The Effects of the Precision of Pointer Analysis ⋆

Marc Shapiro and Susan Horwitz

University of Wisconsin – Madison
1210 West Dayton Street, Madison, WI 53706 USA
{mds,horwitz}@cs.wisc.edu

Abstract. In order to analyze programs that manipulate pointers, it is necessary to have safe information about what each pointer might point to. There are many algorithms that can be used to determine this information, with varying degrees of accuracy. However, there has been very little previous work that addresses how much the relative accuracies of different pointer-analysis algorithms affect "transitive" results: the results of a subsequent analysis.
We have carried out a number of experiments with flow-insensitive, context-insensitive pointer analyses to address the following questions:

- How are the transitive effects of pointer analysis affected by the precision of the analysis?
- How good are the "direct" effects of pointer analysis (the sizes of the computed points-to sets) at predicting the transitive effects?
- What are the time trade-offs?

We found that using a more precise pointer analysis does in general lead to more precise transitive results. However, the magnitude of the payoff in precision depends on the particular use of the points-to information. We also found that direct effects are good predictors of transitive effects, and that increased precision in points-to information not only causes a subsequent analysis to produce more precise results, it also causes the subsequent analysis to run faster.

1 Introduction

Compilers often perform a variety of dataflow analyses, such as live variable analysis, to permit safe code optimization. Such analyses can also be used by other programming tools; for example, to provide feedback about possible logical errors, to aid in program understanding and debugging, or to aid in testing. In the presence of pointers, these analyses become more difficult. For example, consider the following code fragment:

```
x = 0;
printf("%d", *p);
x = 1;
```

If variable p can point to x during the execution of these statements, then x is live after the first assignment, and the assignment itself is considered to be live. If p cannot point to x, then the first assignment is dead (because of the subsequent assignment, x = 1); a compiler can safely ignore the first assignment, and a programming tool might report it as indicating a possible logical error.

⋆ This work was supported in part by the National Science Foundation under grant CCR-9625656, and by the Army Research Office under grant DAAH04-85-1-0482.

To gather the information needed for such analyses, the compiler can first perform a pointer analysis to determine, for each dereference of each pointer p, a safe approximation to the set of objects to which p might be pointing at that dereference.

Previous work on pointer analysis has concentrated on the design and analysis of new algorithms. While such work often includes some data produced by running the new algorithm, this data is usually limited to reporting the times required to analyze programs, plus some information about the sizes of the points-to sets that were produced (possibly comparing the times and sizes of the new algorithm to those required/produced by some other algorithm). (For examples, see [LR92], [And94], [Ruf95], [Ste96], [EGH94], [WL95] or [SH97].) While this is certainly interesting, it is not clear whether this information can be used to predict the usefulness of the pointer-analysis algorithm in a larger context. For example, in the case discussed above, we don't really care about the size of p's points-to set, all we care about is whether p might point to x.

For this paper we have carried out a number of experiments to address the following questions:

- How are the *transitive* effects of pointer analysis affected by the precision of the analysis? For example, how much benefit is gained by using a more precise pointer analysis as the first step in solving a dataflow problem such as live variable analysis?
- How good are the "direct" effects of pointer analysis at predicting the "transitive" effects? For example, what is the correlation between the ratio of the sizes of the points-to sets computed by two different pointer analyses, and the sizes of the live variable sets computed using the results of the two pointer analyses?
- What are the time trade-offs? For example, is the extra time required to compute more precise points-to information offset by a decrease in the time required for the subsequent dataflow analysis?

To carry out our experiments, we implemented:

- Four different pointer analyses (described in Section 2.1).
- Three different dataflow analyses (described in Section 2.2), each of which makes use of points-to information.
- Interprocedural slicing using system dependence graphs (described in Section 2.3). Points-to information is used to build the system dependence graphs.

We measured how much the precision of the different pointer analyses affected the results of the dataflow analyses, the sizes of the system dependence graphs, and the results of slicing. In all cases, we also measured how much time was required for both the pointer analyses and for the subsequent processing that made use of the points-to information.

Our results (presented in section 3) can be summarized as follows:

- Using a more precise pointer analysis *does* in general lead to more precise "transitive" results. However, the magnitude of the payoff in precision depends on the particular use of the points-to information. In our experiments, the difference in precision ranged from 0% to 800%.
- "Direct" effects (points-to set sizes) *are* good predictors of "transitive" effects (*e.g.*, sizes of live variable sets). However, since the effects of the precision of pointer analysis vary depending on how the points-to sets are used, halving the sizes of the points-to sets does not necessarily lead to a doubling in the accuracy of the transitive results. In our experiments, we found that the expected improvement in indirect effects due to doubling the accuracy of the pointer analysis ranged from 0% to 42%.
- Increased precision in points-to information not only causes a subsequent analysis to produce more precise results, it also causes the subsequent analysis to run faster. (Whether this offsets the longer times needed for the more precise pointer analysis depends, of course, on the relative time requirements of the pointer analysis and the subsequent analysis.)

2 Background to the experiments

Our experiments were run on a set of 61 C programs including Gnu Unix utilities, Spec benchmarks, and programs used for benchmarking by Landi [LRZ93] and by Austin [ABS94]. Tests were carried out on a Sparc 20/71 with 256 MB of RAM. Our experiments in which pointer analysis was followed by dataflow analysis all involved the following five steps:

Step 1: Parse a program, building its control-flow graph.
Step 2: Use one of the four pointer-analysis algorithms to analyze the program.
Step 3: Annotate each node of the control-flow graph with the sets of variables used, killed, and defined at that node.[2] For nodes that involve pointer dereferences, use the results of the pointer analysis to determine the use/kill/def sets. (For example, if the node represents the statement "$*p = *q$", the node's use set includes everything that might be pointed to by q, its kill set is empty, and its def set includes everything that might be pointed to by p.)
Step 4: Traverse the annotated control-flow graph, producing data to be used to solve one of the three dataflow-analysis problems.
Step 5: Perform the dataflow analysis.

The dataflow analysis problems were solved using the method described in [RHS95]. The input to the dataflow "engine" is a set of graphs (one for each function in the program), and a set of dataflow functions (one for each edge of the graph). The dataflow functions must be distributive, they must map finite sets to finite sets, and the meet operation must be set union. The output is the set of dataflow facts that hold at each node of the graph. If the dataflow problem is intraprocedural (*i.e.*, there is only one function), the dataflow facts provide the "meet over all paths" solution [Kil73]. If the dataflow problem is interprocedural, the dataflow facts provide the "meet over all interprocedurally valid paths" solution to the dataflow problem (*i.e.*, the algorithm performs precise–or context-sensitive–interprocedural dataflow analysis).

The worst-case time needed to solve a dataflow problem using this method depends on whether the problem is intra- or interprocedural, on the size of the dataflow domain, and on certain properties of the dataflow functions. We chose the three dataflow problems that were used in our experiments to span the range of possibilities, from time proportional to (*size of call graph* $\times$ *number of globals*) to (*sum of sizes of control-flow graphs* $\times$ *number of variables*[3]). This is discussed in more detail in Section 2.2 below, which describes the three dataflow problems.

The experiments in which pointer analysis was followed by building the program's system dependence graph and then computing interprocedural slices also involved steps $1-3$ listed above. Steps 4 and 5 were replaced by:

Step 4: Use the annotated control-flow graph to build the program's system dependence graph.
Step 5: Slice the system dependence graph with respect to all output statements (*i.e.*, calls to library functions like *printf*).

The system dependence graphs were built and sliced using the algorithms defined in [HRB90] [HRSR94] (see Section 2.3 for more detail).

[2] A variable x is considered to be *killed* at a node n if the value of x is definitely overwritten when n is executed. It is considered to be *defined* at n if its value is only partially overwritten (*e.g.*, x is an array, and only one element of the array is overwritten at n), or if its value may or may not be overwritten (*e.g.*, n represents the assignment "$*p = 0$" and p might point to either a or to b; in this case, both a and b are considered to be defined–but not killed–at n).

2.1 The Pointer Analyses

Our experiments made use of four different pointer analyses. The analyses are listed below from most to least precise; in each case, the worst-case running time is given in terms of N, the size of the program. The term $\alpha(N)$ is the (very slowly growing) inverse Ackermann's function that arises in the context of fast union/find data structures [Tar83]. The four pointer analyses we used were:

1. The analysis defined by Andersen in [And94], which has worst-case time $O(N^3)$.
2. The 3-category, log-N-runs analysis defined by Shapiro and Horwitz in [SH97], which has worst-case time $O(N\alpha(N)\log N)$.
3. The analysis defined by Steensgaard in [Ste96], which has worst-case time $O(N\alpha(N))$.
4. A naive analysis that simply assumes that every pointer variable can point to any variable whose address is taken somewhere in the program, or to any location allocated from the heap. This analysis has worst-case time $O(N)$.

All of the analyses are applicable to the C language, including pointer arithmetic and calls via pointers to functions, but not setjmp/longjmp, functions with variable-length argument lists, or signals. They are all both *flow insensitive* and *context insensitive*.[3] A *flow-sensitive* pointer analysis takes into account the flow of control in a program, and computes a points-to set for every pointer at every program point. In contrast, a flow-*insensitive* analysis treats the program as an unordered set of statements, and thus only computes one points-to set for each pointer–which is valid, though usually overly conservative, at all program points. A *context*-sensitive analysis takes into account the fact that a function may be called with different calling contexts, and that when a function returns, control is transferred back to the site of the most recent call. In contrast, a context-insensitive analysis essentially treats a function call as a *goto* to the start of the function, and a function return as a multi-way *goto* back to all call sites.

All of the analyses use a simple memory model that collapses each of the following into a single variable:

– different fields of a single struct
– different elements of a single array
– different instances of a local variable in a recursive function
– different heap locations allocated by an instance of a call to malloc (or a similar function), which might be executed multiple times.

The first three analyses (Andersen's, Shapiro's, and Steensgaard's) can be thought of as building a graph in which nodes represent variables, and an edge $n \rightarrow m$ means that n might point to m. The fundamental difference between Andersen's algorithm (the most precise of the three) and Steensgaard's algorithm (the least precise of the three) is that the graphs built by Andersen's algorithm can have unbounded out-degree, while the graphs built by Steensgaard's algorithm can only have out-degree one. This means that a node in Steensgaard's graph may represent more than one variable, while in Andersen's graph, each node represents exactly one variable. The coarser granularity of Steensgaard's graphs makes the algorithm both faster and less precise than Andersen's.

A detailed comparison of the two algorithms is beyond the scope of this paper; however, the example shown in Figure 1 provides some intuition. Note that in this example, Steensgaard's approach determines that b might point to e, and that d might point to c. While this is safe (no actual points-to information is omitted), it is overly conservative.

Shapiro's algorithm (the second most accurate analysis used in our study) is similar to Steensgaard's algorithm in that the graphs that it builds have limited out-degree; however, it is different in two respects:

---

[3] To be consistent with Steensgaard's implementation, a small amount of context sensitivity was introduced into the analyses by essentially in-lining all library functions. However, all calls to user-defined functions were handled in a context-insensitive manner.

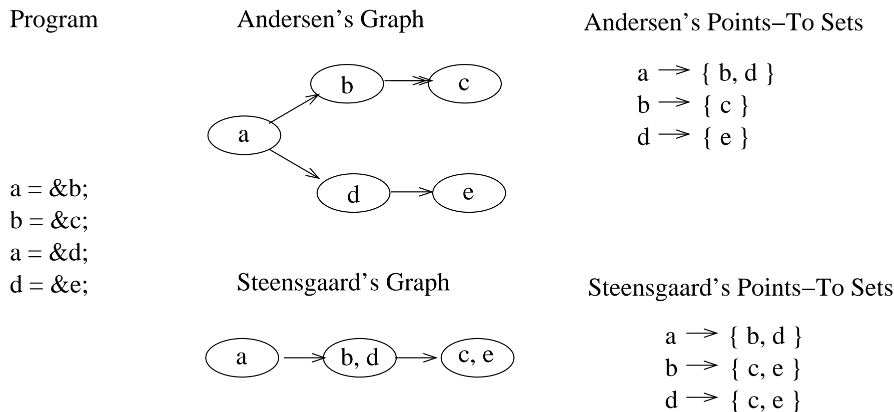

Fig. 1. Example contrasting Andersen's and Steensgaard's points-to analysis algorithms

1. The graphs are limited to out-degree 3, rather than out-degree 1 (this improves the accuracy of the analysis at the cost of slowing it down by a constant factor).
2. The algorithm uses $log_3 N$ passes (where N is the number of variables in the program), building a different graph on each pass. It is guaranteed that for every pair of variables (x, y), there will be at least one pass on which x and y are represented by distinct nodes of the graph. The final points-to information is computed by combining the information from each of the passes.

Both the accuracy and the worst-case time requirements of this algorithm fall between those of Andersen's and Steensgaard's algorithms. (On the example of Figure 1, this algorithm would require two passes, and would produce the same results as Andersen's algorithm.)

As we will see in Section 3, the actual times for Shapiro's analysis are often much slower than the times for both Steensgaard's and Andersen's analyses. This is mainly due to our implementation, rather than to some inherant limitation of the algorithm. While Shapiro's analysis will necessarily be slower than Steensgaard's (since it involves multiple passes, the first of which is essentially the same as Steensgaard's algorithm), we believe that it can be sped up to run in time much closer to that of Steensgaard's algorithm than the current implementation. For large programs, it should be significantly faster than Andersen's algorithm, since it is close to linear, while Andersen's algorithm is cubic in the size of the program. Even with our current inefficient implementation, Shapiro's algorithm does run faster than Andersen's on 6 of our large test programs.

As mentioned above, our fourth analysis (the naive analysis) simply assumes that every pointer might point to every variable whose address is taken somewhere in the program (as well as pointing to all heap-allocated storage). On the example of Figure 1, that algorithm would determine that variables a, b, and d might all point to b, c, d, and e.

2.2 The Dataflow Analyses

To test the transitive effects of the four pointer analyses described above, we implemented three different dataflow analyses (listed below from least to most ambitious):

1. GMOD analysis.
2. Live variable analysis.
3. Truly live variable analysis.

These analyses are described in the following subsections. Recall that all of the analyses rely on a pointer analysis having been performed first, so that for every statement in the program it is known which variables might be used, killed, and defined by that statement.

GMOD analysis: The goal of this analysis is to determine, for each function in a program, which global variables might be modified, either directly or transitively, by that function. (In the presence of pointers, a local variable becomes global if its address is made accessible to other functions.)

GMOD analysis is performed on the program's call graph rather than on its control-flow graph. However, this isn't really important: our dataflow framework simply requires that there be an underlying graph; it doesn't matter if that graph represents calls or flow of control. Thus, GMOD can be implemented as follows:

1. For each function f, the set IMOD_f is the set of global variables that might be directly modified by f (*i.e.*, the effects of the functions called by f are not included).
2. The graph on which the dataflow analysis is performed is the program's call graph, with edges reversed, with a new "start" node, and with some new edges added so that every node is reachable from the start node.
3. The dataflow function associated with the edges out of the graph node that represents function f is: $\lambda S.S \cup \text{IMOD}_f$.

Implemented this way, the GMOD problem is an intraprocedural, locally separable (*i.e.*, gen/kill) problem. The size of the dataflow domain is the number of global variables that are (directly) modified in at least one function. Thus, the worst-case time to solve the GMOD problem using our dataflow engine is O(*size of call graph* $\times$ *number of global variables*).

The accuracy of GMOD analysis is affected by the accuracy of the points-to information that is used to determine which variables might be modified at each node of the control-flow graph, as well as by the accuracy of the points-to information that is used to determine which function(s) might be called at each "indirect" call (*i.e.*, each call via a pointer). For example, if control-flow graph node n represents the statement "*p = 0", then n's def set will be the set of variables that p might point to. If n is in function f, then f's IMOD set will include all of those variables as well. Clearly, a more accurate pointer analysis (that computes smaller points-to sets) will lead to more accurate GMOD sets. Similarly, if function f includes a call via a pointer p, f's GMOD set will include all of the variables that might be modified (directly or transitively) by all of the called functions. Again, a more accurate points-to analysis (that computes a smaller points-to set for p) will lead to more accurate GMOD sets.

Live variable analysis: This is the standard dataflow analysis problem in which variable x is considered to be live at node n of a program's control-flow graph iff there is a path in the graph from n to the end of the program, on which x is used before being defined.

Recall that one of our goals is to determine how the transitive effects of pointer analysis are affected by the accuracy of the pointer analysis. In the case of live variable analysis (as well as truly live analysis, described in the next section), there are at least two interesting quantities that can be considered to be those "transitive" effects:

- The sizes of the live sets computed at each node of the control-flow graph.
- The number of dead assignments (assignments to variables that are not live immediately after the assignment).

The live ranges of variables can be used by a compiler in performing register allocation, and in general, smaller live sets mean smaller live ranges. Thus, if a more accurate pointer analysis leads to smaller live sets, it will probably lead to better register allocation as well. Since we have not implemented register allocation, we cannot study that directly, so we have instead studied the sizes of the live sets produced by our live analysis using each of the four different pointer analyses.

As discussed in the Introduction, a dead assignment can be removed by a compiler, or can be flagged as a possible logical error. Thus, the number of dead assignments discovered after performing live analysis is also an interesting transitive effect.

Like GMOD analysis, live variable analysis is a locally separable problem. The size of the dataflow domain for each function is the number of variables visible in that function (possibly including local variables of other functions whose addresses have been written into the heap or made accessible via parameters). The worst-case time for live variable analysis using our dataflow engine is thus O(*sum of sizes of control-flow graphs* $\times$ *max(number of visible variables)*).

The accuracy of live variable analysis is affected by the accuracy of the points-to information that is used to determine which variables might be used at each node of the control-flow graph, as well as by the accuracy of the points-to information that is used to determine which function(s) might be called at each indirect call. For example, if node n represents the statement "x = *p", then all of the variables in p's points-to set are considered live before node n. A more accurate pointer analysis will lead to more accurate (smaller) live sets. Similarly, if node n represents a call via a pointer p, then all of the variables that are live at the beginnings of *all* of the functions that might be pointed to by p are considered live before node n.

Truly live variables: This is a non-locally-separable (and more accurate) version of the live-variables problem in which variable x is considered to be truly live at node n iff there is a path from n to the end of the program on which x is used in a truly live context before being defined. By definition, a use in a truly live context means: in a predicate, or in a call to a library function, or in an expression whose value is assigned to a truly live variable [GMW81]. Because it is non-locally-separable, the truly-live-variables problem is in some sense a harder problem than the live-variables problem; the worst-case time for this analysis is O(*sum of sizes of control-flow graphs* $\times$ *max(number of visible variables)*^{3}).

The accuracy of truly live variable analysis is affected by the accuracy of the points-to information that is used to determine which variables might be defined and used at each node of the control-flow graph, as well as by the accuracy of the points-to information that is used to determine which function(s) might be called at each indirect call. For example, if node n represents the statement "*p = *q", then all of the variable in q's points-to set are considered truly live before node n if any of the variables in p's points-to set is truly live after node n. Again, it is clear that a more accurate pointer analysis will lead to more accurate (smaller) truly live sets.

2.3 Interprocedural Slicing

The slice of a program with respect to an output statement S (a call to a library function like *printf*) is a projection of the program such that for all possible inputs, the same value is output at S in both the original program and the slice (if statement S is executed more than once, then the two versions of the program must produce the same *sequence* of values at S) [Wei84].

An efficient method for computing context-sensitive interprocedural slices was described in [HRB90] [HRSR94]; that method was used in our implementation. The method involves the use of a program representation called the *system dependence graph*, or SDG. The nodes of a program's SDG are the same as the nodes of its control-flow graph; however, the edges are different: the SDG includes edges that represent the program's data and control dependences, rather than its sequential control-flow. Building a program's SDG involves the following steps:

1. Build the program's control-flow graph.
2. Compute GMOD sets; for each function f, add each variable in GMOD_f as an extra parameter to f.
3. Using the control-flow graph, compute the program's intraprocedural data and control dependences.
4. Compute the transitive data dependences due to function calls.

The total time required to build an SDG is dominated by the time for step 4, which is O((*number of control-flow graph edges* $\times$ *max(number of parameters)* + *number of call sites* $\times$ *max(number of parameters*$)^3$). Once a program's SDG is built, an interprocedural slice can be computed in time proportional to the size of the slice.

The size of a program's SDG as well as the accuracy of the slices computed using the SDG are affected by the accuracy of the points-to information that is used to determine which variables might be defined and used at each node of the control-flow graph, as well as by the accuracy of the points-to information that is used to determine which function(s) might be called at each indirect call. For example, if node n represents the statement "x = *p", then in the SDG, n will be the target of data dependence edges from nodes that represent definitions of all of the variables in p's points-to set. Therefore, a more accurate pointer analysis will lead to an SDG with fewer edges. This in turn will affect the sizes of the slices computed using that SDG.

3 Experimental results

To determine whether and how much the choice of pointer analysis affects the other analyses, we considered the effect of pointer analysis on the following quantities for each test program:

- gmodS – the sum of the sizes of the GMOD sets for each function
- gmodT – the time to perform the GMOD analysis
- liveS – the sum of the sizes of the live variable sets at each node of the CFG
- deadS – the number of dead assignments
- liveT – the time to perform the LIVE analysis
- tliveS – the sum of the sizes of the truly-live variable sets at each node of the CFG
- tdeadS – the number of truly-dead assignments
- tliveT – the time to perform the TRULY-LIVE analysis
- sdgS – the number of vertices and edges in the SDG
- sdgT – the time to build the SDG
- sliceS – the average size of a slice with respect to an output statement
- sliceT – the time to compute the slices

These represent "transitive effects" of pointer analysis. We also measure the "direct effects" with these quantities:

- ptaS – the average size of a pointer variable's points-to set
- ptaT – the time to perform the points-to analysis

We measured these quantities for each of the four pointer analyses, and for each of our example programs. Raw data for 14 of the test programs is given in Figures 2 and 3. The values given for "Lines of Code" are for preprocessed code with blank lines removed.

The following three subsections address the questions posed in the Introduction:

1. What are the transitive effects of the choice of pointer analysis? That is, how much (if at all) does the choice of pointer analysis affect the quantities listed above?
2. In particular, how (if at all) does each quantity depend on the sizes of the points-to sets?
3. How does the choice of pointer analysis affect the total time for performing analyses?

Throughout this section, we use the subscript i to range over the pointer analyses, and the subscript j to range over the example programs. The number of pointer analyses is I and the number of example programs is J. We use Q to represent a particular quantity of interest, and Q_{ij} to represent its value for pointer analysis i and example program j.

3.1 Effect of choice of pointer analysis on "transitive" quantities

To determine whether the choice of pointer analysis affects quantity Q, we assume that

$$Q_{Aj} \approx P_{A/B}\, Q_{Bj}$$

where A and B are two pointer analyses, and $P_{A/B}$ does not depend on the example program j. To make this an equality, we introduce an error factor E_{ABj}:

$$Q_{Aj} = P_{A/B}\, Q_{Bj}\, E_{ABj}$$

We want to find a value for $P_{A/B}$ that makes each E_{ABj} close to 1, so we minimize $\sum_j (\log E_{ABj})^2$. This leads to defining $P_{A/B}$ as the geometric mean of the ratios

$$\frac{Q_{Aj}}{Q_{Bj}}$$

(the ratio of the transitive quantity produced using pointer analysis A to the quantity produced using pointer analysis B for each example program j):

$$P_{A/B} = \left(\prod_j \frac{Q_{Aj}}{Q_{Bj}} \right)^{1/J}$$

The values for the $P_{A/B}$ are given in Figure 4, and graphed in Figures 5 and 6.

If $P_{A/B}$ is 1, then the change in pointer analysis did not affect the quantity Q_{ij}. Values much different from 1 indicate a strong dependence on the pointer analysis. For example, for gmodS, $P_{\text{Steens/Anders}} = 1.895$. This means that the GMOD sets were on average almost twice as large (and therefore more imprecise) when computed using Steensgaard's pointer analysis as using Andersen's pointer analysis. For gmodT, $P_{\text{Steens/Anders}} = 1.724$, so it took about 70% longer to compute those less accurate GMOD sets. For tdeadS, $P_{\text{Steens/Anders}} = 0.994$, so the number of truly-dead assignments was very slightly smaller with Steensgaard's analysis than Andersen's.

Test Program	Lines of Code	Pointer Analysis	gmodS	liveS	deadS	tliveS	tdeadS	sdgS	sliceS	ptaS
compress	657	Naive	3,343	215,368	18	211,504	153	16,407	259.5	23
		Steens	223	96,996	18	90,553	153	11,785	242.8	2
		Shapir	201	94,800	18	88,615	153	11,565	242.8	1
		Anders	188	94,568	18	85,924	153	11,546	242.8	1
gcc.main	1285	Naive	1,857	214,372	46	213,671	130	28,912	762.5	21
		Steens	458	182,658	49	181,421	135	26,121	767.2	3
		Shapir	403	182,642	49	181,421	135	26,121	767.2	2
		Anders	359	183,460	49	182,258	136	26,162	754.7	2
ratfor	1531	Naive	3,793	261,291	76	260,867	95	50,506	902.2	50
		Steens	1,395	224,743	76	224,290	96	46,130	877.1	19
		Shapir	1,283	224,743	76	224,290	96	46,130	877.1	17
		Anders	834	223,356	76	222,922	96	42,685	877.1	4
cdecl	2577	Naive	4,055	392,670	56	377,665	113	–	–	39
		Steens	660	328,279	56	312,574	118	–	–	15
		Shapir	521	328,279	56	312,574	118	–	–	12
		Anders	464	319,867	56	304,786	118	–	–	3
lex	2645	Naive	9,349	1,211,383	93	1,188,546	435	101,216	574.0	44
		Steens	1,224	944,810	93	874,891	461	86,019	574.0	10
		Shapir	1,123	929,310	93	859,391	461	85,978	574.3	9
		Anders	734	800,632	93	746,564	461	76,033	574.3	2
unzip	3261	Naive	14,627	851,988	30	–	–	–	–	50
		Steens	1,175	543,352	34	501,717	171	–	–	4
		Shapir	981	526,161	34	485,716	171	–	–	1
		Anders	961	526,041	34	485,604	171	–	–	1
gcc.cpp	4061	Naive	12,692	1,342,582	251	1,338,273	407	88,889	1,232.7	67
		Steens	2,793	975,349	254	969,015	414	66,644	1,166.0	29
		Shapir	2,478	973,826	254	967,887	414	66,558	1,166.0	25
		Anders	1,499	945,142	254	940,417	414	63,144	1,117.7	12
gzip	4584	Naive	41,635	2,949,512	93	–	–	235,653	352.8	106
		Steens	5,925	2,455,654	95	2,441,615	287	117,550	325.5	40
		Shapir	4,254	2,436,995	95	2,422,076	287	115,224	325.5	29
		Anders	2,389	2,239,493	95	2,218,273	287	88,892	324.5	5
bc	6745	Naive	18,606	2,098,224	123	1,987,200	323	388,338	579.3	97
		Steens	14,321	1,702,935	125	1,624,323	329	321,576	504.6	70
		Shapir	13,372	1,697,233	125	1,608,693	329	321,576	504.6	64
		Anders	3,031	1,529,305	125	1,475,685	330	276,436	500.6	13
ispell.freq	6830	Naive	1,629	89,829	1,658	86,050	1,689	89,891	271.6	59
		Steens	801	24,529	1,658	23,987	1,691	59,629	269.0	26
		Shapir	390	21,307	1,658	20,769	1,691	58,172	266.6	13
		Anders	132	6,224	1,658	2,684	1,691	51,862	266.6	4
grep	7433	Naive	35,321	3,403,244	203	–	–	180,602	454.1	119
		Steens	8,984	2,668,839	203	2,595,240	335	133,888	384.1	56
		Shapir	8,211	2,656,349	203	2,582,792	335	133,870	384.1	49
		Anders	3,210	2,605,513	203	2,530,699	341	124,530	383.8	13
sed	8022	Naive	45,247	3,836,898	261	–	–	146,899	421.0	132
		Steens	33,507	3,180,788	269	–	–	127,759	386.5	75
		Shapir	27,325	3,176,469	269	–	–	127,463	386.5	58
		Anders	19,229	2,947,275	269	–	–	118,413	386.5	29
less	12152	Naive	39,130	5,459,572	79	–	–	–	–	130
		Steens	17,233	2,771,403	85	2,576,684	412	–	–	59
		Shapir	13,384	2,767,857	85	2,573,392	412	–	–	42
		Anders	10,833	2,693,598	86	2,493,743	413	–	–	24
make	15564	Naive	192,240	–	–	–	–	–	–	215
		Steens	33,667	12,038,436	128	–	–	–	–	89
		Shapir	30,021	11,976,716	128	–	–	–	–	77
		Anders	22,771	11,309,614	128	–	–	–	–	43

Fig. 2. A sample of the raw "size" data. Entries with a dash (–) indicate missing data due to the dataflow analysis or slice computation running out of memory.

Test Program	Lines of Code	Pointer Analysis	gmodT	liveT	tliveT	sdgT	sliceT	ptaT
compress	657	Naive	2.68	17.45	185.54	72.18	2.09	0.18
		Steens	0.73	5.84	5.48	23.96	1.85	0.73
		Shapir	0.70	5.68	5.25	20.78	1.84	3.12
		Anders	0.71	5.78	5.05	20.92	1.85	0.75
gcc.main	1285	Naive	1.75	13.70	36.02	77.35	1.28	0.19
		Steens	0.78	9.76	11.35	64.39	1.28	0.79
		Shapir	0.72	9.73	11.18	56.61	1.26	3.05
		Anders	0.59	9.77	11.21	58.91	1.25	1.29
ratfor	1531	Naive	3.31	16.86	62.10	234.55	2.32	0.18
		Steens	1.55	12.13	14.73	157.43	2.13	0.85
		Shapir	1.65	12.18	13.33	139.28	2.09	3.73
		Anders	0.95	11.35	11.69	95.50	2.08	1.32
cdecl	2577	Naive	2.35	26.40	177.01	–	–	0.36
		Steens	1.05	18.28	60.33	–	–	1.38
		Shapir	0.86	18.25	60.54	–	–	6.94
		Anders	0.73	16.94	40.48	–	–	2.41
lex	2645	Naive	5.14	88.98	591.81	1208.90	14.71	0.62
		Steens	1.76	48.28	87.13	612.16	14.72	3.48
		Shapir	1.80	46.65	84.84	464.26	14.39	20.16
		Anders	1.12	38.71	40.27	393.26	14.15	5.19
unzip	3261	Naive	6.42	62.04	–	–	–	0.31
		Steens	0.86	24.01	30.71	–	–	1.42
		Shapir	0.69	23.23	23.69	–	–	7.18
		Anders	0.67	23.20	22.94	–	–	1.52
gcc.cpp	4061	Naive	8.42	111.56	2630.20	1966.36	21.21	0.58
		Steens	2.55	55.30	171.70	408.40	17.60	3.33
		Shapir	2.26	55.55	269.97	221.29	16.36	16.83
		Anders	1.38	48.41	97.71	231.59	14.57	12.47
gzip	4584	Naive	18.30	200.80	–	3632.48	4.20	0.67
		Steens	3.60	107.32	509.76	652.31	5.37	4.51
		Shapir	1.97	104.77	425.35	578.82	5.58	21.88
		Anders	1.34	94.16	127.88	368.14	5.82	8.72
bc	6745	Naive	10.79	128.57	3108.20	4546.95	11.66	0.68
		Steens	8.29	100.06	1444.94	4059.73	12.04	6.64
		Shapir	8.01	95.77	1245.28	3601.62	11.22	28.47
		Anders	2.55	74.19	134.40	2006.68	10.12	16.00
ispell.freq	6830	Naive	7.43	80.42	1072.88	1139.36	8.88	0.77
		Steens	3.87	24.63	57.32	321.43	9.65	4.75
		Shapir	1.11	16.12	20.39	165.62	9.19	20.35
		Anders	0.64	11.90	9.65	141.71	9.48	7.64
grep	7433	Naive	23.28	266.75	–	27449.98	3.42	0.86
		Steens	7.57	110.93	2373.87	2834.70	2.97	7.54
		Shapir	6.68	106.87	2074.78	2201.78	2.91	39.52
		Anders	2.99	84.84	576.63	1316.51	3.42	23.30
sed	8022	Naive	32.37	282.97	–	20359.45	4.28	0.79
		Steens	21.42	199.14	–	11373.49	3.58	9.10
		Shapir	17.94	196.04	–	9650.29	3.44	41.09
		Anders	13.00	145.75	–	5977.48	3.76	57.25
less	12152	Naive	16.87	280.58	–	–	–	0.54
		Steens	6.45	149.32	1698.64	–	–	4.20
		Shapir	3.31	143.74	1565.37	–	–	16.94
		Anders	2.46	135.26	1029.42	–	–	15.20
make	15564	Naive	135.37	–	–	–	–	1.66
		Steens	28.81	982.88	–	–	–	20.04
		Shapir	26.42	625.58	–	–	–	103.14
		Anders	16.95	669.80	–	–	–	146.23

Fig. 3. A sample of the raw "time" data, in CPU seconds. Entries with a dash (–) indicate missing data due to the dataflow analysis or slice computation running out of memory.

Q	Naive/Steens		Naive/Shapir		Naive/Anders		Steens/Shapir		Steens/Anders		Shapir/Anders	
	$P_{A/B}$	Error	$P_{A/B}$	Error	$P_{A/B}$	Error	$P_{A/B}$	Error	$P_{A/B}$	Error	$P_{A/B}$	Error
gmodS	4.635	10.4%	6.556	11.2%	8.783	10.9%	1.414	5.3%	1.895	7.7%	1.340	4.6%
gmodT	2.807	8.3%	3.633	9.8%	4.838	10.9%	1.295	4.6%	1.724	6.7%	1.331	4.6%
liveS	1.528	4.6%	1.575	4.8%	1.672	6.2%	1.030	0.9%	1.093	2.6%	1.061	2.3%
deadS	0.978	0.8%	0.978	0.8%	0.974	0.8%	? 1.000	0.0%	? 0.997	0.3%	? 0.997	0.3%
liveT	1.941	4.9%	2.094	5.4%	2.251	6.0%	1.085	2.0%	1.164	2.7%	1.072	1.6%
tliveS	1.528	6.0%	1.592	6.3%	1.720	11.1%	1.039	1.3%	1.121	4.7%	? 1.078	4.3%
tdeadS	0.945	1.2%	0.940	1.4%	0.939	1.3%	? 0.996	0.4%	? 0.994	0.4%	0.998	0.1%
tliveT	5.677	16.6%	7.630	19.6%	9.803	21.1%	1.410	8.6%	1.894	11.2%	1.343	6.3%
sdgS	1.223	3.6%	1.239	3.7%	1.281	4.0%	1.013	0.3%	1.047	0.9%	1.034	0.8%
sliceS	1.050	1.6%	1.057	2.1%	1.059	2.1%	? 1.007	0.6%	? 1.009	0.6%	? 1.002	0.1%
sdgT	2.007	10.4%	2.632	13.4%	2.863	14.2%	1.311	6.0%	1.426	6.9%	1.088	2.8%
sliceT	1.040	2.0%	? 1.053	3.0%	? 1.061	3.2%	? 1.012	1.5%	? 1.020	1.8%	? 1.008	0.8%
ptaS	4.728	9.4%	7.805	10.4%	12.981	8.4%	1.599	6.2%	2.712	8.5%	1.684	6.2%
ptaT	0.163	7.4%	0.035	7.0%	0.090	16.0%	0.218	2.8%	0.554	9.0%	2.544	9.9%

Fig. 4. The imprecision or slowdown from using a less precise pointer analysis. Each pair of colums is labelled A/B where A is a less precise pointer analysis than B. The farther $P_{A/B}$ is from 1, the larger the effect on quantity Q of choosing pointer analysis A rather than B. Values are marked with a "?" if the interval from $P_{A/B} \div E^2_{A/B}$ to $P_{A/B} \times E^2_{A/B}$ includes 1. For these values, there is only a negligible difference between the two pointer analyses.

The estimated standard error on $\log P_{A/B}$ is

$$\varepsilon(\log P_{A/B}) = \sqrt{\frac{\sum_j (\log E_{ABj})^2}{J(J-1)}}$$

If we define $E_{A/B} = e^{\varepsilon(\log P_{A/B})}$, then it plays a role like a standard error on $P_{A/B}$, except that instead of being additive, it is multiplicative. That is, we expect that if $P_{A/B}$ were determined on another set of test programs, it would most likely fall between $P_{A/B} \div E_{A/B}$ and $P_{A/B} \times E_{A/B}$. This is an interval that is asymmetric about $P_{A/B}$, which is the geometric mean of the endpoints of the interval, rather than the arithmetic mean. Figure 4 includes error values, which are the values of $E_{A/B} - 1$ expressed as percentages. For example, if $E_{A/B} = 1.10$, then the value is reported as 10%.

Most of the values for $P_{A/B}$ are significantly different from 1, indicating that the choice of pointer analysis does have a significant effect on the transitive quantities. Only for the quantity sliceT is $P_{A/B}$ never significantly different from 1. For deadS and tdeadS and sliceS, $P_{A/B}$ is significantly different from 1 only if A is the Naive analysis, while the difference between the non-trivial pointer analyses is slight. However, for gmodS, gmodT, and tliveT, the value $P_{A/B}$ is always 1.3 or more; the choice of pointer analysis has a very strong effect on these.

These results contrast with those reported by Ruf in [Ruf95]. Ruf studied the impact of context sensitivity on flow-sensitive pointer analysis. When he considered the sizes of the points-to sets at *all* program points, he found that using a context-sensitive algorithm did make a difference (the sets were significantly smaller). However, when he considered only those program points at which a pointer was dereferenced, and looked only at those pointers' points-to sets, he found that context sensitivity made very little difference. This suggests that, at least when doing flow-sensitive pointer analysis, the choice of a context-sensitive rather than a context-insensitive analysis will have at best a very weak effect on transitive quantities.

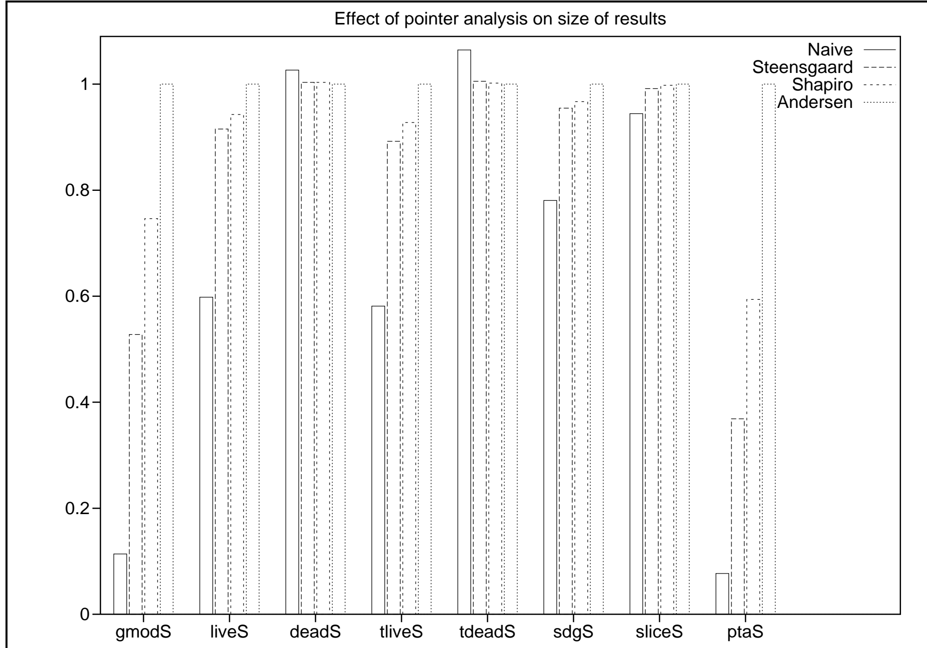

Fig. 5. This is an alternative presentation of the data in Figure 4 for the "size" quantities. The height of each bar is $P_{\text{Andersen/B}}$. This is the factor by which the quantity obtained using pointer analysis B would change if Andersen's pointer analysis were used instead. Recall that for deadS and tdeadS, more precise values are larger, while for the other quantities more precise values are smaller.

3.2 Dependence on size of points-to set

The results of the previous section indicate that in most cases the quantities are dependent on which pointer analyses is used. In this section we explore whether the sizes of the points-to sets (ptaS, which here we refer to as S_{ij}) are good predictors of the quantities, and what those predictions are. To determine the relationship between the points-to set sizes and the quantities, we use a new model

$$Q_{ij} \approx \left(\frac{S_{ij}}{S_{Aj}}\right)^{\delta} Q_{Aj}$$

which as before we make an equality by introducing an error E_{ij}:

$$Q_{ij} = \left(\frac{S_{ij}}{S_{Aj}}\right)^{\delta} Q_{Aj} E_{ij}$$

Here A refers specifically to Andersen's pointer analysis. The value δ is a measure of how much Q_{ij} depends on the sizes of the points-to sets. For positive values of δ, Q_{ij} increases with increasing points-to set size. If $\delta = 1$, then the quantity Q_{ij}, for a fixed j (*i.e.*, a particular example program) is proportional to S_{ij}, the size of the points-to set.

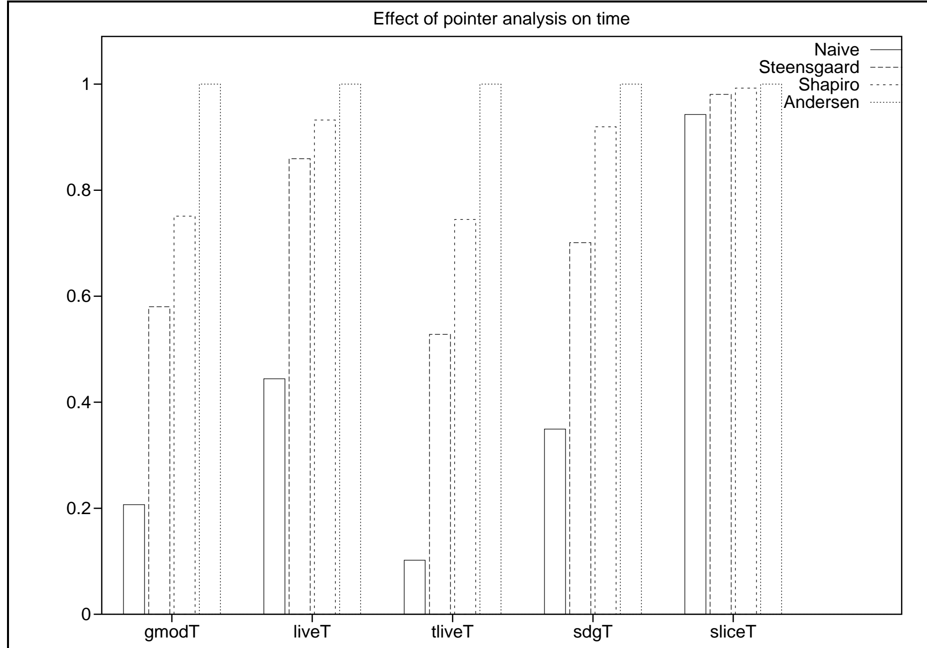

Fig. 6. This is an alternative presentation of the data in Figure 4 for the "time" quantities. The height of each bar is $P_{\text{Andersen/B}}$. This is the factor by which a subsequent analysis using pointer analysis B would speed up if Andersen's pointer analysis were used instead.

If $\delta = -1$, then they are inversely proportional. If $\delta = 0$, they are independent. Note that δ is independent not only of the pointer analysis, but also of the test program. It depends only on the quantity Q under consideration, and is a measure of the sensitivity of Q to points-to set size.

If we take logarithms, the equality looks like a line through the origin

$$log \frac{Q_{ij}}{Q_{Aj}} = \delta \log \frac{S_{ij}}{S_{Aj}} + \log E_{ij}$$

and we can use standard statistical methods for regression through the origin[NW74]. As before, we minimize $\sum_j (\log E_{ij})^2$ and find

$$\delta = \frac{\sum_{ij} \log \frac{Q_{ij}}{Q_{Aj}} \log \frac{S_{ij}}{S_{Aj}}}{\sum_{ij} (\log \frac{S_{ij}}{S_{Aj}})^2}$$

The values for δ are given in Figure 7. We also list the values for 2^δ, which indicates how much each quantity would be expected to increase as a result of doubling the sizes of the points-to sets. This can be considered an abstract $P_{A/B}$, for A with points-to sets twice as large as B. For example, since for gmodS, $2^\delta = 1.705$, if some pointer analysis tended to generate points-to sets that were twice as large as those generated by

Q	δ	ε	2^δ	$2^\varepsilon - 1$
gmodT	0.561	0.021	1.475	1.5%
liveT	0.253	0.011	1.192	0.8%
tliveT	0.791	0.040	1.730	2.8%
sdgT	0.373	0.027	1.295	1.9%
sliceT	0.014	0.007	1.010	0.5%
deadS	-0.007	0.001	0.995	0.1%
tdeadS	-0.018	0.002	0.987	0.2%
gmodS	0.770	0.020	1.705	1.4%
liveS	0.161	0.013	1.118	0.9%
tliveS	0.185	0.022	1.136	1.6%
sdgS	0.086	0.007	1.061	0.5%
sliceS	0.014	0.004	1.010	0.3%

Fig. 7. Dependence of quantities on sizes of points-to sets, and accuracy of points-to set sizes as predictors of quantities. 2^δ is the factor by which the quantity would be expected to increase if the size of the points-to sets doubled. The standard error on δ is ε. The last two columns are similar to the columns in Figure 4.

Andersen's analysis, we would expect that the GMOD sets computed would be 70.5% larger.

To determine whether the points-to set sizes are good predictors of the other quantities, we consider the estimated standard error of δ:

$$\varepsilon \equiv \sqrt{\frac{\sum_{ij}(\log E_{ij})^2}{(IJ-1)\sum_{ij}(\log \frac{S_{ij}}{S_{Aj}})^2}}$$

We also consider the values 2^ε, which are similar to $E_{A/B}$ for a pointer analysis A that tends to generate points-to sets twice as large as another pointer analysis B. That is, we expect values of 2^δ to vary by not much more than a factor of 2^ε if measured with different test programs. The smaller the values of 2^ε, the more accurate the points-to set sizes are as predictors of the results of a subsequent analysis.

As we see from Figure 7, the values of $2^\varepsilon - 1$ are quite small, ranging from 0.1% to 2.8%. This means that points-to set sizes are very good predictors of the quantities listed in the first column.

We also see that the actual predictions range from $\delta = 0.77$ for gmodS to $\delta = -0.007$ for deadS. This means that when points-to set size doubles, gmodS increases by 70%, while deadS hardly changes. This is consistent with the results of the previous section, where we found that some program analyses are much more sensitive than others to the choice of pointer analysis.

3.3 Effect of choice of pointer analysis on total times

In this section we address the question of how the time required to perform a dataflow analysis or to build and slice a system dependence graph is affected by the choice of pointer-analysis algorithm. We use P_{ij} to denote the time for pointer analysis i on test program j; we use Q_{ij} as before to denote the time for a subsequent analysis, and we use T_{ij} to denote the total time for the pointer analysis plus the subsequent analysis. Thus:

$$T_{ij} = P_{ij} + Q_{ij}$$

Figure 8 shows the effect of choice of pointer-analysis algorithm on total time, and also how that time is divided into pointer-analysis time and subsequent-analysis time. The height of each bar is the geometric mean of the ratios:

$$\frac{T_{ij}}{T_{Aj}}$$

(*i.e.*, the ratio of the total time required using pointer analysis P_i to the total time required using Andersen's pointer analysis, for each of the test programs j). An $\times$ indicates how much of the total time was used just for the pointer analysis; *i.e.*, the height of the $\times$ is the geometric mean of the ratios:

$$\frac{P_{ij}}{T_{Aj}}$$

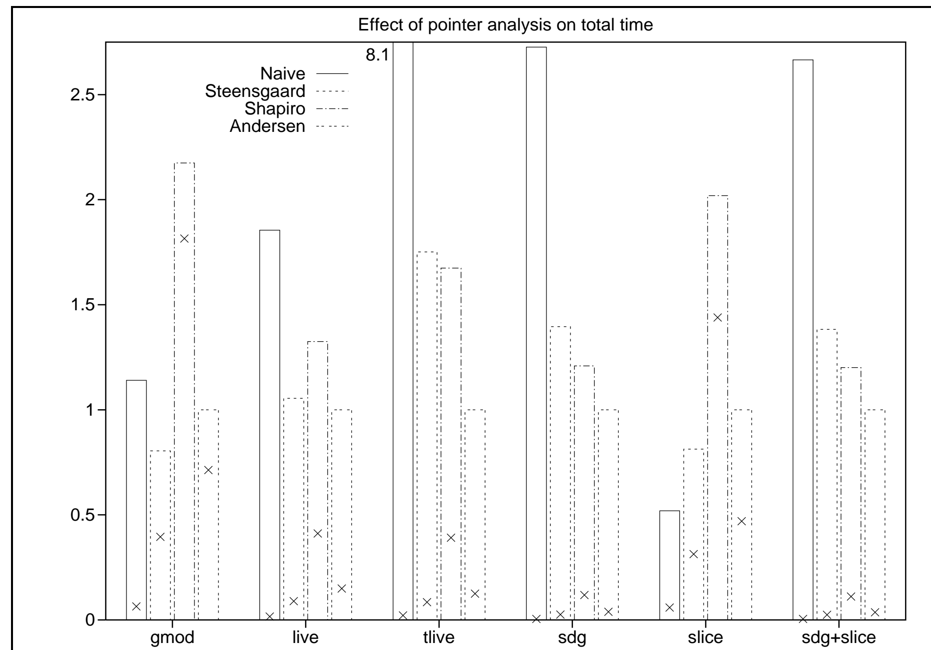

Fig. 8. The total times: time for pointer analysis plus time for subsequent analysis. The $\times$ marks the time for the pointer analysis.

It is interesting to note that for all of the program analyses except GMOD and slicing, the (average) total time is least when the most expensive pointer-analysis algorithm (Andersen's analysis) is used. This is because the increase in the accuracy of the points-to sets computed by Andersen's analysis leads to a decrease in the sizes of the use/kill/def sets computed for each node of the programs' control-flow graphs, as well as a decrease in the sizes of the sets of functions considered to be potentially called at each indirect call site. In turn, these smaller sizes lead to smaller dataflow domains, and thus to faster dataflow analyses.

For GMOD analysis, it is Steensgaard's pointer analysis that leads to the fastest (average) total times. Although the times for GMOD analysis alone are faster using Andersen's analysis than using Steensgaard's (see Figure 3, column gmodT), the GMOD analysis is so efficient that the decrease in GMOD time using Andersen's analysis is not enough to make up for the increase in pointer-analysis time. (But don't forget that the results of GMOD analysis are significantly better using Shapiro's and Andersen's analyses than using Steensgaard's.)

The data for slicing given in Figure 8 (the bars for "slice") are perhaps misleading, since they represent only pointer-analysis time plus slicing time (*i.e.*, they omit the times required to build the system dependence graphs)[4]. A more accurate picture is provided by the final set of bars ("sdg+slice"), which represent pointer-analysis time plus time to build the system dependence graphs plus time to compute slices. In this case, it is again Andersen's algorithm that leads to the fastest (average) times.

The graphs shown in Figures 5 and 8 seem to indicate that Andersen's algorithm is the algorithm of choice: not only does it lead to more accurate final results, but it also leads to the fastest total times in most cases. However, it is important to remember that Andersen's algorithm is, in the worst case, cubic in the size of the program, while Steensgaard's and Shapiro's algorithms are close to linear. Therefore, it is possible that this result (Andersen's algorithm leading to the fastest total times) will not hold as program size increases. If this is true, then it is also possible that the advantage of increased accuracy may eventually be outweighed by the disadvantage of increased running times. In such cases, it is possible that (a more efficient version of) Shapiro's algorithm may become the algorithm of choice, providing a good compromise with fast running time and reasonable accuracy.

Our experiments on large programs were limited by the memory requirements of our dataflow-analysis and slicing implementations. Nevertheless, we can get some indication of how things might change for large programs by looking only at the programs on which Andersen's analysis ran most slowly compared to Steensgaard's analysis: *screen*, *find*, *espresso*, *tar*, *less*, *make*, *gcc.cpp*, *sed*, *grep*, and *triangle*. Figure 9 shows the same results as Figure 8, limited to the ten test programs listed above[5]. In Figure 9, the total times using Shapiro's analysis are much closer to the total times using Andersen's analysis. It is possible that with a better implementation of Shapiro's analysis and with larger test programs, the total times using Shapiro's analysis could be quite a bit faster than those using Andersen's analysis.

4 Conclusions

We find that there are significant transitive effects of pointer analysis. A more precise pointer analysis leads to later analyses that are more precise and faster. Some analyses are more sensitive to the choice of pointer analysis than others. Using a pointer analysis with an average points-to set size twice as large as a more precise pointer analysis led to an increase of about 70% in the size of the GMOD sets, and in the time to perform

[4] As we can see from Figure 2, column sliceS, the slices computed using the Naive analysis are only very slightly larger than those computed using the other analyses. Since the time to compute a slice is proportional to its size, the times for slicing alone are only slightly slower using the Naive analysis than using the more accurate analyses (see Figure 3, column sliceT). Since the Naive analysis is the fastest, the total times (pointer analysis plus slicing) are fastest when Naive analysis is used.

[5] Some of the program analyses ran out of memory on some of these test programs. The results shown in Figure 9 represent data for all ten programs for GMOD analysis, seven programs for live analysis, four programs for truly-live analysis, and just two programs for slicing.

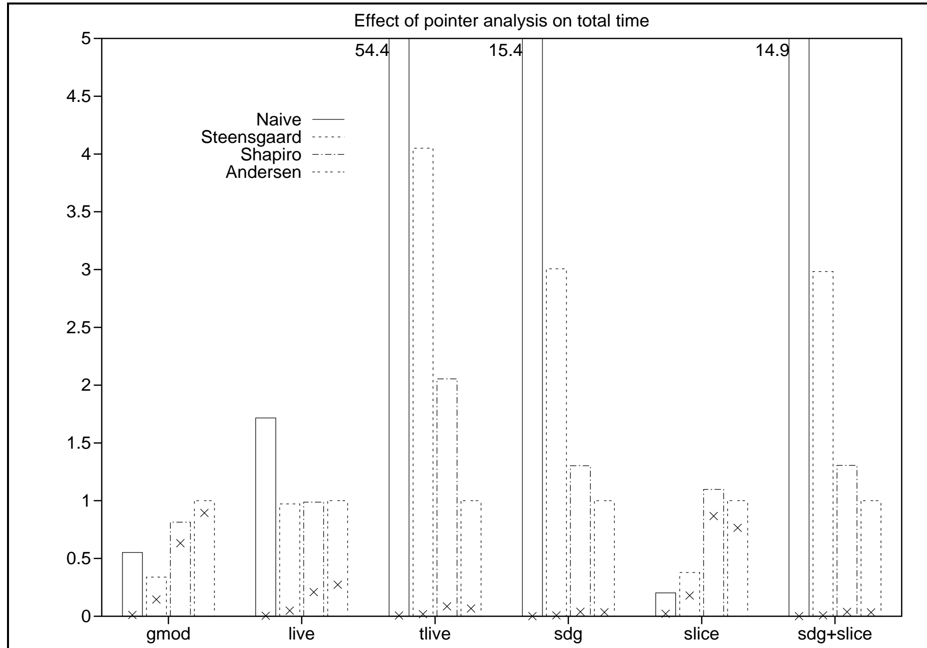

Fig. 9. The total times for the programs on which Andersen's analysis ran most slowly. The × marks the time for the pointer analysis.

truly-live analysis. Other effects were more modest: the number of dead assignments found, and the size of a slice hardly changed.

On small programs, Andersen's algorithm not only produced the most accurate results, it also led to the fastest overall run-times. For some large programs, however, Andersen's algorithm can be very slow, and the overall analysis time might be smaller with Shapiro's algorithm or Steensgaard's algorithm.

Of course, there are many questions that were not addressed by our experiments, including:

1. What are the relative benefits of flow-sensitive and flow-insensitive pointer analyses? When will the additional precision of a flow-sensitive analysis lead to enough of a reduction in subsequent analysis time to offset its cost?
2. Do the results we have obtained for GMOD analysis, live variable analysis, truly-live variable analysis, and interprocedural slicing hold for other program analyses?
3. Do our results (especially the result that more precise points-to sets often lead to much faster dataflow analysis) hold when other dataflow-analysis algorithms are used?
4. Are there properties of a program analysis that can be used to predict the likely benefits of using a more precise pointer analysis, both in terms of the accuracy of the analysis results, and the total time requirements?

5 Acknowledgements

We would like to thank Bob Wardrop and Jonathan Goldstein for their helpful discussions of statistics.

References

[ABS94] T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 290–301, June 1994.

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[EGH94] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN Conference on Programming Languages Design and Implementation*, 1994.

[GMW81] R. Giegerich, U. Moncke, and R. Wilhelm. Invariance of approximative semantics with respect to program transformation. In *GI 81 - 11th GI Annual Conference, Informatik-Fachberichte 50*, pages 1–10, New York, NY, 1981. Springer-Verlag.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[HRSR94] S. Horwitz, T. Reps, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 11–20, December 1994. (Available on the WWW from URL http://www.cs.wisc.edu/wpis/papers/fse94.ps).

[Kil73] G.A. Kildall. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206, January 1973.

[LR92] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 235–248, June 1992.

[LRZ93] W. Landi, B. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 56–67, June 1993.

[NW74] John Neter and William Wasserman. *Applied Linear Statistical Models*, chapter 5.5, pages 156–159. Richard D. Irwin, Inc., 1974.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM Symposium on Principles of Programming Languages*, pages 49–61, January 1995. (Available on the WWW from URL http://www.cs.wisc.edu/wpis/papers/popl95.ps).

[Ruf95] E. Ruf. Context-sensitive alias analysis reconsidered. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 13–22, June 1995.

[SH97] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *ACM Symposium on Principles of Programming Languages*. ACM, New York, January 1997.

[Ste96] B. Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[Tar83] R. Tarjan. Data structures and network flow algorithms. volume CMBS44 of *Regional Conference Series in Applied Mathematics*. SIAM, 1983.

[Wei84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[WL95] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

This article was processed using the LaTeX macro package with LLNCS style