Extended Whole Program Paths *

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

Abstract

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 unrecoverable data dependences. The code for the checks is designed to minimize the increase in the program execution time and the extended control flow trace size when compared to directly collecting control flow and dependence traces. Our experiments show that compressed eWPPs are only 4% *of the size of combined compressed control flow and dependence traces and their collection requires* 20% *more runtime overhead than overhead required for directly collecting the control flow and dependence traces.*

1 Introduction

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

Control flow trace 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 [4]) visited during the program execution. These traces can be analyzed to determine execution frequencies of shorter program paths [12]. 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 [3, 5, 7, 18] and *path prediction* and *instruction fetching* by architecture researchers [8]. 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) [12].

Dependence trace Dependence, data and control, traces have also been used. Compiler researchers have used these profiles for performing *data speculative optimizations for itanium* [14, 15] and computation of dynamic slices [1, 9, 19]. The latter have been used for software *debugging* [2, 10], *testing* [11] and providing security by compiler researchers and architecture researchers have used slicing to study the characteristics of *performance degrading load instructions* [21], *thread creation using slicing* [13], and *studying instruction isomorphism* [17]. The dependence history can be collected as follows. The control dependence trace can be *recovered* by analyzing the control flow trace. Also, the *register data dependences* (i.e., flow of value from a *def* to a *use* through a register) can be recovered using the control flow trace. However, recovery of *memory dependences* (i.e., flow of value through memory – flow from a store to a load) requires detection of def-use information during run-time [1]. This trace is very long because each dependence must identify the statements and their execution instances involved in the dependence. Essentially, since the memory dependences 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.

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 [16] compressed control flow trace representation called the whole program path [12]; and those that are used when traces are *held in memory* for analysis such as the timestamped representations of control flow trace [20] and dependence trace [19]. In this paper, we develop an ap-

*Supported by grants from Microsoft, IBM, Intel and NSF grants CCR-0324969, CCR-0220262, CCR-0208756, CCR-0105535, and EIA-0080123 to the Univ. of Arizona.

proach for producing a compact representation of the traces stored on disk.

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 traces can be quite large. Table 1 gives an idea of the sizes of traces for sample runs. It gives the sizes of the control flow and dependence traces and factors by which they can be compressed. As we can see, the length of the dependence trace is significantly longer than the length of the control flow trace. Moreover, as shown, the compressibility of dependence trace using both Sequitur [16] and VPC [6] is quite inferior to that of control flow trace. Thus, even if the dependence traces are compressed before being stored on disk, they can be quite long.

Table 1. Trace sizes and compressibility.

Program	Uncomp. (MB)		Comp. Factor			
			Sequitur		VPC	
	Cont.	Dep.	Cont.	Dep.	Cont.	Dep.
256.bzip2	154	540	57	1.37	61	5.3
186.crafty	184	604	77	1.53	25	5.7
252.eon	115	612	767	1.24	610	8.3
254.gap	72	528	362	1.51	179	7.2
164.gzip	197	408	90	1.18	116	4.5
181.mcf	291	687	1265	1.18	3417	6.2
197.parser	226	642	161	1.49	221	6.1
253.perlbmk	185	537	1542	1.2	49	4.8
300.twolf	177	513	59	1.25	29	4.6
255.vortex	182	618	3033	1.26	113	6.2
175.vpr	186	525	78	1.2	38	4.8
Average	179	565	681	1.31	442	5.8

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 embedded implicitly 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* 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 trace and dependence 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 original control flow trace and the dependence trace. The enabling program transformations are carefully designed to enable the recovery of dynamic memory dependences at a small incremental cost in terms of program execution time increase and increase in control flow trace size.

Our experiments show that on an average the sizes of compressed eWPPs are only 4% of the sizes of combined compressed WPP and dependence traces. The uncompressed extended control flow trace is 38% smaller and can be captured with 20% more runtime overhead than the combined uncompressed control flow and dependence trace.

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 presents algorithms for recovering dynamic memory dependences from the extended control flow trace. Section 4 describes some important optimizations while Section 5 presents the experimental results. We conclude in section 6.

2 Extended Whole Program Paths

As the data presented in Table 1 shows, control flow traces are shorter in length than dependence 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. 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 [12] but significantly less so for dependence traces. While compression based on value predictors, VPC [6], 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 form of a grammar that can be readily analyzed. For instance [12] shows 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 trace. Note that the dependence 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* from which we can extract dynamically exercised memory dependences. To enable recovery of dynamic memory dependences the extended trace should include additional information. In designing this extended trace we have the following goals:

- The additional information contained in the extended trace should be in form of control flow so that the existing compression algorithm by Larus [12] can be used to compress the extended trace.
- The incremental cost of generating 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 due to 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 1. Let us assume that memory dependences $e_1(def_1, use)$ and $e_2(def_2, use)$ are *potential* memory dependences, due to the aliasing, that may or may not be manifested during a particular execution of the path. While the control flow trace will capture each execution of the path, additional information on the addresses referenced by the 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.

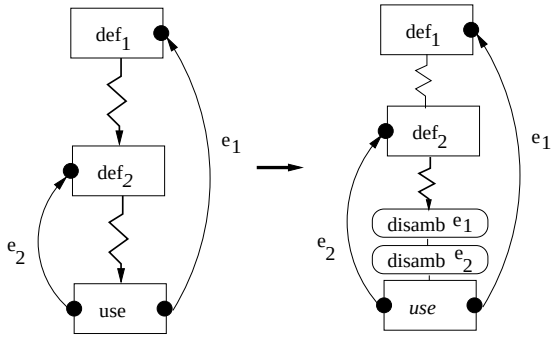


Figure 1. Dynamic disambiguation.

To minimize the cost of the *disambiguation checks*, we do the following. We first identify all the potential memory dependences for each use using static analysis. We then 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

While, 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 2 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.

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.

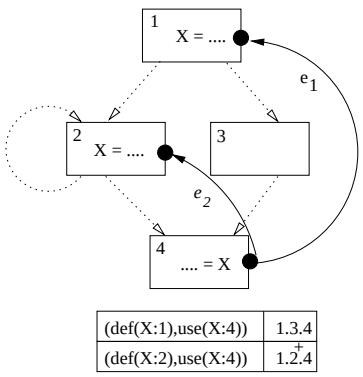
Figure 3 illustrates this situation. The definition in node 2 assigns a value through a pointer. Let use assume that a *points-to* analysis indicates that the pointer P may point to variable X or not. For path 1.3.4, we are guaranteed that dependence edge e_1 is exercised. However, for all other paths 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 is either *definition-clear* w.r.t X or if the def is definitely killed along the path, 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

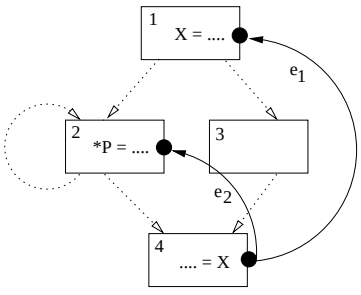
Free capture is only possible when the def and the use are guaranteed to refer to the same address. If the def and use may, but not necessarily, refer to the same address, the disambiguation check must be performed at run-time. If the def always refers to the same variable (say X), while the use may 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 the 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.



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

Figure 2. Fully-free.



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

Figure 3. Partially free.

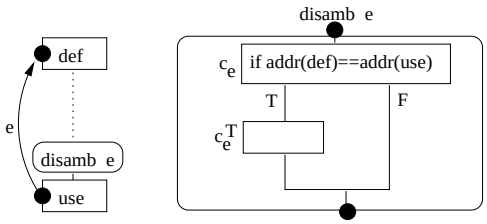


Figure 4. Fixed-cost check.

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 4. 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 5 illustrates a situation in which fixed-cost checks are needed to capture the three memory dependences corresponding the use in node 5. In this example we assume that it is known that pointer P is not 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/Y, disambiguation checks are needed to compare the addresses of X/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 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.

2.3 Variable-Cost Capture

In case of free-dependences both def and use were guaranteed to always refer to same address while in the case of fixed-cost dependences the def was always guaranteed to refer to the same address while 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 6. When the execution proceeds along path $1.2^{+}.4$, 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 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 7 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 continued to be performed till 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 the number of addresses in the buffer. The size of the buffer also continues to grow as the program executes. The example of Figure 6 once transformed using the *variable-cost* disambiguation results in the code shown in Figure 8.

3 Recovery Algorithms

In this section, we discuss the following. We first discuss how the memory dependences are recovered from the

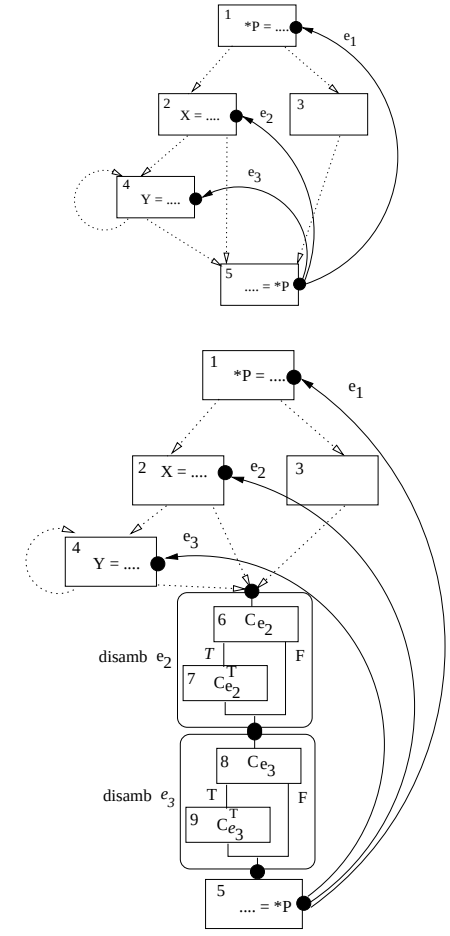


Figure 5. Fixed-cost disambiguation.

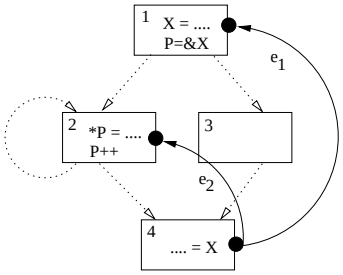


Figure 6. Any-instance dependence.

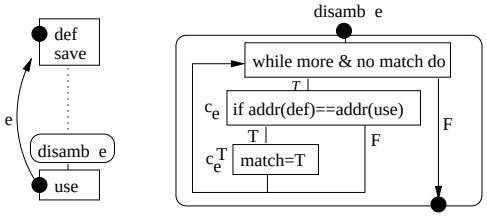


Figure 7. Variable-cost check.

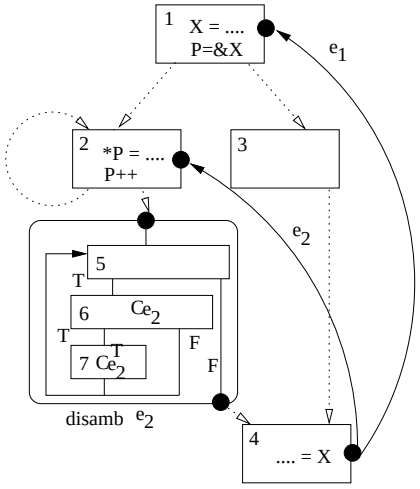
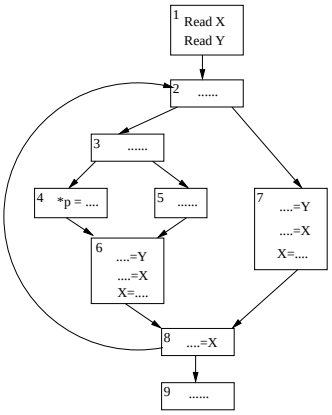


Figure 8. Variable-cost disambiguation.

control flow and dependence traces and then expressed as annotations on the static program representation. We then discuss how to recover the memory dependences from the extended control flow trace and annotate the static program representation.



Execution Trace

t_s	CF	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	...

Figure 9. Control flow & dependence trace.

Consider the execution traces in Figure 9. The control flow trace CF consists of the sequence of basic block ids executed and the data dependence trace DD gives the statement number (basic block - id in this example since no basic block has two stores to the same address) and the instance of the definition at every use. Note that when $*p$ is encountered, we assume that it corresponds to the contents of the address of X. For example, the definition corresponding to the use at $t_s = 14$ comes from the second execution instance of basic block 6, that is from the definition of X at $t_s = 13$. Given a *use* of a value stored at an address at some execution point, the corresponding *def*, the program statement and the instance can be directly obtained from the dependence trace. Now lets consider how the dynamic control flow and dependences can be annotated on the static

program representation. First executions of basic block are assigned timestamps in the order of their execution. The column ts of Figure 9 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 in Figure 10, the time stamps prefixed 'B'). 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 10 shows the labels that identify the dynamic memory dependences next to each dependence edge. We now describe the process of recovering the memory dependences and annotating the static program representation from the extended control flow trace.

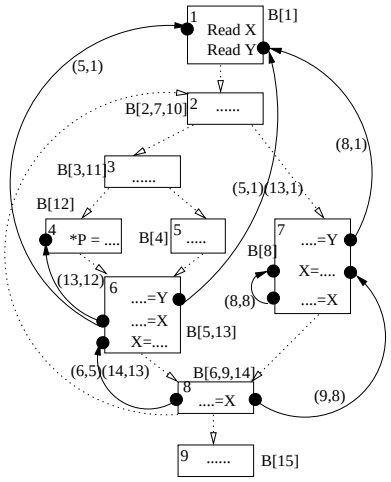
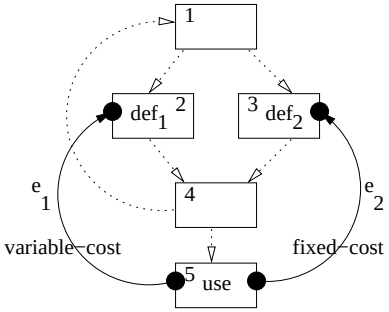


Figure 10. Annotated trace representation.

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 the trace: the control flow signatures of disambiguation checks that immediately precede a use; and the control flow information that identifies and orders the reaching definitions executed prior to reaching the use. The control flow signatures of disambiguation checks identify the definitions and their instances that could have produced the value referenced at the use. The control flow information identifies the order in which these potential definitions were executed and hence it enables us to identify the latest definition which is actually the one that is involved in the dependence.

Consider the example shown in Figure 11. The control flow trace shows that prior to reaching the use in 5, def_1 is executed 3 times and def_2 is executed once. The control flow signature produced preceding 5 indicates that the value produced could have been produced either by def_2 in 3 due to appearance of signature $(C_{e_2}, C^T_{e_2})$ or by the second execution of def_1 in 2 as indicated by signature $(C_{e_1}, C^T_{e_1})$). However, when we examine the preceding control flow trace we find that second execution of def_1 in 2 follows the execution of def_2 in 3. Thus, we conclude that memory dependence edge e_1 was exercised under this execution. Thus, we see that the extended control flow trace contains sufficient information to establish the dynamic memory dependences precisely.



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

Control flow signature of disambiguation checks before 5:
$(C_{e_1})(C_{e_2}\,C^T_{e_2})(C_{e_1}\,C^T_{e_1})(C_{e_1})$

Figure 11. Recovery example.

4 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 instrumentation code, the space and time cost of executing it, and the compressibility of resulting trace are improved.

4.1 Instrumentation Code Size

So 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, due to 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.

To solve the above problem we use two phase profiling consisting of the *filtering phase* and the *collection phase*. In the filtering phase the program is instrumented to identify all memory dependence edges that are exercised at least once during execution and in addition based upon their behavior we classify the dependences 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*. Instrumentation needed for the filtering phase is similar to one used in [1] for his second approximate slicing algorithm – mapping between addresses and statements that defined it last is maintained to detect all exercised memory dependence edges.

4.2 Trace Length & Compressibility

Each time a load is encountered, the disambiguation codes for all the corresponding stores (sources of the dependence) are executed. Therefore the corresponding trace produced can be long in length. A simple optimization can ensure that we simply 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. The instrumentation code for only this source is then executed – the purpose of the trace produced is then to simply identify the precise instance of the defining store that corresponds to this store. This optimization is shown in Figure 12. Note that not only the length of the trace produced is reduced, but also the cost of executing the instrumentation code.

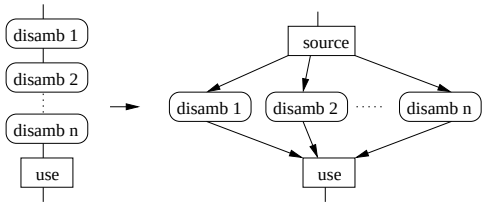


Figure 12. Optimizing Trace Length.

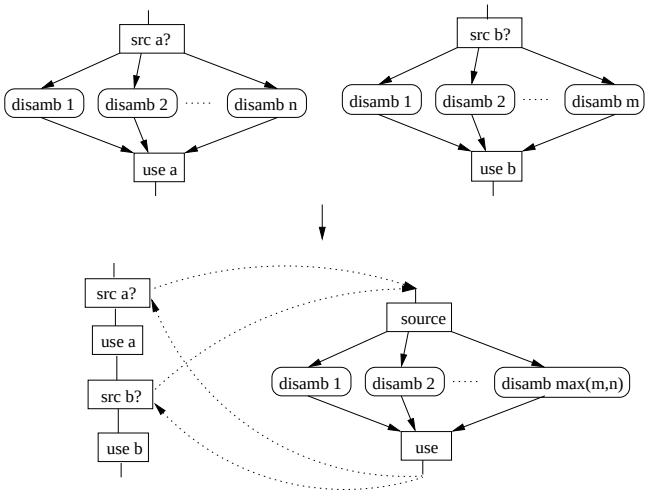


Figure 13. Optimizing Trace Compressibility.

Next we consider another optimization that is aimed at sharing the instrumentation code across different uses (i.e., 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 below. 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 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 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, by examining the control flow trace produced by the instrumentation code itself we know the source of the dependence (1 to n) and the specific execution instance of the source that is involved. The compressibility improves because each disambiguation involves executing a common piece of code and hence these basic blocks repeat in the extended control flow trace. Sequitur is then able to effectively capture these repetitions and compress them.

4.3 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 increase the runtime overhead. We greatly reduce this cost 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 lookup 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 lg(1 billion) = 30. For the benchmark runs we considered, on an average, we only needed 10 checks for every dynamic dependence exercised. The recovery algorithm can also be easily adapted to disambiguate using binary search rather than linear search.

Figure 14 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 shown for 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.

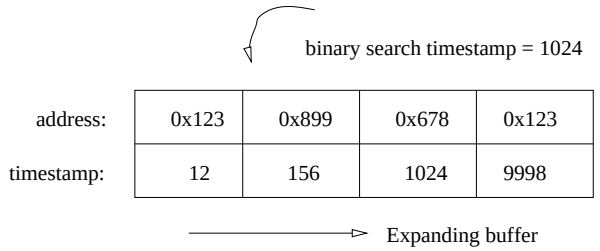


Figure 14. Reducing the number of checks.

4.4 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 200K entries. In addition, we put a check to detect if the buffer size is exceeded. If more buffer space is needed, we allocate a large buffer and copy into it the past history from the old buffer. We found that the number of times copying was required is extremely small. On an average, for all the benchmarks we considered, the total amount of buffer space needed was around 500 MB.

5 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. This was very convenient to implement in Phoenix. The Intermediate representation with which we worked was the low-level *x86* instruction set. This allowed us to clearly distinguish between register dependences 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 an average, 76.4% of all dependences were register dependences for program runs that execute in hundreds of millions of instructions (see Table 2). The SPEC 2000 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 versions of each binary was created apart from the original. The first version was to capture control flow traces and memory dependence traces. The second version was to capture 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. Based upon this implementation we carried out an experimental evaluation whose results are described next.

Table 2. Register vs. Memory dependences.

Program	Instructions (millions)	Register (%)	Memory (%)
256.bzip2	402	78 %	22 %
186.crafty	459	79.7 %	20.3 %
252.eon	378	72 %	28 %
254.gap	425	82.9 %	17.1 %
164.gzip	423	83.8 %	16.2 %
181.mcf	429	71 %	29 %
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 %

5.1 Trace Sizes

We first consider the various trace sizes. In Table 3, the sizes of *uncompressed* control flow (CF), data dependence (DD), extended control flow (eCF) traces are given. The corresponding *compressed* trace sizes, i.e. WPP, cDD, and $eWPP$ respectively, are also given. As we can see, on an average, the eCF is smaller than $CF + DD$ by 38% while $eWPP$ is smaller than $WPP + cDD$ by 96% with Sequitur and 79% with VPC. In other words, whether we use uncompressed traces or compressed traces, our extended control flow trace is superior to combined control flow and dependence trace.

Table 4. Reason for reduced $eWPP$ size.

Program	Sequitur		VPC	
	Smaller eCF (%)	Comp. of eCF (%)	Smaller eCF (%)	Comp. of eCF (%)
256.bzip2	59.8 %	40.2 %	47.3 %	52.7 %
186.crafty	51 %	49 %	51.5 %	48.5 %
252.eon	43.4 %	56.6 %	43.2 %	56.8 %
254.gap	31.6 %	68.4 %	32 %	68 %
164.gzip	53.6 %	46.4 %	52.7 %	47.3 %
181.mcf	25.3 %	74.7 %	25 %	75 %
197.parser	30.5 %	69.5 %	30.6 %	69.4 %
253.perlbmk	35.5 %	64.5 %	36.4 %	63.6 %
300.twolf	41.5 %	58.5 %	42 %	58 %
255.vortex	37.7 %	62.3 %	38.1 %	61.9 %
175.vpr	57 %	43 %	57.3 %	42.7 %
Average	42.4 %	57.6 %	41.5 %	58.5 %

From the data in Table 3 we can see that reduced size of $eWPP$ is due to two reasons. First the size of eCF is smaller than the size of $CF + DD$. This is because the dependence trace must store at every load, the instruction and its instance that produced the definition. For long traces, the number of bits needed to store this information can be very high. Table 4 shows the % by which eCF accounts for in reducing the size of $CF + DD$ and the % by which the compression of eCF accounts for in getting the final trace size to that of $eWPP$. This translates into 42.4% smaller

Table 3. Compressed trace sizes.

Program	Uncompressed			Compressed (Sequitur)			Compressed (VPC)		
	CF+DD (MB)	eCF (MB)	eCF/ CF+DD	WPP+cDD (MB)	eWPP (MB)	eWPP/ WPP+cDD	WPP+cDD (MB)	eWPP (MB)	eWPP/ WPP+cDD
256.bzip2	154 + 540 = 694	380	0.43	2.7 + 394 = 397	46.8	0.12	2.5 + 136 = 139	30.3	0.22
186.crafty	184 + 604 = 788	392	0.49	2.4 + 395 = 397	11.6	0.03	7.2 + 106 = 113	18.6	0.16
252.eon	115 + 612 = 727	414	0.57	0.15 + 494 = 494	6	0.01	0.19 + 74 = 74	2.1	0.03
254.gap	72 + 528 = 600	411	0.69	0.2 + 350 = 350	2.2	0.006	0.4 + 73 = 74	9.3	0.13
164.gzip	197 + 408 = 587	288	0.49	2.2 + 346 = 348	28.6	0.08	1.7 + 90= 92	19.4	0.21
181.mcf	291 + 687 = 978	735	0.75	0.23 + 573 = 573	16.7	0.03	0.09 + 110 = 110	7.7	0.07
197.parser	226 + 642 = 868	609	0.70	1.4 + 431 = 432	21	0.05	1 + 105 = 106	20.9	0.20
253.perlbmk	185 + 527 = 722	466	0.65	0.12 + 448 = 448	1.5	0.003	3.8 + 110 = 114	18.4	0.16
300.twolf	177 + 513 = 690	417	0.60	3 + 410 = 413	32	0.08	6.1 + 112 = 118	39.4	0.33
255.vortex	182 + 618 = 800	500	0.63	.06 + 490 = 490	4.5	0.01	1.6 + 100 = 101	12.2	0.12
175.vpr	186 + 525 = 711	318	0.45	2.4 + 438 = 440	17.4	0.04	4.9 + 110 = 115	24.8	0.22
Average	179 + 565 = 727	448	0.62	1.4 + 434 = 435	17.1	0.04	2.7 + 83 = 86	18.5	0.21

eCF's for Sequitur as indicated by column labeled *Smaller* eCF. Second reason for compacted $eWPP$ is the effective compression of eCF by Sequitur and VPC. On an average this accounts for 57.6% reduction in trace size as indicated by column *Compression of* eCF. As we can see, both of these effects are important in achieving smaller $eWPP$s.

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 5. From this data we can see that on an 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 towards reducing the size of eCF.

Table 5. Distribution of memory dep. types.

Program	No-Cost (%)	Fixed (%)	Varying (%)
256.bzip2	40.8 %	3.1 %	56.1 %
186.crafty	48.5 %	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 %
300.twolf	29.4 %	8.2 %	63.4 %
255.vortex	22.3 %	1.5 %	76.2 %
175.vpr	60.9 %	3 %	36.1 %
Average	30.4 %	3.9 %	65.7 %

5.2 Runtime Overhead

The execution time cost of the disambiguation checks is mainly due to the address comparisons performed. In particular, the greater the number of such comparisons, the greater its cost. In Table 6 the average number of comparisons performed per dynamic data dependence are given in column $Checks/Dep$. These results were obtained as a result of applying number of optimizations described in Section 4. However, the one significant contributing factor is using binary search instead of linear search. If we had not performed these optimizations then the number of checks needed at run-time would have gone up by a significant amount, as shown in the column $UChecks/Dep$. making collection of these traces impractical.

Table 6. Address Comparisons.

Program	UChecks/ Dep.	Checks/ Dep.	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

Table 7 shows the run-time overhead needed to collect these traces. The running time of the 3 versions of each program, that is, the original version, instrumented version for collecting control flow and dependence traces (V_{CD}), 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_{CD} and V_E, the time spent on processing (CPU) and IO are separately shown. Although the CPU time spent in V_E is higher than V_{CD}, coming from the checks needed per dependence, the IO time is much less, extended traces are smaller than dependence traces. On an average, there is a 20% increase in runtime overhead when collecting extended control flow traces when compared to collecting control flow and dependence traces.

6 Conclusion

In this paper we presented a unified trace representation that enables the capture of complete control flow and data

Table 7. Run-time Overhead in seconds.

Program	Original	CF + DD (V_{CD})	eCF (V_E)
	CPU	CPU + IO	FP + CPU + IO
256.bzip2	5	102 + 502 = 604	171 + 120 + 387 = 678
186.crafty	5	80 + 578 = 658	400 + 91 + 446 = 937
252.eon	3	82 + 654 = 736	214 + 107 + 513 = 834
254.gap	3	152 + 488 = 640	303 + 168 + 399 = 870
164.gzip	5	67 + 407 = 474	124 + 82 + 332 = 538
181.mcf	7	101 + 496 = 597	194 + 114 + 403 = 711
197.parser	7	66 + 454 = 520	138 + 85 + 381 = 604
253.perlbmk	5	68 + 664 = 732	380 + 90 + 544 = 1014
300.twolf	6	67 + 552 = 619	120 + 86 + 441 = 647
255.vortex	4	121 + 638 = 759	308 + 141 + 520 = 969
175.vpr	7	82 + 550 = 632	100 + 102 + 374 = 576
Average	5	90 + 544 = 634	223 + 108 + 431 = 762

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 alternative, i.e. combination of control flow and dependence traces. We also presented algorithms for traversing our extended trace to recover data dependences or chains of data dependences. Such information is useful in carrying out variety of code optimizations as well as other software engineering applications such as dynamic program slicing.

Acknowledgements: 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

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

[2] H. Agrawal, R. DeMillo, and E. Spafford, "Debugging with Dynamic Slicing and Backtracking," *Software Practice and Experience*, 23(6):589-616, 1993.

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

[4] T. Ball and J.R. Larus, "Efficient Path Profiling," *IEEE/ACM International Symposium on Microarchitecture*, 1996.

[5] 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.

[6] M. Burtscher, "VPC3: A Fast and Effective Trace-Compression Algorithm," *Joint International Conference on Measurement and Modeling of Computer Systems* (SIGMETRICS), pages 167-176, June 2004.

[7] 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.

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

[9] B. Korel and J. Laski, "Dynamic Program Slicing," *Information Processing Letters*, 29(3):155-163, 1988.

[10] B. Korel and J. Rilling, "Application of Dynamic Slicing in Program Debugging," *AADEBUG*, pages 43-58, 1997.

[11] M. Kamkar, "Interprocedural Dynamic Slicing with Applications to Debugging and Testing," *PhD Thesis*, Linkoping University, 1993.

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

[13] S-W. Liao, P.H. Wang, H. Wang, J.P. Shen, G. Hoflehner, and D.M. Lavery, "Post-Pass Binary Adaptation for Software-Based Speculative Precomputation," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 117-128, 2002.

[14] J. Lin, T. Chen, W-C. Hsu, P-C. Yew, R.D-C. Ju, T-F. Ngai, S. Chan, "A Compiler Framework for Speculative Analysis and Optimizations," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 289-299, 2003.

[15] J. Lin, W-C. Hsu, P-C. Yew, R. Ju, and T-F. Ngai, "A Compiler Framework for Recovery Code Generation in General Speculative Optimizations," *PACT*, pages 17-28, 2004.

[16] 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.

[17] Y. Sazeides, "Instruction-Isomorphism in Program Execution," *Value Prediction Workshop*, June 2003.

[18] C. Young and M.D. Smith, "Better Global Scheduling Using Path Profiles," *IEEE/ACM International Symposium on Microarchitecture*, pages 115-123, 1998.

[19] X. Zhang and R. Gupta, "Cost Effective Dynamic Program Slicing," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 94-106, Washington D.C., June 2004.

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

[21] C.B. Zilles and G. Sohi, "Understanding the Backward Slices of Performance Degrading Instructions," *27th International Symposium on Computer Architecture*, 2000.