Context-Insensitive Alias Analysis Reconsidered

Erik Ruf
Microsoft Research
One Microsoft Way, Redmond, WA 98052 USA
erikruf@microsoft.com

Abstract

Recent work on alias analysis in the presence of pointers has concentrated on *context-sensitive* interprocedural analyses, which treat multiple calls to a single procedure independently rather than constructing a single approximation to a procedure's effect on all of its callers. While context-sensitive modeling offers the potential for greater precision by considering only realizable call-return paths, its empirical benefits have yet to be measured.

This paper compares the precision of a simple, efficient, *context-insensitive* points-to analysis for the C programming language with that of a maximally context-sensitive version of the same analysis. We demonstrate that, for a number of pointer-intensive benchmark programs, context-insensitivity exerts little to no precision penalty. We also describe techniques for using the output of context-insensitive analysis to improve the efficiency of context-sensitive analysis without affecting precision.

1 Introduction

Modern compilers and programming environments are becoming increasingly dependent on semantic information extracted via dataflow analysis. In programs containing pointers, many dataflow analyses depend crucially on the ability to approximate the targets of indirect memory operations, so that all potential uses or modifications of a value can be taken into account. Alias analysis[1] provides this approximation; its precision directly effects the quality of virtually all other dataflow analyses.

Early pointer alias analyses were completely flow-insensitive; both Weihl [Wei80] and Coutant [Cou86] computed alias information on a program-wide basis, building a single, global mapping between pointers and their potential referents. Others later found that these methods generated overly large, imprecise approximations, handicapping subsequent analyses [Ryd89, LR92].

A number of algorithms compute program-point-specific alias information by treating calls and returns as n-way branches; that is, combining the control flow graphs of all procedures into a single super-graph and using intraprocedural flow analysis techniques on the enlarged graph. Such context-insensitive[2] techniques can introduce imprecision by exploring call/return paths that cannot occur in any execution of the program. The system of Chow and Rudmik [CR82] operates in this manner; Chase et al [CWZ90] and Deutsch [Deu94] perform more sophisticated modeling intraprocedurally, but are still context-insensitive at the interprocedural level.

Recent research in interprocedural alias analysis has focused on avoiding the spurious alias relationships generated by context-insensitive strategies. One approach explicitly re-analyzes each procedure under multiple contexts: Emami et al [EGH94] create a context for each each acyclic path from the root of the call graph to the current invocation, while Wilson and Lam [WL95] build one context per set of "relevant" aliases holding on entry to the procedure. Another approach tags each alias relationship with information that allows a procedure to propagate only appropriate relationships to each of its callers: Landi [LR92, Lan92] uses information about the alias relationships on entry to the enclosing procedure, while Cooper [Coo89] and Choi et al [CBC93] augment this with an abstraction of the call stack. All context-sensitive approaches have exponential time complexity in the size of the input program unless some effort is made to limit the number of contexts in which a procedure is analyzed.

Context-sensitive alias analysis techniques have been quite successful, in that they run in reasonable amounts of time, and find relatively few aliases (*e.g.*, the average indirect memory operation is found to reference/modify approximately 1.2 memory locations [LRZ93, EGH94]). However, published work in this area includes comparisons only to Weihl's program-wide analysis, so it is unclear how much of this precision derives from the program-point-specific nature of the analysis, and how much is truly due to its context-sensitivity.

This paper presents the results of an experiment to measure the benefits of context-sensitivity. We implemented

[1] In this paper, we use the term "alias analysis" to denote a dataflow analysis for estimating the effects of indirect memory references through pointers. Weaker forms of alias analysis, such as the treatment of call-by-reference parameters and Fortran `COMMON` blocks, are well understood and will not be discussed here

[2] We use the terminology of [EGH94], in which the terms *context-sensitive* and *context-insensitive* are used to distinguish analyses that propagate dataflow facts from procedure returns solely to appropriate call sites from those that propagate to all call sites Other terms with similar meanings include the *imprecise/precise* distinction of [LR92] and the *monovariant/polyvariant* distinction made in partial evaluation and abstract interpretation.

both context-insensitive and context-sensitive versions of a single alias analysis and evaluated the precision of the results. We found that

- In our empirical tests, the context-sensitive analysis does compute more precise alias relationships at some program points. However, when we restrict our attention to the locations accessed or modified by indirect memory references, no additional precision is measured.
- Information computed by the context-insensitive analysis can be very useful in improving the efficiency of the context-sensitive analysis.

Section 2 of this paper introduces a "points-to" formulation of the pointer alias problem; Section 3 describes a simple, efficient, context-insensitive analysis for finding points-to relations. We then (Section 4) extend the analysis to be fully context-sensitive, and compare the results with those from the context-insensitive analysis. Section 5 discusses our results and their relevance to other alias analysis frameworks. We conclude with a brief summary and discussion of future work.

2 Framework

We analyze C programs represented as value dependence graphs, or VDGs [WCES94a, WCES94b]. For purposes of this paper, VDG form can be thought of as an extended version of static single assignment (SSA) form [CFR+91], in which all program dependences (not merely alias-free def/use dependences) are modeled explicitly. Computation is expressed by nodes that consume input values (outputs of other nodes) and produce output values. For example, memory accesses (both direct and indirect) are uniformly represented as `lookup` and `update` operations that consume (and, in the case of `update`, produce) explicit store values. In general, inputs and outputs may be of scalar, pointer, function, aggregate, or store type; the standard control-flow graph representation of programs can be viewed as a degenerate VDG in which all inputs and outputs are of store type. Our analyses apply equally well to control-flow graph representations; they merely run faster on the VDG because it is more sparse [Ruf95].

We use a "points-to" model of aliasing; that is, at each program point, our analysis models the contents of storage locations (*points-to* relations) rather than modeling equivalence classes of location-valued expressions (*alias* relations). In this respect, our analysis is similar to [RM88, HPR89, CWZ90, EGH94, WL95] and different from [Coo89, LR92, CBC93, Deu94]. We chose the points-to model because it fits naturally into our intermediate representation, and because its storage requirements are likely to be smaller.

In our framework, a finite number of *base-locations* name allocation sites: there is one base-location for each variable, and for each static invocation site of memory-allocating library code such as `malloc`.[3] An *access path* consists of an optional base-location, followed by a possibly empty sequence of access operators, each of which denotes either a structure/union member access or an array access.

[3]Increasing the number of base-locations per `malloc`, *e.g.*, by naming such base-locations with a call string instead of a single allocation site, would be a trivial modification.

Access paths starting with base-locations are referred to as *locations*, and denote indirection through the store, while those with empty base-locations, called *offsets*, denote relative addressing into aggregate values. Careful interning of access operators ensures that an access path is aliased only to its prefixes, allowing us to easily model static aliasing due to C's union types Access paths may model single runtime locations (*e.g.*, global variables and local variables of non-recursive procedures), while others model multiple runtime locations (array contents, heap-allocated data, and locals of recursive procedures[4]). Paths corresponding to at most one location at runtime can be *strongly updated* [CWZ90] during analysis, while others cannot.

The output of our points-to analysis is a mapping from outputs of VDG nodes to sets of *points-to-pair* objects, each of which is a pair $\langle a, b \rangle$ of access paths denoting "in the value produced by this output, indirecting through any location (or offset) denoted by path a may return any location denoted by path b." The first element of a points-to pair is the *path*; the second is the *referent*. Our points-to pairs denote *possible*, rather than *definite* relationships; however, we follow [CWZ90] in treating any singleton set of "possibly points to" pairs as a "definitely points to" pair. This allows us to exploit strong updates without additional representational overhead.

We also require the usual caveats of pointer alias analysis, namely, we allow pointer arithmetic only on array interior pointers, and assume that such arithmetic does not cause the pointer to denote storage outside the array. We do not perform array dependence analysis, and thus maintain one approximation to all values stored in the array. Neither signal handlers, `longjmp`, nor casts between pointer and non-pointer types are modeled.

3 Context-Insensitive Analysis

3.1 Algorithm

Our context-insensitive analysis, described in Figure 1, is essentially the "simple algorithm" of [CWZ90, Sections 3 and 4.2], and has the same effect as the intraprocedural portion of [EGH94]. We maintain a set of points-to pairs on every node output in the program,[5] and incrementally grow these sets using a worklist strategy. Whenever a points-to pair is added to a set, all consumers of that output are notified; they in turn make appropriate modifications to the points-to sets on their outputs. Calls and returns are handled like jumps, in that all information at a call's actuals is propagated to all called procedures, and all information at a pro-

[4]A local variable of a recursive procedure may require special handling if its address is taken, because pointers to multiple instances of such variables may be simultaneously live. In the case where the variable is of pointer type, this behavior could lead to an incorrect points-to solution if the variable's location is strongly updated by the analysis. We have experimented with two solutions to this problem Our first scheme is essentially Cooper's [Coo89] model· each such variable variable is assigned two base-locations, one denoting the most recent instance of the variable, the other denoting all other instances on the stack The former can be strongly updated, while the latter cannot. Our second scheme approximates this by assigning only a weakly-updateable base-location to each such variable.

The choice of solution is irrelevant for the benchmarks in this paper, as they do not contain recursive procedures that pass addresses of local pointer-valued variables downward

[5]We could reduce our storage costs (at $\log n$ time cost) via the sparse storage strategy described in [CWZ90], but we have yet to see a need for this in practice

```
analyze program
  worklist := φ
  for each base-location b ∈ program
    flow-out output(b) ⟨ε, b⟩
  while worklist not empty do
    take δ = ⟨input, pair⟩ from worklist
    flow-in input-node(input) δ

flow-out output pair
  if pair ∉ P(output) then
    P(output) := P(output) ∪ {pair}
    for each c ∈ consumers(output)
      add ⟨c, pair⟩ to worklist

flow-in ⟨lookup, loc, store, ovalue⟩ δ
  case δ of
    ⟨loc, ⟨p_l, r_l⟩⟩
      for each ⟨p_s, r_s⟩ ∈ P(store) | r_l dom p_s
        flow-out ovalue ⟨p_s − r_l, r_s⟩
    ⟨store, ⟨p_s, r_s⟩⟩
      for each ⟨p_l, r_l⟩ ∈ P(loc) | r_l dom p_s
        flow-out ovalue ⟨p_s − r_l, r_s⟩

flow-in ⟨update, loc, store, value, ostore⟩ δ
  case δ of
    ⟨loc, ⟨p_l, r_l⟩⟩
      for each ⟨p_v, r_v⟩ ∈ P(value)
        flow-out ostore ⟨r_l + r_v, r_v⟩
      for each ⟨p_s, r_s⟩ ∈ P(store) | ¬(r_l strong-dom p_s)
        flow-out ostore ⟨p_s, r_s⟩
    ⟨store, ⟨p_s, r_s⟩⟩
      for each ⟨p_l, r_l⟩ ∈ P(loc) | ¬(r_l strong-dom p_s)
        flow-out ostore ⟨p_s, r_s⟩
    ⟨value, ⟨p_v, r_v⟩⟩
      for each ⟨p_l, r_l⟩ ∈ P(loc)
        flow-out ostore ⟨r_l + r_v, r_v⟩

flow-in n = ⟨call, fcn, actual_0, ..., result_0, ...⟩ δ
  case δ of
    ⟨fcn, ⟨p_f, r_f⟩⟩
      (omitted)
    ⟨actual_i, ⟨p_a, r_a⟩⟩
      for each c ∈ callees n
        let o = corresponding-formal c actual_i in
          flow-out o ⟨p_a, r_a⟩

flow-in n = ⟨return, value⟩ δ
  case δ of
    ⟨value, ⟨p_v, r_v⟩⟩
      for each c ∈ callers n
        let o = corresponding-result c value in
          flow-out o ⟨p_v, r_v⟩

flow-in ⟨if, pred, then, else, ovalue⟩ δ
  (omitted)

flow-in ⟨primop, name, actual_0, ..., result_0, ...⟩ δ
  (omitted)
```

Definitions of globals/primitives:

- \+ Append function on access paths.
- − Prefix subtraction function on access paths.
- `callees` Function mapping a `call` node to the corresponding function nodes.
- `callers` Function mapping a `return` node to the corresponding `call` nodes.
- `consumers` Function mapping a node output to a set of node inputs.
- `corresponding-formal` Function mapping a function node and a an actual input to the corresponding formal output
- `corresponding-result` Function mapping a `call` node and a return input to the corresponding result output.
- `dom` Binary relation on access paths used to model static may-aliasing in aggregates. A `dom` B if a read (write) of A may observe (modify) a value written to B. In our path representation, this is true if A is a prefix of B.
- `input-node` Function mapping a node input to its node.
- $\mathcal{P}$ Function mapping a node output to a set of points-to pairs.
- `strong-dom` Binary relation on access paths used to model static must-aliasing in aggregates. A `strong-dom` B if a read (write) of A must observe (modify) a value written to B. In our path representation, this is true if A is both strongly updateable (e.g. its *base-location* denotes a single storage location and none of its access operators are array dereferences) and a prefix of B
- *worklist* Queue of ⟨*input*, *points-to-pair*⟩ pairs.

Explanations of `flow-in` methods:

- **lookup** A new location is dereferenced in the store. A new store pair is used to dereference all of the locations.
- **update** A new location generates a store pair for each value, and also propagates all store pairs not strongly updated by this location. A new store pair is propagated if at least one location doesn't strongly update it. A new value generates a store pair for each location pair. (The propagation behavior for non-strongly-updated store pairs in the actual implementation is more efficient than in this pseudocode)
- **call** A new function updates the call graph and performs appropriate repropagation (code omitted here). A new actual value is propagated to the corresponding formal in each callee.
- **return** A new return value is propagated to the corresponding result output at each call site
- **if** Values from both branches propagate to the output; predicate is ignored.
- **primop** Behavior varies by operator.

Figure 1: Context-insensitive analysis.

name	source lines	VDG nodes	alias-related outputs
allroots*	231	554	278
anagram†	648	1018	560
assembler*	2764	4741	2990
backprop†	286	721	421
bc‡	6771	9024	5435
compiler*	2282	3852	2057
compress+	1502	2080	1124
lex315*	1039	1453	716
loader*	1241	2033	1202
part†	684	1677	1105
simulator*	4009	7052	4047
span†	1297	1364	944
yacr2†	3208	5963	3047

Figure 2. Benchmark programs and their sizes in source and VDG form. An "alias-related output" is a node output that can carry pointer or function values; *e.g.*, one whose type is pointer, function, aggregate containing pointer or function, or store Sources (*) William A. Landi, (†) Todd K. Austin, (‡) Free Software Foundation, (+) SPEC92 suite.

cedure's returns is propagated to all of its callers Termination is assured because the number of outputs and points-to pairs are finite, yielding $O(n^3)$ time and space bounds in the worst case ($O(n^2)$ in the average case, in which each pointer has only a small constant number of referents). This algorithm has the desirable property that its convergence time is independent of the scheduling strategy used for the worklist.

Because points-to analysis is not a distributive problem, the effects of points-to pairs arriving on multiple inputs of a node cannot always be computed independently For example, a new points-to pair arriving on the location input of a `lookup` node causes the node to iterate over the points-to pairs on its store input and emit the referents of all pairs whose path might be aliased to the referent of the newly arrived pair. `Update` nodes are even more interesting due to our desire to perform strong updates whenever possible. Since strong updates block points-to pairs on the store input from propagating through the node, we must (1) delay processing of any points-to pairs on the store input until at least one pair has arrived on the location input, and (2) reprocess all strongly-updated (*i e.*, blocked) pairs on the store input once a second pair arrives on the location input. This gives the same effect as the dual-worklist strategy of [CWZ90]. Similar reasoning applies to indirect calls (and the returns of indirectly called procedures), which must propagate old information to new destinations every time a new points-to pair arrives on the function input of a call.

3.2 Results

We tested our context-insensitive algorithm on a variety of small programs described in Figure 2. We chose these programs from those analyzed in other alias analysis publications [LR92, LRZ93, EGH94] and in [ABS94], which instrumented pointer-intensive programs. Under our Scheme-based implementation, analysis times for the benchmark programs range from 1 to 35 seconds.

Figure 3 reports the number of points-to pairs discovered. These aggregate figures are relatively uninformative by themselves; we can learn more by considering an application, such as def/use or mod/ref analysis. Such applications are concerned only with the memory locations referenced by each memory read or write, *e.g.*, the pointers arriving at the location inputs of `lookup` and `update` nodes. Figure 4 reports these statistics. We see that, on average, most indirect memory operations reference very few locations. Our statistics for memory writes are higher than those in [LRZ93] for two reasons. First, we do not construct synthetic locations to represent lexically non-visible variables, meaning that we lose some opportunities for strong updates (*c.f.* Section 5.1). Second, the VDG intermediate representation often coalesces series of structure or array operations into a single memory read followed by a series of aggregate update operations and a single memory write; thus, many array/structure operations are not counted as memory operations in our statistics (*e g.*, in our representation, *assembler* contains 115 indirect write operations, while in that of [LRZ93], it contains 290). Since the majority of array/structure operations reference only one location, this increases the our "average number of locations referenced/modified" statistic.

Three programs, *backprop*, *compiler*, and *span*, have no indirect loads/stores that reference more than one location, indicating that (under the assumption that all memory operations reference only valid pointers, and are executed at least once at runtime) performing a context-sensitive analysis would not add any precision to a def/use or mod/ref application in these cases. We also note that the "maximum locations modified" values for *allroots*, *assembler*, *compiler*, *lex315*, *loader*, and *simulator* are identical to those produced by context-sensitive means in [LRZ93], allowing us to conclude that the worst-case behavior at indirect memory references in these programs was not caused by context-insensitivity. What cannot be concluded from this data is how much improvement we would see under context-sensitive methods; that is the subject of the next section.

4 Context-Sensitive Analysis

4.1 Modifying the Context-Insensitive Algorithm

Our goal in constructing a context-sensitive analysis is not to produce a reasonable compromise between efficiency and precision, but rather to establish an empirical upper bound on the precision of alias analysis in our points-to framework. Thus we are willing to pay an exponential performance penalty and will not, as in [LR92, CBC93] limit our representation to avoid such a penalty. We choose to use assumption-set-based contexts (rather than call-stack-based contexts) because doing so allows us to prune contexts based on the output of the context-insensitive analysis.

We make the analysis of Section 3.1 context-sensitive by altering it to propagate *qualified points-to pairs* rather than ordinary points-to pairs. A qualified pair consists of an ordinary points-to pair (*e.g.*, a path and its referent), along with a set of assumptions, each of which consists of a points-to pair and a formal parameter output on which that pair must hold. The interpretation of the qualified pair $\langle\langle a, c\rangle, \{\langle s, \langle a, b\rangle\rangle, \langle s, \langle b, c\rangle\rangle\}\rangle$ on some output is "a points to c on this output if, on entry to this procedure, a points to b in formal s and b points to c in formal s."[6] These assumption sets are similar to those of [Coo89, LR92, CBC93], except that (1) our assumptions concern points-to, rather than

[6]Assumptions need not be restricted to store outputs; we might just as easily assume that a pointer-valued formal parameter has a particular value, *e.g.*, $\langle\langle \epsilon, a\rangle, \{\langle f, a\rangle\}\rangle$ means "this output has pointer value a if formal f has value a."

name	pointer	function	aggregate	store	total
allroots	123	0	4	254	381
anagram	206	3	13	1394	1616
assembler	1509	0	1798	165622	168929
backprop	142	0	4	497	643
bc	3017	10	1193	333389	337609
compiler	484	0	189	20566	21239
compress	339	2	114	2459	2914
lex315	264	0	33	10269	10566
loader	491	0	77	5753	6321
part	521	0	311	6597	7429
simulator	1921	0	634	176828	179383
span	322	0	484	3244	4050
yacr2	1174	0	141	38949	40264
TOTAL	10513	15	4995	765821	781344

Figure 3: Total points-to relationships, as computed by context-insensitive analysis. Each column indicates the number of points-to pairs that appear on node outputs of the indicated type.

name	indirect references							
	type	total	accessing n locations				max	avg
			1	2	3	≥ 4		
allroots	read	34	16	18	0	0	2	1.53
allroots	write	3	3	0	0	0	1	1.00
anagram	read	56	53	3	0	0	2	1.05
anagram	write	25	25	0	0	0	1	1.00
assembler	read	176	135	17	0	24	60	2.34
assembler	write	115	80	13	0	22	9	1.93
backprop*	read	32	31	0	0	0	1	0 97
backprop	write	21	21	0	0	0	1	1.00
bc*	read	553	462	50	21	19	33	2.16
bc	write	250	216	18	8	8	26	1 50
compiler	read	83	83	0	0	0	1	1 00
compiler	write	50	50	0	0	0	1	1 00
compress	read	77	76	1	0	0	2	1 01
compress	write	84	84	0	0	0	1	1 00
lex315	read	16	7	9	0	0	2	1.56
lex315	write	9	4	5	0	0	2	1 56
loader	read	80	77	2	0	1	7	1.10
loader	write	43	36	1	1	5	9	1 91
part	read	114	56	58	0	0	2	1 51
part	write	49	35	14	0	0	2	1.28
simulator	read	339	323	0	8	8	22	1.22
simulator	write	210	183	5	12	10	13	1 45
span	read	101	101	0	0	0	1	1.00
span	write	45	45	0	0	0	1	1.00
yacr2	read	268	261	7	0	0	2	1 03
yacr2	write	109	98	10	1	0	3	1.11
TOTAL	read	1929	1681	165	29	52	60	1.55
TOTAL	write	1013	880	66	22	45	26	1.39

Figure 4: Points-to statistics for indirect memory reads and writes **Backprop* and *bc* each contain one indirect read that, if executed, would reference only the null pointer value

```
flow-in ⟨lookup, loc, store, ovalue⟩ δ
  case δ of
    ⟨loc, ⟨⟨p_l, r_l⟩, a_l⟩⟩
      for each ⟨⟨p_s, r_s⟩, a_s⟩ ∈ P(store) | r_l dom p_s
        flow-out ovalue ⟨⟨p_s − r_l, r_s⟩, a_l ∪ a_s⟩
    ⟨store, ⟨⟨p_s, r_s⟩, a_s⟩⟩
      for each ⟨⟨p_l, r_l⟩, a_l⟩ ∈ P(loc) | r_l dom p_s
        flow-out ovalue ⟨⟨p_s − r_l, r_s⟩, a_l ∪ a_s⟩

flow-in ⟨update, loc, store, value, ostore⟩ δ
  case δ of
    ⟨loc, ⟨⟨p_l, r_l⟩, a_l⟩⟩
      for each ⟨⟨p_v, r_v⟩, a_v⟩ ∈ P(value)
        flow-out ostore ⟨⟨r_l + r_v, r_v⟩, a_l ∪ a_v⟩
      for each ⟨⟨p_s, r_s⟩, a_s⟩ ∈ P(store) | ¬(r_l strong-dom p_s)
        flow-out ostore ⟨⟨p_s, r_s⟩, a_l ∪ a_s⟩
    ⟨store, ⟨⟨p_s, r_s⟩, a_s⟩⟩
      for each ⟨⟨p_l, r_l⟩, a_l⟩ ∈ P(loc) | ¬(r_l strong-dom p_s)
        flow-out ostore ⟨⟨p_s, r_s⟩, a_l ∪ a_s⟩
    ⟨value, ⟨⟨p_v, r_v⟩, a_v⟩⟩
      for each ⟨⟨p_l, r_l⟩, a_l⟩ ∈ P(loc)
        flow-out ostore ⟨⟨r_l + r_v, r_v⟩, a_l ∪ a_v⟩

flow-in n = ⟨call, fcn, actual_0, ..., result_0, ...⟩ δ
  case δ of
    ⟨fcn, ⟨⟨p_f, r_f⟩, a_f⟩⟩
      (omitted)
    ⟨actual_i, ⟨⟨p_a, r_a⟩, a_a⟩⟩
      for each c ∈ callees n
        let o = corresponding-formal c actual_i in
          flow-out o ⟨⟨p_a, r_a⟩, {⟨o, ⟨p_a, r_a⟩⟩}⟩
        for each r ∈ returns c
          for each p ∈ P(r)
            propagate-return n r p

flow-in n = ⟨return, value⟩ δ
  case δ of
    ⟨value, ⟨⟨p_v, r_v⟩, a_v⟩⟩
      for each c ∈ callers n
        propagate-return c n ⟨⟨p_v, r_v⟩, a_v⟩

propagate-return caller return ⟨⟨p_v, r_v⟩, a_v⟩
  let o = corresponding-result caller return
      S = {s | s = {a_a | ⟨⟨p_f, r_f⟩, a_a⟩ ∈ P(actual)}
            where actual = corresponding-actual caller f
              where ⟨f, ⟨p_f, r_f⟩⟩ ∈ a_v}
  for each a ∈ ∏ S
    flow-out o ⟨⟨p_v, r_v⟩, a⟩
```

Figure 5: Transfer functions of Figure 1, modified for context-sensitive analysis. This pseudocode does not include the optimizations described in Section 4.2. The code in propagate-return operates as follows. for each element of the assumption set a_v of the *qualified-pair* being returned, we compute a set s of assumption sets which, if holding at the call site, would be sufficient to satisfy that element of a_v. The Cartesian product of these sets gives us all possible caller assumption sets sufficient to satisfy the assumptions on the callee's return value. For each element of this product, we propagate an appropriately qualified version of the return value to the caller

alias, relations, and (2) we do not limit the size of assumption sets.

Our rules for propagation of assumptions (shown in Figure 5) are similar to those in [Lan92]. Assumptions are introduced and removed at procedure calls and returns. When a new qualified-pair p arrives at a call, it is propagated to the corresponding formal f of each potential callee as before; however, the propagated pair is given the assumption set $\{\langle f, p\rangle\}$, indicating that p will hold within the called procedure only if it held on entry. When a qualified-pair reaches a `return` node, its assumptions are checked against the pairs holding at each call site, and it is propagated only to those call sites satisfying all of its assumptions; the propagated value is given new assumptions corresponding to the assumption sets of the actual parameter qualified-pairs used to satisfy the assumptions.

Additional assumptions are introduced at `lookup` and `update` nodes because a points-to pair on output may hold only if multiple points-to pairs hold on input For example, if $\langle\langle\epsilon, a\rangle, \{\langle f_1, \langle\epsilon, a\rangle\rangle\}\rangle$ holds on the location input of a `lookup`, and $\langle\langle a, b\rangle, \{\langle f_2, \langle c, b\rangle\rangle\}\rangle$ holds on the store input, then $\langle\langle\epsilon, b\rangle, \{\langle f_1, \langle\epsilon, a\rangle\rangle, \langle f_2, \langle c, b\rangle\rangle\}\rangle$ holds on the output. In other words, the memory read returns b only when the pointer-valued formal parameter f_1 is a, and when the store-valued formal parameter f_2 maps path c to value b. Similar chaining occurs when processing new qualified-pairs arriving on the inputs of `update` nodes. In the worst case, this behavior can yield assumption sets whose size is exponential in the number of indirect storage operations in the program.

Strong updates also add assumptions. An `update` node propagates a points-to pair on its store input only when its location input contains at least one path that will not definitely overwrite the pair's path. In the context-sensitive analysis, such a pair must be propagated under a different assumption for each non-overwriting location input; in essence, we must enumerate all of the ways in which the input pair could fail to be overwritten.[7] A chain of such `update` nodes quickly yields a large combinatorial explosion.

Finally, we must also introduce assumptions whenever we make use of particular function values in an indirect call. We have not yet implemented this feature in our analysis; hence, our function pointer results are context-insensitive. This is not an issue for our benchmark programs, which make only light use of indirect function calls (we have hand-verified that this context-insensitivity does not affect any of our empirical results).

After the context-sensitive analysis has completed, we compute the set of ordinary points-to pairs on each node output by stripping the assumption sets from the qualified points-to pairs on that output and removing duplicates. Some context-sensitive analyses [PLR92, LRZ93] prefer to use the qualified information directly; this would be easy to accommodate.

4.2 Implementation

The analysis described in the previous subsection is too inefficient to run on any but the smallest of examples. We use several techniques to improve its efficiency.

One important optimization is a subsumption rule on assumption sets. Any qualified points-to pair $\langle p, B\rangle$ reach-

[7] A language of assumptions having a negation operator would permit this to be expressed more concisely; however, this would make composition and comparison of assumptions far more difficult.

ing an output where $\langle p, A \rangle$ already holds may be discarded whenever $A \subset B$. In other words, if p already holds under A, there is no need to store (or process) the fact that p holds under some stronger assumption B.

Another class of optimizations uses the results of our efficient context-insensitive analysis to prune the number and size of assumption sets that are generated:

- Assumptions about location values need not be introduced at indirect memory operations that the context-insensitive analysis has proven to reference/modify only a single location. That is, if we adopt the standard assumptions that all intraprocedural paths are executed, and that all memory reads and writes dereference only non-null pointers, then it must be the case that such a node will reference the same location under all calling contexts, removing the need for tracking assumptions about the location.[8]

  As can be seen from Figure 4, this optimization applies to 87% of the indirect reads and writes in our test programs. Once we consider that only indirect reads and writes of pointer and function values affect the analysis results, only 9% of the indirect reads and 7% of the indirect writes need to introduce assumptions.

- We can also limit the growth of assumption sets due to strong updates. The context-insensitive analysis provides an upper bound on the set of locations modified by each `update` node; any location not in the context-insensitive estimate for a node cannot possibly be modified by that node. Thus, all qualified points-to pairs on the store input that can be shown to be unmodified may be passed on without the need to add a new assumption about the `update`'s location argument. Since the vast majority of `update` nodes modify only one or two locations (average is 1.39 locations modified), almost all points-to pairs can be propagated though `update` nodes without the need for additional assumptions.

We were unable to measure the speedup due to these optimizations because the unoptimized algorithm could only be applied to very small examples. These optimizations do not alter the exponential character of the analysis, but they have served to make its execution feasible. With the optimizations in place, the context-sensitive algorithm executes only slightly more (10%) transfer functions (applications of `flow-in` in the pseudocode) than the context-insensitive algorithm, but as many as 100 times more "meet" operations (applications of `flow-out`). The net result is that the context-sensitive algorithm is 2-3 orders of magnitude slower than the context-insensitive algorithm on our larger test programs (*assembler* and *bc* each take several hours to analyze). This performance is acceptable for our purpose (establishing an upper bound on precision), but it does limit the algorithm's practicality.

Optimization techniques similar to our use of context-insensitive information might also be useful in systems such as [LR92] that maintain bounded-size assumption sets. Such systems presently must arbitrarily choose which assumptions to discard when the bound is reached; in many cases, information from context-insensitive analysis could be used to judiciously discard those assumptions that cannot possibly affect precision.

[8] If we didn't make this assumption, we might be able to rule out some *intraprocedural* paths under some contexts, and reduce the number of locations referenced to zero. In such cases, it would be imprecise to make use of the single-location result from context-insensitive analysis, as that result might be overly conservative.

4.3 Results

Figure 6 summarizes the number of points-to pairs generated by our context-sensitive analysis for the benchmark programs of Figure 2. In most cases, the context-sensitive analysis generates fewer (2%, on average) points-to pairs than the context-insensitive analysis.

This information, however, gives no indication of how much conservative behavior in subsequent analyses would be induced by the spurious points-to information. When we consider only the points-to pairs reaching the location inputs of indirect memory references, we find that *the spurious information does not affect the solution at all*; the results for indirect memory references are *identical* to the context-insensitive results of Figure 4. Thus, for use/def or mod/ref applications, our fully context-sensitive analysis provides no precision benefit on our test programs.

We initially found this result rather surprising, but, after further reflection, it appears reasonable given our framework and test programs. In the next section, we describe several factors contributing to this result, and explain how different problem formulations, implementations, or benchmark suites may affect the precision benefits of context-sensitivity.

5 Discussion

Our results raise two questions:

1. Why does context-insensitive analysis generate so few spurious points-to pairs?
2. Why don't the spurious points-to pairs affect the paths reaching location inputs of indirect memory operations?

Since our result is empirical, we cannot provide complete explanations that are valid for all test programs; indeed, it is easy to construct programs where context-sensitivity provides an arbitrarily large benefit. The remainder of this section gives our working hypotheses.

5.1 Lack of spurious points-to pairs

The lack of spurious points-to pairs can be attributed both the the design of our analysis framework and to characteristics of the benchmark programs used to test it.

5.1.1 Design Choices

Some of the apparent precision of the context-insensitive analysis is due to choices we made in our analysis framework and its implementation:

- Treatment of "invisible" locations. Many alias analysis algorithms [LR92, EGH94, CBC93] explicitly construct synthetic locations (called "invisible variables" or "representative aliases") to represent storage addresses not lexically visible to a procedure (*e g* , a caller's local variable, whose address is passed to the procedure as a parameter or in a global variable), and perform mapping operations on procedure call and return. Our algorithms do not map invisible locations

name	pointer	function	aggregate	store	total	total (insensitive)	percent spurious
allroots	123	0	4	254	381	381	0 0
anagram	206	3	13	1204	1426	1616	11.8
assembler	1509	0	1798	162972	166279	168929	1.6
backprop	142	0	4	497	643	643	0.0
bc	3017	10	1193	325749	329969	337609	2 3
compiler	484	0	189	20484	21157	21239	0.4
compress	333	2	114	2392	2841	2914	2.5
lex315	264	0	33	10269	10566	10566	0 0
loader	491	0	77	5445	6013	6321	4 9
part	521	0	311	6540	7372	7429	0.8
simulator	1921	0	634	175268	177823	179383	0 9
span	320	0	473	3092	3885	4050	4.1
yacr2	1174	0	141	36204	37519	40264	6 8
TOTAL	10505	15	4984	750370	765874	781344	2.0

Figure 6: Points-to relationships, as computed by context-sensitive analysis Each of the first five columns indicates the number of points-to pairs that appear on node outputs of the indicated type The "total insensitive" columns indicates the total number of points-to pairs obtained by the context-insensitive method; the final column shows the percentage of context-insensitive pairs found to be spurious by the context-sensitive method.

to such synthetic representatives, but merely use them directly (with the exception of locations representing locals whose addresses pass through a recursive call).

Mapping can improve precision by allowing more strong updates to be performed; bodies of called procedures will sometimes see a single synthetic location where our analysis will see a set of invisible locations. However, mapping may worsen the precision of context-insensitive analysis. Consider two locations that participate in a points-to relationship at a call site, but which are invisible to the called procedure. Under a context-insensitive analysis, the corresponding synthetic locations must also have a points-to relationship. Depending on the mapping scheme used, this may force all invocations of the procedure body to be analyzed in a context where these synthetic locations have a points-to relationship, even though this relationship may not apply at some sites. Treating the invisible variables explicitly increases costs (due to less memoization), but allows us to avoid this pollution in the context-insensitive case.

- Handling of heap allocation sites. Our analysis constructs only a single representative base-location for each invocation site of heap memory allocators (malloc, realloc, etc). Thus, all clients of a heap-allocated data abstraction will manipulate the same paths, irrespective of whether context-insensitive or context-sensitive analysis are used. More precise heap analyses [Har89, CWZ90, CBC93, Deu92, Deu94] allow multiple representatives per allocation site, yielding a larger pool of locations, and thus a larger set of spurious points-to relations in the context-insensitive case.[9] A similar argument applies to the handling of arrays (treating various subscript ranges independently will increase the number of distinct access paths).

- Program representation. Before our points-to analyses run, our compiler has already performed significant amounts of value numbering, loop invariant code motion, and dead code removal, along with an SSA-like transformation that removes non-addressed variables from the store [Ruf95, Section 3]. It may be the case that, without these optimizations, more spurious (though irrelevant) points-to pairs would be generated.

- Problem formulation. Because an alias-pair formulation of the problem manipulates a set of location-valued expressions that is potentially far larger than our set of access paths (since location-valued expressions may include multiple levels of indirection), the problem of spurious pairs is likely to be larger in such a framework.

5.1.2 Benchmark Characteristics

The structure of our test programs also contributes to the lack of spurious points-to pairs. In these small programs, abstract data types types typically have only one client. Thus, even under context-insensitive analysis, there is little danger of cross-pollution between an abstraction's clients. While this is difficult to quantify, we believe it to be one of the more important factors leading to our result, and the most likely to change when larger programs are analyzed.

Additionally, these programs have relatively sparse call graphs; while procedures average 4.2 callers, 54% of procedures have only one caller.[10] Procedures with multiple callers tend to be near the leaves of the call graph, decreasing the number of contexts they induce in their callees.

Finally, these programs exhibit only shallow nesting of pointer datatypes; the vast majority of pointers are single-level (*i.e.*, they reference scalar datatypes). This means that many procedures that perform indirect reads and writes do not modify points-to relationships in their callers. For example, utility code for performing string operations reads many character-valued pointers, but only writes to character-valued (rather than pointer-valued) memory locations. Shal-

[9] There is an interesting paradox here: more precise analyses often produce what appear to be inferior statistics (in terms of the number of paths accessed/modified by an indirect memory operation) because they begin with a larger pool of potential locations. Research results would be easier to compare if statistics were reported in terms of the *fraction* of potential locations referenced/modified, instead of the absolute number

[10] These statistics do not include library procedures known not to affect the points-to solution; these are modeled as the identity function on stores, avoiding any danger of context pollution

All points-to pairs (context-insensitive)

	referent			
path	function	local	global	heap
offset	<0.1%	0.1%	1.2%	0.7%
local	0.0%	0.0%	14.0%	5.8%
global	0.0%	0.0%	43.1%	12.6%
heap	0.0%	<0.1%	5.6%	16.8%

Spurious points-to pairs only

	referent			
path	function	local	global	heap
offset	0.0%	0 0%	<0.1%	0.1%
local	0.0%	0.0%	34.1%	8.1%
global	0 0%	0.0%	3.1%	29 9%
heap	0 0%	0.1%	5.1%	19 5%

Figure 7: Context-insensitive and spurious points-to pairs, broken down by path and referent types. "Offset" paths include all access paths without base-locations; *i.e.,* pointer, function, and aggregate edges in the VDG The "local" designation includes all procedure locals and parameters, "global" includes string literal storage.

low nesting also means that a single spurious pair is less likely to generate a cascade of other spurious relationships downstream when that pair is dereferenced, since the dereference is unlikely to yield a pointer value.

5.2 Lack of consequences due to spurious pairs

The lack of adverse consequences of spurious points-to pairs can be attributed to several factors. The first is the problem definition itself. If the goal of points-to analysis is to model all relationships at all program points (perhaps for use in a debugger or programming environment), then spurious points-to pairs are clearly a problem. If, however, we restrict the problem, then only certain subclasses of the spurious pairs (in this case, those on pointer-valued outputs consumed by indirect memory operations) are relevant. In every test case other than *compress* and *span*, all of the spurious pairs are on store-valued outputs and are thus uninteresting (the spurious pointer pairs in *compress* and *span* are produced by library calls whose return values are not used; such spurious pairs on "dead code" can also be considered harmless).

For an "uninteresting" spurious points-to pair to induce no "interesting" spurious pairs downstream, it must be the case that either (1) no downstream code references this pair's path, or (2) references to this pair's path also read another, non-spurious, pair with the same referent. Case (1) is typically a consequence of coding style, in that most callers to a procedure performing a side effect will detect only some of the potential effects. For example, a common paradigm has callers pass addresses of pointer-valued local storage to a procedure which then modifies that storage. A particular caller will typically not detect the (spurious) side effects performed on other callers' storage because it doesn't look there. If the callee writes the same pointer value (perhaps the address of a buffer in global or heap space) to all callers' storage, the caller won't detect the spurious side-effects to its storage either. (Due to the largely single-level nature of pointers in our benchmarks, the callee usually writes a scalar, in which the callee's write won't induce any points-to pairs in the first place).

Case (2) can occur in a number of ways, including the merging of points-to sets at intraprocedural control flow joins, and the coarse handling of arrays and heap allocation. Sometimes this case appears through pure serendipity; *e.g.*, the *part* benchmark independently construct two linked lists that are both manipulated via the same set of routines, resulting in cross-pollution from context insensitivity. However, early in its execution, the program exchanges elements between the lists, forcing each list's locations to model all of the values held by the other list's locations. Thus, any spurious pair pointing into the incorrect list will reference the correct values anyway.

Circumstantial evidence supporting both possibilities can be found in Figure 7, which shows the distribution of path and referent types for all context-insensitive points-to pairs and for spurious points-to pairs only. A far larger proportion of the spurious pairs model local variables, which are more likely to be dead at any given point (*e.g.*, incorrectly returning a pair whose path is a local variable of procedure A to procedure B won't cause problems because B will not dereference A's local). Spurious pairs are also more likely to point to heap storage, which (due to the relative scarcity of heap locations in our model, and the inability to perform strong updates on heap locations) is more likely to correctly contain a larger variety of locations than global storage, which is strongly updateable.

With the exception of the mapping of invisible locations to synthetics, the arguments of Sections 5.1 and 5.2 apply equally well to other context-sensitive pointer alias analyses [LR92, CBC93, EGH94]. Thus, we believe that they, too, will realize only minor precision benefits from context-sensitivity on our benchmark suite.

6 Conclusion and Future Work

We have performed what we believe to be the first empirical comparison between program-point-specific context-insensitive and context-sensitive alias analysis techniques. Our conclusion, that adding context-sensitivity to a points-to analysis provides little or no precision benefit on our test cases, should not be taken to mean that we do not find context-sensitive analyses to be a useful line of research. On the contrary, we believe that context-sensitivity will be essential once we begin analyzing larger programs under a richer model of the heap. However, we do believe that present benchmarks and analysis results form inadequate proof of the utility of context-sensitive techniques in practice.

In addition, we have found that context-insensitive techniques can be implemented quite efficiently, and can produce information that is surprisingly precise, as well as useful in improving the efficiency of more sophisticated techniques. We plan to continue working in this vein, using simple, efficient techniques to limit the search space of more expensive, more precise analyses.

Acknowledgements

Roger Crew, Michael Ernst, Ellen Spertus, Bjarne Steensgaard, and Daniel Weise co-developed the VDG-based programming environment in which this work is situated. Bjarne Steensgaard and Daniel Weise commented on drafts of this paper. We would like to thank William Landi and Todd Austin for sharing their benchmark suites with us.

References

[ABS94] T. M Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 290–301. ACM Press, 1994.

[CBC93] J -D Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245 ACM Press, Jan. 1993.

[CFR+91] R. Cytron, J Ferrante, B K Rosen, M. N Wegman, and F. K. Zadeck Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4) 451–490, Oct 1991

[Coo89] B. G. Cooper. Ambitious data flow analysis of procedural programs. Master's thesis, University of Minnesota, May 1989.

[Cou86] D S. Coutant Retargetable high-level alias analysis. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 110–118. ACM Press, 1986.

[CR82] A. L. Chow and A Rudmik The design of a data flow analyzer. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 106–119. ACM Press, 1982.

[CWZ90] D. R Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, June 20-22, 1990

[Deu92] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *International Conference on Computer Languages*, pages 2–13 IEEE, Apr. 1992.

[Deu94] A. Deutsch. Interprocedural may-alias analysis for pointers. Beyond k-limiting. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 230–239. ACM Press, 1994.

[EGH94] M Emami, R Ghiya, and L J Hendren Context-sensitive interprocedural analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256. ACM Press, June 1994

[Har89] W. L Harrison III. The interprocedural analysis and automatic parallelization of Scheme programs *Lisp and Symbolic Computation*, 2(3/4).179–396, 1989.

[HPR89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of the SIGPLAN'89 Symposium on Compiler Construction*, pages 28–40, June 1989 Published as SIGPLAN Notices Vol 24, Num. 7.

[Lan92] W. A Landi *Interprocedural Aliasing in the Presence of Pointers* PhD thesis, Rutgers University, Jan 1992

[LR92] W. Landi and B. G. Ryder A safe approximate algorithm for interprocedural pointer aliasing In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248 ACM Press, June 1992

[LRZ93] W Landi, B G Ryder, and S Zhang Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67 ACM Press, June 1993

[PLR92] H. D. Pande, W Landi, and B. G. Ryder. Interprocedural reaching definitions in the presence of single level pointers. Technical Report lcsr-tr-193, Laboratory for Computer Science Research, Rutgers University, Oct 1992.

[RM88] C. Ruggieri and T P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 285–293, Jan. 1988.

[Ruf95] E. Ruf Optimizing sparse representations for dataflow analysis. In *ACM SIGPLAN Workshop on Intermediate Representations (IR'95)*, pages 50–61, Jan 1995 Proceedings available as Microsoft Research technical report MSR-TR-95-01

[Ryd89] B. G. Ryder Ismm Incremental software maintenance manager. In *Proceedings of the IEEE Computer Society Conference on Software Maintenance*, pages 142–164, 1989.

[WCES94a] D. Weise, R. F Crew, M Ernst, and B. Steensgaard. Value dependence graphs: Representation without taxation In *Proceedings 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 297–310, Jan 1994

[WCES94b] D. Weise, R F. Crew, M. Ernst, and B. Steensgaard Value dependence graphs: Representation without taxation. Technical Report MSR-TR-94-03, Microsoft Research, Redmond, WA, Apr. 13, 1994.

[Wei80] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables In *Proceedings of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, Jan 1980.

[WL95] R P Wilson and M S Lam. Efficient context-sensitive pointer analysis for C programs In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation* ACM Press, June 1995 (this volume)