# Whole Execution Traces

Xiangyu Zhang Rajiv Gupta
The University of Arizona
Department of Computer Science
Tucson, Arizona 85721

## Abstract

*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. Due to 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 relavent 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 do not adversely 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 (e.g., in form of dynamic slices of WETs). 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.*

## 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. Therefore researchers have developed compression techniques to limit the memory required to store different types of profiles.

Lossless compression techniques for several different types of profiles have been separately studied. Compressed representations of *control flow* traces can be found in [14, 26]. These profiles can be analyzed for presence of hot program paths or traces [14] which have been exploited for performing path sensitive optimizations [24, 2, 10] and path-sensitive prediction techniques [11]. *Value profiles* have been compressed using value predictors [3] and used to perform code specialization [6], data compression [27], value speculation [15], and value encoding [23]. *Address profiles* have also been compressed [7] and used for identifying hot data streams that exhibit data locality which can help in finding cache conscious data layouts [18] and developing data prefetching mechanisms [8, 12]. *Dependence profiles* have been compressed in [25] and used for computation of dynamic slices [25], studying the characteristics of performance degrading instructions [28], and studying instruction isomorphism [21].

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. Now if a request is made for the profile information related to the execution of a statement, we will have to search through each of the compressed streams to collect 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 open the door to understanding program behavior

involving interactions among the different program characteristics captured by these profiles. This will lead to exploration of advanced compiler and architecture techniques which 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 relavent 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. give a position of a value in the stream, it is possible to find the immediately preceding and following values with equal ease.

We have implemented the unified WET representation using the Trimaran compiler infrastructure [22]. 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 [13, 17]. In [13] program inputs with smaller program runs that effectively characterize program behavior were identified for Spec benchmarks. In [17] 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.

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 that is used to compress 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. Conclusions are given in section 6.

## 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 of the 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 and then $time$ is 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 $[e_1 e_2 ...]$. 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.

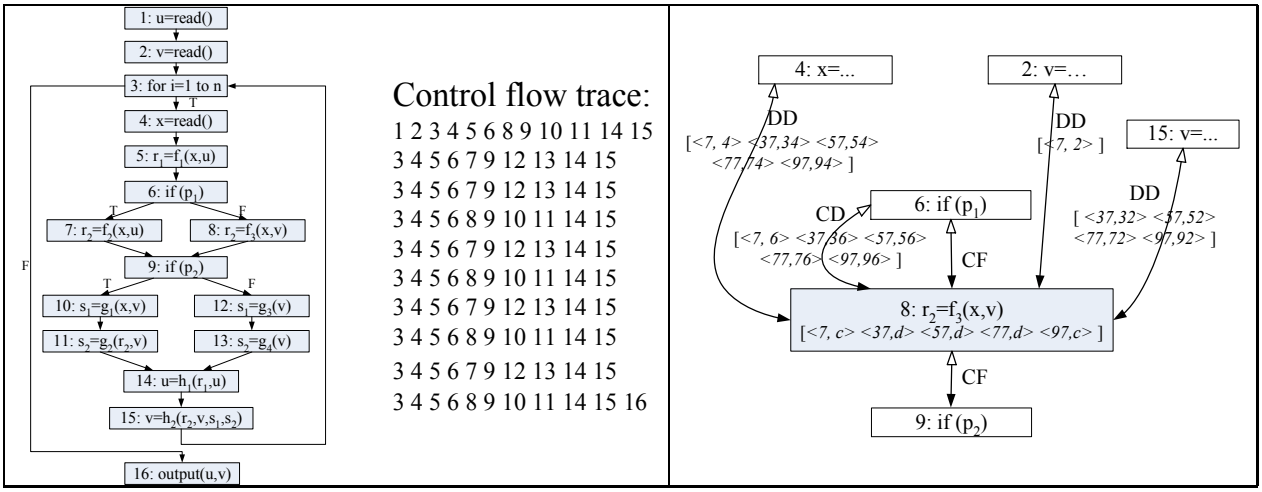
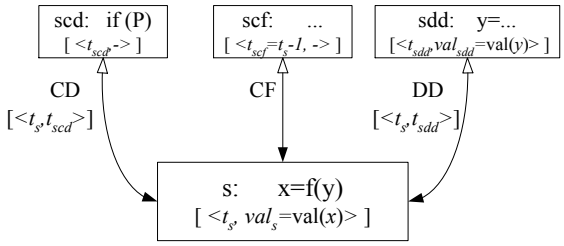



**Figure 1. An example: (a) CFG and its control flow trace; (b) WET subgraph of node 8.**

**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 it 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.

**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 demonstrates that the WET representation incorporates all of the control flow, data and control dependence, value, and address profile information. Also given the bidirectional nature of edges, the graph and hence the profiles can be traversed in forward or backward direction starting from any execution point.

*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.

*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.

*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.

*WET slices.* We define the backward WET slice of a value produced by an execution instance of a statement as the subgraph of WET that captures all of the profile information that led to the computation of the value. By traversing the WET in backward direction along data and control dependences we can compute this slice. A forward WET slice can be found by forward traversal of data and control dependence edges originating from the computed value. The forward slice of WET shows all values whose computation was influenced by the value at which the slice computation originates. WET slices are examples of queries that request all types of profile information contained in WET.

We have shown the organization of all types of profile data in the WET representation which allows 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 the WET graph: (a) timestamps labeling the nodes; (a) 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.

## 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 [1] 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.

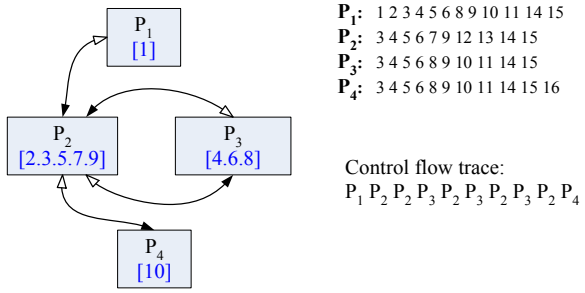



**Figure 2. Reducing number of timestamps.**

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 4 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 [21]. 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 indices 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.

$$Values[i] = UValues[Pattern[i]]$$

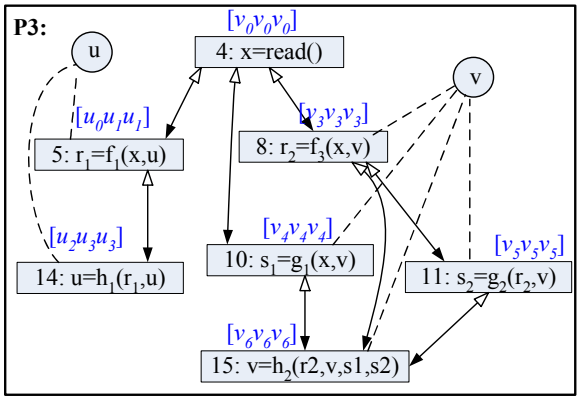
Before

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

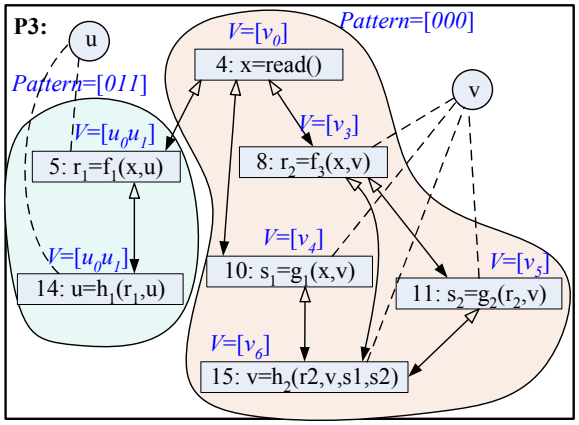
After: Pattern=[0101]

| 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.




(a) Before compression.




(b) After compression.

**Figure 3. Value compression.**

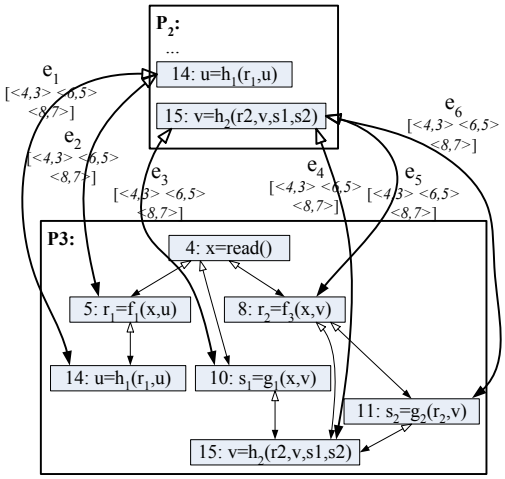
In the above discussion the situation considered is such that all of the statements shared a single pattern. In general, multiple patterns may be desirable because different subsets of statements may depend upon different subsets of inputs that are either received from outside the node or are read through 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 depend upon exactly the same inputs into the same group. Next if a group depends upon 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, for each group a pattern is found and the values are compressed according to the groups pattern.

In Figure 3 formation of groups for node $P3$ is illustrated. The first 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.

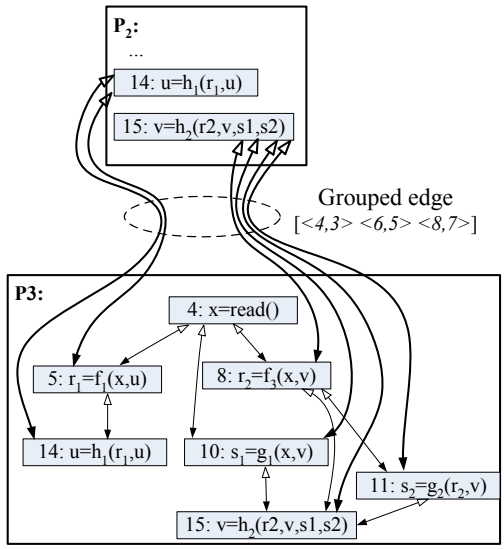
### 3.3 Timestamp pairs labeling edges

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 [25]. To describe how timestamp pairs can be reduced, we divide the data dependences into two categories: edges that are *local* to a Ball Larus path; and edges that are *non-local* as they cross Ball Larus path boundaries.

Let us consider a node $n$ that contains a pair of statements $s_1$ and $s_2$ such that *local* data dependence edge exists due to flow of values from $s_1$ to $s_2$. For every timestamp pair $< t_{s_2}, t_{s_1} >$ labeling the edge it is definitely the case that $t_{s_1} = t_{s_2}$. In addition, if $s_2$ *always receives* the involved operand value from $s_1$, then we do not need to label this edge with timestamp pairs. This is because the timestamp pairs that label the edge can be inferred from the labels of node $n$. If node $n$ is labeled with timestamp $t_n$, under the above conditions, the data dependence edge must be labeled with the timestamp pair $< t_n, t_n >$. It should be noted that by creating nodes corresponding to Ball Larus paths, opportunities for elimination of timestamp pair labels increase greatly. This is because many non-local edges get converted to local edges.




(a) Removing labels on local edges.




(b) Sharing labels across non-local edges.

**Figure 4. Reducing timestamp pairs.**

Let us consider *non-local* edges next. Often there are multiple data dependence edges that are introduced between a pair of nodes. It is further often the case that these edges have identical labels. In this case we can save space by creating a representation for a group of edges and save a single copy of the labels.

While the above approach reduces the space required to store timestamp pairs, it is worth noting that there is no negative impact on ease with which profile information can be accessed. The techniques described above are simple and very frequently applicable. More aggressive techniques for removing timestamp pairs can be found in [25].

The example in Figure 4 illustrates the above technique. The detailed data dependences corresponding to Ball Larus path $P3$ are shown. The first figure shows what the graph looks like after removal of all timestamp pairs that label the edges local to $P3$ while the second figure shows the sharing of timestamp pairs across non-local edges from Ball Larus path $P2$ to $P3$.

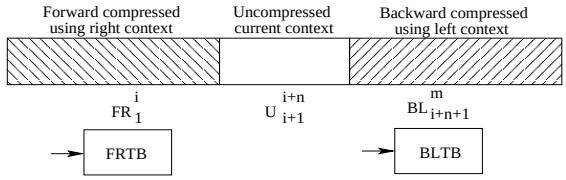
## 4 Stream Compression

For the next step in compression we view the information labeling the WET as consisting of streams of values arising from 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); and a 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 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 towards the last. Examples of such algorithms include compression algorithms designed from value predictors which were used for compressing value and address traces in [3, 4]. 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 [16] which was used for compressing control flow traces in [14] and address traces in [7] yields a representation which can be traversed in both directions. However, it is well known that Sequitur is nearly not as effective as the above unidirectional predictors when compressing value streams [3].

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 into a *bidirectional* one. Let us consider the highly effective FCM predictor [19, 20]. 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); oth-

erwise 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 an index. The compressed stream has two types of entries $< T >$ and $< F, v >$ where $T$ and $F$ are one bit values indicating true and false while $v$ is an uncompressed value. If a value is correctly predicted by the look up table $TB$ using the left context, an entry $< T >$ 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 an entry $< F, v >$ is created in the uncompressed stream while the look up table $TB$ is updated using $v'$ to enable future predictions. Clearly for a stream that is *forward compressed* in the above fashion, only forward traversal is possible.




**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 [9]. In general a stream that has been compressed using a *bidirectional predictor* with a *context size* of $n$ values can be viewed as consisting of three parts: $[FR_1^i][U_{i+1}^{i+n}][BL_{i+n+1}^m]$ where $[FR_1^i]$ is obtained by *forward compressing* ($F$) values 1 through $i$ using *right context* ($R$); $[U_{i+1}^{i+n}]$ are the $n$ uncompressed values ($U$) that form the *context* of the bidirectional predictor; and $[BL_{i+n+1}^m]$ is obtained by *backward compressing* ($B$) values $i+n+1$ through $m$ using the *left context* ($L$). Two look tables, $FRTB$ for $[FR_1^i]$ and $BLTB$ for $[BL_{i+n+1}^m]$, are also maintained. Thus, from the context of $n$ uncompressed values $[U_{i+1}^{i+n}]$ and $FRTB/BLTB$ the values $FR_i/BL_{i+n+1}$ can be uncompressed.

*Backward traversal* by one step shifts the context from $[U_{i+1}^{i+n}]$ to $[U_i^{i+n-1}]$ which is achieved by first uncompressing $FR_i$ into $U_i$ and then compressing $U_{i+n}$ into $BL_{i+n}$. $FRTB$ and $BLTB$ are also updated. *Forward traversal* by one step shifts the context from $[U_{i+1}^{i+n}]$ to $[U_{i+2}^{i+n+1}]$ which is achieved by first uncompressing $BL_{i+n+1}$ into $U_{i+n+1}$ and then compressing $U_{i+1}$ into $FR_{i+1}$. $FRTB$ and $BLTB$ are also updated in this case. The details of forward an backward traversal by one step are given in Figure 5.

We have shown how values are compressed and uncompressed during traversal. Initially all values are compressed by repeated application of above compression operation. In order to ensure that enough left context and right context is available to compress values at the beginning and end of the stream, we assume that the stream is extended by $n$ values each at the two ends where $n$ is the size of the context.

**Backward Traversal**

– Uncompress $FR_i$ and update $FRTB$.
  $index = hash(U_{i+1}^{i+n})$
  `if` $FR_i =< T >$ `then`
    $U_i = FRTB[index]$
  `else` Let $FR_i =< F, v >$
    $U_i = FRTB[index]$
    $FRTB[index] = v$
  `endif`

– Compress $U_{i+n}$ and update $BLTB$.
  $index = hash(U_i^{i+n-1})$
  `if` $BLTB[index] = U_{i+n}$ `then`
    $BL_{i+n} =< T >$
  `else`
    $BL_{i+n} =< F, BLTB[index] >$
    $BLTB[index] = U_{i+n}$

**Forward Traversal**

– Uncompress $BL_{i+n+1}$ and update $BLTB$.
  $index = hash(U_{i+1}^{i+n})$
  `if` $BL_{i+n+1} =< T >$ `then`
    $U_{i+n+1} = BLTB[index]$
  `else` let $BL_{i+n+1} =< F, v >$
    $U_{i+n+1} = BLTB[index]$
    $BLTB[index] = v$
  `endif`

– Compress $U_{i+1}$ and update $FRTB$.
  $index = hash(U_{i+2}^{i+n+1})$
  `if` $FRTB[index] = U_{i+1}$ `then`
    $FR_{i+1} =< T >$
  `else`
    $FR_{i+1} =< F, FRTB[index] >$
    $FRTB[index] = U_{i+1}$

**Figure 5. Traversing by one step.**

The example in Figure 6 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 forwards or backwards, the sequence of states encountered is the same.

**Bidirectional compression derived from a Last n predictor.** Another predictor which has been used for unidirectional compression is the last $n$ predictor [15, 5]. We also derived a bidirectional compression algorithm using the last $n$ predictor. This is because studies have shown that while

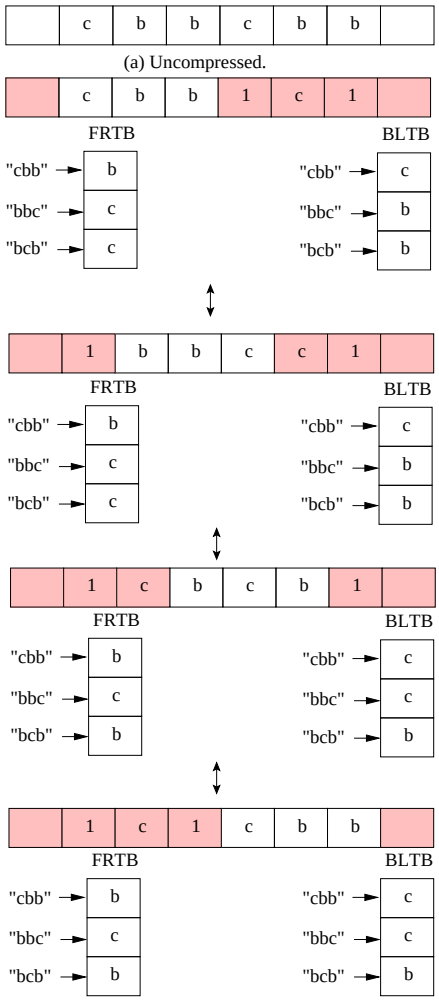



**Figure 6. Bidirectional FCM compression.**

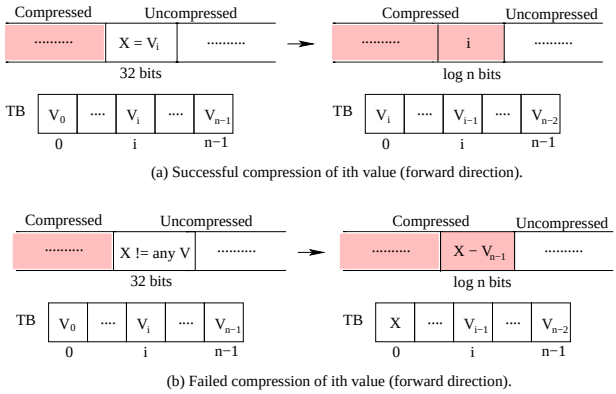



**Figure 7. Bidirectional Last n compression.**

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 [3]. 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 7. Backward compression is similar. It should be noted that unlike the bidirectional FCM predictor only a single look up table $TB$ is used for both forward and backward compression.

**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 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 [9]), last n, and last n stride methods. For each type we created three versions with differing context size.

## 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 [22] compiler infrastructure to profile several benchmarks from the `SpecInt` 2000 and 1995 suites. 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 representation.

### 5.1 WET Sizes

Table 1 lists the benchmarks considered and the lengths of the program runs which vary from 365 and 751 Million intermediate level statements. The effect of our two

tier compression strategy is also summarized in Table 1. 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 an average our approach enables saving of the whole execution trace corresponding to program run of 647 Million intermediate statements using 331 Megabytes of storage.

**Table 1. WET sizes.**

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

Now let us examine the effectiveness of our two tier compression strategy in detail. Table 2 shows the sizes of node labels, timestamp and value sequences, before and after compression while Table 3 presents the same information for edge labels.

**Table 2. Effect of compression on node labels.**

| Benchmark | $t_s$ labels | | | $val_s$ labels | | |
|---|---|---|---|---|---|---|
| | 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 3. Effect of compression on edge labels.**

| Benchmark | Edge labels | | |
|---|---|---|---|
| | 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 |

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 – even though our value compression algorithm is aggressive, the compression ratios for value sequences are modest in comparison to those for timestamp sequences.

In Figure 8 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 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.

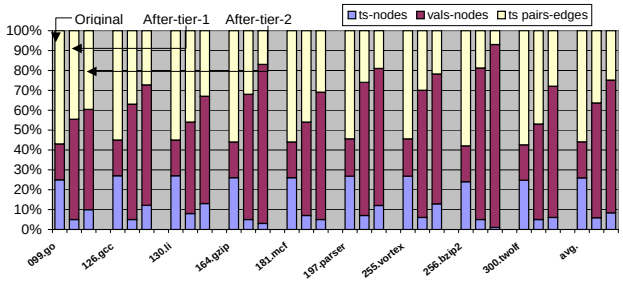



**Figure 8. Relative sizes of WET components.**

Next we show that WETs can be augmented with significant amounts of architecture specific information with 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 4 shows the additional storage taken by the above uncompressed execution histories. Clearly the amount of additional storage needed even without compression is quite small.

**Table 4. 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 |

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 9 the compression ratios (y-axis) for different trace lengths (x-axis) are plotted for each of the benchmarks. From the results in Figure 9 we notice that for 7 out 9 programs the compression ratios either improve or roughly remain the same as the length of the run increases.

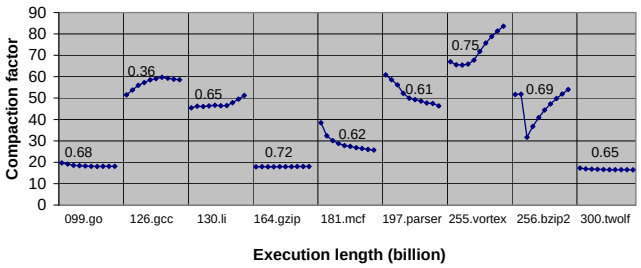



**Figure 9. Scalability of compression ratio.**

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 code statements took approximately 331 Megabytes of storage. Therefore we can extrapolate that if our machine has 2 Gigabyte of RAM, which is a common configuration these days, we can hold the WET corresponding to a program run involving execution of 3.9 Billion Trimaran intermediate code statements in memory. This 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 & Response Times for Queries

For the purpose of timing experiments we used shorter traces. Table 5 gives the lengths of the program runs used in the subsequent experiments. The runs used vary from 114 and 139 Million intermediate level statements. The times taken to construct the compressed WETs for these program runs are also given in Table 5 – the times for different programs are close because the lengths of the program runs are quite close.

In the preceding section we studied the effectiveness of our compression strategy and the scalability of our compressed WET representation. Recall that 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 tier compression and after second tier compression, i.e. after full compression.

*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 6. The total control flow trace size, the time to extract this entire trace, and rate at which it is extracted are given. We can see that on an average the entire control flow trace can be extracted in roughly 20 seconds in either direction. The response times after tier-1 compression 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.

**Table 5. WET construction times.**

| Benchmark | Stmts Executed (Millions) | Construction Time (Minutes) |
|---|---|---|
| 099.go | 132.52 | 34.52 |
| 126.gcc | 139.46 | 34.09 |
| 130.li | 126.78 | 36.84 |
| 164.gzip | 123.06 | 34.10 |
| 181.mcf | 137.31 | 39.69 |
| 197.parser | 122.12 | 39.13 |
| 255.vortex | 119.50 | 34.43 |
| 256.bzip2 | 128.25 | 39.47 |
| 300.twolf | 114.85 | 34.14 |
| Avg. | 127.09 | 36.27 |

*Query for 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 7 the size of complete load value trace, response time for extracting this trace, and the rates at which it is extracted after tier-1 compression 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 little over 47 seconds and 210 seconds 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 compressed representation to tier-2 compressed representation.

*Query for per instruction load/store address traces.* Let us consider requests for load and store address traces on per instruction basis. Such traces can be useful for designing address predictors for data prefetch mechanisms. In Table 8 the size of complete address trace, response time for extracting this trace, and the rates at which it is extracted after tier-1 compression 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 114 seconds and 226 seconds 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 compressed representation to tier-2 compressed representation.

**Table 6. Response times for control flow traces.**

| Benchmark | CF trace (MB) | Forward | | | | Backward | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Tier-1 | | Tier-2 | | Tier-1 | | Tier-2 | |
| | | time (sec.) | MB/sec | time (sec.) | MB/sec | time (sec.) | MB/sec | time (sec.) | MB/sec |
| 099.go | 530.09 | 146.49 | 3.62 | 147.43 | 3.60 | 136.58 | 3.88 | 136.80 | 3.87 |
| 126.gcc | 557.85 | 5.45 | 102.36 | 6.12 | 91.15 | 5.59 | 99.79 | 6.03 | 92.51 |
| 130.li | 507.12 | 3.16 | 160.48 | 3.68 | 137.80 | 3.20 | 158.47 | 3.70 | 137.06 |
| 164.gzip | 492.24 | 1.10 | 447.49 | 1.28 | 384.56 | 1.03 | 477.90 | 1.26 | 390.67 |
| 181.mcf | 549.25 | 4.07 | 134.95 | 4.14 | 132.67 | 3.71 | 148.05 | 4.20 | 130.77 |
| 197.parser | 488.50 | 5.21 | 93.76 | 6.14 | 79.56 | 5.13 | 95.22 | 5.67 | 86.16 |
| 255.vortex | 477.98 | 4.84 | 98.76 | 5.39 | 88.68 | 4.55 | 105.05 | 5.24 | 91.22 |
| 256.bzip2 | 512.99 | 1.13 | 453.97 | 1.56 | 328.84 | 1.10 | 466.35 | 1.38 | 371.73 |
| 300.twolf | 459.40 | 10.36 | 44.34 | 10.70 | 42.93 | 9.86 | 46.59 | 10.31 | 44.56 |
| Avg. | 508.38 | 20.20 | 171.08 | 20.72 | 143.31 | 18.97 | 177.92 | 19.40 | 149.84 |

**Table 7. Response times for per instruction load value traces.**

| Benchmark | Ld value trace (MB) | After Tier-1 | | After Tier-2 | |
|---|---|---|---|---|---|
| | | sec. | MB/sec | sec. | MB/sec |
| 099.go | 77.99 | 185.03 | 0.42 | 301.02 | 0.26 |
| 126.gcc | 77.45 | 38.99 | 1.99 | 45.89 | 1.69 |
| 130.li | 66.98 | 9.56 | 7.01 | 64.00 | 1.05 |
| 164.gzip | 70.58 | 35.77 | 1.97 | 587.70 | 0.12 |
| 181.mcf | 74.58 | 42.38 | 1.76 | 425.20 | 0.18 |
| 197.parser | 70.09 | 14.80 | 4.74 | 34.08 | 2.06 |
| 255.vortex | 86.92 | 7.73 | 11.24 | 45.98 | 1.89 |
| 256.bzip2 | 75.31 | 6.72 | 11.21 | 13.62 | 5.53 |
| 300.twolf | 71.13 | 85.70 | 0.83 | 375.03 | 0.19 |
| Avg. | 74.56 | 47.41 | 4.57 | 210.28 | 1.44 |

**Table 8. Response times for per instruction load/store address traces.**

| Benchmark | Address trace (MB) | After Tier-1 | | After Tier-2 | |
|---|---|---|---|---|---|
| | | sec. | MB/sec | sec. | MB/sec |
| 099.go | 115.79 | 377.79 | 0.31 | 556.22 | 0.21 |
| 126.gcc | 127.64 | 61.28 | 2.08 | 65.56 | 1.95 |
| 130.li | 112.92 | 15.29 | 7.39 | 202.16 | 0.56 |
| 164.gzip | 96.12 | 41.88 | 2.30 | 256.30 | 0.38 |
| 181.mcf | 98.85 | 52.81 | 1.87 | 443.45 | 0.22 |
| 197.parser | 111.47 | 21.29 | 5.24 | 60.65 | 1.84 |
| 255.vortex | 145.55 | 12.73 | 11.43 | 98.22 | 1.48 |
| 256.bzip2 | 126.78 | 12.00 | 10.57 | 21.40 | 5.92 |
| 300.twolf | 96.80 | 116.86 | 0.83 | 330.53 | 0.29 |
| Avg. | 114.66 | 79.10 | 4.67 | 226.05 | 1.43 |

*Query for WET slices.* Finally we consider query for a WET slice. 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 [25]. The average times needed to extract a WET slice after tier-1 and tier-2 compression are little over 14.34 seconds and 90.98 seconds respectively.

**Table 9. WET slices (avg. over 25 slices).**

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

This query is particularly interesting because its response contains all types of profile information. While it is straightforward to determine the profile information that must be contained in a WET slice given our WET representation, providing the same information using a combination of existing compressed representations for different types of profiles can be expected to be very tedious and very expensive. Till recently no algorithm existed that could perform dynamic slicing of long program runs. Our compression algorithms have made dynamic slicing feasible for reasonably long program runs.

Finally 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.

The results in this section have shown the versatility of the compressed WET representation in quickly responding to queries with 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), those that are instruction specific but also require limited traversal (load/store address traces), and those that require traversal to collect different types of related profile information (WET slices).

## 6 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 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 an 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 code statements can be stored in 2 Gigabytes of memory which is commonly available on machines 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.

### Acknowledgements

This work was supported by grants from Microsoft, Intel, and National Science Foundation grants CCR-0324969, CCR-0220334, CCR-0208756, CCR-0105355, and EIA-0080123 to the Univ. of Arizona.

## References

[1] T. Ball, and J. Larus, "Effi cient Path Profi ling," *IEEE/ACM International Symp. on Microarchitecture* (MICRO), pages 46-57, Dec. 1996.

[2] R. Bodik, R. Gupta and M.L. Soffa, "Complete Removal of Redundant Expressions," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 1-14, Montreal, Canada, June 1998.

[3] M. Burtscher and M. Jeeradit, "Compressing Extended Program Traces Using Value Predictors," *12th International Conference on Parallel Architectures and Compilation Techniques* (PACT), pages 159-169, September 2003.

[4] M. Burtscher, "VPC3: A Fast and Effective Trace-compression Algorithm," pages 167-176, *SIGMETRICS* 2004.

[5] M. Burtscher and B.G. Zorn, "Exploring Last n Value Prediction," *International Conference on Parallel Architectures and Compilation Techniques* (PACT), pages 66-76, October 1999.

[6] B. Calder, P. Feller and A. Eustace, "Value Profi ling," *The 30th IEEE/ACM International Symposium on Microarchitecture* (MICRO), pages 259–269, December 1997.

[7] T.M. Chilimbi, "Effi cient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 191–202, Snowbird, Utah, June 2001.

[8] T.M. Chilimbi and M. Hirzel, "Dynamic Hot Data Stream Prefetching for General-Purpose Programs," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 199-209, 2002.

[9] B. Goeman, H. Vandierendonck, and K. Bosschere, "Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Effi ciency," *Seventh International Symposium on High Performance Computer Architecture* (HPCA), pages 207-216, Jan. 2001.

[10] R. Gupta, D. Berson, and J.Z. Fang, "Path Profi le Guided Partial Redundancy Elimination Using Speculation," *IEEE International Conference on Computer Languages* (ICCL), pages 230-239, Chicago, Illinois, May 1998.

[11] Q. Jacobson, E. Rotenberg, and J.E. Smith, "Path-Based Next Trace Prediction," *The 30th IEEE/ACM International Symposium on Microarchitecture* (MICRO), December 1997.

[12] D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," *International Symposium on Computer Architecture* (ISCA), pages 252–263, 1997.

[13] A.J. KleinOsowski and D.J. Lilja, " MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *Computer Architecture Letters*, Vol. 1, June 2002.

[14] J.R. Larus, "Whole Program Paths," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 259-269, Atlanta, GA, may 1999.

[15] M. H. Lipasti and J. P. Shen, "Exceeding the Dataflow Limit via Value Prediction," *29th IEEE/ACM International Symposium on Microarchitecture* (MICRO), pages 226-237, Dec. 1996.

[16] C. G. Nevil-Manning and I.H. Witten, "Linear-time, Incremental Hierarchy Inference for Compression," *Data Compression Conference* (DCC), Snowbird, Utah, IEEE Computer Society, pages 3–11, 1997.

[17] E. Perelman, G. Hamerly, M.V. Biesbrouck, T. Sherwood, and B. Calder, "Using SimPoint for Accurate and Effi cient Simulation," *ACM SIGMETRICS the International Conference on Measurement and Modeling of Computer Systems*, June 2003.

[18] S. Rubin, R. Bodik, and T. Chilimbi, "An Effi cient Profi le-Analysis Framework for Data Layout Optimizations," *The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL), Portland, Oregon, Jan. 2002.

[19] Y. Sazeides and J.E. Smith, "The Predictability of Data Values," *30th IEEE/ACM International Symposium on Microarchitecture* (MICRO), December 1997.

[20] Y. Sazeides and J.E. Smith, "Implementations of Context Based Value Predictors," Technical Report ECE-97-8, University of Wisconsin-Madison. Dec. 1997.

[21] Y. Sazeides, "Instruction-Isomorphism in Program Execution," *Value Prediction Workshop* (held with ISCA), June 2003.

[22] *The Trimaran Compiler Research Infrastructure*. Tutorial Notes, November 1997.

[23] J. Yang and R. Gupta, "Frequent Value Locality and its Applications," *ACM Transactions on Embedded Computing Systems* (TECS), special inaugural issue on memory systems, Vol. 1, No. 1, pages 79-105, November 2002.

[24] C. Young and M.D. Smith, "Better Global Scheduling Using Path Profi les," *IEEE/ACM International Symposium on Microarchitecture* (MICRO), pages 115-123, 1998.

[25] X. Zhang and R. Gupta, "Cost Effective Dynamic Program Slicing," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), Washington D.C., June 2004.

[26] Y. Zhang and R. Gupta, "Timestamped Whole Program Path Representation and its Applications," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 180-190, Snowbird, June 2001.

[27] Y. Zhang and R. Gupta, "Data Compression Transformations for Dynamically Allocated Data Structures," *International Conference on Compiler Construction* (CC), Grenoble, France, April 2002.

[28] C.B. Zilles and G. Sohi, "Understanding the Backward Slices of Performance Degrading Instructions," *ACM/IEEE 27th International Symposium on Computer Architecture* (ISCA), 2000.