IBM Research Report

Distributed Program Tracing

Diptikalyan Saha
IBM Research - India

Pankaj Dhoolia
IBM Research - India

Gaurab Paul
IIT Kharagpur, India

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted is accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T.J. Watson Research Center, Publications, P.O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.
Abstract

Dynamic program analysis techniques depend on accurate program path traces. Program instrumentation is commonly used to collect these traces, which causes overhead to the trace collection process. Various techniques have addressed reducing such overhead by minimizing the number of probes/witnesses used to collect traces. In this paper we present a novel distributed trace collection framework wherein, a program is executed multiple times with the same input for different sets of witnesses. The partial traces such obtained are then merged to create the whole program trace. Such divide-and-conquer strategy enables parallel collection of partial traces, thereby reducing the total time of collection. The problem is particularly challenging as arbitrary distribution of witnesses cannot guarantee correct formation of the traces. We provide and prove the necessary and sufficient conditions for the divide-and-conquer strategy. We use the framework to collect traces of field SAP-ABAP programs using breakpoints as witnesses as instrumentation cannot be performed due to practical constraints. To optimize such collection we extend Ball-Larus’ optimal edge-based profiling algorithm to an optimal node-based algorithm. We demonstrate the effectiveness of the framework for collecting traces of SAP-ABAP programs.

1 Introduction

In recent times many efficient and accurate path based analysis have been developed in the application areas of program comprehension, debugging, bug reproduction, test case generation, fault localization, verification, and more recently fault repair. Many analysis such as path differencing, concolic evaluation, dynamic tainting, and dynamic slicing require exact program path, whereas many compiler optimizations require path profiles. Because of its widespread usage, partial and full trace collection problem is widely studied and efficient algorithms have been developed to address the overhead of trace collection.

Essentially there are three kinds of techniques for trace collection. Instrumentation approaches [1] insert logging instructions in the source/byte-code/binary representation of the program. Kernel/OS level tracing techniques [14, 10] store a log of kernel level events. In situ debugging techniques [7, 20] attach a separate debugger program to the target program and then allow interactive governance of the target program’s execution by stepping thru instruction by instruction, or using breakpoints. All these approaches essentially inspect a subset of program’s run-time behaviors. Following Ball-Larus [1], we use the term witness to denote the set of entities in the static programs that are used to monitor their run-time behavior. Each such witness can occur multiple times in programs execution trace.

Addition of witnesses to monitor program’s run-time behavior, results into run-time overhead during the execution of the targeted software. For example, Libl et al. report an overhead ranging from 2% to 181% on various benchmarks profiling assertion invocations [12], and in a study Diep et al. reported an overhead of 150% to capture call chains [4]. Lowering such overhead is particularly important for profiling software in the field where it can have a direct impact on the user.

Existing works on trace collection present algorithms for optimizing the cost of trace collection by reducing the amount of witnesses, or placing the witnesses in lower execution frequency code regions to reduce the overhead of collection. In this context Ball-Larus in their seminal work [1] presented algorithms for minimal-cost tracing problem. Their solution improved over existing solutions of placing witnesses in all basic blocks, and witnesses in

\[\text{1 In the literature various terms such as logging instruction, probe, monitor, and witness have been used to describe the static entities whose occurrences are inspected at run-time to derive full or partial traces}\]
conditional targets. They provided an optimal edge-based solution which identifies control
flow graph edges to be witnessed for trace collection.

An approach to reduce the overhead of profiling software is to leverage multiple variants of
the same software running at several sites, where each variant contains a subset of witnesses,
and where the subset size can be bounded to meet the overhead tolerance limits [5, 16].
Diep et al. [4] have considered the problem of distribution of witnesses across variants that
maximizes the likelihood of capturing a representative part of the program behavior exercised
in the field. These techniques have been used to obtain profile information that are order-
independent on various runs of the software. For examples, these techniques are not adequate
to obtain a complete execution path of a program, which can be used in debugging field
failures.

In this paper, we introduce a novel framework for distributed trace collection which creates
multiple sets of witnesses. For each set of witnesses, the program is executed (with same
input) and partial trace is collected. All partial traces are merged together to produce the
trace of the program. Such a framework enables faster trace collection using parallelism as
partial traces can be collected in parallel. To our knowledge this is the first approach for
parallel trace collection.

This work also addresses the problem of overhead of trace collection in the field/production
systems which do not allow instrumentation for trace collection. Instrumentation by a user
agent to the production software can change the intended functionality of the software, or
log sensitive information. In these cases, only a system provided utility may be permitted
to debug the code. The debugger may be used to automatically obtain the trace by setting
breakpoints (as witnesses) and controlling/logging the debugger events like breakpoint stops.
However, automated breakpoint based collection is slow. An example of such a system is
SAP-ABAP where we apply our framework to collect traces. Effective application of our
framework for breakpoint based trace collection required to extend Ball-Larus’s optimal
distributed profiling algorithm to node-based profiling.

The distributed path collection framework uses a divide and conquer strategy to collect
the entire execution path. The program is run multiple times, each with a different set of
witnesses. We call the set of witnesses, a profile, and the set of all the profiles, a profile
distribution. For each profile, an execution of the program generates a partial path. We call
a profile distribution realizable if it is possible to reconstruct the entire execution path (for
all possible inputs) by merging all the partial paths.

The challenge of effective distributed path collection is two fold. First, an arbitrary
choice of witnesses into profile distributions, does not guarantee partial paths that may be
unambiguously merged to construct the entire execution path. Therefore, a necessary and
sufficient condition for a realizable path distribution needs to be formulated. Second, the
objective of minimizing the collection time for entire execution path, depends on optimizing
the profile distribution for the available machines for parallel collection.

The techniques presented in this paper address the challenges discussed above. The
contributions of this paper are summarized below.

- We introduce a novel distributed trace collection framework where the trace of a termi-
nating program is collected using a divide and conquer strategy. This is applicable for
both instrumentation and debugger based trace collections.
Figure 1: Example control flow graph with witness based on basic-block strategy and partial paths. Bold dots represent witnesses.

- We show that arbitrary distribution of witnesses (edge-based or node-based) does not guarantee realizable distribution (Section 2). To solve this problem we provide a necessary and sufficient condition for a realizable distribution. This is, in fact, the basis of our framework (Section 4.1).

- We prove that optimal parallel distribution is an NP-Complete problem by reducing the problem of multiprocessor scheduling problem. We further discuss several strategies of forming profile distributions and provide a generic algorithm for profile distribution (Section 4.2). We also consider the profile distribution when each profile size is bounded.

- We provide an algorithm to reconstruct an execution trace from the partial paths (Section 4.3).

- We extend the Ball-Larus’ solution for optimal edge-based tracing problem to optimal node-based tracing. Such a solution is particularly useful for breakpoint based path collection (Section 5).

- We demonstrate the effectiveness of our solution for breakpoint based collection of traces for SAP-ABAP programs. Parallel trace collection reduces the entire collection time by 3 folds in comparison to non-distributed techniques (Section 6).

2 Motivating Examples

We now illustrate the challenges of realizable and effective distributed trace collection with the help of some examples.

Consider a distribution of the basic block profile (all basic blocks contain witnesses) of the control flow graph shown in Figure 1. Note that the overall set of witnesses \{entry, b1, b2, b3, b4, b5, exit\} is sufficient to observe the entire execution path. Now assume that a particular profile distribution is: \{\{entry, b1, b2, b5\}, \{b3, exit\}, \{b4\}\}, and the respective partial paths observed for each execution, are as shown in the figure. In this execution the loop has executed twice, and each time it has taken a different target of the conditional \(b2\). However, looking at the partial path traces it is not possible to determine the execution order of \(b3\) or \(b4\). This
example illustrates that arbitrary distribution of witnesses is not sufficient for whole path reconstruction.

Note that the problem with arbitrary distribution illustrated in the example of Figure 1 is not due to the choice of initial set of witnesses to distribute. If instead of distributing the basic block based set of witnesses, an optimized set of witnesses \{b3, b4\} was distributed into 2 different profiles \{\{b3\}, \{b4\}\}, the partial traces would’ve been as shown in Figure 2. Obviously, in this case as well, it is not possible to infer the execution order of b3 and b4 by looking at the partial traces.

Also note that other naive distribution strategies such as clubbing witnesses by functions, together into a profile, lead to the same problem. In Figure 3 a function bar is called at location c1 in foo and is returned at r1. The two profiles in a distribution \{\{c1\}, \{b4\}\} are sufficient to disambiguate paths considered at the level of each function in isolation. However the partial paths may not be merged to infer the execution order of the b3 and b4 while constructing the entire execution path.

In Section 4 we present the necessary and sufficient condition for realizable distribution. A simple manifestation of such condition, called function-wise grouping strategy is mentioned here. In this strategy each function is assigned to a profile which contains a realizable profile (set of witnesses that can determine all paths through the function) for the function. This will prohibit intra-procedural problems described in Figures 1 and 2. Additionally, functions, that can be called multiple times, have extra witnesses to cover all execution paths from entry.
to exit. For example, in Figure 3, the function \texttt{bar} can have another witness at any node in addition to \texttt{b4} to cover the path from \textit{entry} to \textit{exit} through \texttt{b3}. Presence of such a witness in the same witness set as \texttt{b4} guarantees to resolve the relative order of \texttt{b3} and \texttt{b4} in inter-procedural calls to function \texttt{bar} as in the example of Figure 3.

Even though the above strategy solves the realizable distribution problem, it may not be effective in all cases for reducing the overall trace collection time. The time for parallel trace collection depends on the maximum time taken by any machine. The trace collection time in any machine is proportional to the number of witnesses hit in the actual run. The number of hits depends on the witness choice and actual execution path for the run. Optimal tracing algorithms like Ball-Larus, approximate the behavior of the run by assigning frequencies or weights to edges or vertices of control flow graph. Such weightings can be obtained either by empirical measurement (profiling) or by estimation. Such profiling helps to put witnesses in lower execution frequency area. Thus, in any machine, witnesses are chosen to minimize the sum of weights of the witnesses.

Ball-Larus present an optimal edge-witness based algorithm to minimize the total witness weight for each function (discussed in detail in Section 3). The function-wise grouping strategy is effective for optimizing the total witness weights for each function. However, for effective distribution across machines the total witness weight of all functions have to be distributed \textit{evenly}. Say if total witness weights of function \texttt{f1} is much greater than that of function \texttt{f2}, the expected time for trace collection in the above strategy of function-wise witness set formation will be dominated by the machine containing the witnesses of function \texttt{f1}. A more optimized strategy will distribute witnesses of both the functions across machines evenly without compromising realizability. The necessary and sufficient realizability condition enables us to devise such a fine grained distribution strategy.

**Key Points:**

- Arbitrary distribution of witnesses across profiles does not guarantee realizability.
- Function-wise grouping strategy is realizable but may not be optimal for parallel distribution.

## 3 Background

### 3.1 Notations

We first present some useful notations and terms. An Interprocedural Control Flow Graph \texttt{ICFG} = (\texttt{CFGS}, \texttt{CallE}, \texttt{RetE}) corresponding to a program consists of a set of control flow graphs \texttt{CFGS}, a set of interprocedural call \texttt{CallE} edges, and a set of return edges \texttt{RetE}. Each control flow graph \texttt{CFG} in \texttt{CFGS} corresponding to a single function is represented by a directed graph \( G = (V, E, \text{entry}, \text{exit}) \) where \( V \) is the set of nodes that correspond to the basic blocks in the function, \( E \) is the set of directed edges between the nodes, and \texttt{entry} and \texttt{exit} are two distinguished nodes in \( V \). A subset of \( V \) are \texttt{call} nodes.
Each call node denotes a basic block with a call statement at the end. An interprocedural call edge (∈ CallE) flows from a call node to the entry node of the called function and return edge flow from exit node of the called function to the node containing the return location (return node) in the caller. A distinguished CFG in CFGS corresponds to the CFG for main, the starting function of the program. In Figure 3 foo.c1 is the call node, foo.r1 is the return node, ⟨c1, bar.entry⟩ ∈ CallE, and ⟨bar.exit, r1⟩ ∈ RetE.

We denote v → w for (v, w) ∈ E ∪ CallE ∪ RetE and use v →+ w to denote the transitive closure. We assume that there exists at most one edge between two nodes. We denote succ(u) = {v|u → v}, and a node u is a conditional/predicate if |succ(u)| > 1 (this denotes a basic block ending with a conditional statement). We denote |succ(u)| > 1 as cond(u) == true. For a conditional node u, each node in succ(u) is called the target or successor of the conditional/predicate u. In Figure 3 b1 and b2 are conditional node. We use the notation ICFG.E to denote union of all edges in CFGs in ICFG and call and return edges and use ICFG.V to denote union of all nodes in CFGs of ICFG.

A weighting W of an ICFG assigns a non-negative integer values to every edge of ICFG.E subject to Kirchoff’s flow law: for each node (except entry and exit of main and not-called function) v, the sum of the weights of the incoming edges is equal to the sum of the weights of the outgoing edges. The weight of a node (denoted as wt(v, ICFG, W)) is the sum of the weights of all its incoming edges. The cost of a set of nodes is the sum of all weights of the nodes in the set. A weighting is called a basic weighting if for each conditional all successors of the conditional have same weights. Weight in an edge signify the expected number of times (relative to other edges) that this edge is going to be executed. Typically such weighting is available from profiling. The use of such weighting for optimal profiling is described in [1].

A path P (unless stated, it is undirected) can be denoted as a sequence of a nodes ⟨v1, ..., vi, vi+1, ..., vn⟩ such that for every vertices vi, vi → vi+1 or vi+1 → vi. A path is a cycle (undirected) if v1 = vn. A cycle is simple if all nodes in the cycle are distinct. A path or cycle is directed if, for every vi, vi → vi+1. A directed path is an execution of a function when v1 is the entry of the function and vn is the exit of the function. Let u, v, w be three consecutive vertices in a cycle. There is a fork at v if u → v → w, a join at v if u → v ← w and pipe (u → v → w or u ← v ← w) otherwise. A cycle is piped if it contains atesleast a pipe. A piped cycle is a diamond if it has more than two distinct edges and contains exactly one fork and one join. A diamond is not necessarily a simple cycle. For example, in the example shown in Figure 4 there are 4 diamonds ((b2, b3, b5, b4, b2), (b5, b6, b11, b10, b8, b7, b5), (b5, b6, b11, b10, b9, b7, b5), (b7, b8, b10, b9, b7)) and 6 cycles in the CFG.

A node-based profile is a set of nodes Vwit ⊆ ICFG.V. Similarly, an edge-based profile is a set of edges Éwit ⊆ ICFG.E. A profile (denoted as Pr) is either node-based or edge-based. A collection S corresponding to a profile Pr and an execution P of a function can be denoted by a sequence of nodes S = {b1, ..., bn}, such that each bj ∈ Pr is in the execution path P, and S is a subsequence of P. A profile Pr is realizable iff it is possible to infer any path P from a collection S corresponding to Pr and P. We call such an inference procedure as path reconstruction.

Given a weighting W of an ICFG, the cost of a profile is defined as the cost of the set of elements (either node/edges) in the profile.
Figure 4: (a) All Basic Block profile , (b) Condition Target Profile, (c) Optimal Edge-Profile for basic weighting; Bold dots represent witnesses

3.2 Related Work

**Node-based Realizable Profiles.** In the first simple strategy to obtain a realizable profile, one breakpoint per basic block is set in the program (i.e. \( V_{wit} = V \)), and in the path reconstruction phase \( P = S \), the collection itself. A more efficient method to create a realizable profile is to include only targets of conditionals in the profile [9]. Both the profiles corresponding to a single CFG are shown in Figure 4(a) and (b). The black dots (on nodes or edges) represent the elements in the profile.

**Edge-based Realizable Profiles.** Below we describe the Ball-Larus solution to find optimal realizable edge-based profile. A path in CFG \( G \) is witness-free with respect to a set of edges \( E_{wit} \) iff no edge traversed in the path is in \( E_{wit} \). Given an edge \( p \to q \) where \( p \) is a conditional node in the CFG \( G \) and \( q \) is its target, the witness-set\(^2\) for conditional \( p \) to node \( q \) is a set of edges \( E_{witness}(G, E_{wit}, p, q) = \{ p \to q | p \to q \in E_{wit} \} \cup \{ x \to y | x \to y \in E_{wit}, \exists \text{ witness-free path } p, q, \ldots, x \} \cup \{ EOF | \exists \text{ witness-free path } p, q, \ldots, \text{exit} \} \).

A set of edges, \( E_{wit} \) is a realizable profile, iff for each predicate \( p \) and its successors \( q_1, \ldots, q_n \), for all pairs \( q_i, q_j \) where \( i \neq j \), the 
\[ \text{Ewitness}(G, E_{wit}, p, q_i) \cap \text{Ewitness}(G, E_{wit}, p, q_j) = \phi. \]

We call the above theorem, the necessary and sufficient path condition for a realizable edge profile. Intuitively, each edge in \( \text{Ewitness}(G, E_{wit}, p, q) \) can potentially be the first witness edge visited on a path from \( p, q, \ldots, \text{exit} \). If two successors have disjoint witness sets then existence of a witness in trace from either set determines which successor was taken after execution of the conditional. The necessary and sufficient condition presented above is equivalent to the statement given below [1].

\[ \text{Ewit is realizable iff } E - \text{Ewit contains no diamonds or directed cycles.} \]

We call the above condition as necessary and sufficient CFG condition for edge-based realizable profile. Although, the above two conditions are equivalent, the path-based condition is useful in forming the path reconstruction algorithm, whereas the CFG condition is essential in creating the witness profile.

\(^2\)We use the term ‘witness-set’ for a branch target whereas ‘witness set’ for set of witnesses.
Informally, the above profile realizability condition means that $Ewit$ 'breaks' all diamonds and directed cycles. Intuitively, a profile breaks a diamond or a directed cycle if it contains sufficient witnesses such that it is possible to identify which path is taken in the diamond or directed cycle. $Ewit$ breaks a diamond (or directed cycle) if $Ewit$ contains an edge in the diamond (or in the directed cycle) \(^3\).

Figure 4(c) shows an optimal edge profile for basic weighting. All 4 diamonds and 6 directed cycles are broken by the given $Ewit$.

Optimally breaking the diamonds and directed cycles is an NP-Complete problem \([6, 13]\). However, a spanning tree can break all the cycles (diamonds and directed cycles are cycles in undirected sense) in polynomial time. Edges not in such spanning tree of the CFG comprise realizable $Ewit$ and moreover optimal solution for a large class of CFGs.

4 Distributed Tracing

In this section we first present necessary and sufficient condition for realizable witness distribution. Then we present strategies for effective parallel distribution. Finally, we present the trace reconstruction algorithm which forms the execution trace by merging partial traces.

A profile distribution ($WitD$) is a set of profiles. Based on the type of its constituent profiles, a profile distribution can either be edge-based ($EwitD$) or node-based ($VwitD$). Each constituent profile of a distribution is denoted by $WitD_i$. Given a profile distribution, we refer to a trace collection corresponding to each of its constituent profiles as a partial-path. Our task is to construct the entire execution trace by merging all the partial paths. A profile distribution is realizable if it is possible to reconstruct the trace from the partial paths of collection corresponding to its constituent profiles.

4.1 Necessary and Sufficient Condition for Realizable Distribution

We first present the intuition for deriving the necessary and sufficient condition for realizability of breakpoint distribution. Consider again the example shown in Figure 2. The problem with this distribution is that it was not possible to associate the branches $b_3$ and $b_4$ with the loop iterations. We notice that such association of loop iteration with the branch target happens in a realizable profile (that breaks all cycles and diamonds in CFGs). Consider an alternative non-disjoint breakpoint distribution - $\{\{b_3, b_2\}, \{b_4, b_2\}\}$. In this realizable distribution each profile breaks the directed cycles and the diamond, and therefore realizable. Hence from this example we arrive at a conclusion that, a distribution is realizable if all its constituent profiles break all diamonds and directed cycles.

Next, we discuss whether this is a necessary condition, i.e. does the above example imply that each profile of a realizable distribution needs to break all the diamonds and directed cycles? To answer this question let us consider the example in Figure 4 with the distribution $\{\{b_3, b_4\}, \{b_5, b_6\}, \{b_8, b_{10}\}\}$. This distribution is realizable, and each profile in the distribution individually breaks at-least one diamond ($\{b_5, b_6\}$ breaks two) and all the directed edges not in such spanning tree of the CFG comprise realizable $Ewit$ and moreover optimal solution for a large class of CFGs.

\( ^3 \)In Section 5 it is shown that a node-based profile will break a diamond if it contains a non-fork, non-join node of the diamond.
cycles passing through it, but not necessarily all the diamonds and cycles. Thus the answer to the above question is ‘no’.

Informally we derive that if a profile breaks a diamond then it has to break all the directed cycles ‘passing through it’. Collectively, all the profiles need to break all diamonds and all directed cycles.

Based on the intuition built from the examples above, we now formally present the CFG based necessary and sufficient condition for a realizable distribution.

**Theorem 1** A distribution is realizable iff the following conditions hold:

- For each diamond, there exists at least one profile that breaks the diamond and breaks all the directed cycles (if any) that pass through the fork and branch targets of the diamond.
- All diamonds are broken collectively by all the profiles.
- All directed cycles are broken collectively by all the profiles.

We say a profile covers a diamond if the first condition is met by the profile. The proof of this theorem is provided later in this section. Following this theorem the task of distribution is to distribute diamonds and cycle-without-diamonds to different profiles. Cycles-without-diamonds refers to the directed cycles which do not pass through the fork and branch target of any diamond.

The following theorem presents a path-based necessary and sufficient condition for a realizable distribution

**Theorem 2** A breakpoint distribution is realizable iff for each predicate p and its targets

- if \( q_i \to^+ p \) and \( q_j \to^+ p \),

then there must exists a profile \( \text{WitD}_m \) such that the following conditions are met

- \( \text{witness}(\text{ICFG}, \text{WitD}_m, p, q_i) \cap \text{witness}(\text{ICFG}, \text{WitD}_m, p, q_j) = \emptyset \)
- \( \text{allwitness}(\text{ICFG}, \text{WitD}, p, q_i) = \text{witness}(\text{ICFG}, \text{WitD}_m, p, q_i) \)
- \( \text{allwitness}(\text{ICFG}, \text{WitD}, p, q_j) = \text{witness}(\text{ICFG}, \text{WitD}_m, p, q_j) \)

else

\( \text{allwitness}(\text{ICFG}, \text{WitD}, p, q_i) \cap \text{allwitness}(\text{ICFG}, \text{WitD}, p, q_j) = \emptyset \)
Note that, for the predicate \( p \), when its targets can loop back to the conditional, the above theorem asserts that in some profile their entire witness-sets exist and they are disjoint. The disjointedness criteria uniquely determines which conditional target between \( q_i \) and \( q_j \) is taken after conditional. The existence of their entire witness-sets in one profile determines the association of the occurrence of the targets with the loop iterations. If such condition is not met then there can potentially be a case where two partial-paths have two witnesses, one each from the witness-set of each target, which occur in two different instances of a loop. In this case, it is not possible to associate which iteration executes which target - a case seen in Figure 2. However, for conditional target pairs where both of them do not path back to the conditional, witnesses in their disjoint witness-sets can be distributed across profiles. As an example consider the predicate \( b1 \) corresponding to the loop condition in Figure 1. \( \text{exit} \) and \( b2 \) are its two successors. The successor \( \text{exit} \) doesn’t have a path back to \( b2 \). The witness sets for these may be distributed across profiles. During path reconstruction \( b2 \) should get precedence over \( \text{exit} \), as, if it does occur, it can only occur before \( \text{exit} \).

Proof of Theorem 2. We first show prove that given a distribution satisfying the given conditions in the theorem it is realizable. Consider a predicate \( p \) with two branch targets \( q_i \) and \( q_j \).

Case a.1. \( q_i \rightarrow^+ p \) and \( q_j \rightarrow^+ p \).

Consider the profile where both the branch target’s entire witness sets exist and they are disjoint. Based on single profile realizability, this case is realizable.

Case a.2. Complement of case a.1.

Here having disjoint witness set is sufficient as if there exist two profiles, each having a witness from witness-sets of the branch target, then the one to loop back to conditional is the first one to execute. For the case when both the branch targets of them do not loop back to conditional such conflict case won’t even arise.

We now prove the other way using contradiction. Consider the following cases.

Case b.1. \( q_i \rightarrow^+ p \) and \( q_j \rightarrow^+ p \). Disjointedness of witness-sets is necessary. Now consider that in a profile the complete witness-set does not exist for \( q_i \). In this case we can create an example similar to the example in Figure 1 to prove it unrealizable.

Case b.2. Complement of case b.1. Disjointness is necessary.

Proof of Theorem 1. To prove this we show that this theorem is equivalent to the Theorem 2. Say a predicate \( p \) and its two successors \( q_i \), \( q_j \) create a diamond with join-point \( v \). The disjointedness criteria is satisfied by breaking the diamond. Existence of entire witness-sets in the same profile is satisfied by breaking all directed cyclic paths from targets to the conditional.

Key Points:

- The necessary and sufficient condition for realizable distribution is provided in this section. The crux of the condition is that the profile which breaks a diamond has to break all directed cycles that pass through the fork and branch target of the diamond.

- Cycles refer to in this theorem can be cycles the cross function boundaries i.e. the cycles
that contains call and return edges.

- The condition is valid for both edge-based and node-based profiles.
- Following the condition, diamonds and cycles-without-diamonds are distributed to the profiles (for covering and breaking respectively) for realizability.

4.2 Effective Parallel Distribution

Satisfying the necessary and sufficient condition presented before, many possible realizable profile distributions can be generated. In this sub-section effectiveness of various parallel distribution strategies are discussed. Two cases are considered depending on whether or not each constituent profile is bounded by a profile size.

Our model of parallel distribution consists of \( m \) identical machines, each having a static queue which can contain multiple profiles. All the profiles are distributed to the machines before the execution starts. Each collection is run on the same input. Whenever, a collection is complete in one machine, a profile can be removed from queue and run for collection in that machine. The process stops when all the collections are done. Since collections in each machine is done sequentially, the time taken by a machine is the sum of times of all collections performed in the machine.

The trace collection time (or cost of a collection) is the maximum of total times taken by each machine for its collections. Note that the collection time of a profile depends on the number of witness hits by the actual run. The profile distribution algorithm approximates the number of witness hits in a profile by the total weight of the witnesses in the profile. Thus cost of a distribution is defined as the maximum of cost of distribution for each machine, where cost of distributing in a machine is defined as the sum of the profile weights in the machine. The profile weight is defined as the sum of witness weights in the profile.

The overall time of collection will be optimized if the maximum time across all machines is minimized. This suggests even distribution of witness weights across all machines for optimal parallel distribution. However, we found that such distribution is an NP-Complete problem as stated in the below theorem:

**Theorem 3** The optimal parallel distribution problem is NP-Complete.

Proof. Its easy to check that the problem is in NP. We reduce an existing NP-Complete problem called Multiprocessor Scheduling to this problem. In the Multiprocessor Scheduling problem [6], we are given \( m \) identical machines \( M_1, \ldots, M_m \) and \( n \) jobs \( J_1, \ldots, J_n \). Job \( J_i \) has a processing time \( p_i > 0 \) and the goal is to assign jobs to the machines so as to minimize the maximum load. The load of a machine is defined as the sum of the processing times of jobs that are assigned to that machine. For each job \( J_i \) we construct a diamond such that the weight of covering the diamond is same as the processing time of the job. This is done by selecting two nodes in each directed path in the diamond to have weights same as half of the processing time for the job. Each diamond is put in a separate loop. Then all such loops are put in series. The construction ensures that cost of a profile having a set of diamonds is equal to the sum of cost of each diamond in the set. Then the solution to the optimal
distribution problem for this CFG and weighing is same as the solution to Multiprocessor Scheduling problem.

Following this result, profile distribution algorithms uses greedy approximation algorithm (presented in Figure 5) to optimize the profile distribution.

Our second observation is informally stated as follows: when there is no-bound on the profile size then it is possible to combine all the profiles of one machine into a single profile such that the total weights of witnesses in the resultant single profile is no greater than the total weights of witnesses of all the profiles.

We illustrate this using the example shown in Figure 6. A realizable distribution contains three profiles \( \{b_3, b_4\}, \{b_6, b_7\}, \{b_9, b_{10}\} \) where each profile covers a diamond and cycles passing through it. If there are two machines (i.e. \( m = 2 \)) then the distribution can be \( \{b_3, b_4\}, \{b_6, b_7\} \) and \( \{b_9, b_{10}\} \). However, the first machine can break the two diamonds along with the cycles passing through them by a single profile viz. \( \{b_3, b_4, b_6\} \) with lesser cost than the two profiles. We call such pair of diamonds as witness-optimizable whose combination into a single profile requires lesser weight witnesses for covering both the diamonds than sum of witness weights to cover each diamond.

We summarize the important findings before describing the various distribution strategies.

- For effective time for collection, each machine contains only a single profile if the size of profile is unbounded.
- Combining multiple diamonds in the same profile can reduce the total witness weights to cover the diamonds.
- Fine grained weight distribution facilitates even distribution of total weights across machines. For example, say 6 units with weights 1, 2, 3, 4, 5, and 6 are to be optimally distributed to two machines. If six jobs are formed each having a single unit, then chance of optimal distribution increases compare to the case where jobs are formed combining one or more than one units - say for example forming three jobs like this \( \{\{1\}, \{2, 3\}, \{4, 5, 6\}\} \) and then try to distribute optimally between two machines.

We describe three strategies to distribute covering of all diamonds and directed cycles of a program to multiple machines. The three strategies essentially differ in formation of Job (as referred in Figure 5) - which is a set of diamonds and directed cycles-without-diamonds which are definitely covered in the same machine.

```
Algorithm: AssignJobsToMachine(M, J)
Input: A set of identical machines M (\{M_1, ..., M_m\}),
A set of jobs J (\{J_1, ..., J_n\}).

// Greedy: (2 −1/m) approximation
For i = 1 to n do
   // Assign Job J_i to the machine with least current load
   // Update load of the machine that receives job J_i.
   M_k = find_least_current_load(M)
   assignJob(M_k, J_i)
```

Figure 5: Algorithm: Greedy Solution to MultiProcessor Scheduling problem
The interesting nature of this distribution problem is that forming job by combining diamonds and directed cycles may form the job whose cost of optimal realizable profile is lesser than the sum of the profile weights to break the constituent diamond or directed cycle. Thus combination of multiple diamonds and directed cycles to cover in a single machine can be beneficial to reduce the job weight. On the other hand, increase in profile cost by combining jobs may pose as deterrent to even distribution of jobs in the machines.

In all three strategies diamonds and directed cycles from program’s ICFG are determined in the following way. First, all functions that can be potentially called multiple times are identified. A function can potentially be called multiple times if the function has multiple call-sites or called inside a cycle, or called from a function which can be potentially called multiple times. For these functions an edge is added between exit to entry to preserve realizability, for the their call sites an edge is added between call location to return location. Then, for all functions, diamonds and cycles are identified. This strategy, however, cannot identify potentially witness-optimizable diamonds between caller and called functions. Such a case can arise if there exists a diamond at node $r_1$ in Figure 3 in which case it is not possible to cover the diamonds in two functions.

The first strategy, called fine-grained distribution strategy, forms each job as a single diamond or cycle-without-diamond. Thus preferring effectiveness of distribution over reducing weights through finding witness-optimizable diamonds.

The second strategy, mentioned in Section 2 as function-wise grouping strategy, forms each job by combining all diamonds and directed-cycles-without-diamond of a single function, thereby making sure that all witness-optimizable diamonds in the function are covered in a single machine. However, this strategy, even though fairly simple, can have other diamonds in the same job which are not witness-optimizable.

The third strategy is called optimized-grouping which uses heuristics to find potentially witness-optimizable diamonds and forms job with combining only such diamonds. It follows the principle that two diamonds are potentially witness-optimizable if they share the same cycle or if they share an edge between them. The former case is illustrated in the example shown in Figure 6(a). For the later case consider the CFG in Figure 6(b) which has three diamonds $\langle b_1, b_2, b_7, b_6, b_5, b_3, b_1 \rangle$, $\langle b_1, b_2, b_7, b_6, b_4, b_3, b_1 \rangle$, and $\langle b_3, b_5, b_6, b_5, b_3 \rangle$ which share edges between them. Although breaking of each diamond will require at least one witness,
Input: ICFG ig, Weighting W, A set of m machines M (M₁,...,Mₘ)  
Output: Mᵢ ↦→ wit  
Algorithm:  
Step 1:  
// find all diamonds and directed cycles  
// and create jobs based on strategies  
D = findAllDiamonds(ig)  
C = findDirectedCyclesWithoutDiamonds(IGF ig)  
J = grouping-strategy(D,C)  
// set of Jobs J,  
// initialize profile in machine  
for each machine Mk  
Mₖ.profile.witness = {}, Mₖ.profile.ce = ig.E  
AssignJobsToMachine(M, J)  
assignJob(Mₖ, Jᵢ)  
if not bounded  
EdgeBasedIncrementalAdd2Machine(Jᵢ, Mₖ.profile)  
else // bounded profile size with bound Limit  
  for each directed-cycle C ∈ Jᵢ  
    Job = (C)  
    if Mₖ.profile.witnesses.size + 1 > Limit  
      Mₖ.profile = new Profile()  
      Mₖ.profile.witness = {}, Mₖ.profile.ce = ig.E  
    EdgeBasedIncrementalAdd2Machine(Job₁, Mₖ.profile)  
  for each directed-cycle C ∈ Jᵢ  
    Job = (C)  
    if Mₖ.profile.witnesses.size + 2 > Limit  
      Mₖ.profile = new Profile()  
      Mₖ.profile.witness = {}, Mₖ.profile.ce = ig.E  
    EdgeBasedIncrementalAdd2Machine(Job₁, Mₖ.profile)  
EdgeBasedIncrementalAdd2Machine(Jᵢ, P)  
for each Diamond D ∈ Jᵢ  
  remove from P.ce all edges of D  
  for each Directed Cycle C ∈ Jᵢ  
    remove from P.ce all edges of C  
// spanning graph computation  
add edges to P.ce till no more cycle is created  
P.witnesses = ig.E - P.ce  
P.cost is the cost of sum of edge weights in P.witnesses  

Figure 7: Algorithm: Edge-witness, bounded and un-bounded Parallel Distribution  

but in the same profile all the three diamonds can be broken using two witnesses.  
The overall algorithm for profile distribution for un-bounded profile size and edge-witnesses  
is shown in Figure 7. In the first step jobs are identified based on one of the three grouping  
strategies. The Figure presents the algorithm to find all diamonds and all directed cycles  
without diamond which form the jobs. Using the greedy approximation algorithm, each job  
is assigned to to the machine with least current load. The assignment procedure re-calculates  
the load of the assigned machine. Once a job is assigned to a machine (recall that in un-  
bounded case there is only one profile in the machine denoted by Mₖ.profile), it is possible  
to determine the optimal set of witnesses Mₖ.profile.witnesses to cover all the diamonds  
and directed-cycles-without-diamond by running the Ball-Larus' optimal edge-profiling  
algorithm from scratch. Such from-scratch evaluation is not scalable when number of jobs are  
large - case seen with the fine-grained distribution strategy. Instead, we devise an incremen-  
tal algorithm to repeatedly recalculate witness sets while assign job to machines. With every  
profile p we keep a set of edges p.ce that are complement of witnesses edges in the profile.
```plaintext
findAllDiamonds (ICFG ig)
AllDiamonds = \{\}\nallForks = find nodes that have more than 1 outgoing directed edges
for each fork ∈ allForks
  for each pair of branch targets b_i and b_j
    join = where directed paths from b_i and b_j intersect
    diamond = fork, join, b_i, b_j, all edges from b_i and b_j to join
    add diamond to AllDiamonds
return AllDiamonds

findDirectedCyclesWithoutDiamonds (ICFG ig)
AllCycles = \{\}\nD = findAllDiamonds (ig)
for each cycle in AllDirectedCycles in ig
  found = false
  for each edge ∈ cycle
    if edge = D.fork → D.b_i
      found = true
      break
  if !found
    add cycle to AllCycles
```

Figure 8: Finding diamonds and directed cycles without diamonds

We explain the algorithm using the example in Figure 6(a). In the first step, three jobs, each containing a diamond is identified which need to be distributed to two machine $M_1$ and $M_2$. profile.ce for each machine is initialized with all the edges in the CFG. When a diamond with fork $b_2$ is added to $M_1$ all edges of the diamond are removed from $M_1.profile.ce$. Then edges are added till no more cycle is formed. In this case it will add the edges $b_2 → b_3$ and $b_2 → b_4$. The edges that are not added in $M_1.profile.ce$ ($b_3 → b_5$, $b_4 → b_5$) constitute its realizable witness set which breaks the diamond. Similar process runs when diamond with fork $b_5$ is added to $M_2$. After that diamond with fork $b_8$ is added to $M_1$. All its four edges are removed from $M_1.profile.ce$. Then three edges of the diamond with higher weights ($b_8 → b_9$, $b_9 → b_11$, and $b_8 → b_10$) are added back to ce without introducing any cycle to the ce. The complement edges of ce ($3 → b_5$, $b_4 → b_5$, and $b_10 → b_11$) form the witness for $M_1.profile$.  

The above three strategies can also be applied with modification in cases when profile size is bounded. In this case each machine can have multiple profiles to run where the size of each profile is bounded by a limit. We refer to this as the bounded parallel distribution problem. We show that optimal 2-bounded parallel distribution problem is NP-Complete, and then generalize it for all bounds.

We again reduce the multiprocessor scheduling problem to bounded parallel distribution problem. For each job, we create a diamond with 2 arm-nodes in the profile such that sum of weights of nodes is equal to the job time. Each diamond is put in a loop and loops are joined in series. Then optimal solution to this problem is the solution to multiprocessor scheduling problem. As an instance of optimal bounded parallel is NP-Complete, thus the problem is NP-Complete.

The given algorithm is modified, in function AssignJob to divide the job into multiple profiles (taking into consideration the latest profile in the Machine) with bounded size before adding to a machine.
Input: ICFG $G$, Set of Stacks $T$ consisting of Partial-Paths $T_i$.
A Map $M : (pc,q_i,q_j) \rightarrow WitD_k$
such that $\text{cond}(pc) = \text{true}$, $WitD_k$ satisfies the necessary
and sufficient condition for $q_i,q_j \rightarrow^+ pc$

Output: Execution $E$

Algorithm:
$pc = \text{entry}(G,\text{main})$
$E = pc$
do
if not $\text{cond}(pc)$ then
    $\text{newpc} = \text{succ}(pc)$
else
    if $\exists q_m \in \text{succ}(pc)$ s.t. $\exists T_i, \text{peek} \in \text{witness}(G,\text{WitD}_i,pc,q_m)$
        $q = q_m$
        for each $q_i \in \text{succ}(pc), q_i \neq q_m$, & $q_i \rightarrow^+ pc$
        $WitD_k = M((pc,q_i,q))$
        if $T_k,\text{peek} \in \text{witness}(G,\text{WitD}_k,pc,q_i)$
            $q = q_i$
        else
            $\exists q \in \text{succ}(pc), s.t. \exists T_i, T_i, \text{peek} \in \text{witness}(G,\text{WitD}_i,\text{pc},q)$
            & $q_m \rightarrow^+ pc$ does not exist
            $\text{newpc} = q$
            for all $T_k$ s.t. $T_k,\text{peek} = q$
            $T_k,\text{pop}$
    $pc = \text{newpc}$
$E += pc$
until $pc = \text{exit}$

Figure 9: Algorithm: Path Reconstruction for Realizable Distribution

Figure 10: Example CFG with partial-paths at different stages of path reconstruction

The empirical comparison of the three strategies is described in Section 6.

4.3 Path Reconstruction from Partial Paths

We now present an algorithm to merge multiple partial-paths corresponding to a realizable
distribution to derive the full execution path. The algorithm is presented in Figure 9. The
illustration of the algorithm for an example is given in Figure 10. There are four targets to
the conditional $b_1$, three of which loop back to $b_1$ and the remaining one is $\text{exit}$. The result of
merging the partial-path is the execution $\langle \text{entry}, b_0, b_3, b_4, b_0, b_1, b_4, b_0, b_2, b_4, b_0, \text{exit} \rangle$. The addition of nodes $\text{entry}$ and $b_0$ is straightforward. After this we try to find out which node
occurred after $b$ among $b_1, b_2$ and $b_3$. We first get $q_m = b_3$ (line 57), then compare against
$b_1$ (Lines 58-61). The map $M$ refers to the profile which is used to break the diamond with
branch targets $q_i$ and $q_j$ and fork $pc$. In the first partial-path in the figure shows that $b3$ is at the top, thus $b3$ definitely occurred before $b1$. We proceed with the second comparison with $b2$. The third partial-path shows that $b3$ occurred before $b2$. Thus the next node executed is $b3$. The partial-paths are now adjusted by removing $b3$ from the top. The rest of the execution reconstruction is similar to this and omitted.

The algorithms described in this section are general to edge-based or node-based distribution making it easily applicable towards breakpoint based and instrumentation based collection.

5 Trace Collection for ABAP Programs

SAP-ABAP program are compiled into low-level code and run by a virtual machine inside SAP system. SAP system does not provide direct access to the low-level code for instrumentation to the user/programmer. Moreover, SAP-ABAP practice does not allow programmer induced instrumentation of code in production systems. Its safety is not considered at par with trusted standard agents like SAP compiler or debugger. In general, most production systems do not allow low-level code instrumentation induced by user, but do permit remote debugging. Here, we present a technique for trace collection using code breakpoints inserted by SAP debugger.

Consider the necessary and sufficient condition for realizable profile distribution and parallel distribution strategies presented in the previous section. The strategies distribute covering of diamonds and breaking of directed-cycles-without-diamonds into multiple profiles, essentially breaking cycles and diamonds. However there are many possible profiles that can break a given set of diamonds and cycles. For example, node based profiles like basic block and conditional targets (described in Section 3) can be used. For edge-based profiles, Ball-Larus’ solution [1] (also described in Section 3) can be used to get an optimal profile to break the diamonds and cycles. However, no such optimal algorithm exists for node based profiling. Breakpoint based solution requires such optimal node-based profiles.

Converting an optimal edge-based realizable profile to an optimal node-based profile is not straightforward. It can be easily seen that, for each edge in the edge-based profile, selecting any one of the source or target node of the edge or even selecting both the source
and target may not in general lead to a realizable profile (see Figure 11). Below we present
the realizability condition for node based profile:

A path in CFG $G$ is *witness-free* with respect to a set of nodes $V_{wit}$ iff no nodes traversed
in the path is in $V_{wit}$. Given an edge $p \rightarrow q$ where $p$ is a conditional node in the CFG, the
*witness-set* for conditional $p$ to node $q$ is

$$V_{witness}(G, V_{wit}, p, q) = \{ q \mid q \in V_{wit} \} \cup \{ y \mid y \in V_{wit}, \exists \text{ witness-free path } p, q, \ldots, x \} \cup \{ EOF \mid \exists \text{ witness-free path } p, q, \ldots, \text{exit} \}.$$ 

**Theorem 4** $V_{wit}$ is a realizable nodes-profile iff for each predicate $p$ and its successors $q_1, \ldots, q_n$, for all pairs $q_i, q_j$ where $i \neq j$, $V_{witness}(G, V_{wit}, p, q_i) \cap V_{witness}(G, V_{wit}, p, q_j) = \phi$.

Proof. We first prove that given a profile satisfying the given conditions, it is realizable.
If the condition is satisfied then for each predicate $p$, we know which target $q_i$ got executed
by inspecting the collection’s top node which contain $V_{witness}(G, V_{wit}, p, q_i)$ as this set is
mutually disjoint for other targets $q_j$ of predicate $p$.

Now we prove that any realizable profile will have the stated conditions met for each
predicate. We prove this by contradiction. Say there exists a predicate $p$ for which we have
two targets $q_i$ and $q_j$ for which

$$\exists v, s.t. v \in V_{witness}(G, V_{wit}, p, q_i) \cap V_{witness}(G, V_{wit}, p, q_j).$$

Now we can create an execution where $v$ appears at some point in the collection. Then it is not possible to infer whether
$q_i$ or $q_j$ executed after $p$. Thus the profile is not realizable - a contradiction.

The following CFG based theorem can be derived from the above path based theorem.

**Theorem 5** $V_{wit}$ is a realizable nodes-profile iff $V_{wit}$ breaks all diamonds and directed
cycles, where, a diamond is broken by $V_{wit}$ when there is a node in $V_{wit}$ which is neither
its fork nor join; and a directed cycle is broken by $V_{wit}$ if it contains a node in $V_{wit}$.\(^4\)

Proof. We prove that above two theorems are equivalent. Say a predicate $p$ and its two
successors $q_i, q_j$ creates a diamond with join-point $v$. Consider the path from $q_i$ to $v$ and $q_j$
to $v$. If there exists no vertex witnesses in these paths not including $v$, a vertex witness at
node $v$ makes the profile unrealizable as

$$\in V_{witness}(G, V_{wit}, p, q_i) \cap V_{witness}(G, V_{wit}, p, q_j) \neq \phi.$$ 

If there is no witness in $v$ then these two paths cannot be distinguished and profile is still realizable.
Thus this puts a necessary and sufficient condition that there should exist at least one v.witness node in these
two paths. This proves the necessary and sufficient condition for diamonds. The rest of
the proof about breaking directed cycles follows from Ball-Larus’s proof [1].

The algorithm for optimal node-based profile is shown in Figure 12. The optimal node-
based profiling algorithm has to consider the case of special diamonds (shown in Figure 11(c))
where choosing a source or target or both, from an optimal edge based profile won’t generate
a realizable node-based profile. To break such diamonds, the algorithm inserts its fork-join
edges in the tree (in Step 1), prior to maximum spanning tree computation (Step 2), such
that these fork-join edges do not belong to the resulting edge based profile (complement of
$MCST$). The Step 3 selects appropriate nodes-based profile from the edge based profile.

\(^4\)Our initial conjecture, in light of necessary and sufficient condition of Ball-Larus’ work, that $V_{wit}$ is realizable iff $V - V_{wit}$
contains no diamonds or directed cycles was incorrect.
get_optimal_profile
Input: A job J
Output: Optimal realizable Vwit
Algorithm:
Step 1: Identification of fork-join pairs
MCST = {} for each diamond d in J with fork f and join j
    if ∃ f → j
        add (f, j) to MCST
Step 2. // Same as optimal Ewit computation
Perform maximum spanning tree by adding edges to MCST.
Ewit' = E − MCST where E is all the edges in J
Step 3: Identification of Vwit
For each edge (src, tgt) ∈ Ewit'
    if src is a fork-node
        add tgt to Vwit
    elseif tgt is a join-node
        add src to Vwit
    elseif src && tgt are neither fork nor join
        add the node with minimum weight to Vwit

NodeBasedIncrementalAdd2Machine(J_i, P)
for each Diamond D ∈ J_i
    remove from P.ce all edges of D
for each Directed Cycle C ∈ J_i
    remove from P.ce all edges of C
for each diamond D in J_i with fork f and join j
    if ∃ f → j
        add (f, j) to P.ce
// spanning graph computation
add edges to P.ce till no more cycle is created
Ewit' = ig.E − P.ce
For each edge (src, tgt) ∈ Ewit'
    if src is a fork-node
        add tgt to P.witnesses
    elseif tgt is a join-node
        add src to P.witnesses
    elseif src && tgt are neither fork nor join
        add the node with minimum weight to P.witnesses
P.cost is the cost of sum of node weights in P.witnesses

Figure 12: Algorithm: Optimal Vwit Computation

The optimality follows from the optimality of Ball-Larus’ algorithm and minimum weight of source/target nodes chosen in Step 3.

6 Experimental Results

In this section we compare the effectiveness of various distribution strategies of parallel trace collection for a set of ABAP programs.

The characteristics of the 12 benchmark ABAP programs are shown in Figure 13. All these programs are real ABAP report programs in SAP. For each benchmark program, a trace is collected. The characteristics of the trace are also shown. The implementation of the algorithms is done in Java and experiments are conducted on a laptop running windows
Figure 13: Benchmark and Test characteristics: S is the number of entities in the program, C is the total number of such entities covered by the trace and H is the number of hits of the entity

<table>
<thead>
<tr>
<th>Bench</th>
<th>Stmt</th>
<th>Basic Block</th>
<th>Diamond</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>1074/185/9084</td>
<td>475/93/4031</td>
<td>638/29/1607</td>
<td>10/2/126</td>
</tr>
<tr>
<td>inv</td>
<td>1517/413/6319</td>
<td>745/219/3253</td>
<td>1878/68/958</td>
<td>12/2/66</td>
</tr>
<tr>
<td>imat</td>
<td>1916/488/4981</td>
<td>756/245/3253</td>
<td>1878/68/958</td>
<td>12/2/66</td>
</tr>
<tr>
<td>inap</td>
<td>1047/159/1297</td>
<td>355/121/375</td>
<td>638/29/1607</td>
<td>10/2/126</td>
</tr>
<tr>
<td>r013</td>
<td>625/520/1073</td>
<td>82/46/105</td>
<td>41/6/22</td>
<td>0/0/0</td>
</tr>
<tr>
<td>rv54</td>
<td>1361/219/223</td>
<td>635/104/108</td>
<td>1814/21/21</td>
<td>21/2/4</td>
</tr>
<tr>
<td>tchb</td>
<td>1361/219/223</td>
<td>635/104/108</td>
<td>1814/21/21</td>
<td>21/2/4</td>
</tr>
<tr>
<td>tcmm</td>
<td>1361/219/223</td>
<td>635/104/108</td>
<td>1814/21/21</td>
<td>21/2/4</td>
</tr>
<tr>
<td>wip</td>
<td>1397/478/5197</td>
<td>541/245/2309</td>
<td>300/65/721</td>
<td>13/8/139</td>
</tr>
<tr>
<td>zrep</td>
<td>56/32/1481</td>
<td>25/17/855</td>
<td>17/7/287</td>
<td>0/0/0</td>
</tr>
<tr>
<td>zrotc</td>
<td>251/112/2706</td>
<td>105/63/1041</td>
<td>16/6/338</td>
<td>0/0/0</td>
</tr>
</tbody>
</table>

Figure 14: Effectiveness of distribution strategies in form of cost of distribution/cost of collection for BL-profile

XP and with dual core 2.53Ghz CPU and 4GB RAM. For each program and its corresponding trace, we find a profile and take an intersection of the path with the profile to determine the total number of hits. The trace collection time is proportional to the total number of hits, with an average of 0.5 seconds/hit.

We use Ball-Larus [1] profiling algorithm (referred to as BL-profile) for assigning weights to the nodes and edges. We also consider a case where the profile weights are taken based on exact execution trace (hereafter referred as trace profile).

Our first experiment studies how the effectiveness of parallel distribution with un-bounded profile size varies with the available machine count (without putting any limit on the number of witnesses), for the three algorithms for distributing jobs using BL-profile. Figure 14 shows the results. For BL-profile and trace profile all graphs are available at Figure 18 and Figure 19. Each data-point (in the format A/B) is a pair consisting of cost of distribution (A), and the cost of collection (B) (refer to Section 4.2), where both costs are defined as maximum cost across all machines. Here is the summary of our observations.

Cost of collection vs. Machine. In Figure 15 we present the variation of cost of collection with respect to number of machines. For each benchmark, we first obtain the cost of collection in one machine (non-distribution). For each number of machine, we take the minimum cost of collection (for various strategies) and take its percentage over non-distributed cost of collection. For benchmarks rls and inap the reduction is very less and for zrep there is absolutely no reduction. Leaving these 3 benchmarks, the average percentage of hits are 62, 49, 44, 38, 32, and 31 for 2 to 7 machines, respectively. Thus using 6 machines we the trace collection time reduces to less than 1/3.

Comparison of distribution strategies. To compare the three grouping strategies we consider...
two profiling algorithms separately. For the trace profile, the profile cost distribution accurately reflects the hit distribution as well. In this case we notice that optimized strategy is always better than the function-wise grouping strategy, however fine-grained distribution generates minimum profile cost in half of the cases. We show two cases where each strategy emerges as a winner in Figure 16(a) and (b). In general we found that for benchmarks with less number of functions show better results with fine grained distribution. This case is particularly noticed in trace profiles where functions are not executed and so the diamonds and directed cycles in those function have zero weights, in effect, the number of non-zero weight functions are smaller. This is not the case for BL-profile as the assignment of weights are done not based on the actual trace. In BL-profile case we see more number of winners for function-wise grouping strategies and optimized-grouping strategy. The four winners of fine-grained distribution strategy are rls, r013, zrep, and zrotc. Interesting to note that in all
these case the benchmarks do not have any cycles (see Figure 13). Note that grouping strategies are more effective in reducing weights of profiles when multiple diamonds are in cycles (as in Figure 6(a)). Absence of any cycle makes fine-grained strategy more effective than the two grouping strategies in those four cases. We show representative cases in Figure 16(c) and (d).

**Effect of bounded profile size.** The Figure 17 shows the effectiveness of parallel distribution (in terms of cost of collection) with variation in the limit of witnesses in each profile. The results are taken with number of machines 3 and with function-wise grouping distribution strategy. The aim of this experiment is to see how witness-optimizable diamonds are distributed with increase in limit. Even though we show the result for various limits, SAP-ABAP debugger limits the number of breakpoints in a debugging session to 30. Since function-wise grouping strategy is better suitable for keeping potential witness-optimizable diamonds together, the experiment is done with this selection strategy. We notice that with increase in limit, in most cases the number cost decreases, with an exception in few cases. We investigate these cases. In these cases the diamonds in each CFG are not witness-optimizable, thus combining them increases the profile size and cost.

### 7 Related Work

**Path collection.** Program path collection has been an area of active research in the software engineering community for quite long, particularly due to its importance in a wide variety of applications. The theory of program tracing thus dates back to four decades. Rammoorthy et al. [17] gave a necessary and sufficient condition for edge based realizable profile for obtaining trace of single procedure programs. They wanted to find a minimal-size solution (minimum number of edges) to the tracing problem which was proven to be NP-Complete [13]. In contrast, Ball-Larus, in their seminal work [1] presented an algorithm to solve minimal-cost solution. They reformulated the condition presented by Rammoorthy et al. [17] and extended to multi-procedure case. The program tracing algorithm presented in [1] presented an optimal edge-based placement. However, this solution is applicable for only instrumentation, as breakpoint based solution requires witnessing nodes and not edges.

**Breakpoint based collection.** To our knowledge [7, 20] are the other techniques that use breakpointing to gather path information. In [7] debugging time breakpoints, set by user (not automatic), are used to find whether a statement is executed (not complete path as

<table>
<thead>
<tr>
<th>bench</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>border</td>
<td>1306</td>
<td>1005</td>
<td>597</td>
<td>687</td>
</tr>
<tr>
<td>illn</td>
<td>1203</td>
<td>729</td>
<td>641</td>
<td>567</td>
</tr>
<tr>
<td>imat</td>
<td>3207</td>
<td>2005</td>
<td>1485</td>
<td>1273</td>
</tr>
<tr>
<td>imap</td>
<td>190</td>
<td>73</td>
<td>47</td>
<td>59</td>
</tr>
<tr>
<td>r013</td>
<td>719</td>
<td>374</td>
<td>374</td>
<td>374</td>
</tr>
<tr>
<td>rls</td>
<td>34</td>
<td>24</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>tcll</td>
<td>750</td>
<td>910</td>
<td>1080</td>
<td>1097</td>
</tr>
<tr>
<td>tcm</td>
<td>6841</td>
<td>5638</td>
<td>4992</td>
<td>4882</td>
</tr>
<tr>
<td>rvs4</td>
<td>59</td>
<td>45</td>
<td>35</td>
<td>32</td>
</tr>
<tr>
<td>wip</td>
<td>492</td>
<td>475</td>
<td>495</td>
<td>435</td>
</tr>
<tr>
<td>zrep</td>
<td>352</td>
<td>421</td>
<td>423</td>
<td>423</td>
</tr>
<tr>
<td>zrotr</td>
<td>421</td>
<td>421</td>
<td>421</td>
<td>421</td>
</tr>
</tbody>
</table>
here) which is used to refine static slice. In [20] breakpoints are put in start of all basic blocks to obtain path. All the other techniques use instrumentation based approach.

**One program, many collections.** Techniques such as [8, 3, 11] have used multiple traces of the same program to localize the bugs. However, they used different faulty and correct inputs for the same program, thereby collecting multiple different trace.
Closest to our work is by Palankar et. al. [15] where order of data access is preserved by merging multiple executions of the same program with same input into a single trace. The division of executions were performed due to limited number of hardware assisted data-breakpoints to watch data accesses. However, they do not try to obtain the exact order of data access and use heuristics (like using information in instruction counter) to arrive at
a close to accurate solution. Recent work [18] performs merge operation on partial calling context obtained using lightweight profiling. However, the merge operation performed is maximal matching with the aim that it captures real calling context tree with high probability.

Diep et al. [4] consider distribution of probes to multiple program variants, where each variant contains a subset of probes, where the subset size can be bounded to meet the overhead requirements. However, the aim there is to profile a set of events, and not generating order of events for a particular input.

Software tomography [2] splits monitoring tasks across many instances of the software, so that partial information may be collected from users by means of light-weight instrumentation and merged to gather the overall monitoring information. Even though the framework sounds similar, it has been applied for merging profiling information and not for constructing any information which require respecting order of events.

8 Conclusion

In this paper we have presented a novel divide-and-conquer based trace collection technique in which partial traces may be collected and subsequently merged to produce the complete trace. We showed that such a division of trace collection is a non-trivial problem, and presented a necessary and sufficient condition to guarantee the recovery of the whole path by merging the partial traces. Since partial collections may be done in parallel (using a hadoop based framework), the total time of trace collection is governed by the maximum total time of collection across machines. This required even distribution of witnesses across machines. We show the optimal distribution is an NP-Complete problem and use the existing approximation solutions to effectively distribute the witnesses. To apply such a technique for breakpoint based collection, we extended Ball-Larus’ optimal edge-based profile algorithm to node-based profile algorithm.

Our work opens up other possibilities of parallel collection of entities where order of elements are of importance. Applying these techniques for obtaining call trace and order of data access can be fruitful. We are investigating other uses of this framework such as multithreaded application using Uniparallelism approach [19] and evaluating more dynamic scheduling techniques for parallel collection.

References


