependence graph, the algorithm finds a set of matching candidate instructions in the other data dependence graph through iterative refinement using the structure of the dynamic data dependence graph.

Next we discuss the details of the root matching and dependent node matching algorithms. During these algorithms we will assume that signatures are already available. Finally we discuss the details of signature generation and matching.

3.1 Matching Root Nodes

Before we present the root matching algorithm, we make the following observations. First a root node of a dDDG may come from any point in the program's control flow graph. This is because root nodes correspond to instructions such as variable initializations, reads, memory allocations etc. all of which can appear any where in a program. Second semantic preserving program transformations that may have been used to create another version of the program can have significant affects on these instructions. In particular, an instruction may be *added or deleted*, it may be *merged* with another instruction or *split* into two instructions, it may be *reordered* with respect to other instructions, or it may have been simply *moved* causing its execution frequency to change.



Affect	Transformation
Add	$* \rightarrow n$
Delete	$m \rightarrow *$
Move	$freq(d) \downarrow$
Move	$freq(g) \downarrow$
Merge & Reorder	$(b, (c_0, c_1)) \rightarrow (c, b)$
Reorder & Split	$(h, i) \rightarrow ((i_0, i_1), h)$

Figure 1: Matching roots of program versions.

An example in Figure 1, illustrates the above observations. First all instructions shown, which are indicated by letters, are assumed to be root nodes that are spread throughout the control flow graph. The dotted edges between the two versions indicates the correspondence between these instructions. The table in the figure shows the transformations that resulted in generation of Version b from Version a. To distinguish between corresponding instruction in the two versions we use the notation i and $\bar{i}$. When multiple instructions in one version correspond to one instruction in the other version, subscripts are used.

Next we present a root matching algorithm that has been designed taking into account the above observations. Given two versions of a program, Version a and Version b, we begin by first generating *temporally ordered* lists of the root nodes, aRoots and bRoots, as follows. Each root node appears once in the list. The order in which nodes appear corresponds to the order in which their *first executions* took place. This ordering is determined from the full control flow trace. Given the two lists, a call to function Match(aRoots,bRoots) in Figure 2 produces a matching of the root nodes. The outcome of the matching process is identification of pairs of nodes from aRoots and bRoots that match and set of nodes aUnMatched and bUnMatched that contain the nodes that do not have a corresponding matches.

```
Match(aRoots,bRoots) {
    $Matched \leftarrow aUnMatched \leftarrow bUnMatched \leftarrow \phi$;
    $i \leftarrow first(aRoots)$;
    $j \leftarrow first(bRoots)$;
    while $aRoots \neq \phi$ do
        if $sign(a_i) = sign(b_j)$ then
            $Matched \leftarrow Matched \cup \{(a_i, b_j)\}$;
            if $freq(a_i) = freq(b_j)$ then
                $i \leftarrow next(aRoots)$;
                $j \leftarrow next(bRoots)$;
                $aRoots \leftarrow aRoots - \{a_i\}$;
                $bRoots \leftarrow bRoots - \{b_j\}$;
            elseif $freq(a_i) < freq(b_j)$ then
                $i \leftarrow next(aRoots)$;
                $aRoots \leftarrow aRoots - \{a_i\}$;
                $freq(b_j) \leftarrow freq(b_j) - freq(a_i)$;
            elseif $freq(a_i) > freq(b_j)$ then
                $j \leftarrow next(bRoots)$;
                $bRoots \leftarrow bRoots - \{b_j\}$;
                $freq(a_i) \leftarrow freq(a_i) - freq(b_j)$;
            endif
        else
            $j \leftarrow next(bRoots)$;
            if $j = nil$ then
                $aUnMatched \leftarrow aUnMatched \cup \{a_i\}$;
                $i \leftarrow first(aRoots)$;
                $j \leftarrow first(bRoots)$;
            endif
        endif
    endwhile
    if $bRoots \neq \phi$ then
        for each $b \in bRoots$ do
            if $\nexists (a, b) \in Matched$ then
                $bUnMatched \leftarrow bUnMatched \cup \{b\}$;
            endif
        endfor
    endif
    return($Matched$,$aUnMatched$,$bUnMatched$)
}
```

Figure 2: Matching temporally ordered roots.

Our algorithm takes one instruction from aRoots at a time (denoted as a_i) and finds the instruction(s) from bRoots that match this instruction. To find a match, the signature of a_i, $sign(a_i)$, is compared with the signatures of instructions in bRoots one at a time. If a match is found with instruction b_j in bRoots, number of cases arise. If frequency of the matching instructions is the same, we consider matching of both a_i and b_j to be complete and they are removed from the lists. If the frequency of a_i is lower, then we consider the matching of a_i to be complete but not that of b_j – thus, a_i is removed but not b_j. If the frequency of b_j is lower, then we consider the matching of b_j to be complete but not that of a_i. Using the above algorithm we enable one instruction in one version to be matched with multiple instructions in the other version. Also, all instructions in each version that do not match any instruction in the other version are also identified.

While the above algorithm appears to be quite simple, it is very effective in practice. We apply the above algorithm to the example of Figure 1 and show how our algorithm identifies the correct match. Let us first consider executions of the two versions of the program on the same input. Let us assume that the execution path followed during these executions is as follows:

Version a: $m\ a\ b\ c_0\ d\ e\ d\ e\ h\ i\ j\ b\ c_1\ f\ g\ f\ g\ h\ i\ j\ k.$
Version b: $\bar{a}\ \bar{c}\ \bar{b}\ \bar{d}\ \bar{e}\ \bar{e}\ \bar{i_0}\ \bar{h}\ \bar{j}\ \bar{c}\ \bar{b}\ \bar{f}\ \bar{f}\ \bar{g}\ \bar{i_1}\ \bar{h}\ \bar{j}\ \bar{k}\ \bar{n}.$

From the above control flow paths we can find ordered list of the root nodes based upon when the instructions were executed the first time in the above trace. In addition, we can also compute the number of times each instruction is executed. Below we give the ordering and frequency of the executed instructions.

aRoots ($\frac{instruction}{freq.}$): $\frac{m}{1}\ \frac{a}{1}\ \frac{b}{2}\ \frac{c_0}{1}\ \frac{d}{2}\ \frac{e}{2}\ \frac{h}{2}\ \frac{i}{2}\ \frac{j}{2}\ \frac{c_1}{1}\ \frac{f}{2}\ \frac{g}{2}\ \frac{k}{1}$
bRoots ($\frac{instruction}{freq.}$): $\frac{\bar{a}}{1}\ \frac{\bar{c}}{2}\ \frac{\bar{b}}{2}\ \frac{\bar{d}}{1}\ \frac{\bar{e}}{2}\ \frac{\bar{i_0}}{1}\ \frac{\bar{h}}{2}\ \frac{\bar{j}}{2}\ \frac{\bar{f}}{2}\ \frac{\bar{g}}{1}\ \frac{\bar{i_1}}{1}\ \frac{\bar{k}}{1}\ \frac{\bar{n}}{1}$

Now let us consider the matching of the above lists. We start with matching m from aRoots but no match is found even after traversing the entire bRoots list. Therefore it is put into aUnMatched. a and b are successfully matched with $\bar{a}$ and $\bar{b}$ and since their frequencies match, they are removed from the lists. The set Match is updated to reflect the found matches. Next c_0 is matched with $\bar{c}$. However, since $\bar{c}$ has a higher frequency it is not removed from the bRoots list. d, e, and h are matched next with $\bar{d}$, $\bar{e}$, and $\bar{h}$. i is matched with $\bar{i_0}$ and $\bar{i_1}$. j is matched with $\bar{j}$. c_1 is matched with $\bar{c}$ and this time $\bar{c}$ is removed from bRoots. f, g, and k are matched with $\bar{f}$, $\bar{g}$, and $\bar{k}$. Now aRoots is empty and since bRoots contains $\bar{n}$, it is moved to bUnMatched.

Match = $\{(a,\bar{a}), (b,\bar{b}), (c_0,\bar{c}), (d,\bar{d}), (e,\bar{e}), (h,\bar{h}), (i,\bar{i_0}), (i,\bar{i_1}),$ $(j,\bar{j}), (c_1,\bar{c}), (f,\bar{f}), (g,\bar{g}), (k,\bar{k})\}$
aUnMatched = $\{m\}$
bUnMatched = $\{\bar{n}\}$

Thus, we see that despite the numerous apparent differences between the two program versions, all roots have been exactly matched for the above example.

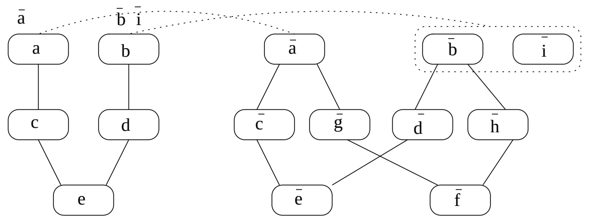
3.2 Matching Dependent Nodes

After matching the root nodes we proceed to the matching of the dependent nodes, i.e. the internal nodes and the leaf nodes in the dDDGs of the two versions. In this step of the matching process we have more information available than the root matching step. For dependent nodes not only do we have the local signatures of individual instructions that are used in matching; in addition, the dependence structure of the dDDG is also matched. For an internal node n in one dDDG version to match an internal node n' in the other dDDG version, the following conditions must hold:

- signatures of n and n' must match;
- signatures of *all* corresponding parents of n and n' must match since the parents supply the operand values; and
- signatures of *some* corresponding children of non-leaf nodes n and n' must match as the results must have similar uses.

The dependence structure matching not only enables more accurate matching, since both the signatures and data dependences of corresponding instructions must match, it also enables faster matching as the signature of an instruction in one version need only be compared with signatures of limited number of instruction signatures from the other version. The latter is true because the matching is driven by the dependence structure.

The matching we describe next consists of multiple passes during which for each instruction in the dDDG of one version, a Match set containing the corresponding matching instructions in the other version are determined. The Match sets are conservatively overestimated and the iterative process continues to refine these sets till eventually no more refinement occurs. The refinement is carried out by repeatedly applying two passes, a *forward pass* and a *backward* pass. Given an instruction n in a dDDG, it may be directly connected by edges to two types of nodes, its parent nodes and its child nodes. During the forward pass, for a given instruction n in dDDG of one version, potential matching candidates from the dDDG of the other version are identified by considering the matching relationships of parent nodes of n. This estimate is further refined by considering the matching relationships of child nodes of n in the backward pass. The refinement process is iteratively applied by repeating the forward and backward passes till no further refinement is possible.



$$\boxed{\begin{array}{c} sign(a) = sign(\bar{a});\ \ sign(b) = sign(\bar{b}) = sign(\bar{i}); \\ sign(c) = sign(\bar{c}) = sign(\bar{g}); \\ sign(d) = sign(\bar{d}) = sign(\bar{h}); \\ sign(e) = sign(\bar{e});\ \ sign(\bar{f})\ -\ no\ match \end{array}}$$

Figure 3: Initial root matches.

We first illustrate the matching process using a simple example and then present a detailed algorithm. Consider the two dDDGs shown in Figure 3 and the relationships between signatures of instructions from the two versions. According to the signatures given, and the dependence structure, nodes a, b, c, d, and e match with nodes $\bar{a}$, $\bar{b}$, $\bar{c}$, $\bar{d}$, and $\bar{e}$ respectively. Now we illustrate how the matching process is carried out.

First the *root matching* is performed using the algorithm described in the preceding section. Let us assume that it produces the result shown above, i.e. $Match(a) = \{\bar{a}\}$ and $Match(b) = \{\bar{b}, \bar{i}\}$. The *forward pass* finds matches for remaining nodes as follows. It examines the nodes in an order consistent with the topological sort order of the nodes in the dDDG. First it examines node c. To find corresponding instructions it first finds a candidate set. Since c has one parent, a, and $Match(a) = \{\bar{a}\}$, the children of $\bar{a}$, i.e. $\{\bar{c}, \bar{g}\}$, can be the potential matches for c and thus they form the candidate set. Now the signature of c is compared with signatures of $\bar{c}$ and $\bar{g}$. Since the signatures match, the first approximation of $Match(c)$ is $\{\bar{c}, \bar{g}\}$. Similarly we determine that $Match(d) = \{\bar{d}, \bar{h}\}$. Next we examine node e which has two parents c and d. The matching nodes of c and d are examined and we find that there are two nodes in the candidate set this time, $\bar{e}$ and $\bar{f}$. This is because both these nodes also have two parents like e which come from sets $Match(c)$ and $Match(d)$ respectively. However, this time when we match signatures, while we find that $\bar{e}$ remains a viable match, $\bar{f}$ is not a match for e. Thus at the end of the forward pass, the $Match$ sets are as shown in Figure 4.

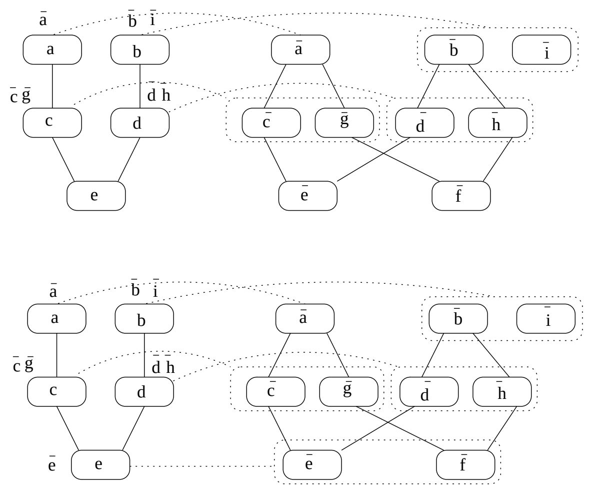


Figure 4: Forward pass: Matching dDDGs.

Now let us perform the *backward pass*. In this pass we make use of the $Match$ set for the only leaf node e to refine the $Match$ sets of its parent nodes c and d and eventually the root nodes. In other words the nodes are examined again in the reverse topological order this time. Lets consider node c first. Node c has one child node e such that $Match(e) = \{\bar{e}\}$. The matching candidates for c are nodes $\bar{c}$ and $\bar{d}$ as they are the two parents of $\bar{e}$. However, after matching the signatures, $\bar{d}$ is eliminated while $\bar{c}$ remains. By intersecting this backward estimate for $Match(c)$ with the earlier forward estimate we conclude that $Match(c) = \{\bar{c}\}$. Similarly $Match(d)$ gets refined to $\{\bar{d}\}$. When we continue the above process to the roots, we find that while $Match(a) = \{\bar{a}\}$ remains the same, $Match(b)$ is refined to $\{\bar{b}\}$. Note that the $Match$ sets at this point represent the desired results (see Figure 5).

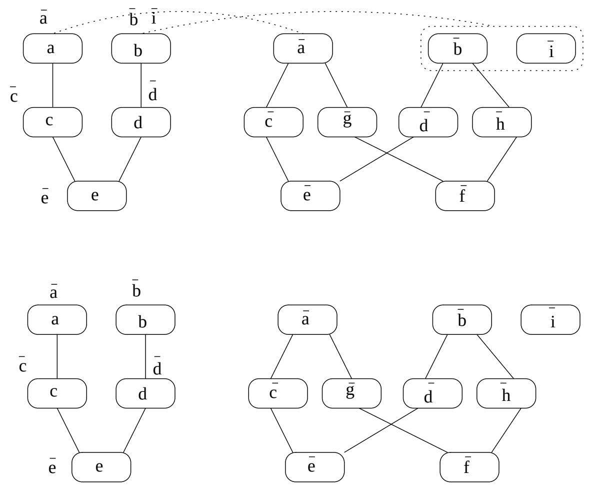
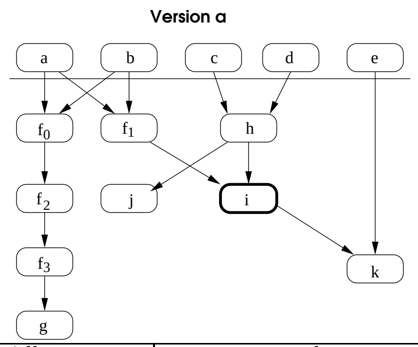


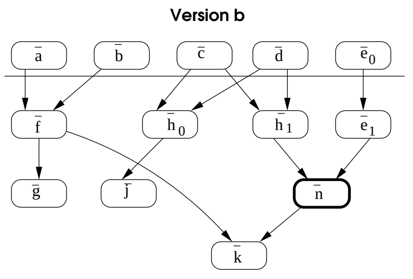
Figure 5: Backward pass: Matching dDDGs.

While in the above example we performed the forward pass once and backward pass once, in general we may have to apply them repeatedly till the $Match$ sets stabilize. The number of repeated applications is bounded by the depth of the dDDG. However, in practice repeated application is almost never required. Although in the above example each instruction in the first dDDG matched with exactly one instruction in the second dDDG, in general this may not be the case. For example, if we change the example such that $sign(e) = sign(\bar{f})$, then c, d, and e will match pairs of nodes $\{\bar{c}, \bar{g}\}$, $\{\bar{d}, \bar{h}\}$, and $\{\bar{e}, \bar{f}\}$ respectively.

The example considered so far has illustrated the key ideas behind our matching algorithm. However, the example considered was simple in one respect. The dDDG of the first version was contained in its exact same form in the corresponding larger dDDG of the other version. However, as mentioned during the development of the root matching algorithm, the second version may be different from the first in its form due to program transformations used to derive the second version from the first. Lets consider another example to illustrate that the same basic algorithm that was described above with some simple but important modifications also works for more general situations.



Affect	Transformation
Add (copy/move)	$e \rightarrow (\bar{e_0}, \bar{e_1})$
Delete (spill/St-Ld)	$(f_2, f_3) \rightarrow *$
Merge (redundancy)	$(f_0, f_1) \rightarrow \bar{f}$
Split (rematerialization)	$h \rightarrow (\bar{h_0}, \bar{h_1})$
Semantic (reassociation)	$(k(i(f_1, h), e)) \rightarrow (\bar{k}(\bar{f}, \bar{n}(\bar{h_1}, \bar{e_1})))$



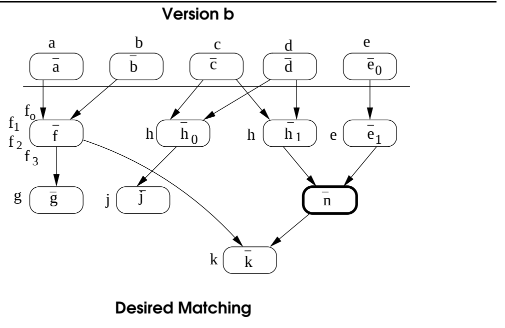


Figure 6: Another example of matching dDDGs.

In Figure 6 two versions of a dDDG are shown that are different in their structure due to the transformations applied to derive version b of the program from version a. Lets look at how the transformations have changed the graph. Differences in register allocation decisions for example, can result in *addition* and *deletion* of nodes. Corresponding to node e we have two nodes $\bar{e_0}$ and $\bar{e_1}$

in the version b – $\bar{e_1}$ corresponds to a MOV instruction that moves the result of $\bar{e_0}$ from one register to another causing the addition of a node. Nodes f_2 and f_3 correspond to spill code – f_2 stores the result of f_0 while f_3 reloads this value for use by g. Better register allocation may eliminate this spill leading to version b of the graph. In version a f_0 and f_1 perform the same computation and thus they are merged during redundancy elimination. Thus, version b contains a single node $\bar{f}$ corresponding to nodes f_0, f_1, f_2, and f_3 in the version a. Node h has been split into $\bar{h_0}$ and $\bar{h_1}$ due to rematerialization. Reassociation has been performed causing intermediate computation i in version a to be replaced by a different intermediate computation $\bar{n}$ in version b.

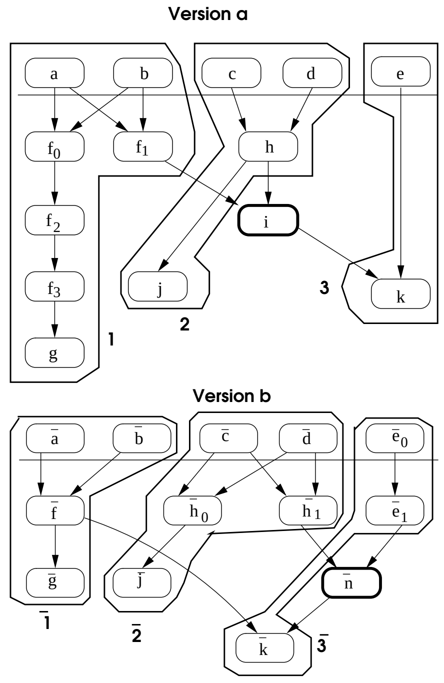


Figure 7: Shapes of matching portions.

The correct matching for this example is also shown in Figure 6 and the matching portions of the two versions are shown pictorially in Figure 7. The important thing to note here is that, unlike the previous example discussed, in this case the corresponding matched portions of the dDDGs do not have an identical size or structure.

Let us now consider how such matches can be discovered. Consider the forward pass as illustrated earlier. Given a node n in one version of the dDDG, a corresponding node cn in the other dDDG that is a candidate for matching with n was related to n as follows. There was a parent of n, say $p(n)$, such that $p(n)$ matched $mp(n)$ and cn was a child of $mp(n)$. Of course there may be many nodes that satisfy this criteria and hence they all are considered as matching candidates. This rule is *generalized* as follows. First $p(n)$ need not be an immediate parent of n but rather it is the closest ancestor of n that has a *non-empty match set*. This rule is needed to get across nodes that do not have any corresponding matches. Note that such nodes may be introduced by transformations. Second once $mp(n)$ is known, the candidates for matching that are considered must include $mp(n)$ and all direct or indirect descendants of $mp(n)$. This rule is needed because after transformations have been applied, we may have chains of nodes in one version that match a single node in another version. The modified relationship between n and cn is illustrated in Figure 8.

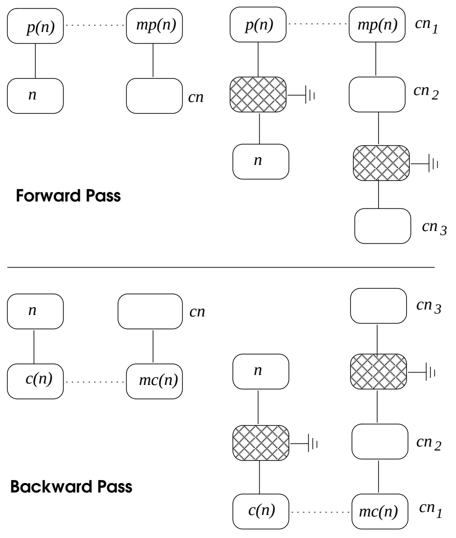


Figure 8: Matching candidates.

Similar changes are made to the backward pass. During the backward pass, given a node n a corresponding node cn that is a candidate for matching with n was related to n as follows. There was a child of n, say $c(n)$, such that $c(n)$ matched $mc(n)$ and cn was a parent of $mc(n)$. This rule is generalized as follows. First $c(n)$ need not be an immediate child of n but rather it is the closest descendant of n that has a non-empty match set. Again this rule is needed to get across nodes that do not have any corresponding matches. Second once $mc(n)$ is known, the candidates for matching that are considered must include $mc(n)$ and all direct or indirect ancestors of $mc(n)$. Again, this rule is needed because chains of nodes in one version may match a single node in another version. This modified relationship between n and cn during the backward pass is illustrated in Figure 8.

If we reconsider the example in Figure 7, we can see how the above generalized rules will enable the match shown to be discovered. When the subgraph 1 is matched to $\bar{1}$, first we are able to match the chain $f_0.f_2.f_3$ to $\bar{f}$. Second even though the immediate child of f_1, i.e. i has an empty match set, we are able cross over i and match k with $\bar{k}$ the immediate child of $\bar{f}$. Thus f_1 is also successfully matched with $\bar{f}$. For similar reasons we are able to successfully match h to $\{\bar{h_0}, \bar{h_1}, \bar{h_2}\}$ and e to $\{\bar{e_0}, \bar{e_1}\}$.

We conclude by presenting the detailed matching algorithm. Before presenting this algorithm we give the precise definitions of *immediate* ancestors/descendants ($iAnc/iDes$) and *all* ancestors/ descendants (Anc/Des). These definitions are key to finding candidate sets during matching and incorporate the rules just discussed. Let us denote a dDDG as $\mathcal{G}(\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is the set of nodes and $\mathcal{E}$ is the set of edges. Furthermore lets denote $\mathcal{E}^-$ to be the subset of edges in $\mathcal{E}$ such that $\mathcal{G}^-(\mathcal{N}, \mathcal{E}^-)$ forms an *acyclic* graph. The definitions of *all* and *immediate* ancestors and descendants are given in Figure 9. Note that the computation of *immediate* ancestors/descendants depends upon the $Match$ sets being empty or not.

The detailed matching algorithm is given in Figure 9 which when given two dDDGs, $dDDG_a = G(\mathcal{A}, \mathcal{E}_a)$ and $dDDG_b = G(\mathcal{B}, \mathcal{E}_b)$, computes the $Match$ set for each node n in $\mathcal{A}$ to be the subset of nodes in $\mathcal{B}$ that are found to match n. As already discussed, following initialization the algorithm iterates over forward and backward passes. During the computation of $Cand$ set for a node n in the forward pass it is ensured through the use of the intersection operator that if a node in Version a uses two operands, the matching node in Version b also uses two operands. On the other hand, during the computation of $Cand$ set for a node n in the backward pass it is ensured through the use of union operator that the number uses of value computed by node n can vary between the two versions. The algorithm terminates when all $Match$ sets stabilize. Following this, the unmatched nodes in $dDDG_a$ are the ones whose $Match$ sets are empty and the unmatched nodes in $dDDG_b$ are those that do not belong to $Match$ set of any node in $\mathcal{A}$.

$$\boxed{\begin{aligned}
Anc(n) &:= \{n\} \cup \bigcup_{(p\rightarrow n)\in\mathcal{E}^-} Anc(p)\\
Des(n) &:= \{n\} \cup \bigcup_{(n\rightarrow c)\in\mathcal{E}^-} Des(c)\\
iAnc(n) &:= \Big(\bigcup_{(a\rightarrow n)\in\mathcal{E}^- \wedge Match(a)\neq\phi} \{a\} \Big)\\
&\quad \cup \Big(\bigcup_{(a\rightarrow n)\in\mathcal{E}^- \wedge Match(a)=\phi} iAnc(a) \Big)\\
iDes(n) &:= \Big(\bigcup_{(n\rightarrow d)\in\mathcal{E}^- \wedge Match(d)\neq\phi} \{d\} \Big)\\
&\quad \cup \Big(\bigcup_{(n\rightarrow d)\in\mathcal{E}^- \wedge Match(d)=\phi} iDes(d) \Big)
\end{aligned}}$$

```
MatchDataDependenceGraphs($DDG_a$,$DDG_b$) {
Given:
    Let $DDG_a = G(\mathcal{A}, \mathcal{E}_a)$; $DDG_b = G(\mathcal{B}, \mathcal{E}_b)$;
Initialization of Match sets.
    for each node $n \in \mathcal{A}$ do
        if $n$ is a root node then
            $Match(n)$ is found by the root matching algorithm.
        else
            $Match(n) \leftarrow \mathcal{B}$
        endif
    endfor
Iterative refinement of Match sets.
    While $Match$ sets continue to change do
    – Forward Pass:
        Let $topo\mathcal{A}$ be the topological sort ordering of $\mathcal{A}$
        for each node $n \in topo\mathcal{A}$ excluding roots do
            $Cand = \bigcap_{a\in iAnc(n)} \big( \bigcup_{\bar{n}\in Match(a)} Des(\bar{n}) \big)$
            $Match(n) = Match(n) \cap$
                $\{\bar{c} : \bar{c} \in Cand \wedge sign(\bar{c}) = sign(n)\}$
        endfor
    – Backward Pass:
        Let $topo\mathcal{A}^{-1}$ be the reverse topo. sort ordering of $\mathcal{A}$
        for each node $n \in topo\mathcal{A}^{-1}$ excluding leaves do
            $Cand = \bigcup_{a\in iDes(n)} \big( \bigcup_{\bar{n}\in Match(a)} Anc(\bar{n}) \big)$
            $Match(n) = Match(n) \cap$
                $\{\bar{c} : \bar{c} \in Cand \wedge sign(\bar{c}) = sign(n)\}$
        endfor
    endwhile
}
```

Figure 9: Matching data dependence graphs.

3.3 Signature Matching

Now we describe how the local histories of instructions are used to form their signatures and how signature matching is performed. When we match the local histories of two instructions, we essentially match the stream of results produced by the instructions. The results produced by an instruction can either represent a stream of data values or a stream of addresses. The most obvious approach is to look for an exact match between the stream of values. However, this is not a good way to match instructions. Recall that program transformations may alter the order in which instruction instances are executed and also the number of times an instruction is executed may increase or decrease. Thus, the list of results for many instructions are unlikely to match exactly. One reason for change in the number of executions is due to elimination of dead instances. By eliminating all dead instances from both versions of the dynamic histories we can avoid this problem. However, removal of dead instances is not the only reason. Several other optimizations, redundancy elimination and speculative code motion, also cause such changes. Therefore we derive simpler signatures from exact list of results such that the derived signatures can be easily matched even if program transformations have affected the corresponding instructions. For instructions with long execution histories such simplifications are unlikely to cause signatures of instructions that do not correspond to each other to match. In case execution histories for some instructions are very small, there is a possibility of coincidental matches. However, in such cases the matching of dependence structure is likely to avoid false matches.

Matching Data Value Streams. An ordered stream of data values is converted into a simpler representation consisting of a vector of *unique values* (U). When matching execution histories, we simply look for *consistency* not equality. Given two instructions I_1 and I_2, we consider their value vectors U_1 and U_2 to match if either U_1 and U_2 contain the same set of values or all values contained in U_1 (U_2) are contained in U_2 (U_1). If values in U_1 are a subset of values in U_2, we consider instruction I_1 to be fully matched and I_2 to be partly matched.

Matching Address Streams. When considering address streams we cannot simply match the unique addresses because the addresses will vary even if they correspond to each other. For enabling matching of addresses, we first convert them to offsets. In case of a heap address the offset is measured from the base address of memory block allocated from the heap. In case of a stack address the offset is measured with respect to the first access via the stack pointer. Once this conversion has been carried out, the comparison of address streams can be performed in the same fashion as that for value streams. This approach is effective because assume that aggressive memory layout optimizations are not performed.

4. EXPERIMENTAL RESULTS

Matching has been implemented in the *Trimaran* system [17]. Our implementation differs from the presented algorithm in that it performs exhaustive comparisons during root matching instead of using temporal ordering to speedup root matching. We generated two versions of VLIW machine code supported under the Trimaran system by generating an *unoptimized* and an *optimized* version of programs. We ran the two versions on the same input, collected their detailed whole execution traces. The execution histories of corresponding functions were then compared. The IMPACT system on which Trimaran is based supports a wide range of optimizations. To enable function by function comparison of dynamic histories, we turned off function inlining.

To evaluate our matching algorithm we carried out two sets of experiments. First we used it to match unoptimized and optimized versions where no errors were present in the optimizer. This experiment was conducted to see how effective (fast and accurate) is our matching algorithm in finding matches when they exist. Second experiment was carried out to evaluate the effectiveness of matching during comparison checking where errors were introduced in the optimized code. Before presenting the results of experiments we describe the benchmarks used in this study.

4.1 Benchmark Characteristics

The program versions used in this evaluation are summarized in Table 1. For each program characteristics of two versions, *unop-*

timized ($.U$) and *optimized* ($.O$), are given. The number of executed functions (functions present) in each program are given. The static number of instructions in each version and the number of instructions executed during program runs are given. As we can see, the static code size and the number of instructions executed differs significantly for the two versions. This is because of aggressive optimizations carried out by IMPACT. Optimization level $O = 4$ was used which performs constant propagation, copy propagation, common subexpression elimination, constant combining, constant folding, code motion, strength reduction, dead code removal, and loop optimizations etc.

Table 1: Program characteristics.

Program	Functions	Instructions U/O	
	Exec. (Exist)	Static Num.	Exec. (millions)
li.U/O	118 (357)	37491/29637	64.7/65.1
m88ksim.U/O	25 (252)	68349/53522	62.0/61.8
twolf.U/O	51 (191)	125260/92807	64.0/63.3
go.U/O	277 (372)	123702/92918	61.7/62.4
vortex.U/O	307 (923)	307526/243678	61.7/60.8
parser.U/O	32 (324)	56526/47560	61.9/62.1

Since the comparison is being carried out between optimized and unoptimized versions of a program, if our algorithm is effective, it should match a very high percentage of instructions from the optimized version with corresponding instructions in the unoptimized version. The number of instructions in unoptimized code that match something in the optimized code is expected to be lower because many statements are eliminated by the optimizations (e.g., redundancy elimination, dead code removal, copy propagation). Finally we expect some instructions in optimized code not to match anything in the unoptimized version due to special features (instructions) of the VLIW machine that are exploited by IMPACT only during generation optimized code (e.g., branch and increment instructions used in software pipelined code, load speculate and load verify instructions).

In Table 2 we present the characteristics of the $dDDG$s of unoptimized and optimized versions of the executed functions. The average number of executed root nodes, leaf nodes, and internal nodes across all executed functions in each program are also given. As we can see, the versions differ significantly not only in the number of nodes they execute but also in the shapes of the dynamic dependence graphs as all three types of nodes differ in their number. This is because the IMPACT system performs both machine independent and machine dependent optimizations very aggressively.

Table 2: dDDG characteristics.

Program. Version	Avg. Exec. Num. Across Funcs U/O		
	Roots	Leaves	Internal Nodes
li.U/O	17.2/15.1	8.7/7.5	22.5/14.0
m88skim.U/O	20.7/18.3	14.4/10.8	40.1/268.0
twolf.U/O	67.1/57.7	28.1/25.0	150.3/102.0
go.U/O	38.8/34.8	29.9/22.5	105.4/65.4
vortex.U/O	53.0/45.4	26.2/21.8	66.9/39.0
parser.U/O	17.7/16.0	12.3/10.2	29.2/19.1

4.2 Accuracy and Cost of Matching

Now we present results when unoptimized version was matched with an optimized version that had no errors introduced in it by the optimizer. The goal of this experiment is to study the accuracy and cost of our matching algorithm.

In Table 3 we summarize the extent to which the execution histories of program versions were matched. The total number of statically distinct nodes that were executed at least once are given for each program version. This number of executed nodes corresponds to the total number of executed nodes matched in all of the functions combined. These are also the nodes that our algorithm attempts to match with each other. The percentage on these nodes in each version for which matches were found in the other version are given. On an average, for over 95% of the nodes in the optimized code, one or more corresponding matches were found. For the unoptimized code this number is lower as expected. This is because, after aggressive optimization, the average number of instructions in the unoptimized version is nearly 25% less than in the unoptimized version (12909 vs. 17292). Many of these instructions have no corresponding instruction in the optimized code.

Table 3: Nodes matched.

Program	Optimized		Unoptimized	
	Nodes	Matched (%)	Nodes	Matched (%)
li	4325	97.0	4989	82.1
m88skim	1398	95.1	1882	81.9
twolf	9419	94.2	12517	86.8
go	28701	91.0	40753	76.9
vortex	32583	97.7	44857	81.7
parser	1450	96.1	1893	84.8
Average	12909	95.2	17292	82.4

Although some of the instructions were not matched by our matching algorithm, this does not necessarily mean that the matches should have been found but were missed by our matching algorithm. Some instructions are not matched due to the features of the VLIW machine used only by the optimized version and optimizations such as strength reduction which introduce computations that produce different intermediate results in the two versions. Therefore, to determine the completeness of the above matches we examined the codes and the matches generated. Since this process had to be performed *manually*, we could not perform it for all the functions. We first looked at many small functions and found that their matches were almost always 100% complete. Then for each program we selected an executed function based upon its complexity: we selected the *largest function* for which *the number of distinct executed instructions in the two versions differed the most*. The number of distinct instructions executed in the optimized ($Exec.O$) and unoptimized ($Exec.U$) versions of selected functions are given in Table 4. We manually found the *actual* total number of matching pairs of instructions by considering all executed instructions from the optimized code. By comparing these matching pairs with those found by our algorithm, we determined the number of pairs that are *missed* and the number that are *false* matches. The results of this experiment given in Table 4 show that we miss very few

Table 4: Matching accuracy.

Fn.Program. Version	Exec.O	Exec.U	Matches		
			Actual	Missed	False
li	78	131	112	0	37
m88ksim	313	426	823	4	72
twolf	765	989	1067	0	220
go	596	939	1362	0	69
vortex	399	677	840	21	336
parser	194	257	243	0	29

pairs although we do find some more false matches. This is not surprising since our signature matching is conservative and thus we are less likely to miss matches and more likely to find some false matches. However, having some false matches is not a serious problem for two reasons. First, for every instruction that we found a false match, we also found the true matches. Second, we believe that false matches can be further reduced without changing the proposed algorithm. Longer and/or multiple runs of the two program versions can be used to refine the matches found by a single run.

Finally we claim that missed matches are more harmful that false matches because user can examine the matches found and eliminate false matches but finding missed matches is far more tedious. Therefore we have designed our algorithm to be on the conservative side, more matches are found than truly exist.

Table 5 gives the total space (in MegaBytes) and time (in seconds) cost of matching. The space cost mainly arises due to the dynamic history used while the time represents the effort it takes to match all of the execution functions. As we can see the time and space costs are reasonable to match executions of fairly large programs involving execution of hundreds of distinct functions.

Table 5: Cost: Space, time, and iterations (averages).

Program	Space (MB)	Time (sec.)	Iterations			
			dDDG Depth	Num. Iter.	After 1 Iter.	Final Iter.
li	4.5	302	5.13	1.81	3.75	1.61
m88skim	29.3	289	5.96	1.88	5.66	2.50
twolf	52.0	362	12.61	2.06	7.15	2.74
go	18.0	387	12.07	1.93	7.42	3.10
vortex	7.8	467	5.89	1.96	7.11	1.89
parser	5.1	265	5.53	2.00	2.93	2.04

We also studied how quickly our iterative matching algorithm stabilizes. In Table 5 the average depth of the dDDGs across all functions is given (dDDG Depth). The average number of iterations (Num. Iter.) it actually took for sets to stabilize is also given. As we can see, although the depths of the graphs can be large, the number of iterations required before the $Match$ sets stabilize is small. The average match set sizes after first iteration (After 1 Iter.) and after algorithm stabilized (Final) are also given to show that they are indeed reduced significantly by our iterative algorithm.

4.3 Matching for Comparison Checking

Next we evaluated matching in context of comparison checking. We injected three different errors into the first three benchmarks. These errors simulate the effect of erroneous data flow results for *common subexpression elimination*, *copy propagation* and *dead code elimination*. During comparison checking matching is used to match as many instructions as possible between the executing unoptimized and optimized versions at regular intervals (execution of 8 million instructions). The instructions that do not match are reported to the user for examination as they may contain an error. We plotted the number of distinct reported instructions as a percentage of distinct executed instructions over time in two situations: when optimized program had no error and when it contained an error (see Figure 10). The points in the graph are also annotated with the actual number of instructions reported. The interval during which error point is encountered during execution is marked.

As we can see, compared to the optimized programs without errors, the number of reported instructions increases sharply after the *error interval* point is encountered. For two out of the three benchmarks, the increases are sharp – 6% to 14% for $m88ksim$ and 3% to 35% for $twolf$. In fact when we look at the actual number of instructions reported immediately before and after the execution interval during which error is first encountered, the number reported increases by an order of magnitude. For li the error is such that erroneous results are not propagated far and thus we see a smaller increase. User examining the reported instructions can fairly quickly focus on erroneous instructions as the number of reported instructions is small in li. In other cases where the number is large, by examining the instructions in the order they are executed, erroneous instructions can be quickly isolated. Other reported instructions are merely dependent upon the instructions that are the root causes of the errors. For example, in $twolf$, out of the over 2000 reported instructions at the end of second interval, we only need to examine the first 15 reported instructions in temporal order to find an erroneous instruction. Note that even when no errors are encountered, some instructions in the optimized version are nevertheless reported. The reporting of some instructions is unavoidable even with a perfect matching algorithm because there is no corresponding match for them in the unoptimized version.

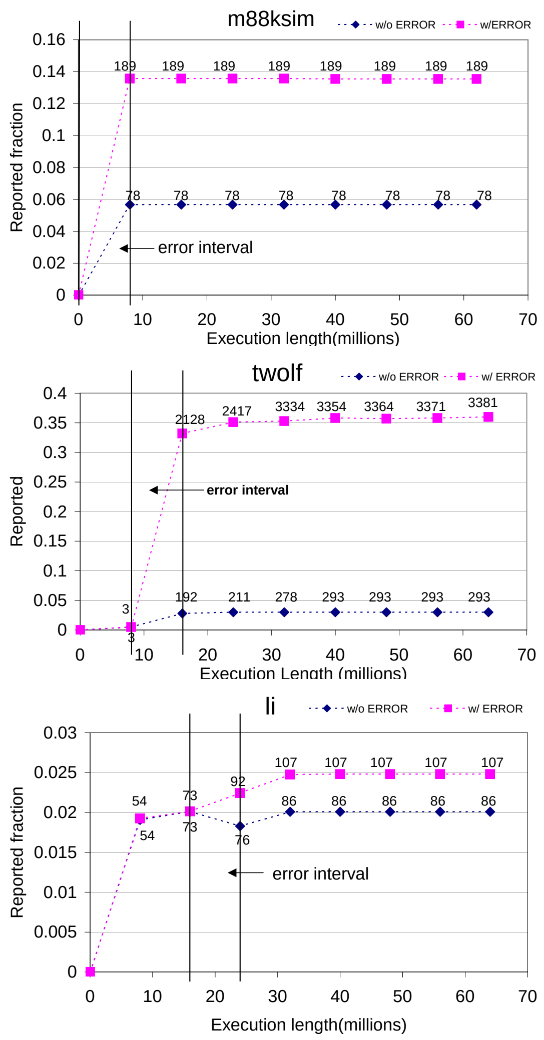


Figure 10: Statements reported for checking.

From the results of the above experiment we can see that erroneous behavior is caught very effectively by our matching algorithm. Thus, we can conclude that our matching algorithm makes the task of implementing comparison checking simple as it does not require the compiler writer to generate mappings between unoptimized and optimized instructions. Moreover, if the compiler does not provide such mappings and no source for the compiler is available (e.g., it is a commercial compiler), we can still implement comparison checking using our matching algorithm.

5. RELATED WORK

Static differencing algorithms. An existing class of algorithms that compare two program versions are *static differencing* algorithms [1, 3, 8, 9]. These algorithms perform differencing at different levels: [9] finds differences using line by line comparison, [1, 8] find differences by comparing control flow graphs, and [3] compares input/output dependences of procedures. With the exception of [3], these algorithms report statements that appear to be different as being different. Moreover, these algorithms work with source or intermediate code representations of the program versions. In contrast our matching algorithms work at binary level and they match instructions that dynamically behave the same even though they statically appear to be different.

Differencing dynamic histories. Research has been carried out on *differencing dynamic histories* of program executions. The benefits of such algorithms for software maintenance have been recognized. In [10] Reps et al. made use of path profiles to recognize Y2K bugs in programs. Wilde [14] has developed a system that enables a programmer to visualize the changes in the dynamic behavior of a program. However, in these works dynamic histories of different executions, corresponding to two different inputs, of a single version of a program are compared. In contrast, our work considers matching of dynamic histories of two program versions on the same input.

Existing matching techniques. There are some existing techniques for matching: procedure extraction [6] requires source or intermediate code matches while BMAT [12] works on binaries. BMAT matches binaries to enable propagation of profile data collected by executing one binary to a transformed binary so that re-execution of the latter could be avoided. Given the nature of this application, it made sense to match binaries statically. On the other hand, in applications such as comparison checking and software piracy detection we are interested in matching the dynamic behaviors of two versions and the execution profiles of the two versions are already available.

6. CONCLUSIONS

In this paper we presented an algorithm for matching execution histories of two program versions. Since results produced by instructions are used to perform the matching, we are able to match instructions that appear to be different but compute the same results. In this way we overcome the problem of matching instructions in presence of different program transformations that may have been applied in generating the two versions. To avoid false matches when instructions coincidentally produce the same results, we also match the dynamic data dependence structures of the computations. Again our algorithm for matching the dependence structure succeeds in matching dependence graphs that behave the same but appear to be different. We demonstrated that using matching we can enable implementation of comparison checking even in the absence of source code of the optimizing compiler. Our ongoing work is exploring the use of matching in software piracy detection.

7. REFERENCES

[1] T. Apiwattanapong, A. Orso, M.J. Harrold, “A Differencing Algorithm for Object-Oriented Programs,” *IEEE International Conf. on Automated Software Engineering*, pages 2-13, 2004.

[2] C. Collberg, C. Thomborson, and D. Low, “Breaking Abstractions and Unstructuring Data Structures,” *IEEE International Conference on Computer Languages*, pages 28-38, Chicago, IL, 1998.

[3] D. Jackson and D.A. Ladd, “Semantic Diff: A Tool for Summarizing the Effects of Modifications,” *IEEE Conference on Software Maintenance*, pages 243-252, Nov. 1994.

[4] C. Jaramillo, R. Gupta, and M.L. Soffa, “Comparison Checking: An Approach to Avoid Debugging of Optimized Code,” *7th European Software Engineering Conference and ACM SIGSOFT 7th Symposium on Foundations of Software Engineering*, LNCS 1687, Springer Verlag, pages 268-284, Toulouse, France, September 1999.

[5] C. Jaramillo, R. Gupta, and M.L. Soffa, “Debugging and Testing Optimizers through Comparison Checking,” *International Workshop on Compiler Optimization Meets Compiler Verification*, Electronic Notes in Theoretical Computer Science 65 No. 2 (2002), held in conjunction with ETAPS, Grenoble, France, April 2002.

[6] R. Komondoor and S. Horwitz, “Semantics-Preserving Procedure Extraction,” *27th ACM SIGPLAN-SIGACT on Principles of Programming Languages*, pages 155-169, 2000.

[7] J.R. Larus and E. Schnarr, “EEL: Machine-Independent Executable Editing,” *SIGPLAN Conference on Programming Language Design and Implementation*, pages 291-300, 1995.

[8] J. Laski and W. Szermer, “Identification of Program Modifications and its Applications to Software Maintenance,” *IEEE Conference on Software Maintenance*, pages 282-290, Nov. 1992.

[9] E.W. Myers, “An O(ND) Difference Algorithm and its Variations,” *Algorithmica*, 1(2):251-266, 1986.

[10] T. Reps, T. Ball, M. Das, and J. Larus, “The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem,” *6th European Software Engineering Conference and ACM SIGSOFT 5th Symposium on Foundations of Software Engineering*, pages 432-449, 1997.

[11] A. Srivastava and A. Eustace, “ATOM - A System for Building Customized Program Analysis Tools,” *SIGPLAN Conference on Programming Language Design and Implementation*, pages 196-205, 1994.

[12] Z. Wang, K. Pierce, and S. McFarling, “BMAT - A Binary Matching Tool for Stale Profile Propagation,” *The Journal of Instruction Level Parallelism*, 2, May 2000.

[13] C. Wang, J. Davidson, J. hill, and J. Knight, “Protection of Software-based Survivability Mechansims,” *International Conference of Dependable Systems and Networks*, pages 193-202, Goteborg, Sweden, July 2001.

[14] N. Wilde, “Faster Reuse and Maintenance Using Software Reconnaissance,” Technical Report SERC-TR-75F, SERC, Univ. of Florida, CIS Department, Gainesville, FL, July 1994.

[15] A. Zeller, “Isolating Cause-Effect Chains from Computer Programs,” *ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering*, Charleston, South Carolina, November 2002.

[16] X. Zhang and R. Gupta, “Whole Execution Traces,” *IEEE/ACM 37th International Symposium on Microarchitecture*, Portland, Oregan, December 2004.

[17] *The Trimaran Compiler Research Infrastructure*. Nov. 1997.

[18] S.S. Muchnick. *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.