Which Pointer Analysis Should I Use?

Michael Hind
IBM Watson Research Center
30 Saw Mill River Road
Hawthorne, NY 10532, USA
hind@watson.ibm.com

Anthony Pioli
Register.com
575 11th Ave
New York, NY 10018, USA
anthony@register.com

ABSTRACT

During the past two decades many different pointer analysis algorithms have been published. Although some descriptions include measurements of the effectiveness of the algorithm, qualitative comparisons among algorithms are difficult because of varying infrastructure, benchmarks, and performance metrics. Without such comparisons it is not only difficult for an implementor to determine which pointer analysis is appropriate for their application, but also for a researcher to know which algorithms should be used as a basis for future advances.

This paper describes an empirical comparison of the effectiveness of five pointer analysis algorithms on C programs. The algorithms vary in their use of control flow information (flow-sensitivity) and alias data structure, resulting in worst-case complexity from linear to polynomial. The effectiveness of the analyses is quantified in terms of compile-time precision and efficiency. In addition to measuring the direct effects of pointer analysis, precision is also reported by determining how the information computed by the five pointer analyses affects typical client analyses of pointer information: Mod/Ref analysis, live variable analysis and dead assignment identification, reaching definitions analysis, dependence analysis, and conditional constant propagation and unreachable code identification. Efficiency is reported by measuring analysis time and memory consumption of the pointer analyses and their clients.

Keywords

Interprocedural pointer analysis, data flow analysis

1. INTRODUCTION

Programs written in languages with pointers can be troublesome to analyze because the memory location accessed through a pointer is not known by inspecting the statement. To effectively analyze such languages, knowledge of pointer behavior is required. Without such knowledge, conservative assumptions about memory locations accessed through a pointer must be made. These assumptions can adversely affect the precision and efficiency of any analysis that requires this information, such as a program understanding system, an optimizing compiler, or a testing tool.

A pointer analysis is a compile-time analysis that attempts to determine the possible values of a pointer. As such an analysis is, in general, undecidable [16, 28], many approximation algorithms have been developed that provide a trade-off between the efficiency of the analysis and the precision of the computed solution. The worst-case time complexities of these analyses range from linear to exponential. Because such worst-case complexities are often not a true indication of analysis time, many researchers provide empirical results of their algorithms. However, comparisons among results from different researchers can be difficult because of differing program representations, benchmark suites, and precision/efficiency metrics. In this work, we describe a comprehensive study of five widely used pointer analysis algorithms that holds these factors constant, thereby focusing more on the efficacy of the algorithms and less on the manner in which the results were obtained.

The main contributions of this paper are the following:

- empirical results that measure the precision and efficiency of five pointer alias analysis algorithms with varying degrees of flow-sensitivity and alias data structures: Address-taken, Steensgaard [34], Andersen [1], Burke et al. [4, 12], Choi et al. [5, 12];
- empirical data on how the pointer analyses solutions affect the precision and efficiency of the following client analyses: Mod/Ref, live variable analysis and dead assignment identification, reaching definition analysis, dependence analysis, and interprocedural conditional constant propagation and unreachable code identification.

The results show (1) Steensgaard's analysis is significantly more precise than the Address-taken analysis in terms of direct precision and client precision, (2) Andersen's and Burke et al.'s analyses provide the same level of precision and a modest increase in precision over Steensgaard's analysis, (3) the flow-sensitive analysis of Choi et al. offers only a minimal increase in precision over the analyses of Andersen and Burke et al. using a direct metric and little or no precision improvement in client analyses, and (4) increasing the precision of pointer information reduces the client analyses' input, resulting in significant improvement in their efficiency.

The remainder of this paper is organized as follows. Section 2 describes background for this work and describes how it differs from similar studies. Section 3 provides an overview of the five pointer algorithms. Section 4 summarizes the client analyses. Section 5 describes the empirical study and discusses the results. Section 6 describes related work and Section 7 summarizes the conclusions.

2. BACKGROUND

A *pointer alias analysis* attempts to determine when two pointer expressions refer to the same storage location. For example, if p and q both point to the same storage location, we say $*p$ and $*q$ are aliases, written as $\langle *p, *q\rangle$. A *points-to analysis* attempts to determine what storage location a pointer can point to. This information can then be used to determine the aliases in the program. This works uses the *compact representation* [5, 12] of alias information, which shares the property of the *points-to* representation [8], in that it captures the "edge" characteristic of alias relations.[1] For example, if variable a points to b, and b points to c, the compact representation records only the following alias set: $\{\langle *a, b\rangle, \langle *b, c\rangle\}$, from which it can be inferred that $\langle ** a, c\rangle$ and $\langle ** a, *b\rangle$ are also aliases. The cost and time when such information is inferred can affect the precision and efficiency of the analysis [22, 12, 20].

Interprocedural data-flow analysis can be classified as *flow-sensitive* or *flow-insensitive*, depending on whether control-flow information of a procedure is used during the analysis [23]. By not considering control flow information, and therefore computing a conservative summary, a flow-insensitive analysis can be more efficient, but less precise than a flow-sensitive analysis. In addition to flow-sensitivity, there are several other factors that affect cost/precision trade-offs including

Context-sensitivity: Is calling context considered when analyzing a function?

Heap modeling: Are objects named by allocation site or is a more sophisticated shape analysis performed?

Struct modeling: Are components distinguished or collapsed into one object?

Alias representation: Is an explicit alias representation or a points-to/compact representation used?

This work holds these factors constant, choosing the most popular and efficient alternatives in each case, so that the results only vary the usage of flow-sensitivity. In particular, all analyses are context-insensitive, name heap objects based on their allocation site, collapse aggregate components, and use the compact/points-to representation.

This work differs from previous studies [33, 35, 7, 21] in the following ways:

- The breadth of pointer algorithms studied; in the only two studies [35, 21] that also include a flow-sensitive analysis, the analysis they study [18] also benefits from being context sensitive and uses a different alias representation (an explicit one) than the (points-to) flow-insensitive analyses it is compared with.
- The number of client analyses reported; this work is the first to report how reaching definitions, flow dependences, and interprocedural constant propagation are affected by the quality of pointer analysis.
- The reporting of memory usage, which is an important aspect in evaluating the scalability of interprocedural data-flow analyses.

3. POINTER ANALYSES

The algorithms we consider, listed in order of increasing precision, are

Address-taken: a flow-insensitive algorithm often used in production compilers that records all variables whose addresses have been assigned to another variable. This set includes all heap objects and actual parameters whose addresses are stored in the corresponding formal. This analysis is efficient because it is linear in the size of the program and uses a single solution set, but can be very imprecise.

Steensgaard [34]: a flow-insensitive algorithm that computes one solution set for the entire program and employs a fast union-find [36] data structure to represent all alias relations. This results in an almost linear time algorithm that makes one pass over the program. Similar algorithms are discussed in [42, 24, 2].

Andersen [1]: an iterative implementation of Andersen's context-insensitive flow-insensitive algorithm, which was originally described using constraint-solving [1]. Although it also uses one solution set for the entire program, it can be more precise than Steensgaard's algorithm because it does not perform the merging required by the union-find data structure. However, it does require a fixed-point computation over all pointer-related statements that do not produce constant alias relations.

Burke et al. [4, 12]: a flow-insensitive algorithm that also iterates over all pointer-related statements in the program. It differs from Andersen's analysis in that it computes an alias solution for each procedure, requiring iteration within each function in addition to iteration over the functions. A worklist is used in the latter case to improve efficiency. Distinguishing alias sets for each function allows precision-improving enhancements such as using precomputed kill information [4, 12].[2] Burke et al.'s analysis can be more precise than Andersen's analysis because it can filter alias information based on scoping, i.e., formals and locals from provably nonactive functions are not considered. It

[1]The minor difference between the compact and points-to representations [12] is not relevant to this work.

[2]This particular enhancement never improved precision over the Burke et al.'s analysis studied in this paper [13]. Thus, the enhanced version of Burke et al.'s analysis that uses precomputed kill information is not included in this study.

```
        T *p, *q, *r;
        void main() {
S1:         p = new T;
S2:         f();
S3:         g(&p);
S4:         p = new T;
S5:         ... = *p;
        }
```

```
        void f() {
S6:         q = new T;
S7:         g(&q);
S8:         r = new T;
        }
```

```
        void g(T** fp) {
            T local;
S9:         if (...)
S10:            p = &local;
            ...
        }
```

Figure 1: Example program

may be less efficient because it computes a solution set for each function, rather than one for the whole program.

Choi et al. [5, 12]: a flow-sensitive algorithm that computes a solution set for every program point. It associates alias sets with each CFG node in the program and uses worklists for efficiency [13].

All analyses incorporate (optimistic) function pointer analysis during the alias analysis by resolving indirect call sites as the analysis proceeds [8, 4].

In theory, each subsequent analysis is more precise (and costly) than its predecessors. This paper will help quantify not only these characteristics, but also how client analyses are affected by the precision of the pointer analyses.

Consider the program in Figure 1, where `main` calls `f` and `g`, and `f` also calls `g`. The Address-taken analysis computes only one set of objects that it assumes all pointers may point to: $\{heap_{S1}$, $heap_{S4}$, $heap_{S6}$, $heap_{S8}$, `local`, `p`, `q`$\}$, all of which will appear to be referenced at $S5$.

Steensgaard's analysis unions two objects that are pointed-to by the same pointer into one object. This leads to the unioning of the points-to sets of these formerly distinct objects. This unioning removes the necessity of iteration from the algorithm. In the example, the formal parameter of `g`, `fp`, may point to either `p` or `q`, resulting in `p` and `q` being unioned into one object. Thus, it appears that they both can point to the heap objects that either can point to: $heap_{S1}$, $heap_{S4}$, $heap_{S6}$, and `local`. At the dereference of `S5`, these four objects are reported aliased to `*p`.

Andersen's analysis also keeps one set of aliases that can hold anywhere in the program, but it does not merge objects that have a common pointer point to them. This leads to $heap_{S1}, heap_{S4}$, and `local` being reported as aliased to `*p`.

Burke et al.'s analysis associates one set with every function, which conservatively represents what may hold at any CFG node in the function, without considering control flow within the function. This distinction allows the removal of objects that are no longer active, such as `local` in functions `main` and `f`. This leads to $heap_{S1}$ and $heap_{S4}$ being aliased to `*p` at `S5`.

Choi et al.'s analysis associates an alias set before (In_n) and after (Out_n) every CFG node, n. For example, Out_{S1} $= \{\langle *p, heap_{S1}\rangle\}$ because $*p$ and $heap_{S1}$ refer to the same storage after $S1$. Choi et al.'s analysis will compute $In_{S5} = \{\langle *p, heap_{S4}\rangle\}$, which is the precise solution for this simple example.

This example illustrates the theoretical precision/efficiency levels of the five analyses we study, from Address-taken (least precise) to Choi et al.'s (most precise). The Address-taken analysis is our most efficient analysis because it is linear and uses only one set. Steensgaard's analysis also uses one set and is almost linear. The other three analyses require iteration, but differ in the amount of information stored from one alias set per program (Andersen), one set per function (Burke et al.), and two per CFG Node (Choi et al.).[3]

The analyses have been implemented in the NPIC system, an experimental program analysis system written in C++. The system uses multiple and virtual inheritance to provide an extensible framework for data-flow analyses [14, 26]. A prototype version of the IBM VisualAge C++ compiler [15] is used as the front end. The analyzed program is represented as a program call (multi-) graph (PCG), in which a node corresponds to a function, and a directed edge represents a call to the target function.[4] Each function body is represented by a control flow graph (CFG), where each node roughly corresponds to a statement. This graph is used to build a simplified sparse evaluation graph (SEG) [6], which is used by Choi et al.'s analysis in a manner similar to Wilson [39]. As no CFG is available for library functions, a call to a library function is modeled based on the function's semantics with respect to pointer analysis. This hand-coded modeling provides the benefits of context-sensitive analysis of such calls. Library calls that cannot affect the value of a pointer are treated as the identity transfer function for pointer analysis. The implementation also assumes that pointer values will only exist in pointer variables, and that pointer arithmetic does not result in the pointer outside of an array. All string literals are modeled as one object. The implementation handles `setjmp/longjmp` in a manner similar to Wilson [39]; all calls to `setjmp` are recorded and used to determine the possible effects of a call to `longjmp`.

To model the values passed as `argc` and `argv` to the `main` function, a dummy `main` function was added, which called the benchmark's `main` function, simulating the effects of

[3]We have found that performing Choi et al.'s analysis using a SEG (sparse evaluation graph [6]) instead of a CFG reduces the number of alias sets by an average of over 73% and reduces analysis time by an average of 280% [13].

[4]Indirect calls can result in several potential target functions.

argc and argv. This function also initialized the _iob array, used for standard I/O. The added function is similar to the one added by Ruf [29, 30] and Landi et al. [19, 17]. Explicit and implicit initializations of global variables are automatically modeled as assignment statements in the dummy main function. Array initializations are expanded into an assignment for each array component.

4. CLIENT ANALYSES

This section summarizes the client analyses used in this study.

4.1 Mod/Ref Analysis

Mod/Ref analysis [20] determines what objects may be modified/referenced at each CFG node. This information is subsequently used by other analyses, such as reaching definitions and live variable analysis. This information is computed by first visiting each CFG node and computing what objects are modified or referenced by the node. Pointer dereferences generate a query of the alias information to determine the objects modified. These results (Mod and Ref sets) are summarized for each function and used at call sites to the function. A call site's Mod/Ref set does not include a local of a function that cannot be on the activation stack because its lifetime is not active. The actual parameters at each call site are assumed to be referenced because their value is assigned to the corresponding formal parameter (pass-by-value semantics). The Mod/Ref analysis makes the simplifying assumption that libraries do not modify or reference locations indirectly through a pointer parameter. Fixed-point iteration is employed when the program has PCG cycles.

4.2 Live Variable Analysis

Live variable analysis [25] determines what objects may be referenced after a program point without an intervening killing definition. This information is useful for register allocation, detecting uninitialized variables, and finding dead assignments. The implementation, a backward analysis, directly uses the Mod/Ref information. It associates two sets of live variables with each CFG node representing what is live before and after execution of the node. Sharing of such sets is performed when a CFG node has only one successor, or when the node acts as an identify function, i.e., it has an empty Mod and Ref set.

All named objects in the Ref set of a CFG node become live before that node. A named object is killed at a CFG node if it is definitely assigned (i.e., it is the only element in the Mod set of a noncall node) and represents one runtime object, i.e., it is not an aggregate, a heap object, or a local/formal of a recursive function. The implementation processes each function once, employing a priority-based worklist of CFG nodes for each function. It is optimistic; no named objects are considered live initially, except at the exit node where all nonlocals that are modified in the function are considered to be live.

4.3 Reaching Definitions Analysis

Reaching definitions analysis [25] determines what definitions of named objects may reach (in an execution sense) a program point. This information is useful in computing data dependences among statements, an important step for program slicing [37] and code motion. The implementation, a forward analysis, uses Mod/Ref information and associates two sets of reaching definitions with each CFG node. Set sharing is performed as in live variable analysis.

All named objects in the Mod set of a CFG node result in new definitions being generated at that node. Definitions are killed as in live variable analysis. Each function is processed once, using a priority-based worklist of CFG nodes for each function. The analysis is optimistic; no definitions are initially considered reaching any point, except for dummy definitions created at the entry node of a function for each parameter or nonlocal that is referenced in the function.

4.4 Interprocedural Constant Propagation

The constant propagation client [26] is an optimistic interprocedural algorithm inspired by Wegman and Zadeck's *Conditional Constant* algorithm [38]. The algorithm tracks values of variables interprocedurally throughout the program and uses this information to simultaneously evaluate conditional branches where possible, thereby determining if a conditional branch will always evaluate to one value. In addition to potentially removing unexecutable code, this analysis can simplify computations and provide useful information for cloning algorithms.

Because this analysis was designed to be combined with Choi et al.'s pointer analysis [26, 27], it uses pointer information directly, rather than using the Mod/Ref sets as was done in reaching definitions and live variable analysis. In this work, the constant propagation analysis is simply run after the pointer analysis is completed. Like Choi et al.'s analysis, the constant propagation algorithm uses nested iteration and a SEG.[5] The algorithm extends the traditional lattice of $\top$, $\bot$, and *constant* to include *Positive*, *Negative*, and *NonZero*. This can help when analyzing C programs that treat nonzero values as true.

5. RESULTS

This study was performed on a 333MHz IBM RS/6000 PowerPC 604e with 512MB RAM and 817MB paging space, running AIX 4.3. The analyses were compiled with IBM's xlC compiler using the "-O3" option. For each benchmark the following are reported for all pointer analyses and clients: precision, analysis time, and the maximum memory usage. Table 1 describes characteristics of the benchmark suite, which contains 23 C programs provided by other researchers [19, 8, 29, 31] and the SPEC benchmark suites.[6] LOC is computed using wc on the source and header files. The column marked "Fcts", the number of user-defined functions, includes the dummy main function, created to simu-

[5]However, the SEG benefits are not as dramatic; most CFG nodes are "interesting" to constant propagation, and thus, the efficiency is typically worse than Choi et al.'s analysis.

[6]The large number of CFG nodes for 129.compress results from the explicit creation of assignment statements for implicit array initialization. Some programs had to be syntactically modified to satisfy C++'s stricter type-checking semantics. A few program names are different than those reported in [29]. Namely, ks was referred to as part, and ft as span [30]. Also, the SPEC CINT92 program 052.alvinn was named backprop in Todd Austin's benchmark suite [3].

Table 1: Static Characteristics of Benchmark Suite.

Name	Source	LOC	CFG Nodes	Fcts	Ptr-Asg Nodes Pct
allroots	Landi	227	159	7	1.3%
052.alvinn	SPEC92	272	229	9	10.0%
01.qbsort	McCat	325	170	8	24.1%
06.matx	McCat	350	245	7	13.5%
15.trie	McCat	358	167	13	23.4%
04.bisect	McCat	463	175	9	9.7%
fixoutput	PROLANGS	477	299	6	4.4%
17.bintr	McCat	496	193	17	8.8%
anagram	Austin	650	346	16	9.5%
ks	Austin	782	526	14	27.4%
05.eks	McCat	1,202	677	30	4.0%
08.main	McCat	1,206	793	41	20.9%
09.vor	McCat	1,406	857	52	28.6%
loader	Landi	1,539	691	30	8.8%
129.compress	SPEC95	1,934	17,012	25	0.2%
ft	Austin	2,156	775	38	18.6%
football	Landi	2,354	2,854	58	1.8%
compiler	Landi	2,360	1,767	40	5.1%
assembler	Landi	3,446	1,845	52	16.6%
yacr2	Austin	3,979	2,070	59	6.6%
simulator	Landi	4,639	2,929	111	6.3%
flex	PROLANGS	7,659	7,107	88	5.2%
099.go	SPEC95	29,637	31,788	373	1.5%

late command-line argument passing. The column marked "Ptr-Asg Nodes Pct" reports the percentage of CFG nodes that are considered pointer-assignment nodes, i.e., the number of assignment nodes where the left side variable involved in the pointer expression is declared to be a pointer.

5.1 Pointer Analysis Precision

The most direct way to measure the precision of a pointer analysis is to record the number of objects aliased to a pointer expression appearing in the program. Using this metric, Andersen's and Burke et al.'s analyses provide the same level of precision for all benchmarks, suggesting that alias relations involving formals or locals from provably non-active functions do not occur in this benchmark suite. Because all client analyses use the alias solution computed by these analysis as their input, there is, likewise, no precision difference in these clients. For this reason, we group these two analysis together in the precision data. The efficiency results of Section 5.6 distinguish these analyses.

A pointer expression with multiple dereferences, such as `***p`, is counted as multiple dereference expressions, one for each dereference. The intermediate dereferences (`*p` and `**p`) are counted as reads. The last dereference (`***p`) is counted as a read or write depending on the context of the expression. Statements such as `(*p)++` and `*p += increment` are treated as both a read and a write of `*p`. A pointer is considered to be dereferenced if the variable is declared as a pointer or an array formal parameter, and one or more of the "*", "->", or "[ ]" operators are used with that variable. Formal parameter arrays are included because their corresponding actual parameter(s) could be a pointer. We do not count the use of the "[ ]" operator on arrays that are not formal parameters because the resulting "pointer" (the array name) is constant, and therefore, counting it may skew results.

The left half of Table 2 reports the average size of the Mod and Ref sets for expressions containing a pointer dereference for each benchmark and the average of all benchmarks.[7] This table, and the rest in this paper, use "–" to signify a value that is the same as in the previous column. For example, the Ptr-Mod for `allroots` is the same for Choi et al.'s analysis and Andersen/Burke et al.'s analyses.

The results show

1. a substantial difference between the Address-taken analysis and Steensgaard's analysis: (i) an average of 30.26 vs. 4.03 and an improvement in all benchmarks that assigned through a pointer for Ptr-Mod, and (ii) an average of 30.70 vs. 4.87 and an improvement in all benchmarks for Ptr-Ref;
2. a measurable difference between Steensgaard's analysis and Andersen/Burke et al.'s analyses: (i) an average of 4.03 vs. 2.06 and an improvement in 15 of the 22 benchmarks that assign through a pointer for Ptr-Mod, (ii) and an average 4.87 vs. 2.35 and an improvement in 13 of the 23 benchmarks for Ptr-Ref;
3. little difference between Andersen/Burke et al.'s analyses and Choi et al.'s analysis: (i) an average of 2.06 vs. 2.02 and an improvement in 5 of the 22 benchmarks that assign through a pointer for Ptr-Mod, and (ii) 2.35 vs. 2.29 and an improvement in 5 of the 23 benchmarks for Ptr-Ref.

In summary, varying degrees of increased precision can be gained by using a more precise analysis. However, as more precise algorithms are used, the improvement diminishes.

5.2 Mod/Ref Precision

The right half of Table 2 reports the average Mod/Ref set size for *all* CFG nodes. This captures how the pointer analysis affects Mod/Ref analysis, which serves as input to many other analyses. The results show

1. a substantial difference between the Address-taken analysis and Steensgaard's analysis: (i) an average of 2.50 vs. 1.04 and an improvement in 22 of 23 benchmarks for Mod, and (ii) 4.48 vs. 1.75 and an improvement in all 23 benchmarks for Ref;
2. a measurable difference between Steensgaard's analysis and Andersen/Burke et al.'s analyses: (i) an average of 1.04 vs. .87 and an improvement in 13 of 23 benchmarks for Mod, and (ii) 1.75 vs. 1.54 and an improvement in 11 of 23 benchmarks for Ref;
3. little difference between Andersen/Burke et al.'s analyses and Choi et al.'s analysis: (i) an average of .871 vs. .867 and an improvement in 3 of 23 benchmarks for both Mod; and (ii) an average of 1.540 vs. 1.536 and an improvement in 4 of 23 benchmarks for Ref.

[7] The modeling of potentially many runtime objects with one representative object may seem more precise when compared to a model that uses more names [29, 20]. For example, if the heap was modeled as one object, all heap-directed pointers would be "resolved" to one object in Table 2.

Table 2: Mod and Ref at pointer dereferences and all CFG nodes. No assignments through a pointer occur in `compiler`. "AT" = Address Taken, "St" = Steensgaard's, "A/B"= Andersen/Burke et al., "Ch" = Choi et al.

	Ptr Mod				Ptr Ref				Mod				Ref			
Name	AT	St	A/B	Ch	AT	St	A/B	Ch	AT	St	A/B	Ch	AT	St	A/B	Ch
allroots	3	2.00	1.00	–	3	2.00	1.38	–	.88	.85	.83	–	1.77	1.58	1.52	–
052.alvinn	14	1.00	–	–	14	1.00	–	–	1.17	.51	–	–	2.24	1.00	–	–
01.qbsort	12	2.00	1.50	–	12	1.76	–	–	.95	.46	–	–	3.39	.94	–	–
06.matx	15	3.00	2.22	–	15	3.25	3.12	–	1.09	.39	.34	–	1.48	.89	.88	–
15.trie	10	1.12	–	1.00	10	1.00	–	–	1.02	.58	–	.52	4.42	.88	–	–
04.bisect	14	1.15	–	–	14	1.00	–	–	2.57	.58	–	–	3.92	1.57	–	–
fixoutput	9	1.80	–	–	9	2.00	–	–	.74	.37	–	–	.78	.56	–	–
17.bintr	7	1.00	–	–	7	1.00	–	–	.62	.30	–	–	2.07	.71	–	–
anagram	17	1.00	–	–	17	1.10	–	–	.90	.45	–	–	2.38	1.02	–	–
ks	17	1.90	1.86	1.62	17	1.79	–	1.74	1.70	.56	.55	.53	3.76	1.35	–	1.34
05.eks	12	1.22	–	–	12	1.02	–	–	1.83	.50	–	–	3.54	1.63	–	–
08.main	13	6.00	3.27	2.61	13	5.14	4.61	3.59	1.76	.63	–	–	5.35	1.32	1.30	1.25
09.vor	19	1.85	1.35	1.32	19	1.92	1.68	1.60	2.04	.63	.62	–	7.91	1.40	1.34	–
loader	47	3.77	2.23	–	47	2.09	1.36	–	5.08	.90	.73	–	9.72	1.78	1.30	–
129.compress	13	1.40	1.07	–	13	2.26	1.11	–	1.68	.80	.78	–	1.66	1.29	1.28	–
ft	10	2.87	1.80	1.72	10	2.66	2.53	2.39	2.14	.90	.74	.73	2.58	1.31	–	1.28
football	32	6.00	2.10	–	32	3.26	1.54	–	1.37	.70	.61	–	4.55	1.83	1.65	–
compiler	0	–	–	–	10	1.00	–	–	3.38	–	–	–	4.45	4.44	–	–
assembler	87	1.24	2.21	–	87	15.14	2.11	–	1.21	1.88	.87	–	15.07	4.09	1.47	–
yacr2	48	1.14	1.11	–	48	1.08	1.02	–	5.32	.53	.52	–	7.80	1.65	–	–
simulator	87	3.16	2.05	–	87	3.95	1.86	–	6.82	.62	.57	–	8.21	1.21	1.06	–
flex	56	5.37	1.78	–	56	5.09	2.03	2.01	5.97	1.60	1.18	–	1.55	3.89	3.44	3.43
099.go	154	42.68	13.64	–	154	51.39	17.03	–	7.31	5.87	3.94	–	4.47	3.98	3.13	–
Average	30.26	4.03	2.06	2.02	30.70	4.87	2.35	2.29	2.50	1.04	0.871	0.867	4.48	1.75	1.540	1.536

Once again varying degrees of increased precision can be gained by using a more precise analysis. However, the improvements are not as dramatic as in the previous metric, resulting in minimal precision gain from the flow-sensitive analysis.

5.3 Live Variable Analysis and Dead Assignment Identification

The first set of four columns in Table 3 reports precision results for live variable analysis. For each benchmark we list the average number of live variables at each CFG node and the average of these averages. Live variable information is used to find assignments to variables that are never used, i.e., a dead assignments. The second set of four columns gives the number of CFG nodes that are dead assignments.

The results show

1. a substantial difference between the Address-taken analysis and Steensgaard's analysis for live variables — on average 34.24 vs. 20.13 and an improvement in all benchmarks — but no difference for finding dead assignments;
2. a significant difference between Steensgaard's analysis and Andersen/Burke et al.'s analyses for live variables — an average of 20.13 vs. 18.36 and an improvement in 13 of 23 benchmarks — but less of a difference for finding dead assignments: an average of 1.91 vs. 1.96 and an improvement in only 1 of 23 benchmarks.
3. a small difference between Andersen/Burke et al.'s analyses and Choi et al.'s analysis for live variables — an average of 18.36 vs. 18.30 and an improvement in 3 of 23 benchmarks — but no difference for finding dead assignments.

In summary, more precise pointer analyses improved the precision of live variable analysis, but Choi et al.'s analysis provided only minimal improvement. In contrast, dead assignments identification was hardly affected by using different pointer analyses.

5.4 Reaching Definitions and Flow Dependences

The third set of four columns in Table 3 reports precision results for reaching definitions analysis. For each benchmark we list the average number of definitions that reach a CFG node. The last set of four columns reports the average number of unique flow dependences between two CFG nodes per function. This metric captures reaching definitions that are used at a CFG node, but counts dependences between the same two nodes only once. Thus, if a set of variables are potentially defined at one node and potentially used at another node, only one dependence is counted because only one such dependence is needed to prohibit code motion of the two nodes or to be part of a slice.

The results show

1. a significant difference between the Address-taken analysis and Steensgaard's analysis: (i) an average of 36.39 vs. 22.04 and an improvement in all 23 benchmarks for reaching definitions, and (ii) an average of 52.51 vs. 44.24 and an improvement in 21 of 23 benchmarks for flow dependences;
2. a measurable difference between Steensgaard's analysis and Andersen/Burke et al.'s analyses: (i) an aver-

Table 3: Live variables, dead assignments, reaching definitions, and flow dependences. "AT" = Address Taken, "St" = Steensgaard's, "A/B"= Andersen/Burke et al., "Ch" = Choi et al.

	Avg live variables at a Node				Total dead assignments				Avg reaching defs at a node				Avg flow deps per function			
Name	AT	St	A/B	Ch	AT	St	A/B	Ch	AT	St	A/B	Ch	AT	St	A/B	Ch
allroots	18.11	17.19	17.08	–	0	–	–	–	20.19	19.23	–	–	36.33	–	–	–
052.alvinn	12.94	7.83	–	–	0	–	–	–	13.08	7.84	–	—	30.75	28.38	–	–
01.qbsort	14.36	6.96	–	–	0	–	–	–	15.54	7.02	–	–	37.29	28.86	–	–
06.matx	21.52	14.97	13.08	–	0	–	–	–	23.12	16.22	14.73	–	41.00	38.00	–	–
15.trie	11.77	5.94	–	5.39	0	–	–	–	13.23	5.22	–	5.11	11.08	9.58	–	–
04.bisect	21.45	12.83	–	–	0	–	–	–	23.24	14.80	–	–	58.67	46.67	–	–
fixoutput	15.39	13.43	–	–	0	–	–	–	15.38	10.31	–	–	74.00	46.60	–	–
17.bintr	7.20	3.24	–	–	0	–	–	–	8.34	3.45	–	–	12.50	9.67	–	–
anagram	17.70	10.67	–	–	0	–	–	–	17.17	10.92	–	–	29.00	25.47	–	–
ks	23.55	13.84	–	–	0	–	–	–	23.24	15.40	–	15.34	72.08	59.69	–	–
05.eks	18.29	10.23	–	–	0	–	–	–	17.89	9.45	–	–	37.93	35.21	–	–
08.main	17.89	10.32	10.26	9.74	7	–	–	–	19.82	12.26	11.68	11.20	36.93	26.50	–	–
09.vor	20.33	6.96	6.85	–	2	–	–	–	23.68	7.35	7.26	–	34.92	26.52	26.20	–
loader	50.16	21.76	16.97	–	0	–	–	–	51.68	22.60	17.83	–	49.52	32.21	31.38	–
129.compress	24.09	14.86	14.60	–	1	–	–	–	25.62	17.59	17.53	–	38.39	36.78	36.56	–
ft	17.06	11.46	11.31	11.10	1	–	–	–	16.79	11.17	11.11	10.89	40.67	37.93	37.78	–
football	38.09	24.70	23.25	–	2	–	–	–	41.08	26.63	25.15	–	69.12	66.93	65.56	–
compiler	43.73	43.70	–	–	0	–	–	–	47.16	47.09	–	–	88.35	88.32	–	–
assembler	82.24	37.36	20.75	–	1	–	–	–	85.40	40.60	22.58	–	86.02	49.00	47.53	–
yacr2	56.00	18.50	18.46	–	1	–	–	–	57.18	21.39	21.35	–	51.86	46.86	–	–
simulator	60.10	10.28	9.70	–	1	–	–	–	61.84	13.30	12.74	–	43.10	33.59	33.53	–
flex	107.70	72.48	65.07	–	1	–	–	–	119.30	84.43	77.17	77.05	116.50	97.22	94.74	–
099.go	87.87	73.37	66.12	–	27	–	28	–	96.91	82.72	74.91	–	111.70	111.10	108.90	–
Average	34.24	20.13	18.36	18.30	1.91	–	1.96	–	36.39	22.04	20.21	20.16	52.51	44.24	43.84	–

age of 22.04 vs. 20.21 and an improvement in 12 of 23 benchmarks for reaching definitions, and (ii) an average of 44.24 vs. 43.84 and an improvement in 9 of 23 benchmarks for flow dependences;

3. a negligible difference between Andersen/Burke et al.'s analyses and Choi et al.'s analysis for reaching definitions — an average of 20.21 vs. 20.16 and an improvement in 5 of 23 benchmarks — but no difference in flow dependences for any benchmark.

In summary, each successively more precise analysis results in an improvement of precision of reaching definitions, but this improvement is diminished when flow dependences are computed. In particular, there is no gain in flow dependences precision using Choi et al.'s analysis over Andersen/Burke et al.'s analyses and only minor improvements in using Andersen/Burke et al.'s analyses over Steensgaard's analysis.

5.5 Constant Propagation and Unexecutable Code Detection

The constant propagation precision results are shown in Table 4. After the benchmark name the first four columns give the number of complete expressions found to be constant. This metric does not count subexpressions such as "`b`" in "`...=b+c;`". The next four columns report the number of unexecutable nodes found by the analysis. The results show

1. a significant difference between the Address-taken analysis and Steensgaard's analysis: (i) an average of 7.8 vs. 10.6 constants found, but an improvement in only 3 of 22 benchmarks, and (ii) an average of 3.2 vs. 25.3 unexecutable nodes detected, but an improvement in only 2 of 22 benchmarks;

Table 4: Constants and unexecutable CFG nodes found. "AT" = Address Taken, "St" = Steensgaard's, "A/B"= Andersen/Burke et al., "Ch" = Choi et al. `099.go` is not included because it exhausts the 200MB heap size.

	Constants				Unexecutable Nodes			
Name	AT	St	A/B	Ch	AT	St	A/B	Ch
allroots	21	–	–	–	12	–	–	–
052.alvinn	2	–	–	–	0	–	–	–
01.qbsort	0	–	–	–	0	–	–	–
06.matx	3	–	–	–	1	–	–	–
15.trie	0	–	–	–	0	–	–	–
04.bisect	0	2	–	–	0	–	–	–
fixoutput	0	–	–	–	0	–	–	–
17.bintr	3	–	–	–	6	–	–	–
anagram	3	–	–	–	6	–	–	–
ks	0	–	–	–	0	–	–	–
05.eks	0	–	–	–	0	–	–	–
08.main	36	–	–	–	16	–	–	–
09.vor	13	–	–	–	0	–	–	–
loader	8	–	9	–	0	–	1	–
129.compress	34	–	–	–	5	–	–	–
ft	6	–	–	–	2	–	–	–
football	0	–	–	–	0	–	–	–
compiler	7	–	–	–	0	–	–	–
assembler	7	–	–	–	0	–	–	–
yacr2	4	–	–	–	1	–	–	–
simulator	10	11	–	–	5	6	–	–
flex	15	74	–	–	16	502	–	–
Average	7.8	10.6	10.7	–	3.2	25.3	25.4	–

2. a negligible difference between Steensgaard's analysis and Andersen/Burke et al.'s analyses: (i) an average of 10.6 vs. 10.7 constants found, an improvement in only 1 of 22 benchmarks, and (ii) an average of 25.3 vs. 25.4 unexecutable nodes detected, an improvement in only 1 of 22 benchmarks;

3. no difference between Andersen/Burke et al.'s analyses and Choi et al.'s analysis in terms of constants found and unexecutable nodes detected.

In summary, constant propagation and unexecutable code detection does not seem to benefit much from increasing precision beyond Steensgaard's analysis.

5.6 Efficiency

The efficiency of an algorithm can vary greatly depending on the implementation [13] and therefore, care must be taken when drawing conclusions regarding efficiency. For example, Fähndrich et al. [9] have demonstrated that the efficiency of a constraint solving implementation of Andersen's algorithm can be improved by orders of magnitude, without a loss of precision, using partial online cycle detection and inductive form.

Table 5 presents the analysis time in seconds of five individual runs for each benchmark. The runs differ only in the pointer analysis used. The times are given for the pointer analysis, the total time for all client analyses, and the sum of these two values. The time reported does not include the time to build the PCG and CFGs, but does include any analysis-specific preprocessing, such as the building of the SEG from the CFG in Choi et al.'s analysis. The last line gives the average for each column expressed as a ratio of the Address-taken analysis for each category: pointer analysis, clients, and total. For example, the average pointer analysis time of Andersen's analysis is 29.60 times that of the average pointer analysis time of the Address-taken analysis, but the average of the client analyses using this information is .84 times the average of the same client analyses using the alias information from the Address-taken analysis. The results show

1. the Address-taken and Steensgaard's analyses are very fast; in all benchmarks these analyses completed in less than a second;

2. the flow-insensitive analyses of Andersen and Burke et al. are significantly slower (approximately 30 times) than the Address-taken and Steensgaard's analyses;

3. the flow-sensitive analysis of Choi et al.'s is on average 80 times slower than the Address-taken analysis and about 2.5 times slower than the Andersen/Burke et al.'s analyses;

4. the client analyses improved in efficiency as the pointer information was made more precise because the input size to these client analysis is smaller. On average this reduction outweighed the initial costs of the pointer analysis for Steensgaard, Andersen, and Burke et al.'s analyses compared to the Address-taken analysis, and brought the total time of the flow-sensitive analysis of Choi et al.'s to within 9% of the total time of the Address-taken analysis.

Table 6 reports the high-water mark of memory usage during the various analyses as reported by the "`ps v`" command under `AIX 4.3`. As before, the amounts are given for the pointer analysis, the total memory for all client analyses, and the sum of these two values. The last line gives the average for each column expressed as a ratio of the Address-taken analysis for each category: pointer analysis, clients, and total. The results show

1. the memory consumption of the Address-taken and Steensgaard's analyses are similar;

2. the memory consumption of the flow-sensitive analysis of Choi et al. can be over 6 times larger than any of the other pointer analysis (`flex`), and on average uses 12 times more memory than the Address-taken analysis;

3. once again, the memory usage of the client analyses improves as the precision of pointer information increases; on average the clients using the information produced by Choi et al.'s analysis used the least amount of memory, which was enough to overcome the twelve-fold increase in pointer analysis memory consumption over the Address-taken analysis.

6. RELATED WORK

Because of space constraints we limit this section to other comparative studies of pointer analyses. A more thorough treatment of related work can be found in [12, 20, 39].

Ruf [29] presents an empirical study of two algorithms: a flow-sensitive algorithm similar to Choi et al. and a context-sensitive version of the same algorithm. The context-sensitive algorithm did not improve precision at pointer dereferences, but Ruf cautioned that this may be a characteristic of the benchmark suite.

Shapiro and Horwitz [32] present an empirical comparison of four flow-insensitive algorithms: Address-taken, Steensgaard, Andersen, and a fourth algorithm [33] that can be parameterized between Steensgaard's and Andersen's analysis. The authors measure the precision of these analyses using procedure-level Mod, live and truly live variables analyses, and an interprocedural slicing algorithm. Their results suggest that a more precise analysis will improve the precision and efficiency of its clients, but leave as an open question whether a flow-sensitive analysis will follow this pattern.

Landi et al. [20, 35] report precision results for the computation of the interprocedural Mod problem using the flow-sensitive context-sensitive analysis of Landi and Ryder [18]. They compare this analysis with an analysis [42] that is similar to Steensgaard's analysis. They found that the more precise analysis provided improved precision, but exhausted memory on some programs that the less precise analysis was able to process.

Emami et al. [8] report precision results for a flow-sensitive context-sensitive algorithm. Ghiya and Hendren [11] empir-

Table 5: Analysis Time in Seconds

Name	Pointer Analysis					Clients					Total				
	AT	ST	An	Bu	Ch	AT	ST	An	Bu	Ch	AT	ST	An	Bu	Ch
allroots	0.01	0.01	0.02	0.03	0.02	0.08	0.08	0.12	0.10	0.09	0.09	0.09	0.14	0.13	0.11
052.alvinn	0.01	0.01	0.02	0.04	0.04	0.22	0.18	0.21	0.20	0.18	0.23	0.19	0.23	0.24	0.22
01.qbsort	0.01	0.01	0.05	0.13	0.30	0.11	0.06	0.09	0.09	0.07	0.12	0.07	0.14	0.22	0.37
06.matx	0.01	0.01	0.06	0.15	0.27	0.20	0.14	0.20	0.16	0.15	0.21	0.15	0.26	0.31	0.42
15.trie	0.01	0.01	0.11	0.19	0.16	0.07	0.07	0.12	0.07	0.08	0.08	0.08	0.23	0.26	0.24
04.bisect	0.01	0.01	0.03	0.05	0.05	0.15	0.10	0.13	0.11	0.10	0.16	0.11	0.16	0.16	0.15
fixoutput	0.01	0.01	0.01	0.06	0.06	0.07	0.06	0.10	0.08	0.07	0.08	0.07	0.11	0.14	0.13
17.bintr	0.01	0.01	0.06	0.05	0.09	0.06	0.07	0.06	0.07	0.05	0.07	0.08	0.12	0.12	0.14
anagram	0.01	0.01	0.09	0.21	0.17	0.34	0.30	0.34	0.30	0.30	0.35	0.31	0.43	0.51	0.47
ks	0.01	0.01	0.08	0.37	0.51	0.34	0.22	0.27	0.21	0.22	0.35	0.23	0.35	0.58	0.73
05.eks	0.01	0.01	0.08	0.17	0.26	0.70	0.65	0.74	0.66	0.68	0.71	0.66	0.82	0.83	0.94
08.main	0.01	0.01	0.66	1.12	1.44	1.14	0.92	1.03	0.95	0.94	1.15	0.93	1.69	2.07	2.38
09.vor	0.01	0.01	3.21	4.05	6.24	1.44	1.09	1.69	1.11	1.04	1.45	1.10	4.90	5.16	7.28
loader	0.01	0.01	0.62	0.59	1.88	2.18	1.84	2.18	1.47	1.46	2.19	1.85	2.80	2.06	3.34
129.compress	0.03	0.04	0.02	0.07	0.08	1.99	1.82	1.97	1.79	1.83	2.02	1.86	1.99	1.86	1.91
ft	0.01	0.01	0.53	1.44	2.27	0.43	0.23	0.40	0.27	0.25	0.44	0.24	0.93	1.71	2.52
football	0.01	0.01	1.34	0.89	1.40	10.38	10.28	11.65	8.50	8.45	10.39	10.29	12.99	9.39	9.85
compiler	0.01	0.01	0.08	0.67	0.64	1.94	2.06	3.09	2.02	2.16	1.95	2.07	3.17	2.69	2.80
assembler	0.01	0.02	5.68	1.77	4.81	6.17	5.21	6.92	3.52	3.26	6.18	5.23	12.60	5.29	8.07
yacr2	0.01	0.01	1.10	2.37	4.92	8.14	6.53	9.61	5.79	5.05	8.15	6.54	10.71	8.16	9.97
simulator	0.02	0.01	1.87	2.19	6.94	16.13	12.80	15.14	8.82	8.42	16.15	12.81	17.01	11.01	15.36
flex	0.02	0.02	4.20	11.59	36.88	44.31	30.84	33.18	29.05	28.26	44.33	30.86	37.38	40.64	65.14
099.go	0.73	0.62	9.38	4.39	9.27	98.96	83.82	74.42	73.18	72.54	99.69	84.44	83.80	77.57	81.81
Ratio to AT	1.00	0.90	29.60	32.92	79.49	1.00	0.81	0.84	0.71	0.69	1.00	0.82	0.98	0.87	1.09

Table 6: Memory Usage in MBs

Name	Pointer Analysis					Clients					Total				
	AT	ST	An	Bu	Ch	AT	ST	An	Bu	Ch	AT	ST	An	Bu	Ch
allroots	0.18	0.25	0.19	0.21	0.18	0.42	0.47	0.47	0.41	0.75	0.60	0.72	0.66	0.62	0.93
052.alvinn	0.13	0.00	1.00	0.22	0.86	0.87	1.73	1.40	0.96	0.50	1.00	1.73	2.40	1.18	1.36
01.qbsort	0.00	0.26	1.36	1.37	0.59	0.59	0.41	0.40	0.28	0.33	0.59	0.67	1.76	1.65	0.92
06.matx	0.00	0.06	2.13	0.50	1.11	1.38	0.82	0.71	0.71	0.74	1.38	0.88	2.84	1.21	1.85
15.trie	0.84	0.29	2.06	0.45	0.97	0.42	0.30	0.24	0.20	0.23	1.26	0.59	2.30	0.65	1.20
04.bisect	1.01	0.61	2.25	0.00	0.50	0.89	0.91	0.55	0.66	0.60	1.90	1.52	2.80	0.66	1.10
fixoutput	0.21	0.43	2.00	0.14	0.28	0.47	0.40	0.40	0.28	0.88	0.68	0.83	2.40	0.42	1.16
17.bintr	0.35	0.00	2.07	0.15	0.62	0.36	0.63	0.28	0.29	0.26	0.71	0.63	2.35	0.44	0.88
anagram	0.50	0.28	1.62	0.04	0.25	1.15	1.10	0.91	0.96	1.37	1.65	1.38	2.53	1.00	1.62
ks	0.26	0.31	2.39	0.42	1.63	2.79	2.38	2.36	2.31	2.86	3.05	2.69	4.75	2.73	4.49
05.eks	0.00	0.10	2.04	0.18	0.75	2.13	1.86	1.69	1.72	1.66	2.13	1.96	3.73	1.90	2.41
08.main	0.19	0.00	3.39	0.76	2.72	2.66	1.89	1.47	1.43	1.42	2.85	1.89	4.86	2.19	4.14
09.vor	0.00	0.01	3.66	1.18	5.00	6.22	4.16	3.80	3.60	3.25	6.22	4.17	7.46	4.78	8.25
loader	0.16	0.01	2.45	0.98	2.33	7.87	5.07	4.46	3.35	3.11	8.03	5.08	6.91	4.33	5.44
129.compress	0.01	0.27	0.00	0.30	0.30	16.68	16.27	16.54	16.25	16.45	16.69	16.54	16.54	16.55	16.75
ft	0.00	0.08	2.36	0.65	2.83	3.37	2.75	2.60	2.51	2.46	3.37	2.83	4.96	3.16	5.29
football	0.01	0.01	6.62	1.86	5.31	32.16	29.93	28.12	25.11	24.79	32.17	29.94	34.74	26.97	30.10
compiler	0.61	1.22	4.20	1.05	2.19	14.60	15.03	14.11	13.80	13.72	15.21	16.25	18.31	14.85	15.91
assembler	0.05	1.06	4.53	1.95	6.36	32.88	24.68	24.25	17.76	17.05	32.93	25.74	28.78	19.71	23.41
yacr2	0.01	0.01	5.19	2.00	8.05	27.27	20.31	19.21	17.72	16.67	27.28	20.32	24.40	19.72	24.72
simulator	0.00	1.46	6.50	2.23	4.95	47.08	31.99	31.16	17.99	17.07	47.08	33.45	37.66	20.22	22.02
flex	1.04	0.00	1.41	3.53	21.30	150.47	118.09	115.58	113.08	110.88	151.51	118.09	116.99	116.61	132.18
099.go	1.62	1.55	1.75	2.44	18.41	114.06	97.96	89.38	88.52	88.53	115.68	99.51	91.13	90.96	106.94
Ratio to AT	1.00	1.15	8.52	3.15	12.19	1.00	0.81	0.77	0.71	0.70	1.00	0.82	0.89	0.74	0.87

ically demonstrate how a version of points-to [8] and connection analyses [10] can improve traditional transformations, array dependence testing, and program understanding.

Wilson and Lam [40, 39] present a context-sensitive algorithm that avoids redundant analyses of functions for similar calling contexts. The algorithm distinguishes structure components and handles pointer arithmetic. Wilson [39] compares various levels of context-sensitivity and describes how dependence analysis uses the computed information to parallelize loops in two SPEC benchmarks.

Diwan et al. [7] examine the effectiveness of three type-based flow-insensitive analyses for a type-safe language (Modula-3). The first two algorithms rely on type declarations. The third considers assignments in a manner similar to Steensgaard's analysis, but retains declared type information. They evaluate the effect of these algorithms on redundant load elimination using statical, dynamic, and upper bound metrics. They conclude that for type-safe languages such as Modula-3 or Java, a fast and simple type-based analysis may be sufficient.

In an earlier paper [13], we describe an empirical comparison of four context-insensitive pointer algorithms: three described in this paper (Choi et al., Burke et al., Address-taken) and a flow-insensitive algorithm that uses precomputed kill information [4, 12]. No alias analysis clients are studied. The paper also quantifies analysis-time speed-up of various implementation techniques for Choi et al.'s analysis.

Yong et al. [41] present a tunable pointer-analysis framework for handling structures in the presence of casting. They provide experimental results from four instances of the framework using a flow- and context-insensitive algorithm, which appears to be similar to Andersen's algorithm. Their results show that for this pointer algorithm distinguishing struct components can improve precision where pointers are dereferenced (the metric used in Section 5.1). They do not address how this affects the precision of client analyses or if similar results hold for other pointer analyses.

Liang and Harrold [21] describe a context-sensitive flow-insensitive algorithm and empirically compare it to three other algorithms: Steensgaard, Andersen, and Landi and Ryder [18], using Ptr-Mod (Section 5.1), summary edges in a system dependence graph, and average slice size as precision metrics. They demonstrate performance and precision mostly between Andersen's and Steensgaard's algorithms. None of the implementations handles function pointers or `setjmp/longjmp`.

7. CONCLUSIONS

This paper describes an empirical study of the precision and efficiency of five pointer analyses and typical clients of the alias information they compute. The major conclusions are

- Steensgaard's analysis is significantly more precise than the Address-taken analysis without an appreciable increase in compilation time or memory usage, and therefore should always be preferred over the Address-taken analysis.
- The flow-insensitive analysis of Andersen and Burke et al. provide the same level of precision. Both analyses offer a modest increase in precision over Steensgaard's analysis. Although this improvement requires additional pointer analysis time, it is typically offset by decreasing the input size (the alias information) and analysis time of subsequent analyses. There is not a clear distinction in analysis time or memory usage between the implementations of these analyses.
- The use of flow-sensitive pointer analysis (as described in this paper) does not seem justified because it offers only a minimum increase in precision over the analyses of Andersen and Burke et al. using a direct metric (such as ptr-mod/ref) and little or no precision improvement in client analyses.
- The time and space efficiency of the client analyses improved as the pointer analysis precision increased because the increase in precision reduced the input to these client analysis.

8. ACKNOWLEDGMENTS

We thank Vivek Sarkar for his support of this work and NPIC group members who have assisted with the implementation. We also thank Todd Austin, Bill Landi, and Laurie Hendren for making their benchmarks available. We thank Frank Tip, Laureen Treacy, and the anonymous referees for comments on an earlier draft of this work. This work was supported in part by the National Science Foundation under grant CCR-9633010, by IBM Research, and by SUNY at New Paltz Research and Creative Project Awards.

9. REFERENCES

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. Available at ftp.diku.dk/pub/diku/semantics/papers/D-203.dvi.Z.

[2] D. C. Atkinson and W. G. Griswold. Effective whole-program analysis in the pressence of pointers. In *ACM SIGSOFT 1998 Symposium on the Foundations of Software Engineering*, pages 131–142, Nov. 1998.

[3] T. Austin. Pointer-intensive benchmark suite, version 1.1. http://www.cs.wisc.edu/~austin/ptr-dist.html, 1995.

[4] M. Burke, P. Carini, J.-D. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Lecture Notes in Computer Science, 892*, pages 234–250. Springer-Verlag, 1995. Proceedings from the *7th Workshop on Languages and Compilers for Parallel Computing.* Extended version published as Research Report RC 19546, IBM T. J. Watson Research Center, September 1994.

[5] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *20th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 232–245, Jan. 1993.

[6] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *18th Annual ACM Symposium on the Principles of Programming Languages*, pages 55–66, Jan. 1991.

[7] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 106–117, June 1998. *SIGPLAN Notices, 33(5).*

[8] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994. *SIGPLAN Notices, 29(6).*

[9] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, 1998.

[10] R. Ghiya and L. J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 24(6):547–578, 1996.

[11] R. Ghiya and L. J. Hendren. Putting pointer analysis to work. In *25th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 121–133, Jan. 1998.

[12] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, July 1999.

[13] M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In G. Levi, editor, *Lecture Notes in Computer Science, 1503*, pages 57–81. Springer-Verlag, 1998. Proceedings from the *5th International Static Analysis Symposium.*

[14] M. Hind and A. Pioli. Traveling through Dakota: Experiences with an object-oriented program analysis system. In *TOOLS USA 2000 - 34th International Conference on Component and Object Technology*, July 2000. Also available as Research Report 21674, IBM T. J. Watson Research Center, February 2000.

[15] M. Karasick. The architecture of Montana: An open and extensible programming environment with an incremental C++ compiler. In *ACM SIGSOFT 1998 Symposium on the Foundations of Software Engineering*, pages 131–142, Nov. 1998.

[16] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, Dec. 1992.

[17] W. Landi. Personal communication, Oct. 1997.

[18] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992. *SIGPLAN Notices* 27(6).

[19] W. Landi, B. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993. *SIGPLAN Notices* 28(6).

[20] W. A. Landi, B. G. Ryder, P. A. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. Technical Report DCS-TR-336, Department of Computer Science, Rutgers University, May 1998.

[21] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In O. Nierstrasz and M. Lemoine, editors, *Lecture Notes in Computer Science, 1687*, pages 199–215. Springer-Verlag, Sept. 1999. Proceedings of the *7th European Software Engineering Confrence* and *ACM SIGSOFT Foundations of Software Engineering.*

[22] T. Marlowe, W. Landi, B. Ryder, J.-D. Choi, M. Burke, and P. Carini. Pointer-induced aliasing: A clarification. *SIGPLAN Notices*, 28(9):67–70, Sept. 1993.

[23] T. J. Marlowe, B. G. Ryder, and M. G. Burke. Defining flow sensitivity in data flow problems. Technical Report RC 20138, IBM T. J. Watson Research Center, July 1995.

[24] J. D. Morgenthaler. *Static Analysis for a Software Transformation Tool.* PhD thesis, UCSD, Aug. 1997.

[25] S. S. Muchnick. *Advanced Compiler Design and Imlementation.* Morgan Kaufmann, 1997.

[26] A. Pioli. Conditional pointer aliasing and constant propagation. Master's thesis, SUNY at New Paltz, 1999. Available at http://www.mcs.newpaltz.edu/tr as Technical Report # 99-102.

[27] A. Pioli and M. Hind. Combining interprocedural pointer analysis and conditional constant propagation. Research Report 21532, IBM T. J. Watson Research Center, Mar. 1999. Also available as SUNY at New Paltz Technical Report #99-103.

[28] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, Sept. 1994.

[29] E. Ruf. Context-insensitive alias analysis reconsidered. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, June 1995. *SIGPLAN Notices, 30(6).*

[30] E. Ruf. Personal communication, Oct. 1997.

[31] Rutgers PROLANGS. http://www.prolangs.rutgers.edu/public.html, 1999.

[32] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In P. V. Hentenryck, editor, *Lecture Notes in Computer Science, 1302*, pages 16–34. Springer-Verlag, 1997. Proceedings from the *4th International Static Analysis Symposium.*

[33] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive point-to analysis. In *24th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 1–14, Jan. 1997.

[34] B. Steensgaard. Points-to analysis in almost linear time. In *23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 32–41, Jan. 1996.

[35] P. A. Stocks, B. G. Ryder, W. A. Landi, and S. Zhang. Comparing flow and context sensitivity on the modifications-side-effects problem. In *International Symposium on Software Testing and Analysis*, pages 21–31, Mar. 1998.

[36] R. Tarjan. Data structures and network flow algorithms. *Regional Conference Series in Applied Mathematics*, CMBS 44, 1983. SIAM.

[37] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[38] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, 1991.

[39] R. P. Wilson. *Efficient Context-Sensitive Pointer Analysis for C Programs.* PhD thesis, Stanford University, Dec. 1997.

[40] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995. *SIGPLAN Notices, 30(6).*

[41] S. H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 91–103, 1999.

[42] S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *4th Symposium on the Foundations of Software Engineering*, pages 81–92, Oct. 1996.