Unified Control Flow and Data Dependence Traces

SRIRAMAN TALLAM
University of Arizona
and
RAJIV GUPTA
University of California, Riverside

We describe the design, generation, and compression of the *extended whole program path* (eWPP), representation that not only captures the control flow history of a program execution but also its data dependence history. This representation is motivated by the observation that, typically, a significant fraction of data dependence history can be recovered from the control flow trace. To capture the remainder of the data dependence history, we introduce *disambiguation checks* in the program whose control flow signatures capture the results of the checks. The resulting extended control flow trace enables the recovery of otherwise irrecoverable data dependences. The code for the checks is designed to minimize the increase in program execution time and the extended control flow trace size when compared to directly collecting control flow and address traces. Our experiments show that compressed eWPPs are only one-quarter of the size of combined compressed control flow and address traces. However, their collection incurs a 5× increase in runtime overhead relative to the overhead required for directly collecting the control flow and address traces, respectively.

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

General Terms: Algorithms, Measurement, Performance

Additional Key Words and Phrases: Control flow trace, address trace, dynamic data dependence trace, profiling

ACM Reference Format:

Tallam, S. and Gupta, R. 2007. Unified control flow and data dependence traces. ACM Trans. Archit. Code Optim. 4, 3, Article 19 (September 2007), 31 pages. DOI = 10.1145/1275937.1275943 http://doi.acm.org/10.1145/1275937.1275943

A preliminary version of this paper appeared in PACT 2005 [Tallam et al. 2005]. This work was supported by Microsoft and NSF grants CNS-0719791, CNS-0708199, CNS-0614707, and CCF-0541382.

Authors' addresses: Sriraman Tallam, University of Arizona, Tuscon, Arizona 85721; email: tmsriram@cs.arizona.edu or tmsriram@gmail.com. Rajiv Gupta, University of California, Riverside, Riverside, California 92502; email: gupta@cs.ucr.edu

1. INTRODUCTION

Execution traces have been collected and analyzed for a wide range of applications, such as developing new optimizations, developing new architectural techniques, and producing reliable software through testing and debugging. Three common types of traces that are useful in the above applications are control flow, dependence, and address traces.

1.1 Control Flow Trace

A control flow trace captures the complete path followed by a program during an execution. It is represented as a sequence of basic block ids (or Ball–Larus path ids [Ball and Larus 1996]) visited during the program execution. These traces can be analyzed to determine execution frequencies of shorter program paths [Larus 1999]. Thus, hot paths in the program can be identified and this knowledge has been used to perform *path-sensitive instruction scheduling* and *optimization* by compiler researchers [Ammons and Larus 1998; Bodik et al. 1998; Gupta et al. 1998; Young and Smith 1998] and *path prediction* and *instruction fetching* by architecture researchers [Jacobson et al. 1997]. Larus demonstrated that complete control flow traces of reasonably long program executions can be collected and stored by developing the compressed representation called the *whole-program path* (WPP) [Larus 1999].

1.2 Dependence Trace

Dependence (data and control) traces have also been used. Compiler researchers have used these profiles for performing *data speculative optimizations for Itanium* [Lin et al. 2003, 2004], speculative optimizations [Chen et al. 2004], and computation of dynamic slices [Weiser 1982; Agrawal and Horgan 1990; Korel and Laski 1988; X. Zhang and Gupta 2004]. The latter have been used for software *debugging* [Agrawal et al. 1993; Korel and Rilling 1997; X. Zhang et al. 2005, 2006], *testing* [Kamkar 1993], and providing security. Architecture researchers have used slicing to study the characteristics of *performance degrading load instructions* [Zilles and Sohi 2000], *thread creation using slicing* [Liao et al. 2002], and *studying instruction isomorphism* [Sazeides 2003]. The dependence history can be collected as follows. The control dependence trace can be *recovered* by analyzing the control flow trace. Every *register data dependence* (i.e., flow of a value from a *def* to a *use* through a register) can also be recovered using the control flow trace. However, recovery of each and every *memory dependence* (i.e., flow of a value through memory—the flow from a store to a load) requires detection of def-use information [Agrawal and Horgan 1990]. This trace is very long because each dependence must identify the statements and their execution instances involved in the dependence. Essentially, since dependence traces are at the granularity of statements while control traces are at the granularity of basic blocks (or Ball–Larus paths), dependence traces are longer than control flow traces.

Alternatively, the dependence information can also be captured by capturing the memory addresses referenced by every load and store instruction during execution. The data dependences can then be *recovered* from the address trace

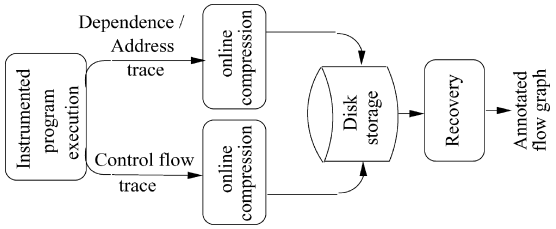


Fig. 1. Trace generation, storage, and use.

by using it in conjunction with the control flow trace and analyzing these traces off-line. Again, address traces are very long, since they are at the granularity of statements.

Let us first briefly see how control flow and data dependence traces are explicitly represented. There are two kinds of explicit representations: those that are more appropriate to use when the traces are *stored on disk*, such as the Sequitur [Nevil-Manning and Witten 1997] compressed control flow trace representation called the whole-program path [Larus 1999] and those that are used when traces are *held in memory* for analysis, such as the timestamped representations of control flow traces [Y. Zhang and Gupta 2001] and dependence traces [X. Zhang and Gupta 2004]. Recent work [Zhao et al. 2006] has shown how to capture a complete profile of a program's control flow, memory reference, and dependence information by exploiting the fact that most of the information can be retrieved by recording the register value changes and using this in conjunction with the control flow trace. In this paper, we develop an approach for producing a compact representation of the traces that are stored on disk as shown in Figure 1. We also discuss recovery algorithms that process these traces stored on disk and convert them to explicit representations that can be held in memory for analysis.

The combination of control flow and dependence traces is useful for a wide range of applications. In fact, frequencies of control flow edges/paths and data dependence edges/chains can be determined from these traces. However, the size of these traces can be quite large. Table I gives an idea of the sizes of the traces for sample runs. These runs were generated using the reference inputs of the **SPEC CPU 2000** integer benchmarks. The traces were collected for instruction counts of approximately between 350 and 400 million. The average length of the abridged control flow traces was around 90 million basic blocks and this corresponds to more than 1% of the trace of the entire run. It gives the sizes of the control flow and memory dependence traces, both when the dependence is captured explicitly (i.e., dependence traces) and implicitly (i.e., address traces). It also shows the factors by which they can be compressed. As we can see, the length of the memory dependence trace is significantly longer than the length of the control flow trace. Moreover, as shown, the compressibility of memory dependence traces using both Sequitur [Nevil-Manning and Witten 1997], a grammar-based compression algorithm, and VPC [Burtscher 2004], a value predictor-based compression algorithm, is quite inferior to that of control flow traces. The table also clearly shows that capturing address traces is superior

Table I. Trace Sizes and Compressibility

	Uncomp. (MB)			Compression Factor					
				Sequitur			VPC		
Program	Cont.	Dep.	Addr.	Cont.	Dep.	Addr.	Cont.	Dep.	Addr.
256.bzip2	154	540	590	57	1.37	4.2	61	5.3	8.4
186.crafty	184	604	638	77	1.53	37.1	25	5.7	17.2
252.eon	115	612	812	767	1.24	1242.0	610	8.3	153.2
254.gap	72	528	593	362	1.51	2.2	179	7.2	5.93
164.gzip	197	408	564	90	1.18	5.1	116	4.5	7
181.mcf	291	687	756	1265	1.18	17.9	3417	6.2	21.5
197.parser	226	642	680	161	1.49	10.5	221	6.1	19.1
253.perlbmk	185	537	652	1542	1.20	52.4	49	4.8	8.3
300.twolf	177	513	559	59	1.25	21.0	29	4.6	7.3
255.vortex	182	618	884	3033	1.26	46.8	113	6.2	16.8
175.vpr	186	525	599	78	1.20	21.7	38	4.8	7.7
Average	179	565	666	681	1.31	132.6	442	5.8	24.8

to capturing explicit dependence traces as they are not significantly larger and can be compressed by a larger degree. Hence, in the rest of the paper we use address traces as the baseline in our experiments.

Even though the address traces of some benchmark programs have good compressibility, overall, address traces do not get compressed as much as control flow traces. Thus, even if the address traces are compressed before being stored on disk, they can be quite long.

In this paper, we develop algorithms for generating an *extended control flow trace* that not only captures the dynamic control flow history but also the dynamic data dependence history. The memory dependences are not captured by an explicit representation of data dependences. Rather, memory dependences are implicitly embedded in the control flow trace. This representation of dynamic memory dependences is motivated by the observation that all dynamic register dependences can be recovered from the control flow trace. To capture the remainder of the dynamic data dependences, i.e., memory dependences, we present program transformations that introduce *disambiguation checks* and whose control flow signatures capture the results of these checks. The resulting extended control flow trace produced enables the recovery of otherwise irrecoverable memory dependences. Thus, our approach replaces the combination of a control flow and address trace with a single *extended control flow trace*, which is compressed to produce the *extended whole-program path* (eWPP) representation. The extended control flow trace is smaller and more compressible than the combination of the original control flow trace and the address trace. The program transformations are carefully designed to enable the capture of dynamic memory dependences at a reasonable cost in terms of the increase in program execution time and the control flow trace size.

Our experiments show that, on average, the sizes of compressed eWPPs are only 25% of the sizes of combined compressed WPP and address traces. Compared to the uncompressed control flow and address trace, the uncompressed extended control flow trace is 47% smaller, but incurs a 5× increase in runtime overhead.

The remainder of the paper is organized as follows. Section 2 describes our new extended trace and the program transformations needed to generate this trace. Section 3 describes some important optimizations that help in capturing and compressing the trace efficiently. Section 4 presents some enhancements to Sequitur that can improve the compression capability of control flow traces. Section 5 discusses how to recover the dependences from these traces and presents algorithms for the same. Section 6 presents results from the experiments we conducted. We conclude in Section 7.

2. EXTENDED WHOLE-PROGRAM PATHS

As the data presented in Table I shows, control flow traces are shorter in length than dependence and address traces. This is because control flow traces consist of a sequence of executed basic blocks (or paths), while dependence traces consist of def-use information, the dynamic memory dependences, and address traces consist of the memory addresses referenced at runtime. Each execution of a basic block or path may involve several memory references. Moreover, Sequitur-based compression techniques are very effective for control flow traces [Larus 1999], but significantly less so for dependence and address traces. While compression based on value predictors, VPC [Burtscher and Jeeradit 2003; Burtscher 2004], provides a greater degree of compression than Sequitur for dependence traces, this benefit comes at a price. Traces compressed using VPC have to be decompressed before they can be analyzed, unlike Sequitur, which produces the compressed trace in the form of a context-free grammar that can be readily analyzed. For instance, Larus [1999] has shown how to traverse the compressed control flow trace to find hot-subpaths.

The above observation motivated us to search for an alternative to the dependence/address trace. Note that the dependence/address trace is needed, because, when used in conjunction with the control flow trace, it enables the recovery of all dynamic memory dependences. The focus of this section is on designing an *extended control flow trace* representation from which we can extract dynamically exercised memory dependences. To enable the recovery of dynamic memory dependences, the extended trace should include additional information. We have the following goals in designing the extended trace representation:

- The additional information contained in the extended trace should be in the form of control flow so that the existing compression algorithm by Larus [1999] can be used to compress the extended trace.
- The incremental cost of generating the additional information should be minimized both in terms of the increase in the size of the trace and the increase in the program execution time because of the generation of the trace.

First, let us consider the additional information that is needed to recover the memory dependences from the control trace. Consider a path from def_1 to *use* that passes through def_2, as shown in Figure 2. Let us assume that memory dependences $e_1(def_1, use)$ and $e_2(def_2, use)$ are *potential* memory dependences, because of aliasing, that may or may not be manifested during a particular

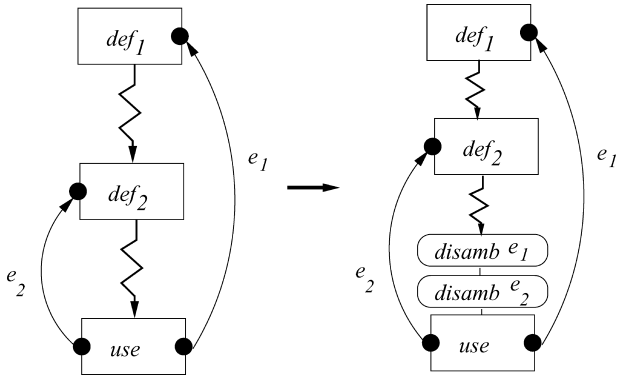


Fig. 2. Dynamic disambiguation.

execution of the path. While the control flow trace will capture each execution of the path, additional information on the addresses referenced by def_1, def_2, and *use* are needed to identify the dynamic memory dependences. Thus, immediately preceding the *use*, *dynamic disambiguation checks* are introduced: $disamb\ e_1$ compares the addresses referenced by def_1 and *use* while $disamb\ e_2$ compares the addresses referenced by def_2 and *use*. As we will see later, the control flow signature of the disambiguation checks captures the result of the comparison (true or false). Thus, the extended control flow trace (i.e., the original control flow trace augmented with the control flow signatures of the disambiguation checks) contains all the information needed to identify the dynamic memory dependences.

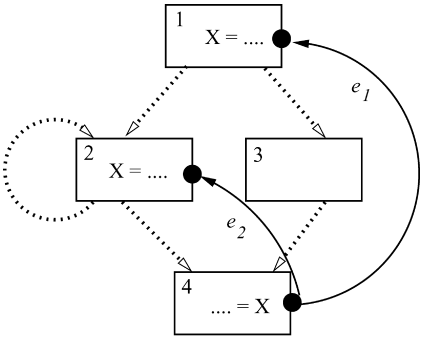
Given a set of potential memory dependences, to minimize the cost of the *disambiguation checks*, we classify each memory dependence into one of three categories: *no-cost*, *fixed-cost*, and *variable-cost* dependence. As the names suggest, the three categories differ in the cost needed for the disambiguation checks. Our program transformations to enable this classification and the collection of the memory dependence history are described next.

2.1 No-Cost Capture

In general, we need to introduce disambiguation checks to capture dynamic memory dependences. However, for a subset of dependences, disambiguation checks are not needed, as the outcomes of these checks can be determined directly from the control flow trace.

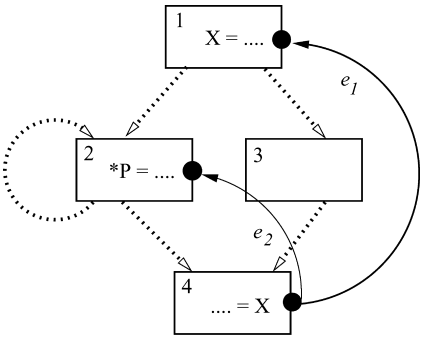
Definition 1. Fully free dependence. A def-use memory dependence is a *fully free dependence* iff under every execution of the program all occurrences of the dependence can be recovered from the program's control flow trace.

Figure 3 illustrates this situation. The two definitions and one use in this example always refer to the same variable, i.e., X. Moreover, for path 1.3.4, we are guaranteed that dependence edge e_1 is exercised and for all other paths that arrive at 4 via 2, dependence edge e_2 is exercised. Thus, the control flow trace is sufficient to identify these dependences when exercised.



(def(X:1),use(X:4))	1.3.4
(def(X:2),use(X:4))	$1.2^{+}.4$

Fig. 3. Fully free.



(def(X:1),use(X:4))	1.3.4
(def(X:1),use(X:4)) or (def(X:2),use(X:4))	$1.2^{+}.4$

Fig. 4. Partially free.

Definition 2. Partially free dependence. A def-use memory dependence is a *partially free dependence* iff, in general, only some occurrences of the dependence can be recovered from the program's control flow trace.

Figure 4 illustrates this situation. The definition in node 2 assigns a value through a pointer. Let us assume that a *points-to* analysis indicates that the pointer P may point to variable X. For path 1.3.4, we are guaranteed that dependence edge e_1 is exercised. However, for all other paths that arrive at 4 via 2, the dependence edge e_1 may or may not be exercised. Thus, the control flow trace only captures partial information for dependence edge e_1, i.e., when exercised through 1.3.4.

The presence of free dependences can be recognized at compile time as follows. First, given a *def* that reaches a *use*, the *def* and *use* must always refer to the same variable (say X). Next, if every path from the *def* to the *use* along

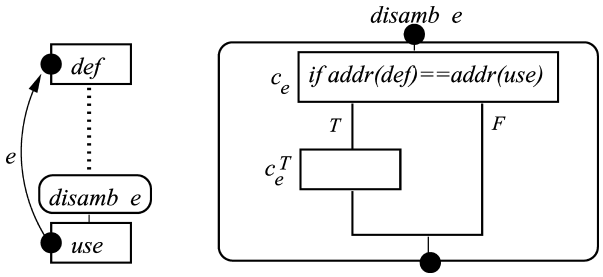


Fig. 5. Fixed cost check.

which the dependence can be exercised is *definition clear* w.r.t X, then the dependence (def, use) is fully free. If the preceding condition is only true for a subset of paths from *def* to *use* (i.e., along at least one of the paths, a definition of a *may-alias* of X is encountered), then this dependence (def, use) is partially free.

2.2 Fixed-Cost Capture

Free capture is only possible when *def* and *use* are guaranteed to refer to the same address. If the *def* and *use* may, but not necessarily, refer to the same address, the disambiguation check must be performed at runtime. If the *def* always refers to the same variable (say X), while the *use* may or may not refer to X, we can introduce a *fixed-cost disambiguation check* to enable detection of instances of this dependence. By a fixed-cost check, we mean that every execution of the *use* will require a constant amount of additional work to perform the disambiguation check for the *def* and *use*, which is a comparison of the address of X with the address read by *use*.

Definition 3. Last-instance dependence. A def-use memory dependence is a *last-instance dependence* iff every occurrence of this dependence is caused by the latest execution of the definition statement prior to the execution of the use statement.

The reason why we can capture some dependences at a fixed cost is because they are *last-instance* dependences. If the *def* always refers to the same variable and if the *def* is executed multiple times prior to executing the *use*, only the last execution of the *def* is relevant to the executed *use* as the *def* assigns to the memory address every time and, hence, is a *last-instance* dependence.

A fixed-cost disambiguation check for dependence edge e, denoted as $disamb\ e$, has the form shown in Figure 5. The control flow signature of $disamb\ e$ is $(C_e.C_e^T)$ if the check finds an address match; otherwise it is (C_e). The key point to note is that the result of the disambiguation check is captured by its control flow signature and is incorporated in the extended control flow trace. There is no need to explicitly save the *def* information for this *use*, i.e., the dependence trace need not be collected.

The example in Figure 6 illustrates a situation in which fixed-cost checks are needed to capture the three memory dependences corresponding to the use in node 5. In this example, we assume that it is known that pointer P is not

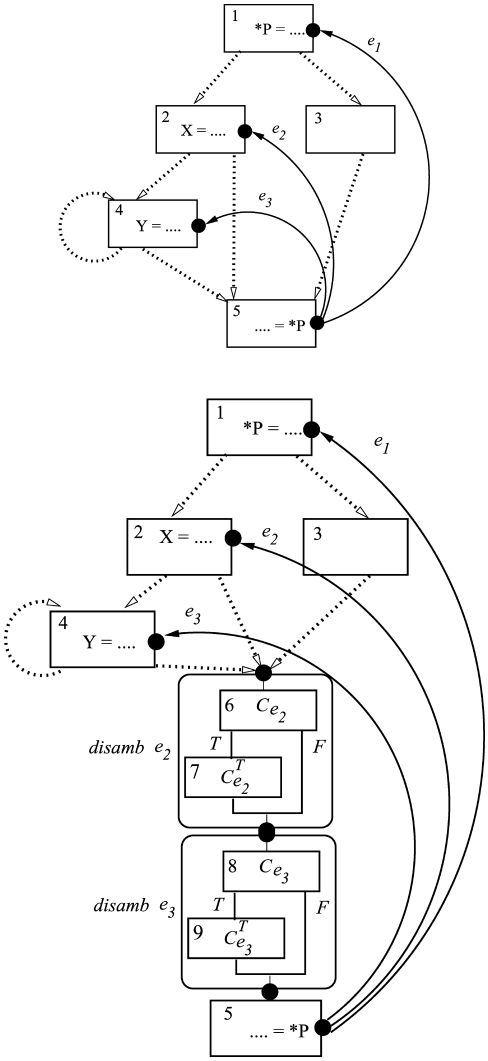


Fig. 6. Fixed-cost disambiguation.

assigned in the code fragment shown. Thus, the *def* in node 1 and the *use* in node 5 always refer to the same address. Assuming that P may or may not point to X or Y, disambiguation checks are needed to compare the addresses of X and Y with $*P$.

It should be noted that in the transformed program, each execution path from 1 to 5 uniquely identifies the exercised memory dependence edge. For example, consider the path 1.2.4.4.6.7.8.5. The disambiguation check signatures (6.7) and (8) indicate that P points to X not Y. The control flow 1.2 indicates that the *def* in node 2 is the latest definition of X before arriving at 5. Thus, we conclude that memory dependence edge e_2 is exercised along this path. Similarly, determinations can be made for all other paths.

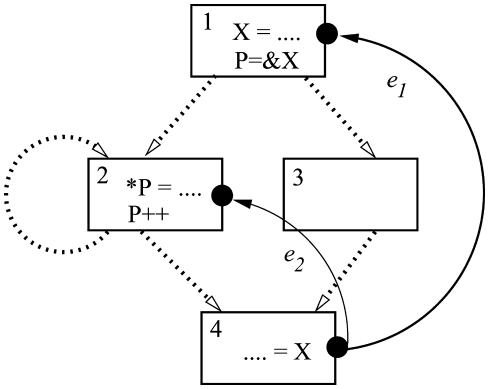


Fig. 7. Any-instance dependence.

2.3 Variable-Cost Capture

In the case of free dependences, both *def* and *use* were guaranteed to always refer to the same address, while in the case of fixed-cost dependences, only the *def* was always guaranteed to refer to the same address as the addresses referred to by the *use* could vary. Now, we consider the final case where both the *def* and *use* can refer to varying addresses.

This final situation is illustrated by the example in Figure 7. When the execution proceeds along path $1.2^{+}.4$ (2^{+} refers to one or more occurrences of 2), the value of X assigned through $*P$ in node 2 reaches the use of X in node 4. While the statements in node 2 may be executed several times, only the first execution of the definition assigns a value to X via $*P$. Thus, the dependence between the definition of $*P$ in node 2 and the use of X in node 4, denoted as $(*P : 2, X : 4)$, is not a last-instance dependence. In fact, by changing the assignment to $P = \&X$ in node 1, we can create situations where the dependence exists between *any-instance* of $*P : 2$ and $X : 4$.

Definition 4. Any-instance dependence. A def-use memory dependence is an *any-instance dependence* if and only if an occurrence of a dependence can be caused by any one of the executions of the definition statement prior to the execution of the use statement.

To capture any-instance dependences we need to do two things. First, all the addresses assigned to by the multiple executions of the definition must be saved in a buffer. Second, at the use, a *variable-cost* check shown in Figure 8 must be inserted. This check compares the *use address* with the *definition addresses* saved in the buffer, one at a time, starting from the latest address. The checks continue to be performed until a match is found or no more addresses remain in the buffer. The complete cost of this check is a variable as it can vary from a minimum of one check to as many checks as there are addresses in the buffer. The size of the buffer also continues to grow as the program executes. The example of Figure 7 once transformed using the *variable-cost* disambiguation results in the code shown in Figure 9.

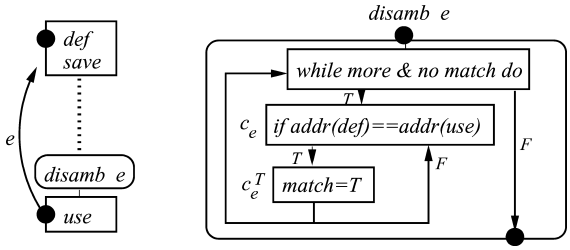


Fig. 8. Variable-cost check.

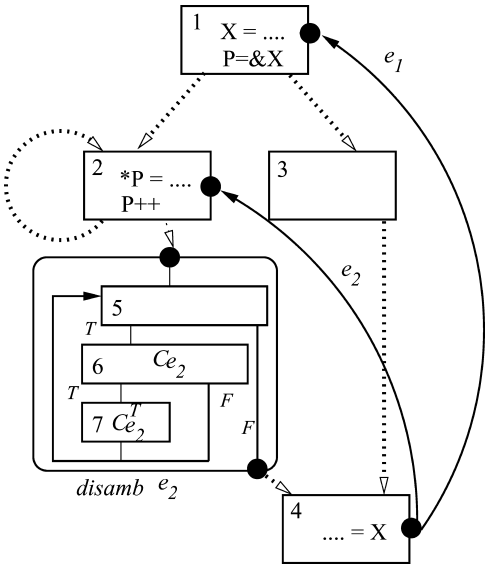


Fig. 9. Variable-cost disambiguation.

3. OPTIMIZING INSTRUMENTATION

In this section, we present a series of optimizations aimed at tuning the insertion and execution of instrumentation code so that the size of the instrumentation code, the space, time cost of executing it, and the compressibility of the resulting trace are improved.

3.1 Instrumentation Code Size

Thus far, in our discussion, we have assumed that all potential memory dependences are identified, classified, and the program is instrumented according to the classification. However, in practice, because of the conservative nature of static analysis, too many spurious memory dependence edges may be present, causing the cost of instrumentation to become very high. The unnecessary instrumentation will not only incur execution time overhead, but will also increase the length of the extended control flow trace and the cost of recovering memory dependences.

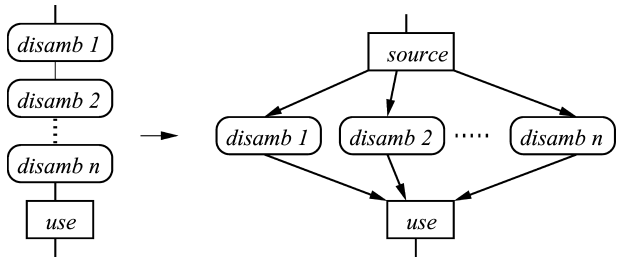


Fig. 10. Optimizing trace length by executing the disambiguation code of the correct store instruction.

To solve the above problem, we use a two-phase profiling scheme that consists of a *filtering phase* and a *collection phase*. In the filtering phase, the program is instrumented to identify all memory dependence edges that are exercised at least once during execution. Also, based upon their behavior, the dependences are classified as no-cost, fixed-cost, or variable-cost. Now that all actually encountered memory dependences have been identified, the program is instrumented only with the disambiguation checks that are needed to capture these dependences. The instrumented program is then run to collect the *extended whole-program path*. The instrumentation needed for the filtering phase is similar to the one used by Agrawal and Horgan [1990] for their second-approximate slicing algorithm—mapping between an address and the statement that defined it last is maintained to detect all exercised memory dependence edges.

This approach is not directly applicable in the presence of nondeterminism, since the second run on the same input may exercise some dependences that were not exercised during the first execution. The absence of instrumentation code for such dependences can cause such dependences to be missed. One solution to this problem is to conservatively introduce instrumentation code to capture all potential memory dependences. Another solution is to capture nondeterministic events in the first run and replay them using the second run so that no new dependences are exercised. For the programs considered in the experiments we performed, the above issue did not arise.

3.2 Trace Length and Compressibility

Each time a load is encountered, the disambiguation codes for all the corresponding stores (potential sources of the dependence) are executed. Therefore, the corresponding trace produced can be very long. A simple optimization can ensure that we only execute the disambiguation code for a single store. We can track the last store for each address at runtime and use it to quickly identify the source of the dependence. We can implement this by using a hash table that is indexed by the memory address and stores the identifier of the source statement that last wrote to this address. The instrumentation code for only this source is executed—the purpose of the trace produced is then to only identify the precise instance of this defining store. This optimization is shown in Figure 10. Note that not only the length of the trace produced is reduced, but also the cost of executing the instrumentation code.

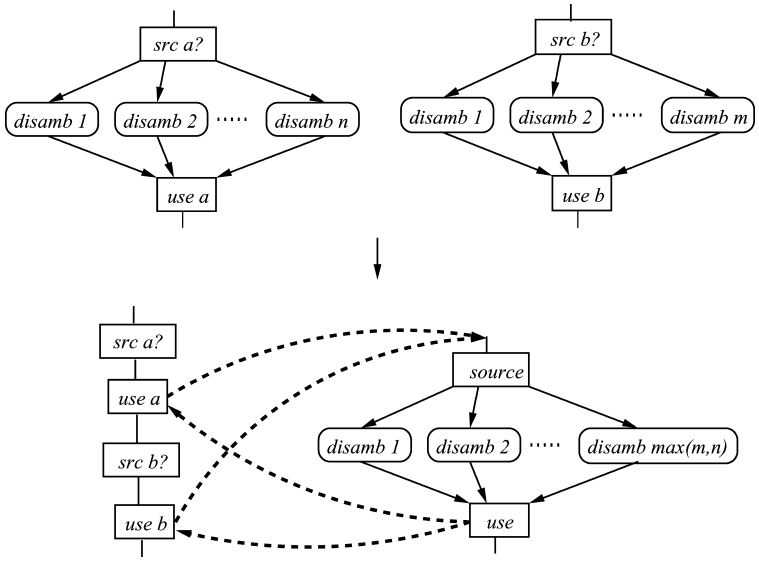


Fig. 11. Optimizing trace compressibility by executing a common piece of code.

Next, we consider another optimization that is aimed at sharing the instrumentation code across different uses (loads). This optimization not only reduces the overall size of the instrumentation code that is inserted, but also increases the compressibility of the trace produced by this instrumentation code. We create a single copy of the instrumentation code as shown in Figure 11. For each load, its corresponding stores are numbered from 1 to n ($\leq maxn$). At each load, the source of the dependence is determined and a call is made to the shared instrumentation providing the source id ($1 \leq id \leq n$) and a pointer to its corresponding buffer. The instrumentation code is then executed, producing traces such that traces for different loads now look similar, thus enabling a greater degree of compression. The control flow trace produced still uniquely identifies the dynamic memory dependences. By finding the source of the call to the instrumentation code from the control flow trace, we can determine which load execution is being processed. Then, by examining the control flow trace produced by the instrumentation code itself, we can know the source of the dependence (1 to n) and the specific execution instance of the source that is involved. Notice that the dependences are encoded implicitly in the trace using the control flow signatures and they need to be recovered to be used in later analysis. Section 5 discusses the algorithms that can be used to recover these dependences. The compressibility of the trace improves because each disambiguation involves executing a common piece of code and, hence, these basic blocks repeat in the extended control flow trace. Sequitur or VPC is then able to effectively capture these repetitions and compress them.

3.3 Reducing the Number of Checks

For the variable-cost transformation, the number of checks could be as high as the number of addresses stored in the buffer. This cost could significantly

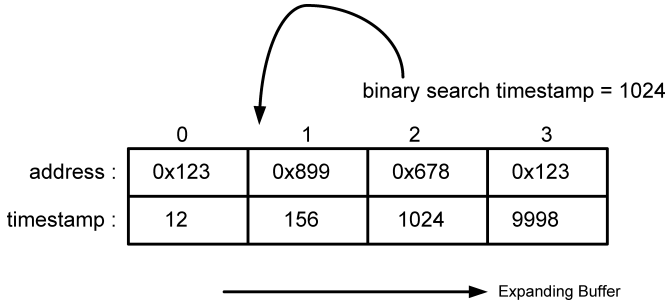


Fig. 12. Reducing the number of checks.

increase the runtime overhead. We greatly reduce this cost by using the following optimization. Instead of using a linear search, we adapt our buffer to allow binary search by saving along with the address, the global timestamp at which the address was written to by the store instruction. At runtime, we also track the global timestamp of the last write to each address. Now, at runtime, when a load is encountered, we look up the timestamp of the latest write to the address being referenced by the load. We search for this address in the buffer using the last write timestamp. Since the timestamps in the buffer appear in ascending order, we can employ binary search to find the relevant timestamp and, hence, determine the distance. Enabling linear search to be replaced by binary search greatly reduces the number of checks required. For instance, if 1 billion instructions are executed, the number of checks for each load cannot exceed $\log_2(1 \text{ billion}) = 30$. For the benchmark runs we considered, on average, we only needed ten checks for every dynamic dependence exercised.

Figure 12 illustrates the above approach. It shows a snapshot of a sample buffer. Let us say that a load corresponding to address $0x678$ is encountered. The last write information for the address will tell us that the timestamp at which the last write to this address was performed is 1024. Now we can search for time stamp 1024 using binary search as the timestamps appear in ascending order. Once the proper entry in the buffer is found, the distance can be determined. Figure 13 shows the code and its CFG that does this search. def_{TS} refers to the array of timestamps, which correspond to the different instances of the definition.

The extended control flow trace will include the control flow signatures from the binary search routine that is executed to capture every variable cost dependence, implicitly capturing the definition and its instance responsible for this dependence. Section 5 discusses how to recover this information from the trace.

3.4 Number of Buffer Entries

For the variable-cost transformation, the buffer used to save the addresses associated with a definition grows as the definition is repeatedly executed. We address this problem as follows. For every store instruction, we preallocate a buffer of size 200 K entries. In addition, we put a check to detect if the buffer size is exceeded. If more buffer space is needed, we allocate a larger buffer that

```
BinarySearch(def_TS[ ], timestamp) {
    length = number of entries in def_TS;
    top = length, bot = 1, mid = (bot + top)/2;
    while (def_TS[mid] ≠ timestamp) do
    {
        if (def_TS[mid] < timestamp)
            B0 : top = mid − 1;
        else
            B1 : bot = mid + 1;
        mid = (bot + top)/2;
    }
    E : end while
}
```

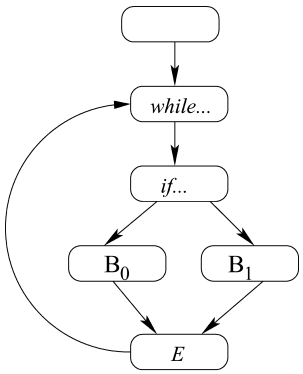


Fig. 13. Code for binary search.

is twice the size of the previous one and copy into it the past history from the old buffer. We selected a relatively large buffer size as we wanted to minimize the runtime overhead because of reallocations and copying. We found that the number of times copying was required is extremely small. On average, for the benchmarks we considered, the total amount of buffer space needed was less than 300 MB.

In the experiments we conducted, the traces were collected for around 400 million instruction executions. If we were to scale the tracing to larger instruction counts, we could run out of memory because of the space needed by this growing buffer. To overcome this problem, we propose the following solution. We can define a maximum buffer size for each store instruction. Once this is about to be exceeded, we can transfer a predetermined amount of the oldest buffer entries to disk to make room for new entries. This solution will require transfers between memory and disk. However, if the maximum buffer size is large, then such transfers will be infrequent as the dependences have a strong history bias. The entries that match are more likely to be closer to the tail of the buffer than its head. We conducted an experiment that confirmed this. For all benchmarks we considered, the matches occurred, on average, within the first 15% of the length of the buffer from the tail. Hence, this will be an effective solution to scale this approach.

4. ENHANCED SEQUITUR ALGORITHM

In this section, we discuss an enhancement we made to Sequitur to improve its compression capability. In particular, we found that this enhancement works very well on control flow traces.

4.1 Overview of Sequitur

Sequitur is a context-free grammar-based compression algorithm that exploits repetitions in the input string to compress it and runs in time linear in the length of the input string [Nevil-Manning and Witten 1997]. As an example, for the input string

abcabcabc

Sequitur produces the following grammar:

$$S \rightarrow AAA$$
$$A \rightarrow abc$$

In this example, Sequitur is effectively able to capture the repetition of the pattern "abc." The Sequitur algorithm manipulates the input symbols so that the following two properties are preserved.

1. *Digram uniqueness.* A digram is a pair of symbols that occur together in the input string. This property states that any digram "xy" can occur, at most, once in the entire grammar. If it occurs more than once, Sequitur introduces a new rule, of the form {R → xy}, that replaces both occurrences of the digram "xy" with the left-hand side of the rule, "R." For example, after "abca" has been processed in the example discussed above, the next symbol causes digram "ab" to occur twice. Thus, Sequitur produces a new rule $A \rightarrow ab$ and transforms the input into "AcA."
2. *Rule plurality.* This property states that the left-hand side of any rule must appear more than once in the entire grammar, on the right-hand side of the other grammar rules. If any rule occurs only once, then Sequitur deletes this rule and substitutes the rule occurrence with its right-hand side. In the example above, after "abcabc" is processed, the resulting grammar rules are $S \rightarrow BB$, $B \rightarrow Ac$, and $A \rightarrow ab$. Now, rule A occurs only once on the right-hand side of the entire grammar. Hence, to preserve this property, Sequitur expands rule $B \rightarrow Ac$ into $B \rightarrow abc$ and deletes the rule $A \rightarrow ab$.

4.2 Enhancement of Sequitur

We were motivated to enhance Sequitur by observing the way it compresses repeating digrams. We first give an example. Consider the input string

wbcwbcxbcxbcybcybczbczbc

Using Sequitur to compress this produces the grammar:

$$S \rightarrow AABBCCDD$$
$$A \rightarrow wbc$$
$$B \rightarrow xbc$$
$$C \rightarrow ybc$$
$$D \rightarrow zbc$$

Although Sequitur detected patterns like "wbc," it missed out the pattern "bc" that got repeated in each of the rules. A better grammar would have been:

$$S \rightarrow AABBCCDD$$
$$A \rightarrow wE$$
$$B \rightarrow xE$$
$$C \rightarrow yE$$

$$D \rightarrow zE$$
$$E \rightarrow bc$$

Notice that the above grammar contains fewer symbols than the first one. The magnitude of this saving grows with the size of the input string.

The reason why Sequitur failed to detect the pattern "bc" is because it first detected the digram "wb," which killed the digram "bc." Had we processed this string offline, instead of an online left to right method that Sequitur uses, and first compressed digram "bc," then we could have generated the ideal grammar for this string. Based upon this observation, we propose the following enhancement to Sequitur. We first show how to generate better grammars by processing the string offline. Then, it is easy to see that the offline algorithm could be made online by buffering the input. Depending on the size of the buffer, a degree of approximation will, however, be introduced. In our experiments, we reported compression results by processing the string offline. Note that offline processing is equivalent to having a buffer whose size is the same as the size of the uncompressed trace.

We now describe our algorithm. We look at the entire string and find frequency counts of each digram occurring in the string. We then start compressing digrams in the descending order of their frequency. As and when a digram is substituted by a rule, the two properties of Sequitur are checked to make sure they are satisfied. In this process, digrams that are more frequent are substituted first rather than the earliest occurring digrams. This ensures that highly frequent digrams are not destroyed in the process of compressing infrequent digrams and, hence, promises smaller grammars. For instance, in the example above, digram bc is the most frequent. Thus, compressing it first gives rise to the grammar

$$S \rightarrow wExEyEzE$$
$$E \rightarrow bc$$

and further compression yields the ideal grammar for this string, already shown previously. The original Sequitur algorithm was modified to take into account these changes. Figure 14 shows the pseudocode for the new Sequitur algorithm.

We experimented with the enhanced Sequitur and found that it is very effective on control flow traces. Table II compares the compression ratios obtained by both the Sequitur versions. On average, enhanced Sequitur can compress the traces further by around 33%. Table II also compares the memory used by both versions of Sequitur and the runtime overhead of using enhanced Sequitur. The enhanced version uses ten times more memory and is five times slower, on average.

5. RECOVERING THE DEPENDENCE EDGES FROM THE TRACES

Thus far, we have discussed how to perform the disambiguation checks in order to capture the different memory dependences. The checks were designed so that the resulting trace is small and compressible and also the runtime overhead is low. Now, we discuss the next step, which is how to extract the memory dependences from these traces after they have been generated. Note that the

```
Enhanced_Sequitur( Trace ) {
    Step 1: Preprocess Trace to find frequency of all digrams
    Let T be the set of all ordered pairs (digram,frequency)
    sorted in descending order of frequency counts.
    Step 2: Compress the Trace
    while T is not empty
    → Pick the most frequent digram from T, say "xy".
    → Create a new rule A_i → xy.
    → Substitute all occurrences of "xy" with A_i.
    → Update the frequency counts of digrams in T. Add new
        ordered pairs for the newly created digrams. Delete
        ordered pair for digram "xy".
    → If any rule R_i violates the Sequitur property of rule
        plurality, expand the rule.
    end while
}
```

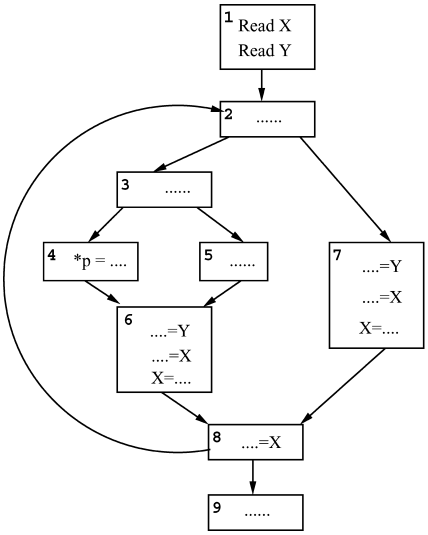
Fig. 14. Enhanced Sequitur algorithm.

Table II. Compression Ratio, Memory Used, and Runtime Overhead: Original Sequitur versus Enhanced Sequitur

	Compression Ratio		Memory Used		Runtime
Program	Original	Enhanced	Original	Enhanced	Enh./Orig.
256.bzip2	57	65	48 M	308 M	4.9
186.crafty	77	126	50 M	368 M	4.8
252.eon	767	1323	22 M	230 M	4.9
254.gap	362	1051	23 M	144 M	4.9
164.gzip	90	102	43 M	394 M	4.0
181.mcf	1265	1572	23 M	582 M	3.8
197.parser	161	297	36 M	452 M	4.5
253.perlbmk	1542	2728	22 M	370 M	4.9
300.twolf	59	73	54 M	354 M	4.8
255.vortex	3033	4100	21 M	364 M	3.4
175.vpr	78	93	46 M	372 M	4.8
Average	681	1048	35 M	358 M	4.5

implicit dependence representation in the extended control flow trace is designed so that the traces can be compactly stored on disk. However, when the dependences have to be used for analysis, they have to be converted into an explicit representation. In this section, we describe how to recover the dependence edges from the extended control flow trace and the address trace and the overhead involved in doing so.

We first discuss how the memory dependences are expressed as annotations on the static program representation. Such annotated reperesentations are very useful when these dependences have to be stored in memory for analysis and have been discussed in Y. Zhang and Gupta [2001] and X. Zhang and Gupta [2004]. We then discuss how to recover the memory dependences from the extended control flow trace and the address trace in order to annotate the static program representation.



Execution Trace

Time Stamp (t_s)	Control Flow Trace (CF)	Dependence Trace (DD)
1	1	···
2	2	···
3	3	···
4	5	···
5	6	Y(1,1), X(1,1)
6	8	X(6,1)
7	2	···
8	7	Y(1,1), X(6,1)
9	8	X(7,1)
10	2	···
11	3	···
12	4	p = &X
13	6	Y(1,1), X(4,1)
14	8	X(6,2)
15	9	···

Fig. 15. Control flow and dependence trace.

Consider the execution traces in Figure 15 in which the dependences are explicitly represented. The control flow trace CF gives the sequence of basic block ids executed. Let us assume that $*p$ corresponds to the contents of the address of X in this run ($p = \&X$). The dependence trace representation in Figure 15 is interpreted as follows. At $t_s = 14$, $X(6, 2)$ means that the use of variable X at this execution point was data dependent on the second execution instance of basic block 6. That is, the definition corresponding to the use at $t_s = 14$ comes from the second execution instance of basic block 6, which is the definition of X at $t_s = 13$. Given a *use* at some execution point, its corresponding *def*, which is the program statement and the instance, can be directly obtained from the dependence trace as the dependences are explicit. Now let us discuss how the dynamic control flow and dependences can be annotated on the static program representation. First, executions of basic blocks are assigned timestamps in the order of their execution. The column t_s of Figure 15 gives the timestamp values for the sample execution. Using these timestamps, the control flow trace can be annotated on the static control flow graph representation by labeling each basic block with the sequence of timestamps at which it was executed (see the timestamps prefixed by "B" in Figure 16). In Figure 16, the control flow edges are represented by dotted lines and the dependence edges are shown by bold lines. A dynamic dependence (data or control) is annotated by labeling a static dependence edge with a sequence of timestamp pairs such that the pair of timestamps identify the execution instances of the statements that were involved in the dynamic dependence. Figure 16 shows the labels that identify the dynamic memory dependences next to each dependence edge. Note that the annotated representation explicitly captures the control flow and data dependences exercised in a program run. We now describe the process of recovering the memory dependences and annotating the static program representation from the extended control flow trace.

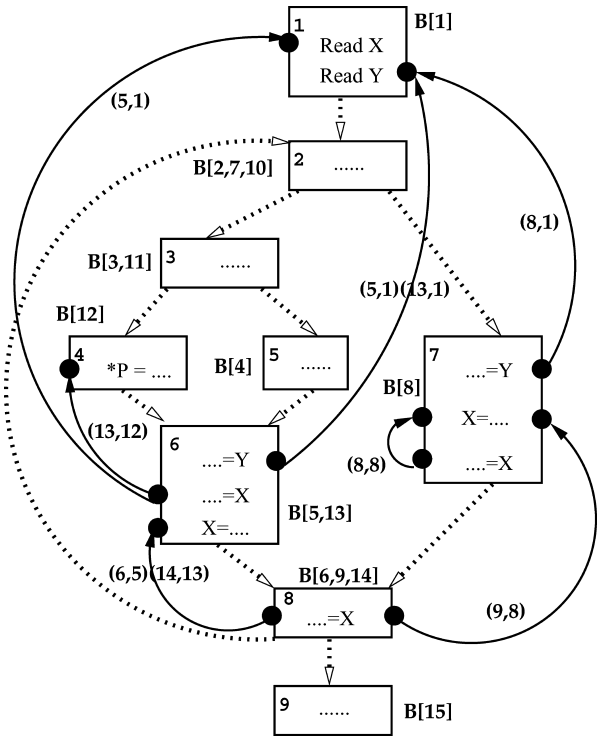
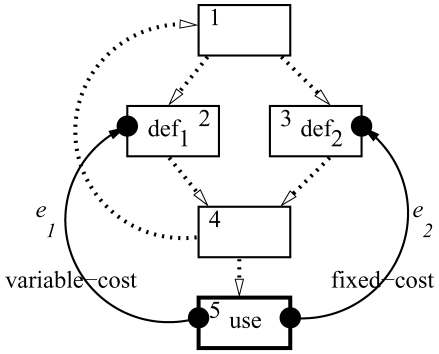


Fig. 16. Annotated trace representation.

Given the extended control flow trace, to recover the definition corresponding to a given execution of a use, we need to put together two types of information contained in the control flow signatures of the disambiguation checks that immediately preceded the use. The control flow signatures of disambiguation checks that contain the definition and the control flow of the binary search routine in Figure 13, which identifies the instance of the definition. Of particular importance is the ordering of the instances of the basic blocks B_0 and B_1. By putting these two pieces together, we can recover the definition and its instance that was involved in the dependence with the current instance of the use. The algorithm to do this is discussed next.

Consider the example shown in Figure 17. The disambiguation code preceding the use is not shown. The trace shows that prior to reaching the use in 5, def_1 is executed three times and def_2 is executed only once. Let the control flow signature of the disambiguation checks preceding 5 be $C_{e_1}.B_0.E$. The control flow signature contains C_{e_1}, which is the signature for def_1, and, hence, def_1 is the definition that produced the value. Also, look at the control flow signature of the binary search routine (Figure 13) preceding the use. Now, def_1 was executed three times in this example and, hence, the length of the timestamp array corresponding to def_1 is 3. The presence of B_0 in the disambiguation check indicates that an address match must have occurred at the first timestamp. Hence, we conclude that the first instance of def_1 must have been responsible for the



Control flow and addresses referenced:
1.2(Y).4.1.3(X).4.1.2(X).4.1.2(Z).4.5(Y)

Control flow signature of disambiguation checks before 5:
$(C_{e_1}.B_0.E)$

Fig. 17. Recovery example.

```
AnnotateControlFlow( ) {
    Let n be a node in the CFG
    time = 1;
    while not eof(trace) do
        n = getnextnode(trace);
        TS(n) = TS(n) ∪ {time};
        time++;
    end while
}
```

Fig. 18. Annotating CFG with control flow information.

dependence. Notice that, in extended control flow traces, the dependences are implicit. The dependence information is actually embedded in the control flow of the disambiguation checks. To recover the exact dependence, we need to interpret these disambiguation checks in the exact reverse of the process we used to embed them.

Next, we present the detailed algorithm for recovering a memory dependence, as illustrated by the above example. It should be recalled that our goal is to process the extended control flow trace and produce the annotations on the static program representation as shown in Figure 16. First we show how the control flow trace can be traversed and annotated on the static representation. As Figure 18 shows, each node n is annotated with a timestamp sequence $TS(n)$. The function $getnextnode()$ returns the identity of the node that was executed immediately after the current node. $TS(n)$ is the set of all timestamps at which node n was executed. *Before*() and *After*() in Figure 19 show how the static graph can be traversed in the reverse and forward direction of actual observed control flow using the $TS()$ annotations. Given an execution point $n(t)$, execution instance of node n at timestamp t, the function *Before*$(n(t))$ returns the identity of the node(n') that was executed immediately before $n(t)$, which is $n'(t-1)$.

$$\forall\ (t,n)\ \text{st}\ t \in TS(n)$$

$$Before(n(t)) = \begin{cases} \phi & if\ t = 1 \\ n'(t-1)\ \ elseif\ (n' \in Pred(n))\ \wedge\ (t-1 \in TS(n')) \end{cases}$$

$$After(n(t)) = \begin{cases} n'(t+1)\ \ if\ (n' \in Succ(n))\ \wedge\ (t+1 \in TS(n')) \\ \phi \qquad otherwise \end{cases}$$

Fig. 19. Definitions of Before and After functions to traverse the eCF along control flow edges.

```
RecoverDependence( u(t) ) {
    Step 1: Search for the definition of the dependence
    by looking for the disambiguation check.
    Let {d1,d2 ... dn} be the reaching definitions of u.
    n(tn) = Before(u(t));
    while n is not of type Ci
        n(tn) = Before(n(tn));
    end while
    /* n is of type Ci ⇒ definition is of type di */
    Step 2: Compute the instance of the definition
    by looking at the signature of binary search
    ArrayA = TS(di);
    length = count(A);
    if addr(di) =definitely addr(u) then
    /* A[length] is the instance, last instance of di */
        return (di(A[length]), u(t))
    top = length, bot = 1, mid = (top + bot)/2;
    n(tn) = After(n(tn));
    while n is not E
        if n is B0
            top = mid − 1;
        else
            bot = mid + 1;
        mid = (top + bot)/2;
        n(tn) = After(n(tn));
    end while
    /* A[mid] is the necessary instance of di */
    return (di(A[mid]), u(t));
}
```

Fig. 20. Recovering the dynamic memory dependence from the extended trace for use u executed at time t by replaying binary search.

Analogously, *After*$(n(t))$ returns the identity of the node(n') that was executed immediately after $n(t)$, which is $n'(t+1)$.

Now let us consider the memory dependence recovery algorithm in Figure 20. The first step in the algorithm examines the control flow signature of the disambiguation checks before the use to determine the definition that was involved in the dependence. This is done by traversing backward from the use point, $u(t)$, until a node whose signature is of the form C_i is found. The signature of this node exactly gives the definition, d_i, of this use. Once this is obtained, the

```
RecoverChain( u0(t0) → (d1, u1) → ··· (dn, un) → dn+1 ) {
  for i = 0 to n do
    (d(t), ui(ti)) = RecoverDependence(ui(ti))
    if d ≠ di+1 then return(nil) endif
    ti+1 = t;
  end for
  return (u0(t0) → (d1, u1)(t1) → ··· (dn, un)(tn) → dn+1(tn+1))
}

ChainFreq( u0(t0) → (d1, u1) → ··· (dn, un) → dn+1 ) {
 for each t ∈ TS(u0) do
  if RecoverChain( u0(t) → (d1, u1) → ··· (dn, un) → dn+1 ) ≠ nil
  then freq++ end if
 end for
 return (freq)
}
```

Fig. 21. Recovering a dynamic data dependence chain and obtaining its frequency.

second step looks at the control flow signature of the binary search routine to determine the exact instance of the definition that produced the value referenced by $u(t)$. To use this signature, we first need to reconstruct the array of timestamp values that was used to embed this signature. The array of timestamp values is already available in $TS(d_i)$. This array contains the timestamp values of execution instances of the definition d_i. Using the control flow signature of the binary search routine, this array is searched to obtain the right instance. The control flow signature gives the binary search order that we used on the timestamp array to embed the dependence and retracing this will give us the exact instance. When the definition and its instance are found, we have recovered the memory dependence. Note that this process is the exact reverse of the process used to embed the dependence. All memory dependences can be recovered and annotated on the program representation in this manner. While a single call to the RecoverDependence recovers a single dynamic memory dependence, we can build upon this function to develop additional functions, such as RecoverChain and ChainFreq, that are able to recover a *dependence chain* and the number of times a dependence chain is encountered during execution (see Figure 21). In Figure 21, d_1 is the definition corresponding to the use $u_0(t_0)$. u_1 is a use at statement d_1 and d_2 is the definition corresponding to use u_1, and so on. We use the RecoverDependence routine to check if a particular chain has been executed.

The algorithm for recovering a memory dependence using an address trace is shown in Figure 22. At every use point, we need to backtrack and find the latest definition that wrote to the same address as the use. Figure 22 shows a simple backtracking scheme. However, the backtracking can be implemented more efficiently by using a hash table as follows. For every definition we encounter, we can store the instance and the definition in the hash table indexed by the address. When we reach a use, we can look up in the hash table to find the latest definition and instance that wrote to the same address.

```
RecoverDependence( u(t) ) {
    Step 1: Back track and find the latest definition and its
    instance which wrote the same address as the use.
    Let the address at u(t) be addr(u)
    match = 0;
    while match ≠ 1
        n(t_n) = Before(u(t)) ;
        if n is a definition
            if addr(n) = addr(u)
                match = 1;
    end while
    return (n(t_n), u(t));
}
```

Fig. 22. Recovering the dynamic memory dependence from the address trace for use u executed at time t.

Table III. Register versus Memory Dependences

Program	Instructions (millions)	Register (%)	Memory (%)
256.bzip2	402	78.0	22.0
186.crafty	459	79.7	20.3
252.eon	378	72.0	28.0
254.gap	425	82.9	17.1
164.gzip	423	83.8	16.2
181.mcf	429	71.0	29.0
197.parser	415	74.2	25.8
253.perlbmk	354	72.2	27.8
300.twolf	405	79.2	20.8
255.vortex	418	69.4	30.6
175.vpr	407	77.5	22.5
Average	410	76.4	23.6

We compare the overheads of recovering the dependences from the address and extended control flow traces in the experimental section.

6. EXPERIMENTAL RESULTS

We have implemented our algorithms using the *Phoenix* Compiler Framework developed by *Microsoft*. The instrumentation code was inserted by using Phoenix to rewrite the binaries of the benchmark programs. The intermediate representation with which we worked was the low-level *x86* instruction set. This allowed us to clearly distinguish between register and memory dependences. Notice that the register dependences can always be detected directly from the control flow trace. Hence, the instrumentation was performed to capture memory dependences. This is important for carrying out a realistic evaluation as for the program runs used in our experiments, on average, 76.4% of all dependences were register dependences for program runs that execute hundreds of millions of instructions (see Table III). The SPEC CPU2000 C benchmarks were used to carry out the experiments (we had to exclude 176.gcc because the current version of Phoenix had problems with this benchmark). Two instrumented

Table IV. Uncompressed Trace Sizes

Program	CF + AT (MB)	eCF (MB)	eCF/CF + AT
256.bzip2	154 + 590 = 744	380	0.51
186.crafty	184 + 638 = 822	392	0.48
252.eon	115 + 812 = 927	414	0.45
254.gap	72 + 593 = 665	411	0.62
164.gzip	197 + 564 = 761	288	0.38
181.mcf	291 + 756 = 1047	735	0.70
197.parser	226 + 680 = 906	609	0.67
253.perlbmk	185 + 652 = 837	466	0.56
300.twolf	177 + 559 = 736	417	0.57
255.vortex	182 + 884 = 1066	500	0.47
175.vpr	186 + 599 = 785	318	0.41
Average	179 + 666 = 845	448	0.53

Table V. Sequitur Compressed Trace Sizes

Program	WPP + cAT (MB)	eWPP (MB)	eWPP/WPP + cAT
256.bzip2	2.4 + 142 = 144	46.8	0.32
186.crafty	1.5 + 17.2 = 19	11.6	0.60
252.eon	**0**.1 + **0**.65 = **1**	**6.0**	**6.00**
254.gap	0.1 + 270 = 270	2.2	0.01
164.gzip	2.0 + 110.5 = 113	28.6	0.25
181.mcf	0.1 + 42.2 = 42	16.7	0.40
197.parser	0.8 + 64.8 = 66	21.0	0.32
253.perlbmk	0.1 + 12.4 = 13	1.5	0.12
300.twolf	2.4 + 26.6 = 29	32.0	1.10
255.vortex	0.04 + 18.9 = 19	4.5	0.24
175.vpr	2.0 + 27.6 = 30	17.4	0.60
Average (excluding 252.eon)	1 + 74 = 75	18	0.24

versions of each binary were created apart from the original. The first version captured control flow and address traces. The second version captured extended control flow traces. Being able to produce the instrumented binaries of each of these using Phoenix allowed us to accurately measure the overheads involved in collecting these traces. We ran the binaries on a system with a 2 GHz Intel processor, 2 GB of RAM, and 100 GB of hard disk space. Based upon this implementation we carried out an experimental evaluation whose results are described next.

6.1 Trace Sizes

We first consider the various trace sizes. In Table IV, the sizes of the *uncompressed* control flow (*CF*), address (*AT*), and extended control flow (*eCF*) traces are given. The traces were collected, on average, for the first 400 million instructions. The corresponding *compressed* trace sizes, i.e., *WPP*, *cDD*, and *eWPP*, respectively, are also given in Tables V and VI. As we can see, on average, the *eCF* is smaller than $CF + AT$ by 47% while *eWPP* is smaller than $WPP + cDD$ by 76% and 70% using Sequitur and VPC, respectively. In other words, whether we use uncompressed or compressed traces, our extended control flow trace is superior to combined control flow and address traces. For 252.eon, using Sequitur, the

Table VI. VPC Compressed Trace Sizes

Program	WPP + cAT (MB)	eWPP (MB)	eWPP/WPP + cAT
256.bzip2	2.5 + 69.9 = 72	30.3	0.42
186.crafty	7.2 + 37 = 44	18.6	0.43
252.eon	0.19 + 5.3 = 5	2.1	0.42
254.gap	0.4 + 100 = 100	9.3	0.10
164.gzip	1.7 + 80.1 = 82	19.4	0.24
181.mcf	0.09 + 35.2 = 35	7.7	0.22
197.parser	1 + 35.6 = 37	20.9	0.56
253.perlbmk	3.8 + 78.3 = 82	18.4	0.22
300.twolf	6.1 + 76.4 = 83	39.4	0.47
255.vortex	1.6 + 52.7 = 54	12.2	0.23
175.vpr	4.9 + 77.8 = 83	24.8	0.30
Average	2.7 + 59 = 62	18.5	0.30

Table VII. Reason for Reduced *eWPP* Size When Compared to Address Trace

	Sequitur		VPC	
Program	Smaller *eCF* (%)	Comp. of *eCF* (%)	Smaller *eCF* (%)	Comp. of *eCF* (%)
256.bzip2	52	48	51	49
186.crafty	53	47	54	46
252.eon	56	44	55	45
254.gap	38	62	39	61
164.gzip	65	35	63	37
181.mcf	30	70	30	70
197.parser	34	66	34	66
253.perlbmk	44	56	45	55
300.twolf	45	55	46	54
255.vortex	53	47	53	47
175.vpr	61	39	61	39
Average	48	52	48	52

eWPP trace size obtained is larger, though not significantly. This aberration is because of the fact that the address trace for this program is highly compressible using Sequitur. The average size of the eWPP is calculated excluding 252.eon.

From the data in Table V, we can see that the reduced size of *eWPP* is a result of two factors. First, the size of *eCF* is smaller than the size of *CF* + *AT*, as a result of our novel representation of dependences. Second, Sequitur and VPC are extremely effective in compressing *eCF* into *eWPP*. Thus, we wanted to see what is the contribution of each of the two factors mentioned in reducing the trace size from *CF* + *AT* to *eWPP*. Table VII shows the results of this experiment. It shows that, on average, for Sequitur compressed traces, 48% of the reduction in trace size came from the first factor (shown under column *Smaller eCF*), i.e., going from *CF* + *AT* to *eCF*. The remaining 52% reduction came from the compression (shown under column *Compression of eCF*), as a result of going from *eCF* to *eWPP*. For VPC too, the contibutions because of both factors were the same. This shows that both the factors, representing the trace as *eCF* and then compressing it, are important in achieving smaller *eWPPs*.

Table VIII. Distribution of Memory Dependence Types

Program	No-Cost (%)	Fixed (%)	Varying (%)
256.bzip2	40.8	3.1	56.1
186.crafty	48.5	0.0	51.5
252.eon	18.8	16.7	64.5
254.gap	3.4	0.6	96.0
164.gzip	72	0.1	27.9
181.mcf	6.9	3.5	89.6
197.parser	9.8	5.6	84.6
253.perlbmk	21.3	0.7	78.0
300.twolf	29.4	8.2	63.4
255.vortex	22.3	1.5	76.2
175.vpr	60.9	3.0	36.1
Average	30.4	3.9	65.7

Table IX. Address Comparisons per Dep. Edge Using Linear and Binary Search

	Checks/ Dep.			
Program	Linear	Binary	Min	Max
256.bzip2	164814	11	1	24
186.crafty	18004	5	1	21
252.eon	35738	9	1	22
254.gap	661199	12	1	23
164.gzip	80493	8	1	22
181.mcf	194896	4	1	22
197.parser	107898	12	1	22
253.perlbmk	33341	8	1	23
300.twolf	170999	18	1	22
255.vortex	158386	10	1	23
175.vpr	26126	9	1	22
Average	150172	10	1	22

We also studied the distribution of three types of dynamic memory dependences: no-cost, fixed-cost, and varying-cost. The resulting data is given in Table VIII. From this data we can see that, on average, 65.7% of the dependences are hard dependences, i.e., varying-cost dependences. However, the number of no-cost memory dependences is also significant (average of 30.4%), which contributes directly toward reducing the size of *eCF*.

6.2 Runtime Overhead in Trace Collection

The execution time cost of the disambiguation checks is mainly because of the address comparisons performed. In particular, the greater the number of such comparisons, the greater the overhead. In Table IX, the average number of comparisons performed per dynamic data dependence is shown in the column named binary under *Checks*/*Dep*. These results were obtained by applying the optimizations described in Section 4. However, the most significant factor in keeping the number of checks small is using binary search instead of linear search. If we had not performed these optimizations, the number of checks needed at runtime would have then gone up by a significant amount, as shown

Table X. Running Time of Instrumented Versions (s)

Program	Original CPU	CF + AT (V_{AT}) CPU + IO	eCF (V_E) FP + CPU + IO
256.bzip2	5	26 + 22 = 48	118 + 160 + 69 = 347
186.crafty	5	28 + 29 = 57	94 + 96 + 21 = 211
252.eon	3	28 + 40 = 68	105 + 113 + 51 = 269
254.gap	3	19 + 27 = 46	170 + 214 + 55 = 439
164.gzip	5	31 + 15 = 46	65 + 71 + 16 = 152
181.mcf	7	30 + 35 = 65	116 + 135 + 21 = 272
197.parser	7	28 + 32 = 60	90 + 100 + 66 = 256
253.perlbmk	5	28 + 35 = 63	90 + 217 + 55 = 362
300.twolf	6	29 + 29 = 58	77 + 87 + 54 = 218
255.vortex	4	32 + 28 = 60	139 + 143 + 48 = 330
175.vpr	7	27 + 30 = 57	86 + 95 + 51 = 232
Average	5	28 + 29 = 57	105 + 131 + 46 = 282

in the column named linear under *Checks*/*Dep.*, making collection of these traces impractical.

Table X shows the runtime overhead needed to collect these traces. The running time of the three versions of each program, that is, the original version, the instrumented version for collecting control flow and address traces (V_{AT}), and the instrumented version for collecting extended traces (V_E) is shown. For V_E, the time spent in the filtering phase alone is shown as *FP*. Also, for versions V_{AT} and V_E, the time spent on processing (CPU) and IO are separately shown. The CPU time spent in V_E is higher than V_{AT}, coming from the checks needed per dependence. The numbers also show the overhead incurred in the filtering phase (FP). On average, there is a 5× increase in runtime overhead when collecting extended control flow traces compared to collecting control flow and address traces.

6.3 Dependence Edge Recovery

Table XI shows the time needed to recover the dependence information from the address and extended traces. As mentioned before, dependences in the extended control flow and address traces are implicitly represented. To be used in analysis, they need to be recovered and the numbers show the time needed to do the same. Notice that it is much harder to recover the dependences from the address traces. Although address traces are quicker to generate, they need, on average, ten times the time to process extended traces for recovering the dependences.

6.4 Decompressing the Traces

While the traces are compressed to be stored compactly on disk, in order to use the trace information in analysis, they need to be decompressed. In Table XII, we give the time taken to decompress compressed extended control flow and address traces. On average, Sequitur-compressed traces take longer, more than twice the time, to decompress than VPC compressed traces. The total time needed to recover the dependences explicitly from compressed traces is the sum

Table XI. Dependence Edge Recovery Time (s)

Program	CF + AT (R_{AT}) CPU + IO	eCF (R_E) CPU + IO
256.bzip2	146 + 5 = 151	18 + 4 = 22
186.crafty	164 + 4 = 168	11 + 5 = 16
252.eon	206 + 5 = 211	16 + 4 = 20
254.gap	212 + 5 = 217	14 + 4 = 18
164.gzip	75 + 4 = 79	15 + 4 = 19
181.mcf	157 + 6 = 163	21 + 4 = 25
197.parser	146 + 4 = 150	16 + 5 = 21
253.perlbmk	241 + 6 = 247	8 + 4 = 12
300.twolf	165 + 5 = 170	9 + 4 = 13
255.vortex	211 + 6 = 217	16 + 4 = 20
175.vpr	141 + 5 = 146	12 + 4 = 16
Average	169 + 5 = 174	14 + 4 = 18

Table XII. Decompression Times (s) for Compressed Traces

Program	Sequitur		VPC	
	WPP + cAT	eWPP	WPP + cAT	eWPP
256.bzip2	183	160	73	59
186.crafty	118	98	53	40
252.eon	201	131	49	69
254.gap	165	159	61	46
164.gzip	103	101	63	32
181.mcf	185	152	71	55
197.parser	257	243	119	79
253.perlbmk	250	193	95	75
300.twolf	242	227	112	82
255.vortex	160	117	83	50
175.vpr	195	168	83	92
Average	187	159	78	61

of the decompression time and the time needed to recover the dependences from the uncompressed traces shown in Table XI. Although there is a 5× increase in runtime overhead in collecting extended traces over collecting control flow and address traces, the time taken to collect, decompress, and recover the trace is of the same order (282 + 18 + 61 = 361 for extended traces and 57 + 174 + 78 = 307 for control flow and address traces).

We would like to point out that an alternative to decompression also exists for traces compressed with Sequitur. If the traces are compressed using Sequitur, they could be analyzed without decompression. This is possible because of the nature of the Sequitur compression algorithm, which compresses the trace into a context-free grammar. An algorithm for identifying hot paths of a specific length by analyzing the compressed control flow trace is given in Larus [1999]. A similar technique could also be developed for *eWPP* to recover dependences.

7. CONCLUSION

In this paper, we presented a unified trace representation for storing traces on disk that enables the capture of complete control flow and data-dependence

histories. The key problem that we solved in designing this unified trace is our ability to effectively convert dynamic memory data dependences between stores and loads into equivalent control flow trace information. The unified trace produced is smaller than the combination of control flow and address traces and has the dependences encoded implicitly in the form of control flow signatures. In order to use these dependences in analysis, we have presented algorithms for traversing our extended trace to recover data dependences or chains of data dependences. Such information is useful in carrying out a variety of code optimizations, as well as in other software engineering applications, such as dynamic program slicing.

ACKNOWLEDGMENTS

We would like to especially thank Hoi Vo of Microsoft Corp. for his support in both obtaining and using the Phoenix Research Development Kit (RDK). We also want to acknowledge the support provided by the entire Phoenix Compiler Infrastructure group at Microsoft in adapting our algorithms to work with Phoenix.

REFERENCES

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

AGRAWAL, H., DEMILLO, R., AND SPAFFORD, E. 1993. Debugging with dynamic slicing and backtracking. *Software Practice and Experience 23,* 6, 589–616.

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

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

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

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

BURTSCHER, M. AND JEERADIT, M. 2003. Compressing extended program traces using value predictors. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, Los Alamitos, CA. 159–169.

CHEN, T., LIN, J., DAI, X., HSU, W.-C., AND YEW, P.-C. 2004. Data Dependence Profiling for Speculative Optimization. In *Proceedings of the 13th International Conference on Compiler Construction*. 57–72.

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

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

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

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

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

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

LIAO, S. W., WANG, P. H., WANG, H., SHEN, J. P., HOFLEHNER, G., LAVERY, D. M., AND ZHANG, X. 2002. Post-Pass Binary Adaptation for Software Speculative Precomputation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 117–128.

LIN, J., CHEN, T., HSU, W.C., YEW, P.C., JU, R.D.C., NGAI, T.F., AND CHAN, S. 2003. A Compiler Framework for Speculative Analysis and Optimizations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 289–299.

LIN, J., HSU, W.C., YEW, P.C., JU, R.D.C., NGAI, T.F. 2004. A Compiler Framework for Recovery Code Generation in General Speculative Optimizations. In *14th International Conference on Parallel Architectures and Compilation Techniques*. 17–28.

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

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

TALLAM, S., GUPTA, R., AND ZHANG, X. 2005. Extended Whole Program Paths. In *14th International Conference on Parallel Architectures and Compilation Techniques*. 17–26.

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

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

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

ZHANG, X., HE, H., GUPTA, N. AND GUPTA, R. 2005. Experimental Evaluation of using Dynamic Slices for Fault Location. In *SIGSOFT-SIGPLAN Sixth International Symposium on Automated and Analysis-Driven Debugging*. ACM, New York. 33–42.

ZHANG, X., GUPTA, N. AND GUPTA, R. 2006. Pruning Dynamic Slices with Confidence. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York. 169–180.

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

ZHAO, Q., SIM, J.E., Wong, W.F., AND RUDOLPH, L. 2006. DEP: detailed execution profile. In *15th International Conference on Parallel Architectures and Compilation Techniques*. 154–163.

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

Received March 2006; revised August 2006 and December 2006; accepted February 2007