Timestamped Whole Program Path Representation and its Applications *

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

ABSTRACT

A *whole program path* (WPP) is a complete control flow trace of a program's execution. Recently Larus [18] showed that although WPP is expected to be very large (100's of MBytes), it can be greatly compressed (to 10's of MBytes) and therefore saved for future analysis. While the compression algorithm proposed by Larus is highly effective, the compression is accompanied with a loss in the ease with which subsets of information can be accessed. In particular, path traces pertaining to a particular function cannot generally be obtained without examining the entire compressed WPP representation. To solve this problem we advocate the application of *compaction* techniques aimed at providing easy access to path traces on a per function basis.

We present a WPP compaction algorithm in which the WPP is broken into *path traces* corresponding to individual function calls. All of the path traces for a given function are stored together as a block. Ability to construct the complete WPP from individual path traces is preserved by maintaining a *dynamic call graph*. The compaction is achieved by eliminating *redundant path traces* that result from different calls to a function and by replacing a sequence of static basic block ids that correspond to a *dynamic basic block* by a single id. We transform a compacted WPP representation into a *timestamped* WPP (TWPP) representation in which the path traces are organized from the perspective of dynamic basic blocks. TWPP representation also offers additional opportunities for compaction.

Experiments show that our algorithm compacts the WPPs by factors ranging from 7 to 64. At the same time information is organized in a highly accessible form which speeds up the responses to queries requesting the path traces of a given function by over 3 orders of magnitude.

*Supported by DARPA award no. F29601-00-1-0183 and National Science Foundation grants CCR-0096122, EIA-9806525, and CCR-9996362 to the University of Arizona.

1. INTRODUCTION

Profile data, in the form of basic block and edge profiles, has been extensively used for guiding the application of performance improving code transformations including global instruction scheduling [11]. Till recently it was believed that collecting path profiles is too expensive. However, Ball and Larus [4] showed that cost of acyclic path profiling is merely double the cost of collecting edge profiles. Encouraged by this result, Larus [18] further demonstrated that collecting a *whole program path* (WPP), which is the complete control flow trace of a program's execution, is also feasible. This is because, although a WPP is typically very large (100's of MBytes), it can be greatly compressed (to 10's of MBytes) and therefore saved for future analysis.

While the compression algorithm proposed by Larus is highly effective, the compression is accompanied with a loss of ease in accessibility to information. For example, path traces pertaining to a particular function cannot generally be obtained without examining the entire compressed WPP representation. This is a serious drawback because typically an application using the WPP can be expected to make a series of requests for profile data for individual functions, that is, each request is only for a small subset of the overall information contained in a WPP. Repeated extraction operations to satisfy these requests are likely to result in high analysis time costs. Therefore it is important to design a representation from which path traces of individual functions can be rapidly accessed.

We believe that the above loss of accessibility is a natural consequence of treating the entire control flow trace as a single data stream during *compression*. As a result the information corresponding to a given function is scattered through out the compressed trace and can in general be located only by examining the entire compressed trace. In order to solve this problem we advocate the application of *compaction* techniques which are aimed at simultaneously reducing the size of the WPP and providing easy access to subsets of information within the WPP. We present one such WPP compaction algorithm in this paper. In this algorithm the WPP is broken into *path traces* corresponding to individual function calls and all of the path traces for a given function are stored together as a block. Therefore information regarding a specific function can be readily accessed. In order to ensure that the complete WPP can be reconstructed from individual path traces, a *dynamic call graph* which links the *path traces* together is also maintained.

The compaction of WPPs is achieved by compacting the path traces using two techniques. First we *eliminate duplicate path traces* that result from different calls to the same function. This technique is very effective and resulted in reductions in the sizes of WPPs by a factors ranging from 5.66 to 9.5 in our experiments. This is because although many functions are called numerous times, they tend to follow one of a small subset of paths through the function body. For example, in a WPP collected from executing `gcc` we found that function `_rtx_equal_p` was called 355189 times but it generated only 35 unique path traces. The second technique we employ replaces a sequence of static basic block ids, that correspond to a *dynamic basic block*, by a single id. A dynamic basic block belonging to a path trace is a sequence of static basic blocks that is always entered from the first block and exited from the last block in the path trace. Since dynamic basic blocks often appear inside loops, they can be repeated many times in a path trace. Thus, replacing them by a single id can significantly reduce the size of the WPP. We typically observed reductions by factors ranging from 1.35 to 4.24 using this technique in our experiments.

While compacted WPPs provide easy access to information, they still do not represent path traces in a form that is suitable for analysis which is performed from the perspective of basic blocks. In particular, we consider a class of applications that perform data flow analysis on the program and utilize the WPP information to gather data flow facts, including frequencies of data flow facts [20, 7, 13, 14, 15], which can be observed to hold during the program execution represented by the WPP. Applications that fall in this category include identification of hot data flow facts for profile-guided code optimizers (static and dynamic) as well as debugging aids (dynamic slicing and currency determination algorithms). We refer to such analysis as *profile-limited data flow analysis.*

To address the above issue, we transform a compacted WPP representation into a *timestamped* WPP (TWPP) representation in which the trace information is organized from the perspective of dynamic basic blocks since data flow analysis is carried out from the perspective of dynamic basic blocks. We demonstrate that TWPPs can be conveniently used by applications that need to perform *profile-limited* data flow analysis including profile-guided optimizers and dynamic debugging algorithms. The TWPP representation also offers opportunities for compaction leading to *compacted TWPPs* with sizes that are significantly smaller than that of *compacted WPPs.* Compaction opportunity arises because when a sequence of timestamp values are used to identify the positions in a path trace corresponding to a block's execution, often these sequences form an arithmetic series which can be represented compactly. Our experiments indicate that the size of a compacted TWPP is often much smaller than that of a compacted WPP.

In summary the contributions of this paper are as follows:

- We replace the compression strategy proposed by Larus [18] by a *compaction* strategy which is also effective in reducing WPP sizes. Moreover compacted WPPs organize trace information in an accessible form which allows quick access to path traces of any given function. [Section 2]
- We propose the *timestamped WPP* (TWPP) representation which organizes trace information from the perspective of dynamic basic blocks. This form is highly suitable for applications that use WPPs to perform *profile-limited data flow analysis.* [Section 2]
- Overall the above techniques were observed to compact the original traces by factors ranging from 7 to 64 and at the same time speedups of over 3 orders of magnitude was observed in responding to queries requesting the path traces of a given function. Our techniques also compares favorably with Larus's technique for compressing WPPs. [Section 3]
- Finally we present *demand-driven* algorithms for profile-limited data flow analysis and illustrate their use in two applications: code optimizers and debugging tools. [Section 4]

2. COMPACTION ALGORITHM

As mentioned earlier, a whole program path (WPP) is a complete control flow trace of a program execution. Consider the program and a sample WPP shown in Figure 1. The WPP shows that the loop in `main` iterates 5 times and in each iteration the function `f` is called. The loop in function `f` iterates 3 times for each call. Looking at the WPP for a small program we observe two things: WPPs for real applications can be expected to be quite large (e.g., 100's of MBytes) and in its current linear form WPP is difficult to use (e.g., in order to extract trace information for a subpath in `main` or function `f`, we must examine the entire WPP). Next we present a step by step transformation of the above WPP to achieve two goals: compaction of the WPP to reduce memory requirements and organization of the WPP information for faster access to path traces of individual functions.

Partitioning WPP into path traces.

We partition the WPP into *path traces* corresponding to individual function calls and all of the path traces for a given function are stored together as a block. Therefore information regarding a specific function can be readily accessed. In order to ensure that the complete WPP can be reconstructed from individual path traces, a *dynamic call graph* (DCG) which links the *path traces* together is also maintained. Figure 2 shows this representation of the WPP for our example program. Clearly from this representation the WPP form of Figure 1 can be easily constructed. More importantly one can rapidly search for occurrences of a given path (intraprocedural or interprocedural). The path traces of interest are located and then examined for desired information. To search for an occurrence of a path in `main` we need to only examine one-sixth of the total trace in Figure 2.

Eliminating redundant path traces.

The WPP can be greatly reduced in size by *eliminating duplicate path traces* generated by different calls to the same function. In Figure 2, corresponding to the 5 calls to function `f`, there are only two unique path traces. Therefore the WPP representation can be transformed to eliminate redundant path traces as shown in Figure 3. This technique is very effective because although many functions are called numerous times, they tend to follow one of a small subset of paths through the function body. For example, in a WPP collected from executing `gcc` we found that function `_rtx_equal_p` was called 355189 times but it generated only 35 unique path traces.

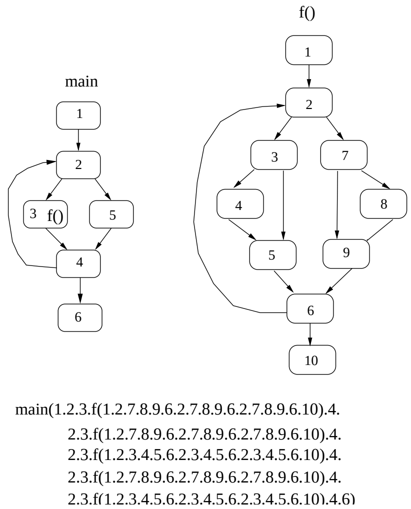

Figure 1: An uncompacted WPP.

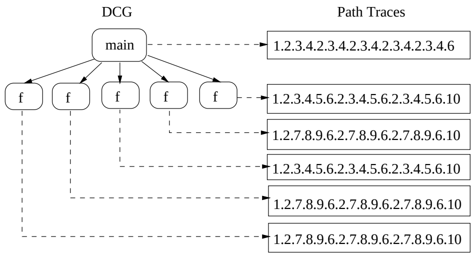

Figure 2: WPP organized using the DCG.

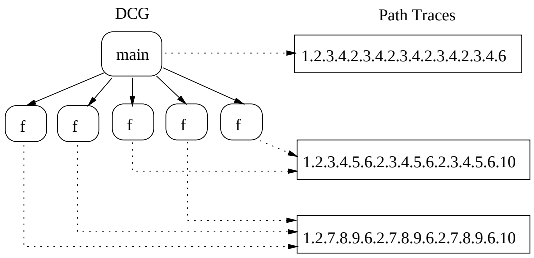

Figure 3: WPP after redundant path trace removal.

Creating dictionaries of dynamic basic blocks.

Another technique that we employ replaces a sequence of static basic block ids that correspond to a *dynamic basic block* by a single id. A *dynamic basic block* (DBB) belonging to a path trace is a sequence of static basic blocks that is always entered from the first block and exited from the last block in the path trace. Since DBBs can often appear inside loops, they are often repeated many times in a path trace. Thus, replacing them by a single id can significantly reduce the size of the WPP.

Each path trace is processed as follows: a dictionary of DBBs is created by constructing a dynamic control flow graph and finding chains of static blocks representing DBBs in it. Each DBB is assigned the block id of the first static block in it and accordingly the path trace is modified by deleting all but the first id in each occurrence of a DBB. Once all compacted path traces and dictionaries are obtained, duplicate path traces and dictionaries are also eliminated. In this transformed form, each node in the dynamic call graph has an associated tuple (t, d) where t is a path trace and d is a dictionary. Figure 4 shows the chains of static basic blocks that form dynamic basic blocks for the three path traces in Figure 3. After creating dictionaries and compacting path traces, we are left with one path trace and two dictionaries for function `f` as shown in Figure 5.

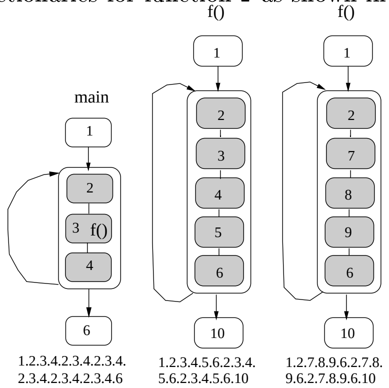

Figure 4: DBBs and dynamic control flow graphs.

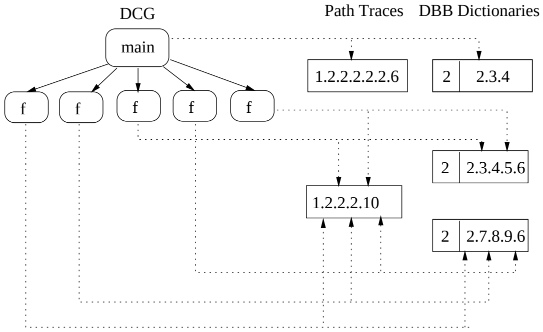

Figure 5: WPP after creating dictionaries of DBBs.

Timestamped WPP representation.

In the WPP representation described so far the execution trace of a given function invocation is represented by a sequence of basic blocks visited during its execution. While such a path trace representation is adequate for identifying hot paths through a program, it is not the most appropriate for performing data flow analysis. Since profile-limited data flow analysis is carried out from the perspective of basic blocks, it is more appropriate to organize the traces from the perspective of dynamic basic blocks. Next we describe the *timestamped* WPP (TWPP) representation which achieves this goal.

The execution of the function can be viewed from the perspective of time steps, where each time step corresponds to the execution of a dynamic basic block. Therefore a path trace for a function call in WPP representation can be viewed as a mapping between time steps, or *timestamps*, and dynamic basic blocks. In contrast, the TWPPs represent a mapping between dynamic basic blocks and an ordered sets of timestamps. Let $\mathcal{T}$, $\mathcal{B}$, and $\mathcal{P}(\mathcal{T})$ denote the set of timestamps, set of dynamic basic blocks, and the power set of timestamps associated with the path trace of a given function call f. A path trace in WPP and TWPP forms is represented by the following mappings:

$$\mathbf{WPPPathTrace}_f : \mathcal{T} \rightarrow \mathcal{B}$$
$$\mathbf{TWPPPathTrace}_f : \mathcal{B} \rightarrow \mathcal{P}(\mathcal{T})$$

Consider the WPP of Figure 5. The WPP trace 1.2.2.2.2.2.6 corresponds to the following $\mathcal{T} \rightarrow \mathcal{B}$ mapping: $\{1 \rightarrow 2,\ 2 \rightarrow 2,\ 3 \rightarrow 2,\ 4 \rightarrow 2,\ 5 \rightarrow 2,\ 6 \rightarrow 2,\ 7 \rightarrow 6\}$. When transformed to TWPP form it is represented by the following $\mathcal{B} \rightarrow \mathcal{P}(\mathcal{T})$ mapping: $\{1 \rightarrow \{1\},\ 2 \rightarrow \{2,3,4,5,6\},\ 6 \rightarrow \{7\}\}$. The complete uncompacted TWPP for this example is shown in Figure 6.

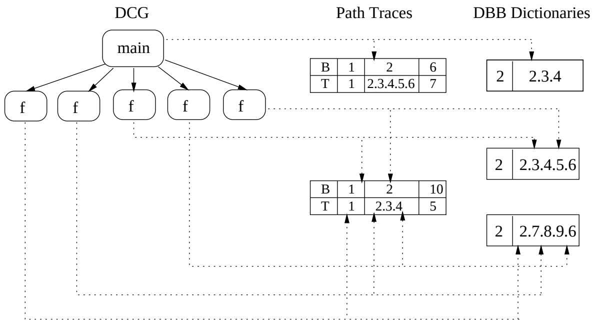

Figure 6: TWPP form.

Compacting TWPP path traces.

The path traces in TWPP form can be further compacted because often a subsequence of timestamp values corresponding a dynamic basic block forms an arithmetic series. This situation arises particularly when the same path within a loop body is traversed repeatedly during different loop iterations. The subsequences that form arithmetic series are compacted yielding a sequence of entries which are of the following form: l (singleton), $l : h$ ($l.l+1.l+2...h$, i.e., series with step 1), or $l : h : s$ ($l.l+s.l+2s...h$, i.e., series with step s). As we can see, depending upon its form, an entry is represented using one, two or three positive integer values. We store the timestamps corresponding to a block merely as a sequence of integers. For correct interpretation of the information we need to encode the boundaries that separate the variable length entries. This information is encoded in the signs (+ or -) of the values and therefore it does not require any increase in the size of the path trace. In particular, the last number in a each entry is stored as a negative number. By using the sign to encode the end of an entry we limit the largest timestamp value that is available to us since we can no longer use unsigned integers. However, our experience with the benchmarks considered shows that the timestamp value does not overflow because individual path traces are much smaller than the complete WPP.

Notice that the sequence of timestamps assigned to dynamic basic block 2 in Figure 6 form an arithmetic series since block

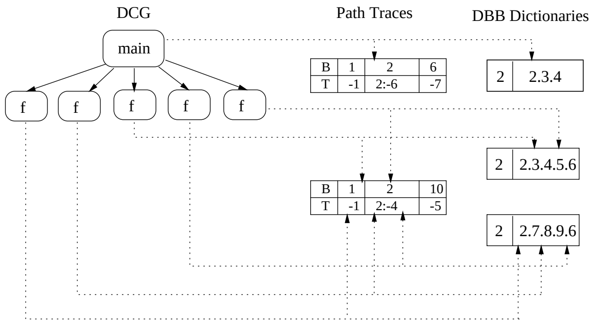

Figure 7: Compacted TWPP.

2 is executed repeatedly during successive loop iterations. Therefore the TWPP can be compacted into: $\{1 \rightarrow \{-1\},\ 2 \rightarrow \{2 : -6\},\ 6 \rightarrow \{-7\}\}$. Notice that the last number in each sequence is a negative number. The complete compacted form of TWPP for our running example is shown in Figure 7.

Compacting the DCG.

The dynamic call graphs resulting from executions of large application programs can also be quite large. Therefore in addition to compacting the path traces, we also compress the DCG. For this purpose we considered the popular dictionary based approaches proposed by Ziv and Lempel [28, 29]. In particular, we used Welch's variation of Ziv and Lempel's adaptive dictionary based technique which is also referred to as the LZW algorithm [26].

3. EXPERIMENTAL RESULTS

We have implemented the algorithm described in the preceding section and used it to compact WPPs for several benchmark programs from the SPECint95 suite. The original WPPs used in the experiments were generated using the Trimaran compiler infrastructure [24]. A WPP consists of two parts: the dynamic call graph (DCG) and the individual traces for function calls (which we will collectively refer to as the WPP traces). The sizes of WPPs used in our experiments are shown in Table 1. The experiments are aimed at studying the effectiveness of our compaction techniques in reducing memory requirements and the effectiveness of organization of the WPP information for faster access.

Program	DCG (MB)	WPP traces (MB)	Total size (MB)
099.go	6.0	170.0	176.0
126.gcc	34.7	489.5	524.2
130.li	6.6	78.3	84.9
132.ijpeg	1.7	266.9	268.6
134.perl	3.4	41.5	44.9

Table 1: Sample input traces used in the experiments.

Compaction study.

Table 2 shows the sizes of the WPP traces in their various forms. As we can see, the three compacting transformations, removal of redundant path traces, creation of DBB dictionaries, and transformation to compacted TWPP form are all very effective in reducing the WPP trace size. The

Program	WPP trace after		Compacted TWPP trace - MB	OWPP / CTWPP
	Redundancy removal - MB	Dictionary creation - MB		
099.go	27.0 (x6.30)	17.1 (x1.58)	17.6 (x0.97)	9.7
126.gcc	86.5 (x5.66)	50.8 (x1.70)	32.9 (x1.54)	14.9
130.li	8.5 (x9.21)	5.3 (x1.60)	1.1 (x4.81)	71.2
132.ijpeg	28.1 (x9.50)	20.8 (x1.35)	5.7 (x3.65)	46.8
134.perl	7.2 (x5.76)	1.7 (x4.24)	0.02 (x85.0)	2075

Table 2: WPP trace compaction due to various transformations.

Program	Compacted DCG (MB)	Compacted TWPP (MB)		Total (MB)	Compaction factor
		Traces	Dictionaries		
099.go	6.6	17.6	2.3	26.5	7
126.gcc	13.8	32.9	4.9	51.6	10
130.li	5.3	1.1	0.04	6.4	13
132.ijpeg	1.0	5.7	0.6	7.3	37
134.perl	0.7	0.02	0.02	0.7	64

Table 3: Overall compaction factor.

ratio of the sizes of original WPP traces (OWPP) and compacted TWPP traces (CTWPP) gives us the compression factor which varies from 9.7 to 2075 for our sample traces. The sizes of the WPP traces after each of the three transformations as well as the compression factors corresponding to each of the transformations are also shown separately in parenthesis in Table 2. The results show that each of the transformations is an important source of compaction.

A large factor of size reduction comes from removing redundant (duplicate) path traces (5.66 - 9.50). The reason for this large reduction becomes clear when we examine the data in Figure 8. This figure gives the percentage of total function calls (plotted along Y-axis) that can be attributed to functions with at most N unique path traces (N is plotted along the X-axis). For `130.li`, `132.ijpeg`, and `134.perl` programs 57-80% of all function calls are attributable to functions with at most 5 unique path traces. For `126.gcc` and `099.go` over 50% of function calls are attributable to functions with at most 25 and 50 unique traces respectively. Given that the number of function calls made during the runs of these benchmarks were in hundreds of thousands, we can see that the degree of redundancy in path traces is very high.

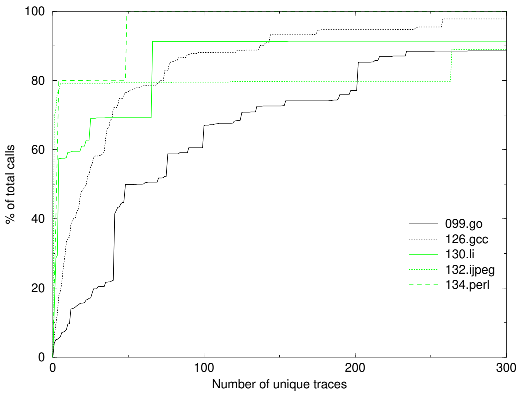

Figure 8: Trace redundancy.

The creation of dictionaries results in compaction of WPP traces by factors ranging from 1.35 to 4.24. The conversion into compacted TWPP form results in further reductions. For four out of five benchmarks, compacted TWPP traces provide reductions in the sizes of WPP traces by factors ranging from 1.55 to 85. The only case in which compacted TWPP trace is slightly larger is the `099.go` program where the compacted TWPP trace was 3% larger than compacted WPP trace prior to its conversion to TWPP form. These results are very encouraging because not only is the TWPP representation suitable for profile-limited data flow analysis, it is also compact.

The breakdown of different components of a WPP and the overall compaction factors for the complete WPP (DCG + WPP trace) are given in the Table 3. For the sample WPPs used in these experiments the overall WPP compaction factor ranges from 7 to 64.

Access time study.

To study the impact of reductions in the WPP size on the speed with which the path traces can be accessed, we conducted an experiment which measured the time it took to extract the path traces corresponding to a single function from the complete WPP. The speedups we can expect result from two sources. First due to the compaction of the WPP we have to read through a smaller file. Second we organize the contents of the file to allow faster access. Followed by the dynamic call graph, the path traces (including dictionaries) of the most frequently called function are stored first and that of least frequently called function are stored last. By remembering the position of information for each function in the file, and storing it as a header in the compacted TWPP file, we can access the path traces for individual functions rapidly.

Table 4 shows the times taken to extract a function's trace in the following scenarios: extraction from uncompacted file (column U); and extraction from compacted file (column C). Both the average and maximum times for U and C are given. On an average the access times are reduced by over 3 orders of magnitude.

Program	U (ms)		C (ms)		Speedup
	avg.	max	avg.	max	C/U(avg.)
099.go	5033	8383	8	1438	629
126.gcc	22879	29672	6	528	3813
132.ijpeg	7615	11447	6	258	1269
130.li	2390	3263	2	124	1195
134.perl	1303	1873	0.2	3	434

Table 4: Extraction times for a single function.

Larus's Sequitur based compression algorithm.

We have also implemented Larus's compression algorithm which is based upon Sequitur [19]. This algorithm produces the compressed WPP representation which is in the form of a grammar that generates a single string - the original trace. We compared the Sequitur generated grammar representation with the TWPPs generated using our approach in two ways: their sizes and the access times to individual function traces.

The results of this comparison are shown in Table 5. On an average, the total size of the grammar produced by Sequitur is smaller than the corresponding size of the compacted TWPP by a factor of 3.92. Now lets consider the time it takes to extract the trace corresponding to a single function from the complete compacted trace. The extraction of a function's trace from the Sequitur generated grammar essentially requires two steps: reading in the grammar and then processing it to generate a subgrammar corresponding to the functions trace. The total time taken for extraction, and the times for each of the steps, are shown in Table 5. These numbers represent averages over all functions present in the respective programs. These times range from 10's to 1000's of milliseconds. In contrast, the TWPP is so organized that we can locate and extract the trace in few (< 10) milliseconds. The access times for Sequitur grammars are greater than access times of TWPPs by factors ranging from 89 to 553. In summary, although TWPPs are larger in size by an average factor of 3.92, they provide access times that are lower by an average factor of 309. These experiments simply show that the two representations embody design decisions with different space time trade-offs.

Program	Compacted size		Extraction time	
	Sequitur (MB)	TWPP (MB)	Sequitur (ms) read+process=total	TWPP (ms)
099.go	8.4	26.5	622 + 1315 = 1937	8
126.gcc	11.2	51.6	898 + 2423 = 3321	6
132.ijpeg	0.7	6.4	544 + 1650 = 2194	6
130.li	7.8	7.3	47 + 132 = 179	2
134.perl	0.4	0.7	29 + 30 = 59	0.2

Table 5: Compacted trace sizes and extraction times.

Apart from the different size and access time characteristics, the two representations also impact on the design of analysis algorithms that will use them. While Larus's techniques is suitable for analysis of hot paths (i.e., collection of data flow facts that hold along frequently executed paths), our representation is suitable for collecting hot data flow facts (data flow facts that hold frequently at various program points). One of the advantages of our approach is that TWPPs are in the form required for profile-limited analysis. In contrast the compressed WPPs produced by Sequitur require some amount of preprocessing before they can be used by an application. In the next section we demonstrate the use of TWPPs in carrying out profile-limited data flow analysis.

4. PROFILE-LIMITED DATA FLOW

Next we present a systematic approach for *profile-limited data flow analysis* which is aimed at answering data flow queries with respect to a given WPP. Examples of such queries are: *does a data flow fact hold?* or *how often does a data flow fact hold at a program point?* with respect to a given WPP. Such analysis is useful for profile-guided compile-time optimization of programs [2, 7, 3, 11, 23, 13, 14, 15], dynamic optimization of programs [3, 6] (here the profile represents a partial execution history of the program which is used to optimize the remainder of the program's execution), and debugging of programs [10, 17, 1].

The analysis we present focuses on gathering *hot data flow facts* that hold during the execution of frequently called functions. Such analysis can be used to clone and create a specialized (optimized) copy of the function. For this analysis, we do not need access to the entire TWPP but only a subset of information corresponding to the function under consideration. In particular, we use a timestamp annotated dynamic control flow graph for the given path trace which is described below.

4.1 Timestamp annotated dynamic CFG

This representation consists of the dynamic control flow graph in which DBBs are annotated with timestamp vectors. This representation is quite adequate for data flow analysis because we can trace the WPP using the timestamp vectors associated with the dynamic basic blocks and limit the exploration of only those control subpaths that appear as part of the WPP during data flow analysis. The following characteristics make this proposed representation particularly attractive for profile-limited data flow analysis.

First it allows efficient backward and forward traversal of the path trace starting from any arbitrary point in the path trace. A timestamp and program point pair (t, n) together specify a particular point in the path trace. The preceding point is $(t-1, m)$ where m is the predecessor of n in the dynamic control flow graph labeled with timestamp $t-1$. Similarly the succeeding point is $(t+1, s)$ where s is a successor of n in the dynamic control flow graph which is labeled with timestamp $t+1$.

Second it allows efficient simultaneous traversals of multiple subpaths in the path trace. A vector of timestamps at a program point $(\vec{\mathcal{T}}, n)$ can be used to represent multiple traversal points. Each element in the vector can be incremented or decremented and resulting timestamps can be matched with timestamps of predecessors and successors to continue simultaneous traversal along multiple subpaths. *Compaction of timestamps directly attributes to the efficiency of traversals.* For example consider a series of timestamps represented by (2:20:2) in our representation. A simple increment/decrement resulting in (3:21:2)/(1:19:2) corresponds to simultaneous forward/backward traversal along 10 subpaths in the path trace.

An indicator of the relative costs of profile-limited analysis and traditional static analysis are the cumulative sizes of static and dynamic flowgraphs (see Table 6). We compared the total number of nodes (N) and edges (E) in the static and dynamic flow graphs. For a given function multiple dynamic flow graphs can result because of multiple unique traces associated with it. The nodes and edges in all of these graphs were counted in computing the cumulative size of the dynamic flow graphs. From the results in Table 6 we can see that the number of nodes and edges in the dynamic graphs are typically much smaller than those in the static graphs. However, the cost of profile-limited analysis is also dependent upon the size of timestamp vector associated with each node. Average size of the timestamp vector is shown in the last column of Table 6 (the value in parenthesis is the size of the vector before compaction - the results show that timestamp vector is significantly reduced in size using our compaction technique). In summary, the data in Table 6 indicates that while, as expected, profile-limited analysis is more expensive than static analysis, it has a reasonable cost.

Program	Static FG		Dynamic FG		
	N	E	$\sum$ N	$\sum$ E	avg. $\lvert\vec{\mathcal{T}}\rvert$
099.go	10823	16236	4739	16591	11.9 (15.7)
126.gcc	66571	104379	8838	20012	14.0 (33.1)
130.li	2701	3536	265	289	51.2 (410.3)
132.ijpeg	5718	8105	754	1213	18.1 (109.7)
134.perl	13117	19539	501	674	3.9 (616.8)

Table 6: Sizes of static and dynamic flow graphs.

4.2 Demand-driven analysis

It is natural to formulate profile-limited analysis in a demand-driven fashion [9, 21]. This is because the applications of profile-limited analysis request information incrementally. For example, during debugging a user typically makes a request for the dynamic slice corresponding to only one variable at a fixed program point (i.e., we only need to compute subset of data flow information for subset of program points). Similarly during profile-guided or dynamic code optimization, subset of profile-limited data flow information may be requested by the optimizer for subset of program points in hot regions of the program [6].

Queries for profile-limited data flow.

A *profile-limited data flow query* is of the form $< \mathcal{T}, n >_d$, where n is a node, $\mathcal{T}$ is a subset of timestamps for n in the path trace, i.e., $\mathcal{T} \subseteq \mathcal{T}(n)$, and d is the data flow fact of interest. This query represents a request for determining whether or not d holds true prior to n's executions corresponding to timestamp values in $\mathcal{T}$. Therefore the query $< \mathcal{T}(n), n >_d$ determines the data flow solution corresponding to all executions of n in a given path trace. The solution to this query allows us to determine if d always holds true, never holds true, and holds true sometimes for the given path trace. In fact solving such queries allows us to determine the frequency with which d holds true with respect to the given path trace [20, 7, 13, 14, 15].

Query propagation.

We consider profile-limited demand-driven *backward* propagation of queries for GEN-KILL problems because they arise both during code optimization and debugging. For simplicity, we consider the analysis of intraprocedural paths. However, in analyzing these paths we will take into account the effects of any function calls that a path trace may contain. Our techniques can be easily extended to handle interprocedural paths by analyzing path traces of multiple functions in concert and propagating queries along interprocedural paths [9].

The demand-driven propagation begins at a point n when the query $< \mathcal{T}, n >_d$ is raised. For GEN-KILL problems it is appropriate to propagate a *timestamp vector*, $\vec{\mathcal{T}}$, which contains one slot for every timestamp, or more precisely, for every entry in the compacted TWPP path trace. The propagation should be viewed as simultaneous (or parallel) search for data flow solutions corresponding to each timestamp in $\mathcal{T}$. Each slot in $\vec{\mathcal{T}}$ is initialized to the timestamp value(s) to which it corresponds. The propagation of this $\vec{\mathcal{T}}$ begins at n.

We must ensure that query propagation is consistent with the path trace under consideration. As discussed earlier in this section, this goal is easily accomplished using the timestamp annotated dynamic control flow graph representation. It is possible to correctly manipulate the timestamp vector during propagation such that the timestamps in the vector are propagated only to the appropriate predecessors. When a node that answers the query (true or false) with respect to a particular timestamp is encountered, the propagation on behalf of that timestamp ceases. Otherwise equivalent queries are generated and propagated along the path trace.

The query $< \vec{\mathcal{T}}, n >$ represents the search for dynamic GEN-KILL points corresponding to timestamps of n for which slots were created in $\vec{\mathcal{T}}$. For carrying out the propagation we must first compute dynamic GEN-KILL sets (i.e, sets wrt to a given TWPP) for a data flow fact d which are denoted as $DGEN_n^d$ and $DKILL_n^d$. Although n is a dynamic basic block, to simplify the presentation we assume that n contains a single statement. If node n contains a call to function f, then the traces for calls made by the n's instances corresponding to $\mathcal{T}(n)$ are examined. The set $GEN_f^d(\mathcal{T}(n))$ $(KILL_f^d(\mathcal{T}(n)))$ contains the subset of timestamps from $\mathcal{T}(n)$ for which call to function f generates (kills) d. If node n simply contains a statement, the dynamic sets are computed from the static GEN and KILL sets for node n denoted below as $SGEN_n$ and $SKILL_n$.

$$DGEN_n^d = \begin{cases} GEN_f^d(\mathcal{T}(n)) & if\ n\ calls\ f \\ \mathcal{T}(n) & elseif\ d \in SGEN_n \\ \phi & otherwise \end{cases}$$

$$DKILL_n^d = \begin{cases} KILL_f^d(\mathcal{T}(n)) & if\ n\ calls\ f \\ \mathcal{T}(n) & elseif\ d \in SKILL_n \\ \phi & otherwise \end{cases}$$

Now let us consider query propagation. The timestamp values in $\vec{\mathcal{T}}$ are each decremented by 1 during every step of backward propagation. Only those resulting timestamp values which are present in $\mathcal{T}(m)$, where m is a predecessor node, are propagated to m. At m the query for a timestamp may be resolved as true (if $t \in DGEN_m^d$) or as false (if $t \in DKILL_m^d$). If it is not resolved, then the above process is repeated starting with the decrementing of the timestamp and propagation continues. It should be noted that only a subset of slots may be relevant for a given predecessor node; thus the other slots will contain a null value denoted by $\perp$. The above rules are stated precisely below and are further illustrated by example applications discussed in the subsequent sections.

Propagation of $<\vec{\mathcal{T}}, n>$
Notation: $\vec{\mathcal{T}}/\mathcal{T}'$ is a timestamp vector st $(\vec{\mathcal{T}}/\mathcal{T}')_i = $ if $(\vec{\mathcal{T}})_i \in \mathcal{T}'$ then $(\vec{\mathcal{T}})_i$ else $\perp$.
Slots in $\vec{\mathcal{T}}$ resolved as true are slots in vectors $\bigcup_{m \in pred(n)} (\vec{\mathcal{T}} - \vec{1}) \ / \ DGEN_m^d$ which do not contain $\perp$.   Slots in $\vec{\mathcal{T}}$ resolved as false are slots in vectors $\bigcup_{m \in pred(n)} (\vec{\mathcal{T}} - \vec{1}) \ / \ DKILL_m^d$ which do not contain $\perp$.   Queries propagated for unresolved slots in $\vec{\mathcal{T}}$ $\bigcup_{m \in pred(n)} < (\vec{\mathcal{T}} - \vec{1}) \ / \ (\mathcal{T}(m) - DGEN_m^d - DKILL_m^d), m >$

4.3 Applications

In this section we illustrate the use of timestamp annotated dynamic CFG and the demand driven analysis described in the preceding section.

4.3.1 Profile-guided Optimization

A profile-guided optimizer identifies data flow facts that are observed to hold for hot regions of the code and exploits them to generate highly optimized code. This approach has been shown to be effective for variety of optimization tasks [7, 3, 11, 23, 13, 14, 15]. Both non-speculative [22, 16] and speculative [14, 15, 7] transformations have been developed for specialization of code along hot program paths.

In this section we illustrate the use of profile-limited analysis in profile-guided optimization. Consider a load instruction which is executed frequently and often causes cache misses to occur. In order to reduce the number of times this instruction is executed, we would like to determine the degree of redundancy in this instruction, that is, how often is the load is redundant because the loaded value is already available in some register. Rough estimate of the frequency of the data flow fact, *the load is redundant*, can be estimated using techniques based upon edge frequencies [20, 5, 7, 8] or acyclic path profiles [4, 13, 14, 15]. However, to obtain a precise value of degree of redundancy we require the use of an analysis based upon WPPs such as the profile-limited analysis.

Let us assume that we are interested in computing the degree of redundancy present in the load instruction in node 4 (4_Load) of the example shown in Figure 9. This load is redundant due to the load in node 1 (1_Load) as long as we arrive at it without visiting node 6 which contains a killing store (6_Store). Let us assume that the loop is executed 100 times during which it follows the given path trace. If we simply consider the execution frequencies of the nodes we cannot determine the degree of redundancy. We know that 4_Load executes 60 times, 1_Load executes 100 times, and 6_Store executes 40 times. However, from these frequencies we cannot tell how often 1_Load is killed by 6_Store prior to reaching 4_Load. While bounds on the degree of redundancy can be computed using techniques in [5, 8], precise degree of redundancy cannot always be found. On the other hand if we make use of profile-limited analysis which exploits the timestamps labeling the nodes, we can easily determine that 4_Load is always redundant for the given path trace. This information can be used by the optimizer to transform the program using code motion and/or restructuring [22, 16, 14, 7].

The query propagation that identifies that the redundancy count for 4_Load is 60, that is, degree of redundancy is 100%, is also shown in Figure 9. As we can see, the degree of redundancy has been computed using a single backward pass through the loop body and only 6 queries were generated in this process. This example illustrates the benefits of demand-driven analysis and compaction of the timestamps. Although the loop executes for 100 iterations, demand-driven analysis begins by considering the 60 iterations during which 4_Load is executed. Instead of dealing with each of the 60 timestamps of 4_Load individually, we are able to efficiently manipulate the compacted timestamps collectively during query propagation. This is analogous to the manner in which many array data flow analysis techniques achieve efficiency by propagating ranges representing array sections, as opposed to propagating individual array elements [12, 27].

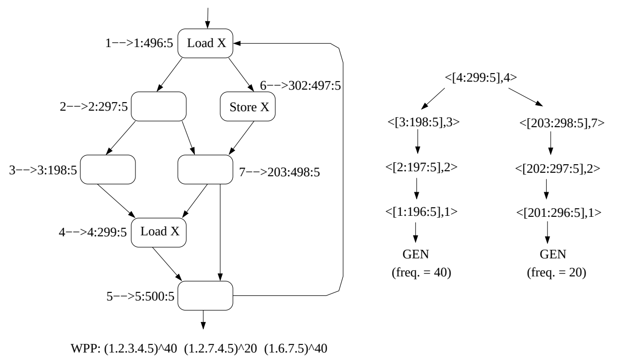

Figure 9: Detecting dynamic load redundancy.

4.3.2 Program Debugging

During debugging the user typically interrupts program execution and requests information specific to that particular program execution. The TWPP corresponding to partial program execution up to the breakpoint can be quite useful in accurately answering user queries. There are two specific debugging problems that can use profile-limited analysis: *dynamic slicing* and *dynamic currency determination* during symbolic debugging of optimized code.

Dynamic program slicing.

Static backward program slicing was first proposed by Weiser as a debugging aid [25]. An even more precise form of slicing, called dynamic slicing was proposed by Korel and Laski [17]. Most recently Agrawal and Horgan [1] developed three dynamic slicing algorithms which trade-off precision in the computed slice with the time it takes to compute the slice. Each of these algorithms constructs a *different* specialized program dependence graph (PDG) to capture the dependences exercised in a given execution. A backward traversal over the graph is used to compute the dynamic slice as a transitive closure over data and control dependences. Each of the above dynamic slicing algorithms can be implemented using one common representation, the timestamped dynamic control flow graph, and thus we can avoid constructing specialized graphs suggested in [1].

Next we describe the implementations of the three dynamic slicing algorithms in [1] and show how they can implemented using our approach. We will illustrate these algorithms using the example program and its execution history shown in Figure 10.

Approach 1: This method marks all executed nodes in the PDG during the execution. The backward traversal to identify the statements in the dynamic slice is allowed to visit only the marked nodes. These marked nodes are essentially the nodes with non-empty timestamp sets in our TWPP representation. Therefore in our implementation the backward traversal of a query through the timestamp annotated CFG is allowed to traverse only nodes that have a non-empty timestamp set. When a dependence is identified under such a traversal, the statement at which the dependence originates is added to the dynamic slice. In our example, all statements are executed. Therefore the dynamic slice is the same as a static slice, which contains all nodes except node 10.

Approach 2: This method marks all executed edges in the PDG during the execution. The backward traversal to identify the statements in the dynamic slice is allowed to only traverse marked edges. Our backward analysis uses timestamps to find dependences can carry out a similar traversal by ensuring that an edge from node n to node m is traversed only if the query at node m contains timestamp t and the timestamp $t-1$ is associated with node n. More over since this algorithm does not distinguish between different timestamps corresponding to a node, when a dependence is found, and new queries are generated at a node, all timestamps of that node are included in the newly generated query for further propagation. In the example, we will be able to get the dynamic slice which inlcudes all nodes except node 3 and 10.

Approach 3: This method duplicates executed node and its dependence edges during the execution so that it can distinguish between the instances of a given statement. This expanded graph is traversed to find the precise dynamic slice. Our backward analysis uses timestamps to find dependences and when a dependence is found we only a single timestamp is added to the newly generated queries. In other words we identify the precise instance of the assignment (for data dependence) and predicate (for control dependence) which is the source of the dependence and generate queries only for the corresponding instances of variables that are read by the assignment or predicate. In our example, note that although statements 8 and 3 are executed, they are not included in the slice because the value of Z at 13 depends only upon the values of Y and J computed by statements 7 and 11.

The detailed propagation of queries for the three algorithms are shown in Figure 11. The queries of the form $<\mathcal{T}, n>_V$ where $\mathcal{T}$ is the timestamp vector, n is the node at which the query is to be evaluated, and V is the variable whose definition is to be found. Therefore, a request for a slice on Z at line 14 is translated into the query $<[30], 14>_Z$. In case of the first algorithm the timestamp is not needed and therefore the query has the form $<*, 14>_Z$. All queries generated are given in the first column of Figure 11. The updated slice after the processing of a query is given in the corresponding entry of the second column and the type of dependence (control or data) that caused the addition of a statement to the slice is also indicated.

```
1 → 1               read N
2 → 2               I = 1
3 → 3               J = 0
4 → 4 : 28 : 8      while I ≤ N do
5 → 5 : 21 : 8          read X
6 → 6 : 22 : 8          if X < 0 then
7 → 7, 23                   Y = f1(X)
8 → 15                  else Y = f2(X)
9 → 8 : 24 : 8          Z = f3(Y)
10 → 9 : 25 : 8         write Z
11 → 10 : 26 : 8        J = 1
12 → 11 : 27 : 8        I = I+1
                    endwhile
13 → 29             Z = Z + J
14 → 30             breakpoint - request slice for Z

Input: (N = 3, X = -4, 3, -2)
WPP: 1.2.3.4.5.6.7.9.10.11.12
     4.5.6.8.9.10.11.12
     4.5.6.7.9.10.11.12
     4.13.14
```

Figure 10: Dynamic slicing example

Finally the worst case time complexity of our implementation is the same as that of Agrawal and Horgan's algorithm. Primary cost of both algorithms comes from processing the control flow trace. Our algorithm must examine the entire trace to compute the TWPP path trace representation while their algorithm must examine the trace to construct a dynamic dependence graph. The main difference between the two approaches is as follows. Agrawal et al. compute all dynamic dependences first and construct a graph using which any dynamic slice request can be processed using a simple traversal. In contrast our approach computes relevant dependences for slicing requests upon demand (like Weiser's algorithm [25]). Since the same dependences may be relevant to different slicing requests, their recomputation must be avoided by caching the computed dependences. In other words our approach builds the dynamic dependence graph incrementally as slicing requests are processed.

Dynamic currency determination.

Profile-limited analysis can be used to address the problem of dynamic currency determination. The user carries out debugging from the perspective of an unoptimized program; however, the code being executed is an optimized version of the program. Therefore when the user requests the value of some variable at a breakpoint, the value of the variable may or may not be current, that is, it may or may not correspond to the value that would have been observed by executing the unoptimized program. As shown in [10], timestamping of basic block executions is needed for dynamic currency determination. The timestamp annotated dynamic flow graph is therefore adequate for solving this problem.

Approach 1:Slicing request: $< *, 14 >_Z$		
Query	Slice	Dependence
$< *, 14 >_Z$	{14}	
$< *, 13 >_Z < *, 13 >_J$	{13,14}	data
$< *, 4 >_Z < *, 4 >_J < *, 4 >_I < *, 4 >_N$	{4,13,14}	control
$< *, 3 >_Z < *, 3 >_I < *, 3 >_N < *, 12 >_Z < *, 12 >_J < *, 12 >_I < *, 12 >_N$	{3,4,12,13,14}	data
$< *, 2 >_Z < *, 2 >_N < *, 11 >_Z < *, 11 >_I < *, 11 >_N$	{2,3,4,11,12,13,14}	data
$< *, 1 >_Z < *, 10 >_Z < *, 10 >_I < *, 10 >_N$	{1,2,3,4,11,12,13,14}	data
$< *, 9 >_Y < *, 9 >_I < *, 9 >_N$	{1,2,3,4,9,11,12,13,14}	data
$< *, 8 >_X < *, 8 >_I < *, 8 >_N < *, 7 >_X < *, 7 >_I < *, 7 >_N$	{1,2,3,4,7,8,9,11,12,13,14}	data
$< *, 6 >_X < *, 6 >_I < *, 6 >_N$	{1,2,3,4,6,7,8,9,11,12,13,14}	data,control
$< *, 5 >_I < *, 5 >_N$	{1,2,3,4,5,6,7,8,9,11,12,13,14}	data
$< *, 4 >_I < *, 4 >_N$	{1,2,3,4,5,6,7,8,9,11,12,13,14}	solved queries

Approach 2: Slicing request: $< [30], 14 >_Z$		
Query	Slice	Dependence
$< [30], 14 >_Z$	{14}	
$< [29], 13 >_Z < [29], 13 >_J$	{13,14}	data
$< [28], 4 >_Z < [28], 4 >_J < [4:28:8], 4 >_I < [4:28:8], 4 >_N$	{4,13,14}	control
$< [27], 12 >_Z < [27], 12 >_J < [3], 3 >_I < [11:27:8], 12 >_I < [3], 3 >_N < [11:27:8], 12 >_N$	{4,12,13,14}	data
$< [26], 11 >_Z < [10:26:8], 11 >_I < [2], 2 >_N < [10:26:8], 11 >_N$	{2,4,11,12,13,14}	data
$< [25], 10 >_Z < [9:25:8], 10 >_I < [9:25:8], 10 >_N$	{1,2,4,11,12,13,14}	data
$< [8:24:8], 9 >_Y < [8:24:8], 9 >_I < [8:24:8], 9 >_N$	{1,2,4,9,11,12,13,14}	data
$< [7, 23], 7 >_X < [15], 8 >_X < [7, 23], 7 >_I < [15], 8 >_I < [7, 23], 7 >_N < [15], 8 >_N$	{1,2,4,7,8,9,11,12,13,14}	data
$< [6:22:8], 6 >_X < [6:22:8], 6 >_I < [6:22:8], 6 >_N$	{1,2,4,6,7,8,9,11,12,13,14}	data,control
$< [5:21:8], 5 >_I < [5:21:8], 5 >_N$	{1,2,4,5,6,7,8,9,11,12,13,14}	data
$< [4:28:8], 4 >_I < [4:28:8], 4 >_N$	{1,2,4,5,6,7,8,9,11,12,13,14}	solved queries

Approach 3: Slicing request: $< [30], 14 >_Z$		
Query	Slice	Dependence
$< [30], 14 >_Z$	{14}	
$< [29], 13 >_Z < [29], 13 >_J$	{13,14}	data
$< [28], 4 >_Z < [28], 13 >_J < [28], 4 >_I < [28], 4 >_N$	{4,13,14}	control
$< [27], 12 >_Z < [27], 12 >_J < [27], 12 >_I < [27], 12 >_N$	{4,12,13,14}	data
$< [26], 11 >_Z < [26], 11 >_I < [26], 11 >_N$	{4,11,12,13,14}	data
$< [25], 10 >_Z < [25], 10 >_I < [25], 10 >_N$	{4,11,12,13,14}	data
$< [24], 9 >_Y < [24], 9 >_I < [24], 9 >_N$	{4,9,11,12,13,14}	data
$< [23], 7 >_X < [23], 7 >_I < [23], 7 >_N$	{4,7,9,11,12,13,14}	data
$< [22], 6 >_X < [22], 6 >_I < [22], 6 >_N$	{4,6,7,9,11,12,13,14}	control
...	...	
$< [3], 3 >_I < [3], 3 >_N$	{2,4,5,6,7,9,11,12,13,14}	data
$< [2], 2 >_N$	{1,2,4,5,6,7,9,11,12,13,14}	data

Figure 11: Implementing Agrawal and Horgan's dynamic slicing algorithms.

The example in Figure 12 illustrates currency determination. Assuming that block 2 contains the last use of the value of variable X, the second assignment to X in block 1 can be moved to block 2 by the partial dead code elimination optimization. During the execution of the optimized code the user may request the value of X at a breakpoint placed in block 3. Depending upon the path taken to arrive at the breakpoint the value of X may or may not be current. The WPP captures the path history and therefore allows us to make the correct assertions as shown in Figure 12. The determination requires propagating a query to locate the definition of X that reaches block 3 and then ensuring that this is the same definition that would have provided the value of X in the unoptimized code.

5. CONCLUDING REMARKS

In this paper we have demonstrated the following.

- WPPs can be *compacted* without compromising accessibility to profile data. The proposed techniques of *redundant path trace elimination* and *dynamic basic block dictionary creation* effectively compact the WPP without compromising accessibility.
- An organization of trace information based upon the *dynamic call graph* and *timestamped dynamic basic blocks* is particularly appropriate for performing profile-limited data flow analysis. This representation is compact and provides rapid access to path traces of a given function.
- Overall our techniques were observed to compact the original traces by factors ranging from 7 to 64 and at the same time speedups of over 3 orders of magnitude were observed in responses to queries requesting all of the trace information of a given function.
- We presented *profile-limited data flow analysis* for GEN-KILL problems and demonstrated its application to data flow frequency analysis for profile-guided optimization as well as debugging of programs.

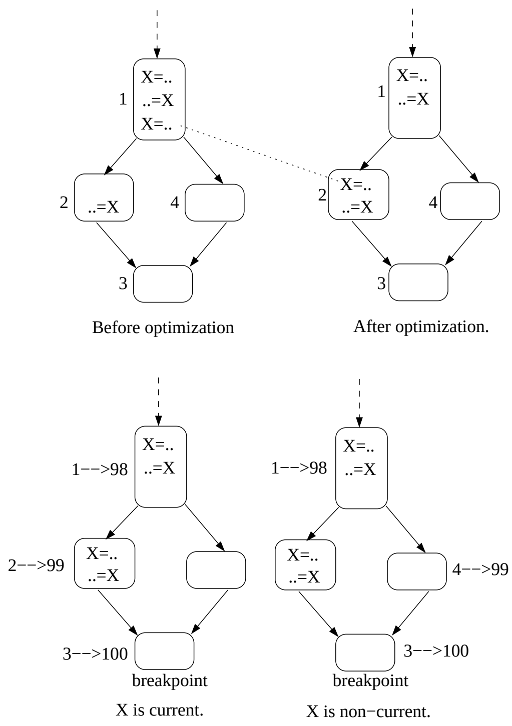

Figure 12: Detecting dynamic currency.

6. REFERENCES

[1] H. Agrawal and J.R. Horgan, "Dynamic Program Slicing," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246-256, White Plains, NY, June 1990.

[2] G. Ammons and J.R. Larus, "Improving Data Flow Analysis with Path Profiles," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 72-84, Montreal, Canada, 1998.

[3] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A Transparent Runtime Optimization System," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1-12, Vancouver, Canada, June 2000.

[4] T. Ball and J. Larus, "Efficient Path Profiling," *29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 46-57, Paris, France, 1996.

[5] T. Ball, P. Mataga, and M. Sagiv, "Edge Profiling Versus Path Profiling: the Showdown," *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 134-148, San Diego, CA, 1998.

[6] R. Bodik, R. Gupta, and V. Sarkar, "ABCD: Eliminating Array Bounds Checks on Demand," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 321-333, Vancouver B.C., Canada, June 2000.

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

[8] R. Bodik, R. Gupta, and M.L. Soffa, "Load Reuse Analysis: Design and Evaluation," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 64-76, Atlanta, Georgia, May 1999.

[9] E. Duesterwald, R. Gupta, and M.L. Soffa, "Demand-Driven Computation of Interprocedural Data Flow," *ACM Transactions on Programming Languages and Systems*, Vol. 19, No. 6, pages 992-1030, November 1997.

[10] D.M. Dhamdhere and K.V. Sankaranarayanan, "Dynamic Currency Determination in Optimized Programs," *ACM Transactions on Programming Languages and Systems*, Vol.. 20, No. 6, pages 1111-1130, November 1998.

[11] J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, C-30:478-490, 1981.

[12] T. Gross and P. Steenkiste, "Structured Dataflow Analysis for Arrays and its use in an Optimizing Compiler," *Software - Practice and Experience*, Vol. 20, No. 2, pages 133-155, Feb. 1990.

[13] R. Gupta, D.A. Berson, and J.Z. Fang, "Path Profile Guided Partial Dead Code Elimination using Predication," *International Conference on Parallel Architecture and Compilation Techniques*, San Francisco, CA, 1997.

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

[15] R. Gupta, D. Berson, and J.Z. Fang, "Resource-Sensitive Profile-Directed Data Flow Analysis for Code Optimization," *30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 558-568, Research Triangle Park, North Carolina, December 1997.

[16] J. Knoop, O. Ruthing, and B. Steffen, "Optimal Code Motion: Theory and Practice," *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 4, pages 1117-1155, 1994.

[17] B. Korel and J. Laski, "Dynamic Program Slicing," *Information Processing Letters*, 29:155-163, October 1988.

[18] J. Larus, "Whole Program Paths," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259-269, Atlanta, GA, May 1999.

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

[20] G. Ramalingam, "Data Flow Frequency Analysis," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 267-277, Philadelphia, PA, May 1996.

[21] T. Reps, S. Horwitz, and M. Sagiv, "Precise Interprocedural Data Flow Analysis via Graph Reachability," *22nd ACM Symposium on Principles of Programming Languages*, pages 49-61, 1995.

[22] B. Steffen, "Property Oriented Expansion," *International Static Analysis Symposium, LNCS 1145, Springer Verlag*, pages 22-41, Germany, September 1996.

[23] M. Smith, "Better Global Scheduling Using Path Profiles," *31th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 115-126, Dallas, Texas, Nov.-Dec. 1997.

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

[25] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, SE-10(4):352-357, July 1984.

[26] T.A. Welch, "A Technique for High-Performance Data Compression," *IEEE Computer*, pages 8-14, June 1984.

[27] X. Yuan, R. Gupta, and R. Melhem, "Demand-Driven Data Flow Analysis for Communication Optimization," *Parallel Processing Letters*, Vol. 7, No. 4, pages 359-370, December 1997.

[28] J. Ziv and A. Lempel, "A Universal Algorithm for Data Compression," *IEEE Transactions on Information Theory*, Vol. 23, No. 3, pages 337-343, May 1977.

[29] J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Transactions on Information Theory*, Vol. 24, No. 5, pages 530-536, September 1978.