Whole Execution Traces and Their Applications

XIANGYU ZHANG and RAJIV GUPTA
University of Arizona

Different types of program profiles (control flow, value, address, and dependence) have been collected and extensively studied by researchers to identify program characteristics that can then be exploited to develop more effective compilers and architectures. Because of the large amounts of profile data produced by realistic program runs, most work has focused on separately collecting and compressing different types of profiles. In this paper, we present a unified representation of profiles called Whole Execution Trace (WET), which includes the complete information contained in each of the above types of traces. Thus, WETs provide a basis for a next-generation software tool that will enable mining of program profiles to identify program characteristics that require understanding of relationships among various types of profiles. The key features of our WET representation are: WET is constructed by labeling a static program representation with profile information such that relevant and related profile information can be directly accessed by analysis algorithms as they traverse the representation; a highly effective two-tier strategy is used to significantly compress the WET; and compression techniques are designed such that they minimally affect the ability to rapidly traverse WET for extracting subsets of information corresponding to individual profile types as well as a combination of profile types. Our experimentation shows that on, an average, execution traces resulting from execution of 647 million statements can be stored in 331 megabytes of storage after compression. The compression factors range from 16 to 83. Moreover the rates at which different types of profiles can be individually or simultaneously extracted are high. We present two applications of WETs, dynamic program slicing and dynamic version matching, which make effective use of multiple kinds of profile information contained in WETs.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers, debuggers, optimization*; C.4 [**Performance of Systems**]: Measurement Techniques

General Terms: Algorithms, Measurement, Performance

Additional Key Words and Phrases: Profiling, control flow, dependences, addresses, values, compression

A preliminary version of this paper appeared in MICRO 2004.
Supported by grants from Microsoft, IBM, Intel, and NSF grants CCR-0324969, CCR-0220262, CCR-0208756, and EIA-0080123 to the University of Arizona.
Authors' address: University of Arizona, Department of Computer Science, Tucson, AZ 85721; email: {xyzhang,gupta}@cs.arizona.edu.

1. INTRODUCTION

A software tool for collection, maintenance, and analysis of detailed program profiles for realistic program runs can greatly benefit compiler and architecture researchers. This is because program profiles can be analyzed to identify program characteristics that can then be exploited by researchers to guide the design of superior compilers and architectures. The key challenge that one faces in developing a software tool is that the amounts of profile information generated during realistic program runs can be extremely large. One approach to reducing the amount of profile data is by using lossy compression or summarization techniques. Lossy compression of variety of profiles has been carried out including, dynamic dependence profiles in Agrawal and Horgan [1990], dynamic control flow in Ball and Larus [1996], and dynamic values in Calder et al. [1997]. Although, for many, applications summarization is adequate, for others they have proved to be inadequate. For example, it has been shown that summarization of dynamic data dependences results in high levels of inaccuracy in dynamic data slices [Zhang et al. 2003].

Researchers have developed lossless compression techniques to limit the memory required to store different types of profiles. These techniques for several different types of profiles have been separately studied. Compressed representations of *control flow* traces can be found in Larus [1999] and Zhang and Gupta [2001]. These profiles can be analyzed for presence of hot program paths or traces [Larus 1999], which have been exploited for performing path-sensitive optimizations [Young and Smith 1998; Ammons and Larus 1998; Bodik et al. 1998; Gupta et al. 1998] and path-sensitive prediction techniques [Jacobson et al. 1997]. *Value profiles* have been compressed using value predictors [Burtscher and Jeeradit 2003; Burtscher 2004] and used to perform code specialization [Calder et al. 1997], data compression [Zhang and Gupta 2002], value speculation [Lipasti and Shen 1996], and value encoding [Yang and Gupta 2002]. *Address profiles* have also been compressed [Chilimbi 2001] and used for identifying hot data streams that exhibit data locality, which can help in finding cache-conscious data layouts [Rubin et al. 2002] and developing data-prefetching mechanisms [Chilimbi and Hirzel 2002; Joseph and Grunwald 1997]. *Dependence profiles* have been compressed in Zhang and Gupta [2004] and used for computation of dynamic slices [Zhang and Gupta 2004], studying the characteristics of performance-degrading instructions [Zilles and Sohi 2000], and studying instruction isomorphism [Sazeides 2003].

Each of the above works has studied the handling of a single type of profile. The next step in profiling research is to develop a software tool that unifies the maintenance and analysis of all of the above types of profiles. An effective tool cannot be created by simply using the above-mentioned techniques in combination. If we use the above techniques as is, the stream of values representing control flow, values, and addresses will be separately compressed. If a request is now made for the profile information related to the execution of a statement, we will have to search through each of the compressed streams to gather this information. In other words, the above representation will not provide easy access to related profile information. The goal of designing a unified representation is to overcome the above drawback and enable the understanding of program behavior involving interactions among the different program characteristics captured by these profiles. This will lead to exploration of advanced compiler and architecture techniques that simultaneously exploit multiple types of profiles.

In this paper we present an unified representation and show that it is possible to maintain and make use of such a representation. There are three key challenges that are addressed in this paper in developing such a unified representation, which we call *whole execution traces* (WETs). First, WETs, provide an ability to relate different types of profiles (e.g., for a given execution of a statement, we can easily find the control flow path, the data dependences, values, and addresses involved). This goal is achieved by designing WET so it is constructed by labeling a static program representation with profile information, such that relevant and related profile information can be directly accessed by analysis algorithms as they traverse the representation. Second, we develop an effective two-tier compression strategy to reduce the memory needed to store WETs. First, we use customized compression techniques for different types of profiles and then we use a generic compression technique to compress streams of values corresponding to all types of profile information. Third, the compression is achieved in such a way that WETs can be rapidly traversed to extract subsets of information corresponding to individual profile types (i.e., control flow, value, address, and dependence) as well as subsets of related information, including all types of profiles (e.g., dynamic slices of WETs corresponding to computation of a value). The customized compression schemes are designed such that they minimally affect the cost of traversing the WETs. The generic compression scheme is designed to enable bidirectional traversal, i.e., given a position of a value in the stream, it is possible to find the immediately preceding and following values with equal ease. In Burtscher and Jeeradit [2003], it is shown that value predictor-based compression techniques outperform other generic compression techniques, such as *gzip*, *bzip*, and *Sequitur* [Nevil-Manning and Witten 1997]. In comparison to Burtscher and Jeeradit [2003], our generic compresion scheme supports bidirectional traversability while, at the same time, having similar compression ability.

We have implemented the unified WET representation using the Trimaran compiler infrastructure [Trimaran 1997]. Extrapolation from our experimental results shows that whole execution trace corresponding to a program run involving execution of 3.9 billion intermediate-code statements can be stored in 2 gigabytes of memory, which is commonly available on machines today. Being able to hold profiles of a few billion instructions in memory is a critical milestone because other works have shown that behaviors of long program runs can be effectively characterized by considering smaller program runs or smaller segments of longer program runs that are few billion instructions long [KleinOsowski and Lilja 2002; Perelman et al. 2003]. In KleinOsowski and Lilja [2002], program inputs with smaller program runs that effectively characterize program behavior were identified for Spec benchmarks. In Perelman et al. [2003] it was shown that by appropriate selection of smaller segment of a longer program run, program's execution can be effectively characterized. We have also evaluated the ease and, hence, the speed at which subsets of profile information can be extracted from WET in response to a variety of queries. Our results show that these queries, which ask for related profile information, can be responded to rapidly. We also present two applications of WETs, dynamic program slicing and dynamic version matching, which make effective use of multiple kinds of profile information contained in WETs.

The rest of the paper is organized as follows. Section 2 describes the uncompressed WET representation. Section 3 presents first tier-compression methods for control flow, data dependences, values, and addresses. The key characteristic of first-tier compression is that it does not negatively impact the speed with which the profile information can be accessed. Section 4 presents a generic scheme for compressing a stream of values. This scheme is used to compress the stream of values corresponding to all types of profile information. Section 5 presents results of experiments aimed at measuring the space and time costs of storing and using WETs. In Section 6, we present two applications of WETs—dynamic program slicing and version matching. Conclusions are given in Section 7.

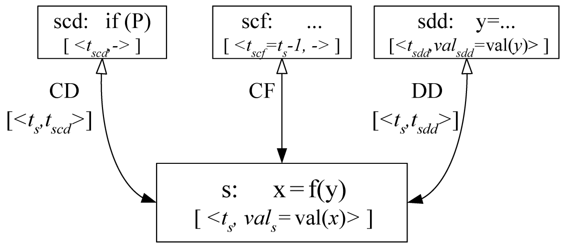
2. WHOLE EXECUTION TRACE

The WET is a unified representation that holds full execution history including, control flow, value, address, and dependence (data and control) histories. WET is essentially a static representation of the program that is labeled with the dynamic profile information. This organization provides a direct access to all relevant profile information associated with every execution instance of every statement. A statement in WET can correspond to a source-level statement, intermediate-level statement, or a machine instruction. In our discussion we assume that each statement is an intermediate-code statement.

In order to represent profile information of every execution instance of every statement, it is clearly necessary that we are able to distinguish between execution instances of statements. The WET representation we develop distinguishes between execution instances of a statement by assigning unique *timestamps* to them. To generate the timestamps, we maintain a *time* counter that is initialized to one and each time a basic block is executed. The current value of *time* is assigned as a timestamp to the current execution instances of all the statements within the basic block; *time* is then incremented by one. Timestamps assigned in this fashion essentially allow us to remember the ordering of all statements executed during a program execution. The notion of timestamps is also key to representing and accessing the dynamic information contained in WET.

The WET is essentially a labeled graph whose form is defined next. A label associated with a node or an edge in this graph is an ordered sequence where each element in the sequence represents a subset of profile information associated with an execution instance of a node or edge. The relative ordering of elements in the sequence corresponds to the relative ordering of the execution instances. We denote a sequence of elements e_1, e_2... as $[e_1 e_2 \ldots]$. For ease of presentation we assume that each basic block contains one statement, i.e., there is one-to-one correspondence between statements and basic blocks.

Definition: The Whole Execution Trace (WET) is represented in form of a labeled graph $G[N, E(CF, CD, DD)]$, where N is the set of statements in the program. Each statement $s \in N$ is labeled with a sequence of ordered pairs: $[< t_s, val_s >]$ where statement s was executed at time t_s and produced the value val_s. Note that, in general, when a node contains multiple statements, instead of a single value in each ordered pair, we have a set of values one each for every statement in the basic block.



E is the set of edges. The edges are bidirectional so that the graph can be traversed in either direction. $(s \rightarrow d)$ denotes direction of the edge that takes us from the source s of the dependence to the destination d of the dependence, while $(s \leftarrow d)$ is used to denote the reverse direction. The edges are subdivided into three disjoint categories.

- *DD* is the set of *data dependence* edges in the program. Each edge $(sdd \rightarrow s) \in DD$ is labeled with a sequence of ordered pairs: $[< t_s, t_{sdd} >]$, where statement s was executed at time t_s using an operand whose value was produced by statement sdd at time t_{sdd}.
- *CD* is the set of *control dependence* edges in the program. Each edge $(scd \rightarrow s) \in CD$ is labeled with a sequence of ordered pairs: $[< t_s, t_{scd} >]$, where statement s was executed at time t_s as a direct result of the outcome of predicate scd executed at time t_{scd}.
- *CF* is the set of *control flow* edges in the program. These edges are *unlabeled*.

The example in Figure 1 illustrates the form of WET. A control flow graph and control flow trace of one possible execution is given in Figure 1a. Since the entire WET for the example is too large, we show the subgraph of WET that captures the profile information corresponding to the executions of node 8. The label on node 8 says that statement 8 is executed five times at timestamps 7, 37, 57, 77, and 97 producing values c, d, d, d, and c, respectively. Executions of statement 8 are control dependent upon statement 6 and data dependent on statements 4, 2, and 15. Therefore, CD and DD edges are introduced whose labels express the dependence relationships between execution instances of statements 6, 4, 2, and 15 with statement 8. Unlabeled control flow edges connect statement 8 with its predecessor 6 and successor 9 in the control flow graph.

2.1 Queries

Next we show how WET can be used to respond to a variety of useful queries for subsets of profile information. The ability to respond to these queries

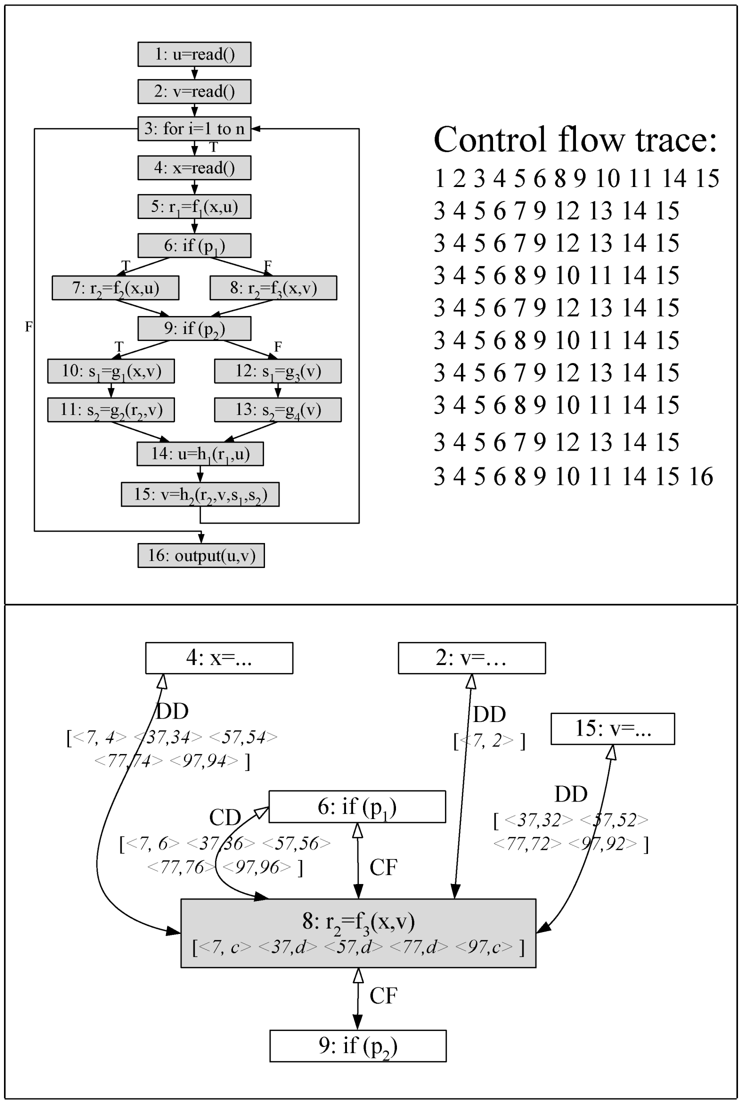


Fig. 1. An example of (a) CFG and its control flow trace; (b) WET subgraph of node 8.

demonstrates that the WET representation incorporates all of the control flow, data and control dependence, value, and address profile information.

2.1.1 *Control Flow Path.* The path taken by the program can be generated from WET using the combination of static control flow edges (*CF*) and the sequences of timestamps associated with nodes (N). If a node is labeled with $< t, - >$, the node that is executed next must be labeled with $< t+1, - >$. Using this observation, we can generate the complete path or part of the program path starting at any execution point.

2.1.2 *Values and Addresses.* The value and address profiles are captured by the values contained in $[< t, v >]$ sequences associated with nodes. Some values represent data while others represent addresses the distinction can be made by examining the use of the values. Values produced by executions of a statement can be obtained by simply examining its $[< t, v >]$ sequence. Addresses corresponding to executions of a specific statement can be obtained by simply examining the $[< t, v >]$ sequences of statements that produce the operands for the statement of interest. On the other hand, if we are interested in examining the sequence of values (addresses) that are produced (referenced) during program execution, we need to follow the control flow path taken as described earlier and then examine the relevant $< t, v >$ pair of each node as it is encountered.

2.1.3 *Data and Control Dependences.* All instances of data and control dependences are captured explicitly by labeled edges (*CD* and *DD*). Chains of data dependences, control dependences, or combinations of both types of dependences can all be easily found by traversing the WET.

We have shown the organization of all types of profile data in the WET representation, which allows a variety of queries to be responded to with ease. Given the large amounts of profile information, the sizes of WETs are expected to be extremely large. Therefore the next challenge we face is to compress the WETs in a manner that does not destroy the ease or efficiency with which queries for information can be handled. In the next two sections we present a two-tier compression strategy that accomplishes these goals.

3. CUSTOMIZED COMPRESSION

The first tier of our compression strategy focuses on developing separate compression techniques for each of the three key types of information labeling in the WET graph: (a) timestamps labeling the nodes; (b) values labeling the nodes; and (c) timestamp pairs labeling the dependence edges. The compression is accompanied with minimal impact on the ease and efficiency of accessing the profiles. These compression strategies essentially aim to minimize the redundancy in the profile information.

3.1 Timestamps Labeling Nodes

The total number of timestamps generated is equal to the number of basic block executions and each of the timestamp labels exactly one basic block. We can reduce the space taken up by the timestamp node labels as follows. Instead of having nodes that correspond to basic blocks, we create a WET in which nodes can correspond to Ball–Larus paths [Ball and Larus 1996] that are composed of multiple basic blocks. Since a unique timestamp value is generated to identify the execution of a node, now the number of timestamps generated will be fewer. In other words when a Ball–Larus path is executed, all nodes in the path share the same timestamp. By reducing the number of timestamps, we save space without having any negative impact on the traversal of WET to extract the control flow trace.

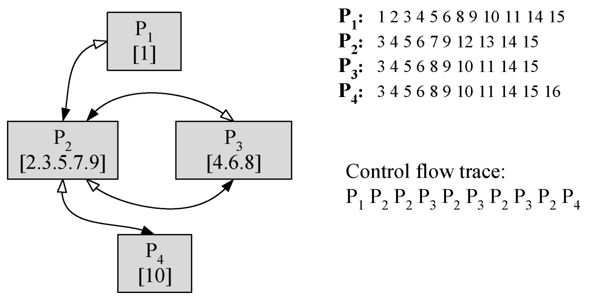

P1 [1]
P2 [2.3.5.7.9]
P3 [4.6.8]
P4 [10]
P1: 1 2 3 4 5 6 8 9 10 11 14 15
P2: 3 4 5 6 7 9 12 13 14 15
P3: 3 4 5 6 8 9 10 11 14 15
P4: 3 4 5 6 8 9 10 11 14 15 16
Control flow trace:
P1 P2 P2 P3 P2 P3 P2 P3 P2 P4

Fig. 2. Reducing number of timestamps.

As an example, the execution shown in Figure 1a involves 103 executions of basic blocks and, hence, generates 103 timestamps. However, the entire execution can be broken down into a total of 10 executions of four distinct Ball–Larus paths, as shown in Figure 2. Thus, if we use the transformed graph shown in Figure 2 where edges represent flow of control across Ball–Larus paths, we only need to generate 10 timestamps as shown.

3.2 Values Labeling Nodes

It is well known that subcomputations within a program are often performed multiple times on the same operand values; this observation is the basis for widely studied techniques for reuse based redundancy removal [Sazeides 2003]. Next we show how the same observation can be exploited in devising a compression scheme for sequence of values associated with statements belonging to a node in the WET.

We describe the compression scheme using the example below in which the value of x is an input to a node and using this value, the values of y and z are computed. Further assume that while the node is executed four times, only two unique values of x (x_1 and x_2) are encountered in the value sequence $Vals[0..3] = [x_1 x_2 x_1 x_2]$. Given the nature of the computation, the values of y and z also follow similar patterns. We can compress the value sequences by storing each unique value produced by a statement only once in the *UVals*[0..1] array. In addition, we remember the pattern in which these unique values are encountered. This pattern is, of course common to the entire group of statements. The pattern [0101] gives the indexes of values in the *UVals*[] array that are encountered in each position. Clearly the *Vals*[0..3] corresponding to each statement can be determined using the following relationship.

$$Vals[i] = UVals[Pattern[i]]$$

Before

Statement	$Vals[0..3]$
x	$[x_1 x_2 x_1 x_2]$
$y = f(x)$	$[y_1 y_2 y_1 y_2]$
$z = g(x, y)$	$[z_1 z_2 z_1 z_2]$

After: Pattern=[0101]

Statement	$UVals[0..1]$
x	$[x_1 x_2]$
$y = f(x)$	$[y_1 y_2]$
$z = g(x, y)$	$[z_1 z_2]$

The above technique yields compression because by storing the pattern only once, we are able to eliminate all repetitions of values in value sequences associated with all statements. The ease with which the sequence of values can be generated from the unique values is a good characteristic of this compression scheme. The compression achieves space savings at the cost of slight increase in the cost of recovering the values from WET.

In the above discussion, the situation considered is such that all of the statements shared a single pattern. In general, multiple patterns may be desirable because different subsets of statements may depend upon different subsets of inputs that are either received from outside the node or are read through input statements within the node. Statements belonging to a node are subdivided into disjoint groups as follows. For each statement the input variables that it depends upon (directly or indirectly) is first determined. Groups are first formed by including all statements that exactly depend upon the same inputs into the same group. Next, if a group depends upon a set of inputs that are a proper subset of inputs for another group, then the two groups are merged. Finally, input statements within the node on which many groups depend is included in exactly one of the groups. Once the groups are formed, a pattern is found for each group and the values are compressed according to the group's pattern.

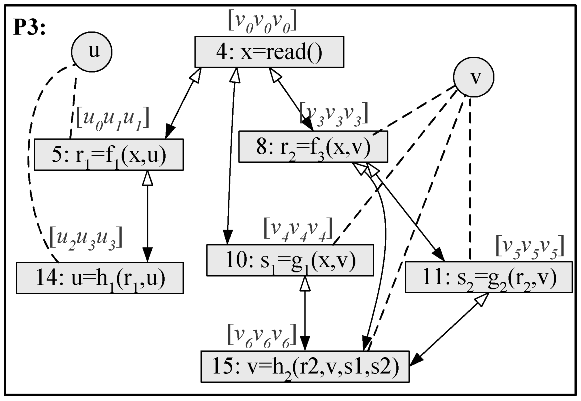
In Figure 3, formation of groups for node $P3$ is illustrated. The top part of the figure shows the value sequences associated with statements before compression. The statements depend upon values of u and v from outside the node and the value of x that is read by a statement inside the node. Two groups are formed because some statements depend upon values of x and v, while other statements depend upon values of x and u. The statement that reads the value of x is added to one of the groups. Once the groups have been identified, patterns are formed for each group as shown.

3.3 Timestamp Pairs Labeling Edges

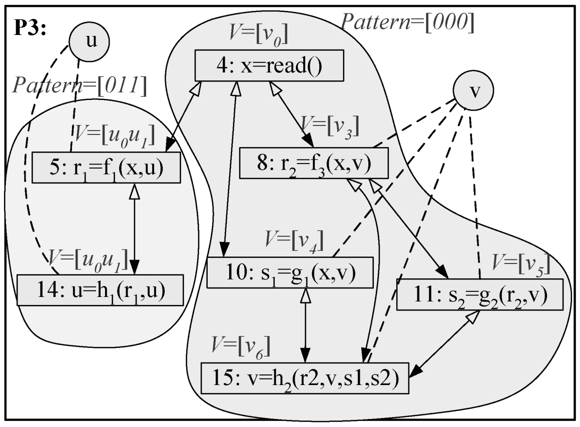
Each dependence edge is labeled with a sequence of timestamp pairs. Next we describe how the space taken by these sequences can be reduced. Our discussion focuses on data dependences; however, similar solutions exist for handling control-dependence edges [Zhang and Gupta 2004].

While, in general, we need to remember all dynamic instances of all dependences, we next show that all dynamic instances need not be explicitly remembered. We develop optimizations that have the effect of eliminating timestamp pairs. These optimizations can be divided into the the following three categories:

1. *Inference*. Static edge is introduced for a dependence and the timestamp pairs corresponding to the dynamic instances of the dependence are *inferred* and thus need not be explicitly remembered.
2. *Transformation*. While some timestamp pairs cannot be inferred from the original static dependence graph, transformations can be applied to the static graph so that the timestamp pairs can be inferred from the transformed graph. Thus, these transformations enable inferring of timestamp pairs.
3. *Redundancy Removal*. There are situations in which different dependence edges are guaranteed to have identical timestamp pairs. Redundant copies of timestamp pairs can thus be discarded.



(a) Before compression



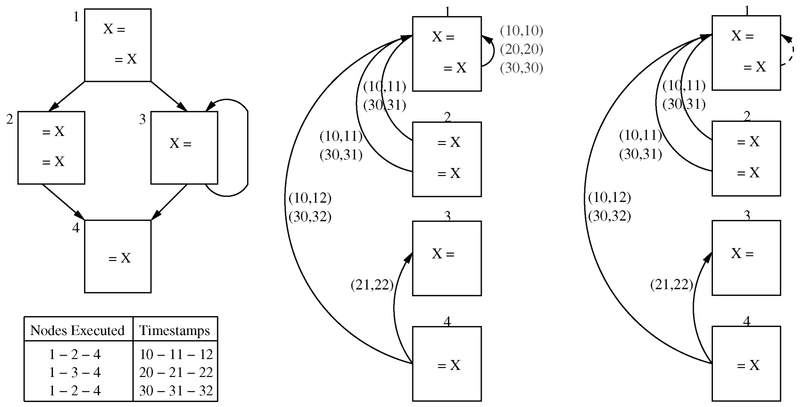
(b) After compression

Fig. 3. Value compression.

Given an execution instance of a use $u(t_u)$, to find the corresponding definition, we need to find the corresponding execution instance of the relevant definition $d(t_d)$. There are two steps to this process: (finding d) in general, many different definitions may reach the use, but we need to find the relevant definition for $u(t_u)$; and (finding t_d) even if the relevant definition d is known, we need to find the execution instance of d, i.e., $d(t_d)$, that computes the value used by $u(t_u)$. The following optimizations show how the above determinations can be made, even in the absence of some timestamp pairs.

3.4 (OPT-1) Inference

3.4.1 *(OPT-1a) Inference of Local Def-Use for Full Elimination.* Consider a definition d and a use u that are *local* to the same basic block, d appears before u, and there is no definition between d and u that can ever prevent d from reaching u. In this case there is one-to-one correspondence between execution instances of d and u. Since d and u belong to the same basic block, the timestamps of corresponding instances are *always the same*, i.e., given a dynamic



Nodes Executed	Timestamps
1 – 2 – 4	10 – 11 – 12
1 – 3 – 4	20 – 21 – 22
1 – 2 – 4	30 – 31 – 32

Fig. 4. Effect of applying OPT-1a.

data dependence $du(t_d, t_u)$ it is always the case that $t_d = t_u$. Therefore, given the use instance $u(t_u)$, the corresponding d is known statically and the corresponding execution instance is simply $d(t_u)$. Thus we do not need to remember dynamic instances individually—it is enough to introduce a static edge from u to d.

When no optimization is used, we begin with a static set of nodes (basic blocks) and introduce all dependence edges dynamically. To take advantage of the above optimization, we simply introduce the edge from u to d statically prior to program execution. No new information will be collected or added at runtime for the use u as the edge from u to d does not need any timestamp labels. In other words, all dynamic instances of the def-use edge from u to d are statically replaced by a single shared representative edge.

The impact of this optimization is illustrated using a dependence graph in Figure 4. Basic block 1 contains a labeled local def-use edge that is replaced by a static edge, which need not be labeled by this optimization. We draw static edges as dashed edges to distinguish them from dynamic edges.

3.4.2 *(OPT-1b) Inference Local Def-Use for Partial Elimination.* In the above optimization, it was important that a certain subpath was free of definitions of the variable involved (say v) so that a dependence edge involving v that is free of labels could be used. In programs with pointers, the presence of a definition of a *may alias* of v may prevent us from applying the optimization, although at runtime this definition may rarely redefine v. To enable the application of preceding optimization in presence of definitions of may aliases of v, we proceed as follows. We introduce a static unlabeled edge from one definition to its potential use. If, at runtime, another may alias turns out to truly refer to v, additional dynamic edges labeled with timestamp pairs will be added. The effect of this optimization is that the timestamp labels corresponding to the statically introduced data dependence are eliminated, while the labels for the dynamically introduced data-dependence edge are not, i.e., labels have been *partially eliminated*.

During traversal, first the labels on dynamic edges are examined to locate the relevant dependence. If the relevant dependence is not found, then it must

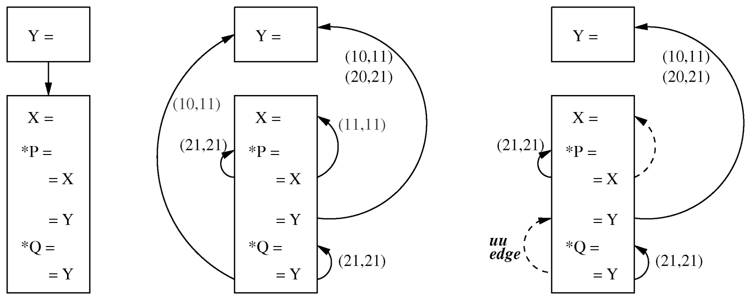


Fig. 5. Effect of applying OPT-1b.

be the case that the dependence involved corresponds to the static edge, which can then be traversed. It should also be clear that greater benefits will result from this optimization if the edge being converted to an unlabeled edge is the more *frequently exercised* dependence edge. Thus, if *profile data* is available, we can make use of it in applying this optimization.

In the example shown in Figure 5, let us assume that $*P$ is a may alias of X and $*Q$ is a may alias of Y. Further assume that the code fragment is executed twice, resulting in the introduction of the following labeled dynamic edges: between the uses of X and definitions of X and $*P$ and between the uses of Y and the definitions of Y and $*Q$. We introduce the following static unlabeled edges: from the use of X to the definition of X (as in OPT-1a) and, later, the use of Y to the earlier use of Y (as in OPT-2a described later). The dynamic edges introduced are from the use of X to the definition of $*P$ and from the later use of Y to the definition of $*Q$. Thus some, but not all, labels have been removed.

3.5 (OPT-2) Transformation

3.5.1 *(OPT-2a) Transform Non-Local Def-Use to Local Use-Use.* Consider two uses u_1 and u_2 such that u_1 and u_2 are *local* to the same basic block, u_1 and u_2 always refer to the same location during any execution of the basic block, and there is no definition between u_1 and u_2 that can cause the uses to see different values. Now let us assume that a non-local definition d reaches the uses u_1 and u_2. In this case, each time u_1 and u_2 are executed, two non-local def-use edges $du_1(t_d, t_{u_1})$ and $du_2(t_d, t_{u_2})$ are introduced. Let u_1 appear before u_2. We can replace the non-local def-use edge $du_2(t_d, t_{u_2})$ by a local use-use edge u_1u_2. The latter does not require a timestamp label because t_{u_1} is always equal to t_{u_2}. By replacing a non-local def-use edge by a local use-use edge, labels on the edge are eliminated. During slicing, an extra edge (the use-use edge) will be traversed. Moreover, use-use edges are treated differently. In particular, a statement visited by traversing a use-use edge is not included in the dynamic slice.

Using static analysis, we can identify uses local to basic blocks, which always share the same reaching definition. Once having identified these uses, we statically introduce use-use edges from later uses to the earliest use in the basic blocks. After having introduced these edges, there will not be any need to collect or introduce any dynamic information corresponding to the later uses.

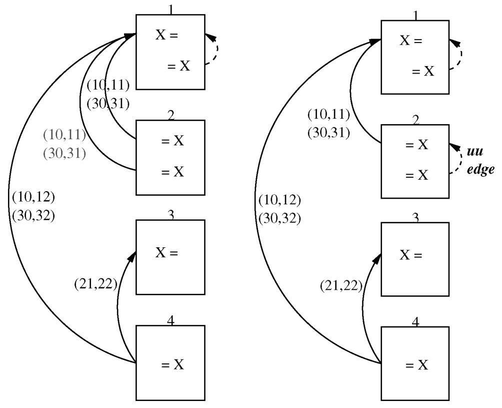
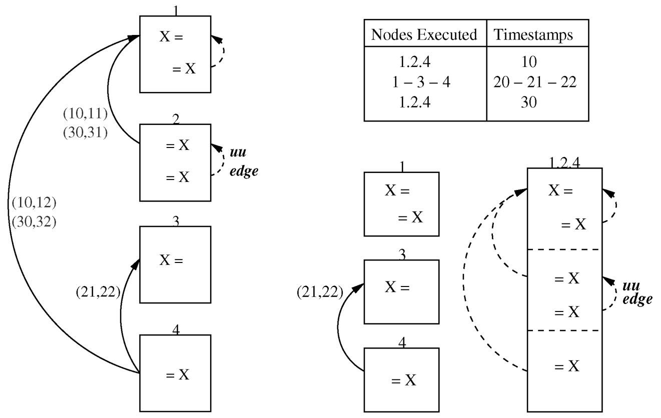


Fig. 6. Effect of applying OPT-2a.

The impact of this optimization is illustrated by further optimizing the dependence graph obtained by applying OPT-1a. As shown in Figure 6, basic block 2 contains a two uses of X each having the same reaching definition from block 1. The labeled non-local def-use edge from the second use to the definition is replaced by an unlabeled static use-use edge by this optimization. We draw use-use edge using a dashed edge.

3.5.2 *(OPT-2b) Transform Non-Local Def-Use to Local Def-Use.* Let us assume that there is a non-local def-use edge $du(t_d, t_u)$ between basic blocks b_d and b_u. Moreover, this edge is always exercised whenever a *path* p, which contains b_d and b_u, is executed. We can convert this non-local dynamic edge into a local dynamic edge $du(t'_d, t'_u)$ by creating a specialized node for p. While for the original edge $du(t_d, t_u)$, the values of t_d and t_u are not equal. For the modified edge $du(t'_d, t'_u)$, the values of t'_d and t'_u are equal. At runtime, if the dependence between d and u is established along path p, then that dependence would be represented by an unlabeled edge local to node for path p. However, if the dependence is established along some path other than p, it is represented using a labeled non-local edge between b_d and b_u.

The consequence of earlier optimizations was that the initial graph that we start out with contains some statically introduced data-dependence edges. The consequence of this optimization is that instead of starting out with a graph that contains only basic block nodes, we start out with a graph that contains additional nodes corresponding to paths that have been specialized. During execution we must detect when specialized paths are executed. This is necessary for construction of the Dynamic Dependence Graph because of the following reasons. The value of global timestamps must be incremented after the execution of code corresponding to a node in the graph. Thus, we no longer will increment the timestamp each time a basic block is executed because nodes representing specialized paths contain multiple basic blocks. At runtime we must distinguish between executions of a block that correspond to its appearance in a specialized path from the rest of its executions so that when we introduce a dynamic data-dependence edge in the graph we know which copy of the block to consider.



Nodes Executed	Timestamps
1.2.4	10
1 – 3 – 4	20 – 21 – 22
1.2.4	30

Fig. 7. Effect of applying OPT-2b.

The impact of this optimization is illustrated by further optimizing the optimized dependence graph from Figure 6. As shown in Figure 7, if we create a specialized node for path along basic blocks 1, 2, and 4, many of the previously dynamic non-local def-use edges are converted to dynamic local def-use edges within this path. The def-use edges established along this path can now be statically introduced within the statically created node representing this path. Thus, the timestamp labels for these def-use edges are no longer required. Since block 2 can only be executed when path 1-2-4 is executed, we do not need to maintain a separate node for 2 once node for path 1-2-4 has been created. However, the same is not true for blocks 1 and 4. Therefore, we continue to maintain nodes representing them to capture dynamic dependences that are exercised when path 1-2-4 is not followed.

After applying multiple optimizations to the dependence graph of Figure 4, we have eliminated all but one of the labels in the dependence graph. In fact this label can also be eliminated by creating another specialized node for path containing blocks 3 and 4.

Finally it should be noted that the above optimization only eliminates labels corresponding to dependence instances exercised along the path for which a specialized node is created. Thus, greater benefits will be derived if the path specialized is a *frequently executed path*. As a result, selection of paths for specialization can be based upon *profile data*.

3.6 (OPT-3) Redundancy Across Non-Local Def-Use Edges

In all the optimizations considered so far, we have identified and created situations in which the labels were guaranteed to have a pair of identical timestamps. Now we present an optimization that identifies pairs of dynamic edges between different statements that are guaranteed to have identical labels in all executions. Thus, the statements can be clustered so that they can share the same edge and thus a single copy of the list of labels. Given basic blocks b_d and b_u such that definitions d_1 and d_2 in b_d have corresponding uses u_1 and u_2 in b_u. If it is guaranteed that along every path from b_d to b_u either both d_1 and d_2 will

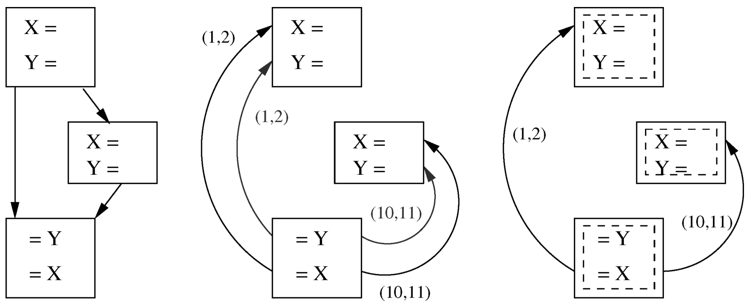


Fig. 8. Effect of applying OPT-3.

reach u_1 and u_2 or neither d_1 nor d_2 will reach u_1 and u_2, then the labels on the def-use edges d_1u_1 and d_2u_2 will always be identical. The example in Figure 8 shows that the uses of Y and X always get their definitions from the same block and thus dependence edges for Y and X can share the labels. A shared edge between clusters of statements (shown by dashed boxes) is introduced by this optimization.

4. STREAM COMPRESSION

For the next step in compression we view the information labeling the WET as consisting of streams of values arising from the following sources: a sequence of $< t, v >$ pairs labeling a node gives rise to two streams, one corresponding to the timestamps (t's) and the other corresponding to the values (v's): The sequence of $< t_{s_2}, t_{s_1} >$ pairs labeling a dependence edge also gives rise to two streams, one corresponding to the first timestamps (t_{s_2}'s) and the other corresponding to the second timestamps (t_{s_1}'s). Each of the above streams are compressed using the same algorithm that is developed in this section.

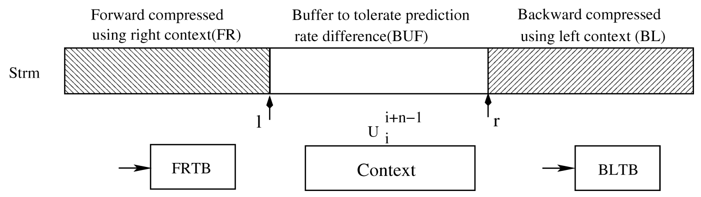
The stream compression algorithm should be designed such that the compressed stream of values can be rapidly traversed. An analysis algorithm using the WET representation may traverse the program representation in forward or backward direction (recall that is why all edges in WET are bidirectional). Thus, during a traversal, it is expected that the profile information and, hence, the values in above streams, will be inspected one after another either in forward or backward direction. Unfortunately most of the existing algorithms for effectively compressing streams are *unidirectional*, i.e., the compressed stream can be uncompressed only in one direction, typically starting from the first value and going toward the last. Examples of such algorithms include compression algorithms designed from value predictors which were used for compressing value and address traces in Burtscher and Jeeradit [2003]. The problem with using a *unidirectional* predictor is that while it is easy to traverse the value stream in the direction corresponding to the order in which values were compressed, traversing the stream in the reverse direction is expensive. The only way to efficiently traverse the streams freely in both directions is to uncompress them first, which is clearly undesirable. Sequitur [Nevil-Manning and Witten 1997] which was used for compressing control flow traces in Larus [1999] and address traces in Chilimbi [2001] yields a representation that can be traversed in both directions. However, it is well known that Sequitur is not nearly as effective as the above unidirectional predictors when compressing value streams [Burtscher and Jeeradit 2003].

To overcome the above problem with existing compression algorithms, we have developed a novel approach to constructing *bidirectional* compression algorithms. The approach can be used to convert an *unidirectional* value predictor-based compression algorithm [Burtscher and Jeeradit 2003] into a *bidirectional* one. Let us consider the highly effective FCM predictor [Sazeides and Smith 1997a, 1997b]. A unidirectional FCM predictor compresses a stream in the *forward direction* such that a value is successfully compressed if it can be correctly predicted from its *left context* (i.e., pattern of immediately preceding n values); otherwise the value is saved in uncompressed form. A look-up table *TB* is maintained to store predictions corresponding to a limited number of left-context patterns encountered in the past. The index of the entry at which the prediction for a pattern is stored is derived by hashing the pattern into a number.

If a value is correctly predicted by the look-up table *TB* using the left context, a bit 1 is created in the compressed stream. If a prediction v provided by the look-up table *TB* using the left context does not match the value v' being compressed, then a bit sequence of $< \overline{v} \cdot 0 >$ is created in the compressed stream while the look-up table *TB* is updated using v' to enable future predictions. Here $\overline{v}$ denotes the bits for v. Clearly for a stream compressed in the above fashion, only forward traversal is possible.

4.1 Bidirectional Compression Derived from the FCM Predictor

Now let us look at the design of a bidirectional predictor. In particular, we look at a bidirectional counterpart of the FCM predictor [Sazeides and Smith 1997a, 1997b]. A bidirectional differential FCM predictor [Goeman et al. 2001] can be constructed in a similar way. Normal FCM is *forward* compressed and then *forward* traversed. If we change the direction of table lookup from using *left* context to using *right* context, which means we use future values to predict the current value instead of using previous values to predict next value, we can get a *forward* compressed and *backward* traversed FCM. Similarly, we can construct a *backward* compressed and *forward* traversed FCM. By using these two FCMs back to back, we get a bidirectional FCM (BFCM).


Forward compressed using right context(FR)
Buffer to tolerate prediction rate difference(BUF)
Backward compressed using left context (BL)
Strm
l
U_i^{i+n-1}
r
FRTB
Context
BLTB

Before we present the algorithms for bidirectional traversal of the value stream, we introduce the notation we use. Let m be the *length* of the uncompressed value stream, n be the *context size*, a BFCM can be viewed as a tuple of $< Strm, FRTB, BLTB, i, l, r, Context >$ where:

- *Strm* is the compressed bit stream composed of two substreams: *FR* and *BL*. *FR* is obtained by compressing values at positions 1 through $(i-1)$ in *forward direction* (F) using *right context* (R). *BL* is obtained by compressing values at positions $(i+n)$ through $(m-1)$ in the *backward direction* (B) using the *left context* (L).
- *Context* is a buffer that contains the current context of n uncompressed values from position i to position $(i+n-1)$.
- *FRTB* is the look-up table for *FR*, while *BLTB* is the look-up table for *BL*.
- Finally, l is the end bit position in *Strm* of *FR*, while r is the starting bit position of *BL* in *Strm*. The reason for providing extra bits (*BUF*) between positions l and r will be discussed in greater detail later—essentially these bits provide extra space needed to accommodate the differences between forward and backward compression rates.

There are four types of basic operations for a BFCM on which the traversal operations are built. *FORWARD_COMPRESS* compresses a value v into *Strm* starting at bit position l using *FRTB*. Parameter *Context* is the right context for v. The difference between this operation and the forward compressing operation in a conventional FCM is that *FORWARD_COMPRESS* uses the *right* context instead of *left* context. Using *right* context to compress *forward* provides the capability to uncompress in the *backward* direction. *BACKWARD_UNCOMPRESS* consumes bits in the backward direction starting at l, which were generated earlier by *FORWARD_COMPRESS* operation, to uncompress the value to the left of the current context. The other two operations, *FORWARD_UNCOMPRESS* and *BACKWARD_COMPRESS* can be constructed in a similar way. The details of all four operations are given in Figure 9.

To traverse one step forward, BFCM first forward uncompresses the value to the right of *Context*, U_{i+n}, by looking at the bits starting at $Strm_r$ and then shifts *Context* one step forward and uses the new *Context* to forward compress the value to the left. Backward traversal can also be similarly defined. The implementation of the traversal operations in terms of the four basic operations is given in Figure 10. Note that we are assuming a 32-bits machine is used. Hence if a value is predicted, it consumes one bit space; if not, it consumes $32+1$ bits of space.

The example in Figure 11 illustrates the above algorithm. The first figure shows a portion of the uncompressed stream while the second figure shows the state of the stream and look-up tables corresponding to four consecutive positions of the context, which consists of three uncompressed values. No matter whether the stream is traversed forward or backward, the sequence of states encountered is the same.

4.2 Accounting for Difference in Forward and Backward Compression Rates

One implementation problem arises due to different prediction rates of the two FCMs. As a result, the amount of space needed to store the stream will vary at different points of the traversal. To handle this problem, our goal is to allocate enough extra space so that at any point during traversal there is enough

```
Basic Operations
$FORWARD\_COMPRESS(v, Strm, l, FRTB, Context)$
(1)  $index = hash(Context)$
(2)  if $FRTB[index] = v$ then
(3)      $Strm_{l \ldots l+1} =< 1 >$
(4)      $l = l + 1$
(5)  else
(6)      $Strm_{l \ldots l+33} =< \overline{v} \cdot 0 >$
(7)      $l = l + 33$
(8)      $FRTB\,[index] = v$
(9)  endif

$FORWARD\_UNCOMPRESS(Strm, r, BLTB, Context)$
(1)  $b = Strm_r$
(2)  $r = r + 1$
(3)  $index = hash(Context)$
(4)  if $b = 1$ then
(5)      $v = BLTB[index]$
(6)  else
(7)      $v = Strm_{r \ldots r+32}$
(8)      $r = r + 32$
(9)      $BLTB[index] = v$
(10) endif
(11) return $v$

$BACKWARD\_COMPRESS(v, Strm, r, BLTB, Context)$
(1)  $index = hash(Context)$
(2)  if $BLTB[index] = v$ then
(3)      $Strm_{r-1 \ldots r} =< 1 >$
(4)      $r = r - 1$
(5)  else
(6)      $Strm_{r-33 \ldots r} =< 0 \cdot \overline{v} >$
(7)      $r = r - 33$
(8)      $BLTB[index] = v$
(9)  endif

$BACKWARD\_UNCOMPRESS(Strm, l, FRTB, Context)$
(1)  $b = Strm_l$
(2)  $l = l - 1$
(3)  $index = hash(Context)$
(4)  if $b = 1$ then
(5)      $v = FRTB\,[index]$
(6)  else
(7)      $v = Strm_{l-32 \ldots l}$
(8)      $l = l - 32$
(9)      $FRTB\,[index] = v$
(10) endif
(11) return $v$
```

Fig. 9. Four basic operations used by BFCM.

```
Traverse
– $< Strm, FRTB, BLTB, Context, i, l, r >$ is the bit stream to traverse;
$STEP\_FORWARD(Strm, FRTB, BLTB, Context, i, l, r)$
(1)   $v = FORWARD\_UNCOMPRESS(Strm, r, BLTB, Context)$
(2)   $t = Context[0]$
(3)   $Context = Context[1...n-1] \cdot v$
(4)   $FORWARD\_COMPRESS(t, Strm, l, FRTB, Context)$
(5)   $i = i + 1$
(6)   return $v$

$STEP\_BACKWARD(Strm, FRTB, BLTB, Context, i, l, r)$
(1)   $v = BACKWARD\_UNCOMPRESS(Strm, l, FRTB, Context)$
(2)   $t = Context[n-1]$
(3)   $Context = v \cdot Context[0...n-2]$
(4)   $BACKWARD\_COMPRESS(t, Strm, r, BLTB, Context)$
(5)   $i = i - 1$
(6)   return $v$
```

Fig. 10. Forward and backward traversal by a single step.

space available to handle the stream. The space allocation is performed in a manner that at any point in time the context (uncompressed values) are held in the *Context* buffer while all other values (forward and backward compressed values) are kept in *Strm* storage. The space allocated between l and r in *Strm* (referred to as *BUF*) accommodates the difference between forward and backward compression rates. For example, when we move forward or backward by one step, it is possible that the value that is uncompressed frees up one bit (i.e., the value had been compressed to one bit) while the value that is compressed requires 33 bits (i.e., the value cannot be successfully compressed). The additional bits allocated accommodate these extra bits. In fact, we compute the *BUF* size to be such that whenever extra space is needed it is available.

To ensure that there is sufficient extra space allocated in *BUF* so that forward and backward traversals never cause the compressed stream size to overflow the allocated space, we use the algorithm described in Figure 12. This algorithm first *forward* compresses all the values into a temporary bit stream *Tmp*. However, *Tmp* is not yet bidirectional traversable. A backward traversal is performed to determine the amount of additional space that needs to be allocated. Lines 13 to 17 in the algorithm preallocate extra space. Condition $r < l + 33$ being true means that the next *BACKWARD_COMPRESS* operation may overwrite the bits generated by *FORWARD_COMPRESS* operation previously. In other words, both the FCMs encounter low prediction rate and then the allocated space may not be sufficient. In this case, BFCM inserts some buffer space between *FR* and *BL*. After *backward* traversing *Tmp* once with allocating buffer space to tolerate different prediction rates, the BFCM $< Strm, FRTB, BLTB, Context, 0, l, r >$ is ready to be used for bidirectional traversal.

4.3 Bidirectional Compression Derived from a Last n Predictor

Another predictor which has been used for unidirectional compression is the last n predictor [Lipasti and Shen 1996; Burtscher and Zorn 1999]. We also derived a bidirectional compression algorithm using the last n predictor. This

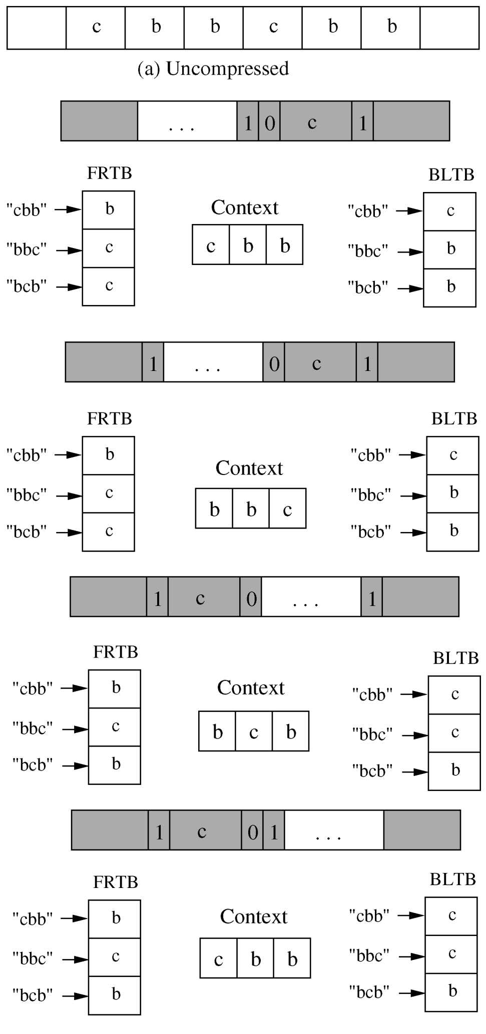


Fig. 11. Example of bidirectional FCM compression.

is because studies have shown that while overall performance of both FCM and Last *n* predictors is quite good, there are also specific situations where one predictor works well while the other does not and vice versa [Burtscher and Jeeradit 2003]. The full details of bidirectional compression algorithm based upon last *n* predictor are omitted due to space limitations. However, the main cases of forward compression of a value are summarized in Figure 13. Backward compression is similar. Unlike the bidirectional FCM predictor only a single look-up table *TB* is used for both forward and backward compression.

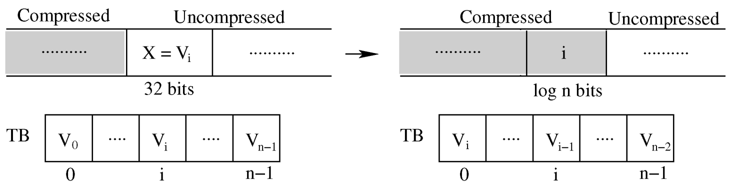
4.4 Selection

For each stream, we selected from one of several bidirectional versions of compression methods. Initially we use all methods to compress the stream. After

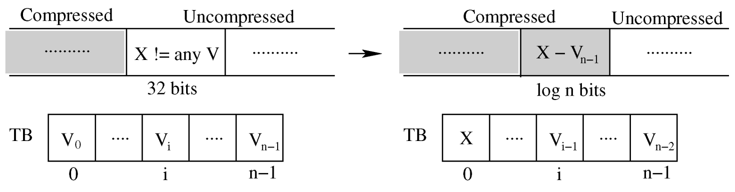
Compress value stream

```
– $Vals$ is the uncompressed stream;
– $Tmp$ is a bit stream with $INFINITE$ length;
$COMPRESS(Vals, vLen)$
(1)  $Context = Vals[0...n-1]$
(2)  $sLen = 0$
(3)  for $i = 0$ to $vLen - n - 1$
(4)     $Context = Context[1...n-1] \cdot Vals[n+i]$
(5)     $FORWARD\_COMPRESS(Vals[i], Tmp, sLen, FRTB, Context)$
(6)  endfor
(7)  $r = sLen$
(8)  $l = sLen$
(9)  for $i = vLen - n - 1$ to $0$
(10)    $v = BACKWARD\_UNCOMPRESS(Tmp, l, FRTB, Context)$
(11)    $v_{com} = Context[n-1]$
(12)    $Context = v \cdot Context[0...n-2]$
(13)    if $r < l + 33$ then
(14)       $Tmp_{(r+EXTRA)...(sLen+EXTRA)} = Tmp_{r...sLen}$
(15)       $r = r + EXTRA$
(16)       $sLen = sLen + EXTRA$
(17)    endif
(18)    $BACKWARD\_COMPRESS(v_{com}, Tmp, r, BLTB, Context)$
(19) endfor
(20) $Strm = Tmp_{0...sLen}$
(21) return $< Strm, FRTB, BLTB, Context, 0, l, r >$
```

Fig. 12. Preparing Strm for bidirectional traversal.



(a) Successful compression of ith value (forward direction)



(b) Failed compression of ith value (forward direction)

Fig. 13. Bidirectional Last n compression.

a certain number of instances, we pick the method that performs the best up to that point. We implemented the FCM, differential FCM (this is an adaptation of FCM that works on strides [Goeman et al. 2001]), last n, and last n stride methods. For each type we created three versions with differing context size.

Table I. WET Sizes

Benchmark	Input	Stmts Executed (Millions)	Orig. WET (MB)	Comp. WET (MB)	Orig./ Comp.
099.go	training	685.28	10369.32	574.65	18.04
126.gcc	ref/insn-emit.i	364.80	5237.89	89.03	58.84
130.li	ref	739.84	10399.06	203.01	51.22
164.gzip	training	650.46	9687.88	537.72	18.02
181.mcf	testing	715.16	10541.86	416.21	25.33
197.parser	training	615.49	8729.88	188.39	46.34
255.vortex	training/lendian	609.45	8747.64	104.59	83.63
256.bzip2	training	751.26	11921.19	220.70	54.02
300.twolf	training	690.39	10666.19	646.93	16.49
Avg.	n/a	646.90	9588.99	331.25	41.33

5. EXPERIMENTAL RESULTS

We have implemented the WET construction and compression techniques presented in this paper. In addition, we have also developed implementations of several queries for subsets of profile information that were described in Section 2. To carry out this work, we used the Trimaran [Trimaran 1997] compiler infrastructure to profile several benchmarks from the `SpecInt` 2000 and 1995 suites. We used `SpecInt` 1995 versions of some programs because the version of Trimaran we used did not compile the `SpecInt` 2000 versions of these programs. The statements correspond to Trimaran's intermediate level statements. The program is executed on the simulator, which avoids introduction of intrusion as no instrumentation is needed. We do not count pseudo statements and we do not maintain result values for intermediate statements that do not have a *def port* (e.g., stores and branches). In our implementation, for labeling dependences, instead of using global timestamps to identify statement instances, we use local timestamps for each statement because this approach yields greater levels of compression. These experiments were carried out on a Pentium IV 2.4 GHz machine with 2 gigabyte RAM and 120 gigabyte hard disk. Our evaluation focuses on two main aspects of WETs. First, we evaluate the practicality of WETs by considering the sizes of WETs in relation to the length of the program execution. We also examine in detail the effectiveness of the two-tier compression strategy. Second, we evaluate the speeds with which queries requesting subsets of profile information can extract the information from a compressed WET.

5.1 WET Sizes

Table I lists the benchmarks considered and the lengths of the program runs which vary from 365 and 751 million intermediate-level statements. WETs could not be collected for complete runs for most benchmarks, although we tried using Trimaran-provided inputs with shorter runs. The effect of our two-tier compression strategy is summarized in Table I. While the average size of the original uncompressed WETs (Orig. WET) is 9589 megabytes, after compression their size (Comp. WET) is reduced to 331 megabytes, which represents a compression ratio (Orig./Comp.) of 41. Therefore, on average, our approach

Table II. Effect of Compression on Node Labels

	t_s Labels			val_s Labels		
Benchmark	Orig. (MB)	Orig./ Tier-1	Orig./ Tier-2	Orig. (MB)	Orig./ Tier-1	Orig./ Tier-2
099.go	2614.12	37.96	47.13	1847.09	2.48	6.33
126.gcc	1391.60	50.06	126.63	945.03	3.15	17.62
130.li	2822.26	32.47	105.88	1894.48	3.83	17.33
164.gzip	2481.32	30.33	152.76	1733.13	1.66	4.02
181.mcf	2728.12	22.12	127.09	1875.21	2.37	7.02
197.parser	2347.92	30.61	101.82	1615.57	2.05	12.45
255.vortex	2324.87	53.51	176.55	1641.31	3.51	23.82
256.bzip2	2865.81	55.24	1171.6	2154.85	2.46	10.61
300.twolf	2633.64	27.36	69.49	1873.52	2.13	4.36
Avg.	2467.74	37.74	230.99	1731.13	2.63	11.51

Table III. Effect of Compression on Edge Labels

	Edge Labels		
Benchmark	Orig. (MB)	Orig./ Tier-1	Orig./ Tier-2
099.go	5908.12	9.00	26.00
126.gcc	2901.26	15.37	118.94
130.li	5682.32	11.36	84.74
164.gzip	5473.42	10.13	60.37
181.mcf	5938.54	7.62	46.56
197.parser	4766.38	15.57	133.92
255.vortex	4781.46	21.75	212.35
256.bzip2	6900.52	32.06	455.44
300.twolf	6159.03	7.05	34.43
Avg.	5390.12	14.43	130.31

enables saving of the whole execution trace corresponding to a program run of 647 million intermediate-level statements using 331 megabytes of storage.

Now let us examine the effectiveness of our two-tier compression strategy in detail. Table II shows the sizes of node labels, timestamp, and value sequences, before and after compression, while Table III presents the same information for edge labels.

The above results show that while the average compression ratios of all the *timestamp sequences* are very high (231 for nodes and 130 for edges), the same is not true for *value sequences* that label the nodes (compression ratio for these is only 11.5). Compression of values is much harder although our value compression algorithm is aggressive, the compression ratios for value sequences are modest in comparison to those for timestamp sequences. We would like to point out that the optimizations described to reduce dynamic dependence history size (i.e., edge labels of timestamp pairs) were very effective. Only 6% of the dynamic dependences are explicitly maintained after the proposed optimizations are applied.

In Figure 14, the relative sizes of the three main components of profile data (node timestamp sequences, node value timestamp sequences, and edge timestamp sequences) are shown before compression (Original), after first-tier

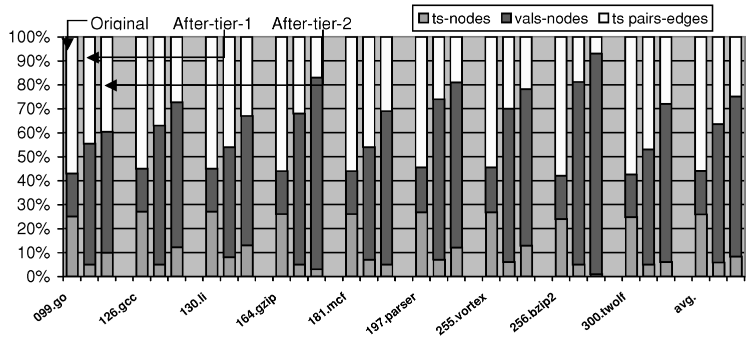


Fig. 14. Relative sizes of WET components.

Table IV. Architecture-Specific Information

Benchmark	Space (MB)		
	Branch	Load	Store
099.go	9.68	12.17	5.87
126.gcc	5.28	6.11	4.09
130.li	11.36	11.16	8.08
164.gzip	9.64	10.95	5.87
181.mcf	12.88	11.93	3.32
197.parser	9.24	10.67	6.39
255.vortex	7.24	14.79	10.49
256.bzip2	10.08	14.51	3.26
300.twolf	9.76	13.38	4.94
Avg.	9.46	11.74	5.81

compression (After Tier-1), and after second-tier compression (After Tier-2). As we can see, the contribution of value sequences to the total size increases in percentage following each compression step, since the degree of compression achieved for value sequences is lower.

Next we show that WETs can be augmented with significant amounts of architecture-specific information with a modest increase in WET sizes. In particular, lets consider the augmentation of WETs with branch misprediction, load miss, and store miss information. For a single execution of a branch, load, or store the history of misprediction or cache miss can be remembered using one bit. Table IV shows the additional storage taken by the above uncompressed execution histories. Clearly the amount of additional storage needed is quite small.

Next we study the *scalability* of our approach so that we can estimate the limit on the length of a program run for which we can realistically keep the whole execution trace in memory. For this purpose, we study the impact of trace length on the compression ratios. In Figure 15, we divided the executions into ten intervals for each benchmark (x axis) and then measured the compression ratios (y axis) up to each interval. From the results in Figure 15, we notice that for 7 out of 9 programs, the compression ratios either improve or roughly remain the same as the length of the run increases. For benchmark *256.bzip2*,

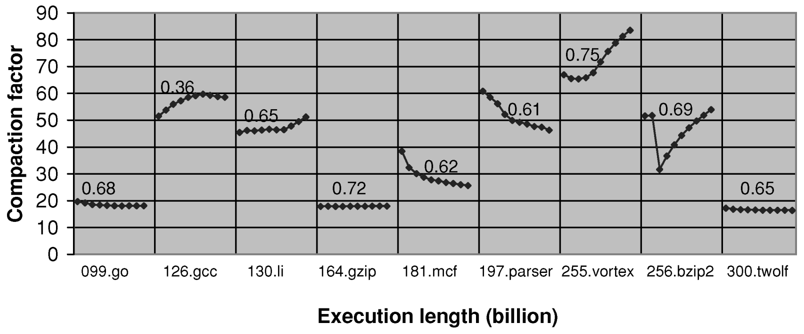


Fig. 15. Scalability of compression ratio.

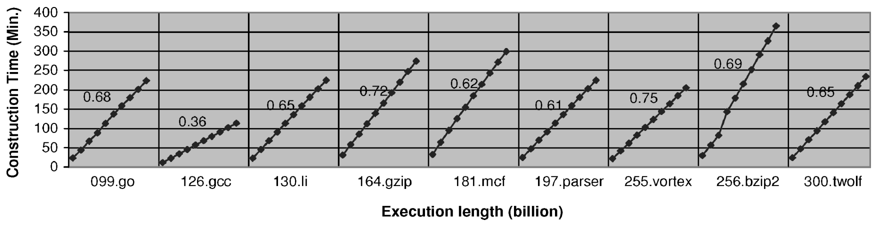


Fig. 16. WET construction times.

we observe a sharp decrease of the compress ratio from the second to the third interval. We believe that is due to the switch of program phases. The new phase is substantially more difficult to compress compared to the previous one. As this phase finishes, its effect gradually fades out and the compression ratio gradually recovers.

Let us assume that the compression ratio remains constant across the length of a program run. Further recall that our earlier experiments show that the compressed WET for execution of 647 million Trimaran intermediate-level code statements took approximately 331 megabytes of storage. Therefore, we can extrapolate that the WET corresponding to a program run involving execution of 3.9 billion Trimaran intermediate-level code statements consumes 2 gigabyte of space, which is the normal RAM size for a workstation. It is a fairly long trace and thus can be used effectively to study program behavior when designing compilers and architectures.

5.2 WET Construction Times and Response Times for Queries

The times taken to construct the compressed WETs for the program runs are presented in Figure 16. Similar to the prior experiment, we divided the executions into ten equal-length intervals and then collected the cumulative construction times up to each interval. We can see that it takes 200–300 min to construct the WETs for most of the runs, depending on the execution lengths. We can also observe that the construction time increases almost linearly with the execution length.

In the preceding section, we studied the effectiveness of our compression strategy and the scalability of our compressed WET representation. Recall that

Table V. Response Times for Control Flow Traces

		Forward			
		Tier-1		Tier-2	
Benchmark	CF Trace (MB)	Time (sec.)	MB/sec	Time (sec.)	MB/sec
099.go	2614.12	513.97	5.09	536.57	4.87
126.gcc	1391.60	11.47	121.33	13.11	106.15
130.li	2822.26	14.05	200.87	17.06	165.43
164.gzip	2481.32	16.08	154.31	19.06	130.18
181.mcf	2728.12	17.71	154.04	21.15	128.99
197.parser	2347.92	13.81	170.02	15.66	149.93
255.vortex	2324.87	21.81	106.60	25.11	92.59
256.bzip2	2865.81	7.90	362.76	9.41	304.55
300.twolf	2633.64	45.22	58.24	47.53	55.41
Avg.	2467.74	73.56	148.14	78.30	126.46

	Backward			
	Tier-1		Tier-2	
Benchmark	Time (sec.)	MB/sec	Time (sec.)	MB/sec
099.go	471.27	5.55	491.21	5.32
126.gcc	11.12	125.14	12.77	108.97
130.li	14.62	193.04	16.02	176.17
164.gzip	15.25	162.71	18.02	137.70
181.mcf	16.73	163.07	22.27	122.50
197.parser	13.83	169.77	16.23	144.67
255.vortex	21.64	107.43	24.59	94.55
256.bzip2	7.43	385.71	9.11	314.58
300.twolf	39.04	67.46	41.26	63.83
Avg.	67.88	153.32	72.39	129.81

the WET representation and compression techniques were designed so as to allow access to related profile information with ease and, hence, with good speed. Next, we study the *response times* to various queries for profile information. The response times are provided by using the WET after only first- and second-tier compression, i.e., after full compression.

5.2.1 *Query for Control Flow Trace.* Let us consider the request for the control flow trace. Such a request can be made with respect to any point either along the execution flow (forward) or in the reverse direction (backward). The rates at which control flow trace can be extracted from WET in either direction are given in Table V. The total control flow trace size, the time to extract this entire trace, and the rate at which it is extracted are given. We can see that, on average, the entire control flow trace can be extracted in roughly 75 sec in either direction. The response times after tier-1 and tier-2 compression are very close and so are the times in forward and backward direction. This indicates that the bidirectional compression algorithm that we use is very effective for node timestamp labels.

5.2.2 *Query per Instruction Load Value Traces.* Let us consider requests for load values on per instruction basis. Such traces can be useful for designing load value predictors. In Table VI, the size of complete load value trace, response

Table VI. Response Times per Instruction Load Value Traces

Benchmark	Ld Value Trace (MB)	After Tier-1		After Tier-2	
		sec.	MB/sec	sec.	MB/sec
099.go	354.86	775.36	0.46	1105.70	0.32
126.gcc	203.11	73.82	2.75	89.10	2.28
130.li	340.95	52.22	6.53	192.20	1.77
164.gzip	367.49	143.09	2.57	4568.65	0.08
181.mcf	362.70	229.22	1.58	6911.11	0.05
197.parser	354.86	80.80	4.39	191.08	1.86
255.vortex	450.03	36.81	12.23	178.26	2.52
256.bzip2	439.91	44.39	9.91	79.32	5.55
300.twolf	405.95	372.04	1.09	1537.15	0.26
Avg.	364.43	200.86	4.61	1650.29	1.63

Table VII. Response Times per Instruction Load/Store Address Traces

Benchmark	Address Trace (MB)	After Tier-1		After Tier-2	
		sec.	MB/sec	sec.	MB/sec
099.go	549.35	1410.27	0.39	2006.47	0.27
126.gcc	335.52	122.54	2.74	149.89	2.24
130.li	572.16	79.90	7.16	588.00	0.97
164.gzip	564.38	179.73	3.14	2790.63	0.20
181.mcf	463.98	270.11	1.72	7004.47	0.07
197.parser	557.64	110.00	5.07	354.84	1.57
255.vortex	761.67	59.87	12.72	341.57	2.23
256.bzip2	549.40	54.23	10.13	139.68	3.93
300.twolf	554.15	512.49	1.08	1590.79	0.35
Avg.	545.36	311.02	4.91	1662.93	1.32

time for extracting this trace, and the rates at which it is extracted after tier-1 and after tier-2 compression are shown. The average times to extract the entire load value trace after tier-1 and tier-2 compression are a little over 200 and 1650 sec, respectively. Since the values are not compressed as effectively as timestamps, as expected, there is a notable increase in response time as we go from using tier-1 to tier-2 compressed representation.

5.2.3 *Query per Instruction Load/Store Address Traces.*

Let us consider requests for load and store address traces on a per-instruction basis. Such traces can be useful for designing address predictors for data prefetch mechanisms. In Table VII, the size of complete address trace, response time for extracting this trace, and the rates at which it is extracted after tier-1 and after tier-2 compression are shown. The average time to extract the entire address trace after tier-1 and tier-2 compression are little over 311 and 1662 sec, respectively. Since addresses are simply part of values in WET representation and values are not compressed as effectively as timestamps, there is a notable increase in response time as we go from using tier-1 to tier-2 compressed representation.

The results in this section have shown the versatility of the compressed WET representation in quickly responding to queries with a wide range of characteristics: those that require traversal of the graph (control flow traces), those that are instruction specific, requiring only the information labeling a node (load value traces), and those that are instruction specific but also require limited traversal (load/store address traces).

6. APPLICATIONS OF WETS

In this section we present two important applications that require multiple types of profiles: dynamic program slicing and version matching. We demonstrate that using WETs we are able to address the goals of these significant applications in an efficient and effective manner.

6.1 WET-Based Dynamic Program Slicing

A static program slice of a variable v at a program point p represents the set of statements that can contribute to the computation of the variable's value at p under some program execution. The notion of static program slicing was first proposed by Mark Weiser [Weiser 1982] as a debugging aid. He gave the first *static slicing* algorithm. For programs that make extensive use of pointers, the highly conservative nature of static data-dependency analysis leads to highly imprecise and considerably larger program slices. Since the main purpose of slicing is to identify the subset of program statements that are of interest for a given application, conservatively computed large slices are undesirable. Recognizing the need for accurate slicing, Korel and Laski proposed the idea of *dynamic slicing* [Korel and Laski 1988]. The dependences that are exercised during a program execution are identified and a precise dynamic dependence graph is constructed. Dynamic program slice of a variable is computed by traversing the dynamic-dependence graph and computing the transitive closure over data and control dependences, starting at the definition of variable at point of interest. It has been shown that the dynamic slices can be considerably smaller than static slices [Venkatesh 1995].

The importance dynamic slicing extends well beyond debugging of programs [Agrawal et al. 1993; Korel and Rilling 1997]. Increasingly applications aimed at improving software quality, reliability, security, and performance are being identified as candidates for making automated use of dynamic slicing. Examples of these applications include: detecting spyware that has been installed on systems without the user's knowledge, carrying out dependence-based software testing [Duesterwald et al. 1992; Kamkar 1993], measuring module cohesion for purpose of code restructuring [Gupta and Rao 2001], and guiding the development of performance enhancing transformations, based upon estimation of criticality of instructions [Zilles and Sohi 2000] and identification of instruction isomorphism [Sazeides 2003].

While the notion of dynamic slicing is very useful for the above-mentioned applications [Agrawal et al. 1993; Duesterwald et al. 1992; Kamkar 1993; Gupta and Rao 2001], an impediment to their widespread use in practice has been the high cost of computing them. As we have already seen, the sizes of dynamic-dependence graphs can be very large and thus it is not possible to keep them in memory for realistic program runs. To address this problem, we recently proposed the LP algorithm in Zhang et al. [2003], where the dynamic-dependence

Table VIII. WET Slices[a]

Benchmark	Stmts Executed (Millions)	Tier-1 (sec.)	Tier-2 (sec.)	Tier-2/ Tier-1
099.go	132.52	58.31	412.44	7.07
126.gcc	139.46	10.91	17.74	1.63
130.li	126.78	10.00	121.42	12.14
164.gzip	123.06	4.20	102.33	24.34
181.mcf	137.31	17.47	76.07	4.35
197.parser	122.12	1.55	4.69	3.02
255.vortex	119.50	4.75	18.09	3.81
256.bzip2	128.25	2.76	3.90	1.42
300.twolf	114.85	19.10	62.15	3.25
Avg.	127.09	14.34	90.98	6.78

[a]Avg. over 25 slices.

graph is constructed demand in response to dynamic slicing requests from the execution trace that is saved on disk. While this approach greatly reduces the size of a dynamic-dependence graph held in memory, the on-demand construction of the dynamic-dependence graph is quite slow, since it requires repeated traversals of the trace stored on disk. We found that even after optimizations aimed at speeding up graph traversal, it took 5 to 25 minutes to compute a single dynamic slice.

The WET contains all the dynamic information necessary to compute dynamic slices. The compacted WETs can be kept in memory and used to compute dynamic slices of reasonably long program runs. Given a value computed by the execution of a code statement during program execution, the WET slice is a backward slice over the WET representation starting from the value of interest. This slice captures the complete flow of control, flow of values across dependences, and address references that directly or indirectly impacted the computation of the value of interest. Thus a WET slice provides a superset of information provided by a traditional dynamic slice [Korel and Laski 1988]. In this application, we cut the prior runs at the boundaries of from 114 and 139 million intermediate-level statements, which are very close to the trace lengths used in our previous works [Zhang et al. 2003; Zhang and Gupta 2004]. As shown in Table VIII, the average times needed to extract a WET slice after tier-1 and tier-2 compression are little over 14.34 and 90.98 sec, respectively. These times are far less than the times of 5 to 25 min provided by our prior algorithm in Zhang et al. [2003]. We would like to point out that response times for the `099.go` benchmark are higher than other programs. Due to complex control flow structure of `099.go` each node had several incoming edges and thus it took longer to identify the appropriate relevant edge during traversal.

6.2 Matching

Although compile-time optimizations are important for improving the performance of programs, applications are typically developed with the optimizer turned off. Once the program has been sufficiently tested, it is optimized prior to its deployment. However, the optimized program may fail to execute correctly

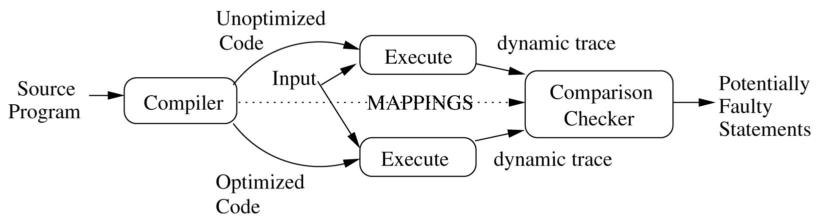


Fig. 17. Comparison checker.

on an input although the unoptimized program ran successfully on that input. In this situation, the fault may have been introduced by the optimizer through the application of an unsafe optimization or a fault present in the original program may have been exposed by the optimizations. Determining the source and cause of fault is, therefore, important.

In prior work, we had developed a technique called *comparison checking* to address the above problem [Jaramillo et al. 1999]. A comparison checker executes the optimized and unoptimized programs and continuously compares the results produced by *corresponding instruction executions* from the two versions (see Figure 17). At the earliest point during execution that the results differ, they are reported to the programmer who can use this information to isolate the cause of faulty behavior. *It should be noted that not every instruction in one version has a corresponding instruction in the other version because optimizations, such as reassociation, may lead to instructions that compute different intermediate results.* While the above approach can be used to test optimized code thoroughly and assist in location of fault if one exists, it has one major drawback. In order for the comparison checker to know which instruction executions in the two versions correspond to each other, the compiler writer must write extra code that determines *mappings* between execution instances of instructions in the two program versions. Not only do we need to develop a mapping for each kind of optimization to capture the effect of that optimization, we must also be able to compose the mappings for different optimizations in order to produce the mapping between the unoptimized and fully optimized code. The above task is not only difficult and time consuming, it must be performed each time a new optimization is added to the compiler.

We have developed a WET-based approach for automatically generating the *mappings*. The basic idea behind our approach is to run the two versions of the programs and regularly compare their execution histories. The goal of this comparison is to find *matches* between the execution history of each instruction in the optimized code with execution histories of one or more instructions in the unoptimized code. If execution histories match closely, it is extremely likely that they are, indeed, the corresponding instructions in the two program versions. At each point, executions of the programs are interrupted and their histories are compared with each other. Following the determination of matches, we determine if faulty behavior has already manifested itself and, accordingly, potential causes of faulty behavior are reported to the user for inspection. For example, instructions in the optimized program, which have been executed numerous times but do not match anything in the unoptimized code, can be

Table IX. Nodes Matched

Program	Executed (millions)	Distinct Executed	Matched (%)
099.go	62.4	28701	91.0
130.li	65.1	4325	97.0
164.gzip	66.8	16913	93.5
181.mcf	64.6	4469	91.8
197.parser	62.1	1450	96.1
255.vortex	60.8	32583	97.7
256.bzip2	29.7	12520	96.9
300.twolf	63.3	9419	94.2
Avg.	59.35	13797.5	94.8

reported to the user for examination. In addition, instructions that matched with each other in earlier part of execution but later did not match can be reported to the user—this is because the later phase of execution may represent instruction executions after faulty behavior manifests itself. The user can then inspect these instructions to locate a fault(s).

The key problem that we must solve to implement the above approach is to develop a matching process that is highly accurate. We have designed a WET-based *matching algorithm* that consists of the following two steps: *signature matching* and *structure matching*. A signature of an instruction is defined in terms of the frequency distributions of the result values produced by the instruction and the addresses referenced by the instruction. If signatures of two instructions are consistent with each other, we consider them as being *tentatively* matched. In the second step we match the structures of the data dependence graphs produced by the two versions. Two instructions from the two versions are considered to match, if their was a tentative signature match between them and the instructions that provided their operands also matched each other. More details of this algorithm can be found in Zhang and Gupta [2005].

We studied the accuracy of our matching algorithm we implemented to match execution traces of unoptimized and optimized binaries produced by the Trimaran compiler [Trimaran 1997]. This compiler is based on the IMPACT system that performs a host of optimizations including constant propagation, copy propagation, common subexpression elimination, constant combining, constant folding, code motion, strength reduction, dead code removal, and loop optimizations, etc. When we matched the executed instructions in the two versions, we achieved highly accurate matches. In Table IX, we show matching results for the same set of benchmarks except *126.gcc*, because, for some reason, Trimaran did not compile *126.gcc* with all the optimizations off. The column *Executed* gives the number of statements in millions that were executed during the program run. The column *Distinct Executed* gives the number of statically distinct instructions that were executed at least once. Finally, the column *Matched* tells us for what percentage of instructions that were executed at least once our matching algorithm found the correct match or matches. On average, for almost 95% of the executed statements in the optimized code, one or more corresponding true matches were found. Other instructions are not matched because none truly exist—this is due to the features of the VLIW machine used only by the optimized version and optimizations such as reassociation. Therefore, our matching algorithm is highly effective in the presence of aggressive transformations. Our algorithm is also reasonably fast—matching of runs over 60 million instructions took between 4 to 6 min. We believe that further optimization of our initial implementation can further reduce this time.

7. CONCLUSIONS

In this paper we presented the design and evaluation of a compressed whole execution trace representation. The representation presents complete profile information, including control flow, value, and addresses, and control and data-dependences, in an integrated manner such that a wide range of queries requiring single or multiple types of profile information can be responded to quickly and with ease. We presented compression techniques that are highly effective in reducing the sizes of WETs. In particular, on average, compressed WET representation for an execution of 647 million statements can be stored in 331 megabytes of storage. Our extrapolation shows that whole profile information corresponding to a program run involving execution of 3.9 billion intermediate-level code statements can be stored in 2 gigabytes of space which can easily fit in the main memory today. Thus, the compressed WET representation can form the basis of a powerful tool for analyzing profile information to discover program characteristics that can be exploited to design better compilers and architectures. We demonstrated the power and efficiency of WETs by developing WET-based algorithms for two applications: dynamic program slicing and version matching. Although our WETs were designed for sequential programs, we can adapt the WET representation to allow multithreaded programs. A stream of values corresponding to thread ids can be maintained. In addition, edges corresponding to synchronization events can be maintained.

REFERENCES

Agrawal, H and Horgan, J. 1990. Dynamic program slicing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 246–256.

Agrawal, H., DeMillo, R., and Spafford, E. 1993. Debugging with dynamic slicing and backtracking. *Software Practice and Experience 23*, 6, 589–616.

Ammons, G. and Larus, J. R. 1998. Improving data flow analysis with path profiles. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 72–84.

Ball, T. and Larus, J. R. 1996. Efficient path profiling. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. IEEE/ACM, New York, 46–57.

Bodik, R., Gupta, R., and Soffa, M. L. 1998. Complete removal of redundant expressions. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 1–14.

Burtscher, M. 2004. VPC3: A fast and effective trace-compression algorithm. In *Proceedings of the SIGMETRICS Conference*. ACM, New York, 167–176.

Burtscher, M. and Jeeradit, M. 2003. Compressing extended program traces using value predictors. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 159–169.

Burtscher, M. and Zorn, B. G. 1999. Exploring last n value prediction. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 66–76.

CALDER, B., FELLER, P., AND EUSTACE, A. 1997. Value profiling. In *Proceedings of the 30th IEEE/ACM International Symposium on Microarchitecture*. IEEE/ACM, New York, 259–269.

CHILIMBI, T. M. 2001. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 191–202.

CHILIMBI, T. M. AND HIRZEL, M. 2002. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 199–209.

DUESTERWALD, E., GUPTA, R., AND SOFFA, M. L. 1992. Rigorous data flow testing through output influences. In *Proceedings of the 2nd Irvine Software Symposium*. 131–145.

GOEMAN, B., VANDIERENDONCK, H., AND BOSSCHERE, K. 2001. Differential FCM: increasing value prediction accuracy by improving table usage efficiency. In *Proceedings of the Seventh International Symposium on High Performance Computer Architecture*. IEEE-CS. 207–216.

GUPTA, N. AND RAO, P. 2001. Program execution based module cohesion measurement. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*. IEEE. 144–153.

GUPTA, R., BERSON, D., AND FANG, J. Z. 1998. Path profile guided partial redundancy elimination using speculation. In *Proceedings of the IEEE International Conference on Computer Languages*. IEEE. 230–239.

JACOBSON, Q., ROTENBERG, E., AND SMITH, J. E. 1997. Path-based next trace prediction. In *Proceedings of the 30th IEEE/ACM International Symposium on Microarchitecture*. IEEE/ACM, New York, 14–23.

JARAMILLO, C., GUPTA, R., AND SOFFA, M. L. 1999. Comparison checking: an approach to avoid debugging of optimized code. In *Proceedings of the ACM SIGSOFT 7th Symposium on Foundations of Software Engineering and 8th European Software Engineering Conference*. LNCS 1687, Springer Verlag, New York, 268–284.

JOSEPH, D. AND GRUNWALD, D. 1997. Prefetching using Markov predictors. In *Proccedings of the International Symposium on Computer Architecture*. IEEE/ACM, New York, 252–263.

KAMKAR, M. 1993. Interprocedural dynamic slicing with applications to debugging and testing. PhD Thesis, Linkoping University, Sweden.

KLEINOSOWSKI, A. J. AND LILJA, D. J. 2002. MinneSPEC: a new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters 1*.

KOREL, B. AND LASKI, J. 1988. Dynamic program slicing. *Information Processing Letters 29*, 3, 155–163.

KOREL, B. AND RILLING, J. 1997. Application of dynamic slicing in program debugging. In *Proceedings of the Automated and Algorithmic Debugging*. 43–59.

LARUS, J. R. 1999. Whole program paths. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 259–269.

LIPASTI, M. H. AND SHEN, J. P. 1996. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th IEEE/ACM International Symposium on Microarchitecture*. IEEE/ACM, New York, 226–237.

NEVIL-MANNING, C. G. AND WITTEN, I. H. 1997. Linear-time, incremental hierarchy inference for compression. In *Proceedings of the Data Compression Conference*. IEEE-CS. 3–11.

PERELMAN, E., HAMERLY, G., BIESBROUCK, M. V., SHERWOOD, T., AND CALDER, B. 2003. Using SimPoint for accurate and efficient simulation. In *Proceedings of the SIGMETRICS the International Conference on Measurement and Modeling of Computer Systems*. ACM, New York, 318–319.

RUBIN, S., BODIK, R., AND CHILIMBI, T. 2002. An efficient profile-analysis framework for data layout optimizations. In *Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 140–153.

SAZEIDES, Y. 2003. Instruction-isomorphism in program execution. In *Proceedings of the Value Prediction Workshop*.

SAZEIDES, Y. AND SMITH, J. E. 1997a. The predictability of data values. In *Proceedings of the 30th IEEE/ACM International Symposium on Microarchitecture*. IEEE/ACM, New York, 248–258.

SAZEIDES, Y. AND SMITH, J. E. 1997b. Implementations of context based value predictors. Technical Report ECE-97-8, University of Wisconsin-Madison.

Trimaran Compiler Research Infrastructure, 1997. Tutorial Notes.

Venkatesh, G. A. 1995. Experimental results from dynamic slicing of C programs. *ACM Transactions on Programming Languages and Systems 17*, 2, 197–216.

Weiser, M. 1982. Program slicing. *IEEE Transactions on Software Engineering SE-10*, 4, 352–357.

Yang, J. and Gupta, R. 2002. Frequent value locality and its applications. *ACM Transactions on Embedded Computing Systems 1*, 1, 79–105.

Young, C. and Smith, M. D. 1998. Better global scheduling using path profiles. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. IEEE/ACM, New York, 115–123.

Zhang, Y. and Gupta, R. 2001. Timestamped whole program path representation and its applications. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 180–190.

Zhang, Y. and Gupta, R. 2002. Data compression transformations for dynamically allocated data structures. In *Proceedings of the International Conference on Compiler Construction*. LNCS 2304, Springer Verlag, New York, 14–28.

Zhang, X., Gupta, R., and Zhang, Y. 2003. Precise dynamic slicing algorithms. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*. IEEE/ACM, New York, 319–329.

Zhang, X. and Gupta, R. 2004. Cost effective dynamic program slicing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 94–106.

Zhang, X. and Gupta, R. 2005. Matching execution histories of program versions. In *Proceedings of the Joint 10th European Software Engineering Conference and 13th SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, to appear. 189–206.

Zilles, C. B. and Sohi, G. 2000. Understanding the backward slices of performance degrading instructions. In *Proceedings of the IEEE/ACM 27th International Symposium on Computer Architecture*. IEEE/ACM, New York, 172–181.

Received March 2005; revised July 2005; accepted July 2005