Pointer Analysis – A Survey

Vishwanath Raman
Computer Science
University of California, Santa Cruz
Santa Cruz, CA 95064
vishwa@soe.ucsc.edu

December 1, 2004

1 Introduction

Pointer analysis has been studied for over two decades. It answers the following question – For variables of pointer types, what are the set of objects they may point to at runtime. The analysis is also called Points-to analysis. Pointer analysis is used in a variety of applications, including live variable analysis for register allocation and constant propagation, static checking of potential runtime errors, such as NULL pointer dereferencing and for applications such as static schedulers that need to track resource allocations and usage precisely. In recent times it has been used in the context of detecting buffer overruns and print format string violations, that are known gateways that compromise security. Similar to other static techniques, pointer analysis is plagued by decidability issues and any solution found is always an approximation for most languages.

This document describes the contributions of a set of papers on the subject. The first section describes the problem using an example. The second section gives some background material. The third outlines implementation choices. The fourth and fifth describe the contributions of two papers that take a novel approach to analyzing pointers. The sixth describes the contribution of a paper on a practical application of pointer analysis and is followed by a conclusion.

2 Motivation

Pointer analysis finds its roots in alias analysis. Alias analysis attempts to create equivalence relations over abstract memory locations. Two variables are aliases for each other if they point to the same memory location. Changing the contents of one will indirectly change the other. Traditionally, all variables that are pair-wise equivalent are determined and sets of such pairs

are created. Assuming the alias relation is transitive the variables in each set are all aliases for each other. Pointer analysis is similar to alias analysis but has a tighter space requirement.

Consider pointer analysis for the following C program:

```
Example 1

int main (void) {
  int x, y, *p, **q, (*fp)(char*, char*);
  p = &x;
  q = &p;
  *q = &y;
  fp = &strcmp;

  return 0;
}

A safe and minimal points-to map for the above program is -
```
[fp $\rightarrow$ $\{strcmp\}$, q $\rightarrow$ $\{p\}$, p $\rightarrow$ $\{x,\ y\}$]

```
End of Example 1
```

A trivial points-to map exists for all languages. Languages with unrestricted pointers have a trivial points-to map where each pointer points to all variables. A well-typed language has a trivial points-to map where each pointer points to all variables of the same type. This is the worst-case conservative estimate of a points-to set in the absence of doing any pointer analysis. The problem of pointer analysis can therefore be restated as the process of refining this worst-case points-to set to get better approximations with respect to candidate target applications.

Each application demands a different granularity in the accuracy of the points-to set. For compiler optimizations, that are semantics preserving rewrites of the source program, the accuracy of pointer analysis impacts the performance of the final generated code, whereas for applications such as static schedulers, that attempt to track the usage of resources such as semaphores accurately, fairness is tainted in the absence of most accurate yet conservative points-to sets.

3   Background

This section describes some of the definitions and abbreviations we use in the remaining sections of this document with examples for added clarity.

3.1 Definition-use (def-use) analysis

An assignment statement contains both a definition and one or more uses. The left hand side of the assignment is a definition for the variable being assigned. The variables in the right hand side expression are all use points of the most recent definitions of those variables. These most recent definitions are also known as reaching definitions. The determination of all def-use pairs in order to construct a def-use chain is called definition-use analysis.

3.2 Static single assignment form (SSA form)

This is an intermediate representation of a program, where each variable is uniquely renamed at each statement in the program that contains a definition for that variable. SSA forms and their variants are used extensively in compiler optimizations and in various other applications including pointer analysis [4].

```
Example 2

1.  X = 1;     // will become X₀ = 1
2.  Y = 2;     // will become Y₀ = 2
3.  X = 3;     // will become X₁ = 3
4.  Z = X + Y; // will become Z₀ = X₁ + Y₀

End of Example 2
```

In the above transformation Z uses the reaching definitions X_1 and Y_0 at line 4. From the perspective of compiler optimizations a use of renaming is the reduction of the live ranges of variables, enabling better register allocation.

3.3 Dominance frontier

A language with conditional statements will have confluence points for multiple control paths in the CFG. Such confluence points are called dominance frontiers. A node in the CFG dominates another node, if every path leading to the latter includes the former. Definitions from dominating nodes will unconditionally reach confluence points, whereas definitions originating from nodes that are not dominating reach dominance frontiers predicated by the branching conditions. For such definitions, definition selectors are inserted at the dominance frontiers to choose between multiple reaching definitions. These functions are called ϕ functions in SSA forms and γ functions in gated SSA forms, where the branching predicates are used as gates in the latter. Consider the following example -

Example 3

```
1.  int *u;
2.  int a = 0;      // a_0 = 0
3.  if (P) {
4.     u = &a;       // u_0 = &a
5.  } else {
6.     u = NULL;     // u_1 = NULL
7.  }                // u_2 = φ(u_0, u_1)
8.  a = 1;           // a_1 = 1
9.  if (P) {
10.    a = *u;        // a_2 = u_2
11.    ...
12. }
```
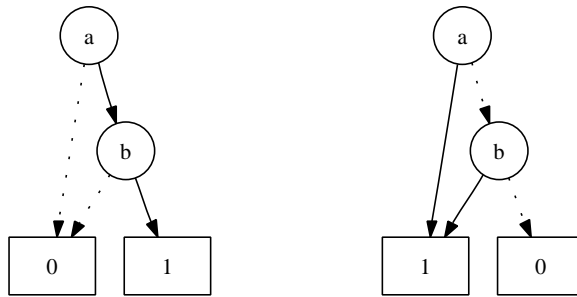
End of Example 3

The confluence point of the branch on line 3, namely line 7 has a new definition for variable u, in terms of a ϕ function that chooses between one of the two reaching definitions of u. Note that the ϕ function does not discriminate between the two branches of the if condition, whereas a gating γ function will.

3.4   Binary Decision Diagrams (BDDs)

BDDs are used in practice for many applications including synthesis and verification of hardware designs. They are a cornerstone of model checking algorithms. A BDD is a directed acyclic graph used to represent Boolean functions or the state space covered by a set of Boolean variables. They are canonical representations of Boolean formulae given an order on the variables used in the formulae.

The following example shows BDDs for Boolean AND and OR functions of two variables a and b. The dotted lines are edges corresponding to a branch when the source variable is zero, and the solid lines correspond to a branch when the variable is one. The destination node of the dotted edge from a, represents a co-factor of the function f with respect to ¬a, while the destination node of the solid edge from a, represents the co-factor of f with respect to a. The terminal nodes are constants one and zero, representing the truth or falsehood of the Boolean formula. When a BDD is interpreted as a set, all paths leading to the one node are members of the set.

Example 4

```
Functions:  f = a ∧ b and f = a ∨ b
Sets:  S = {11} and S = {10, 11, 01} or {1X, 01}
```

```
End of Example 4
```

4   Implementation alternatives

The granularity of the accuracy of the points-to set, dictated by target applications, diverges considerably from analyses that are intra-procedural to inter-procedural with or without flow sensitivity and path sensitivity. A points-to analysis with full flow and path sensitivity is usually prohibitively expensive and can be applied only to interesting procedures or paths.

4.1   Flow-sensitive vs. Flow-insensitive

A flow sensitive analysis will consider the flow of values in a program or procedure in deciding the points-to set whereas a flow-insensitive analysis will be conservative and consider all assignments to a pointer variable as equally likely and hence produce a courser points-to set. For example an analysis over the SSA form, where all variables are renamed, can help create program point specific sets as opposed to procedure based summary sets.

4.2   Path-sensitive vs. Path-insensitive

not sure about the example???

Consider example 3. For the reaching definition of u at line 10, a path-insensitive analysis will use u_2, whereas a path-sensitive analysis will use u_0, since the same predicate P is used in both conditional statements.

4.3   Intra-procedural vs. Inter-procedural

Intra-procedural analysis is local to a procedure and creates a summary of points-to sets based on local information alone. For parameters and global variables being used in the procedure worst-case assumptions are made, namely that they are all UNKNOWN. Pointers returned by function calls

are also considered UNKNOWN [1]. In inter-procedural analysis, each procedure is analyzed with definitions of incoming parameters and global variables that are defined and/or used in that procedure. For inter-procedural analysis, we need the entire program, with the complete call-graph.

One way of achieving inter-procedural analysis is to inline all procedures with suitable variable renaming or to copy procedures to create as many instances as the number of points where they are called. Each copy is a variant of the procedure being called. Each variant can then be analyzed using context sensitive definitions of actual parameters. The latter method is known as procedure cloning. For recursive procedures, a limit known as k-limit is used in the literature to describe unrolling the procedure k times to improve analyses accuracy.

In summary, flow-insensitive analyses are fast while sacrificing accuracy, as we will see from "Points-to analysis in almost linear time" [2] later in this document. Flow-sensitive analyses are slow and are in general ad-hoc. The same holds for path-insensitive vs. path-sensitive and intra-procedural vs. inter-procedural analyses.

5   Pointer analysis using a type system

The work of Bjarne Steensgaard is one of the more formal approaches to the problem of points-to analysis. The paper titled "Points-to analysis in almost linear time" [2] describes an inter-procedural flow-insensitive, path-insensitive points-to analysis in almost linear time by type inference. The type system introduced by Steensgaard is based on a storage model of variables. A storage model describes the relationships between abstract memory storage locations. A storage model is realized as a Storage Shape Graph (SSG), where the vertices are abstract memory locations and the edges represent the points-to relation. In essence, a pointer variable occupies some storage and points to some storage, contributing an edge from one abstract memory location to another. Steensgaard's algorithm can be used to compute an SSG. From the SSG one can extract points-to information or alias information.

IMPORTANT

The language of choice is critical for a type system based inference of points-to sets. A type preserving subset of C that excludes casting of pointers and pointer arithmetic lends itself to such an analysis. The type inference based approach assumes that the program being analyzed has a type preserving operational semantics. A general language of pointers is defined, together with a type system for its storage model and a set of inference rules. Points-to analysis reduces to the problem of assigning types to all locations (variables) in a program, such that the statements in the program are well-typed given the set of inference rules. At the end of the assignment of types, two locations are assigned different types, unless they have to be

described by the same type in order for the program to be well-typed.

Consider the following language -

```
S ::= x = y
   | x = &y
   | x = *y
   | x = op(y₁.. .yₙ)
   | x = allocate(y)
   | *x = y
   | x = fun(f₁.. .fₙ) → (r₁.. .rₙ) S*
   | x₁.. xₘ = p(y₁.. .yₙ)
```

The types -

```
τ ::= ⊥ | ref(V → α)
V ::= P(absloc)
α ::= (τ * λ)
λ ::= ⊥ | lam(α₁.. . αₙ)(αₙ₊₁.. . αₙ₊ₘ)
```

Some type rules -

```
x:ref(_ → α) ∧ y:ref(_ → α) ⇒ welltyped(x = y)
x:ref(_ → (τ * _)) ∧ y:τ ⇒ welltyped(x = &y)
x:ref(_ → (ref(_ → α) * _)) ∧ y:τ ⇒ welltyped(x = allocate(y))
```

Some inference rules -

```
x = &y:
  let ref(_ → (τ₁ * _)) = type(ecr(x))
  and τ₂ = type(ecr(y)) in
  if τ₁ ≠ τ₂ then
    join(τ₁, τ₂)

x = allocate(y):
  let ref(_ → (τ * λ)) = type(ecr(x)) in
  if type(τ) = ⊥ then
    let e₁ = MakeECR()
    and e₂ = MakeECR()
    and e₃ = MakeECR() in
    type(e₁) ← ref(_ → (e₂ * e₃))
    type(ecr(x)) ← ref(_ → (e₁ * λ))
```

7

In the above, S is a statement of the language, op is an operator and the other names have obvious meanings. Functions are modeled as taking multiple parameters and returning multiple values, which is a very generic interpretation of functions. In practice functions can modify global variables or parameters passed by reference in a language like C. The τ type is the type of a location, that can either be a bottom type (unassigned type) or a reference function that maps an abstract location to an α term. An α term is an aggregate that begins with a location type, followed by a function type. The λ type is the function type or the bottom type. The type V represents a set of abstract locations. By interpreting the contents of locations as an aggregate, the type system accommodates a notion of structures in C, where the address to an object of a structure type is the same as the address of the first field in that structure [ISO 1990]. The analysis though is not field sensitive.

The type rules and inference rules mentioned above are a subset of Steensgaard's type system. We have enumerated rules to derive simple assignments, assignments with the address-of operator and the more interesting one for allocate. The modeling of locations is very generic. Each location is assumed to contain an aggregate of the form τ * λ, with pointers to locations and functions.

In inference rules, the acronym ECR (or ecr) stands for Equivalence Class Representation. New locations are created only through the allocate calls and hence the inference rule for allocate creates a new equivalence class for α terms in x and y. MakeECR() creates an empty equivalence class with the bottom type $\perp$ as the return value. So, all new locations have the bottom type. They change as use points for the newly allocated locations are encountered in the program.

Note that a location can point to a location reference. Therefore, the system models pointers to pointers. The join operation in the inference rules unifies types. For example, in the inference rule for assignment using an address-of operator, since the program is well-typed, the type for the location referenced by the location on the left should be the same as the type for the location on the right. A join is therefore performed on the two encountered types τ_1 and τ_2. This involves renaming the location types τ_1 and τ_2 to be the same and replacing every occurrence of either τ_1 or τ_2 to be the new type.

The algorithm works on the entire program and is inter-procedural. It is assumed that the variables have been rendered unique across multiple call instances of functions. Each statement is visited exactly once. The processing of a program statement changes the typing of the locations in that statement so that the statement is well-typed.

Example 5

```
1.   a = &x              a:   τ₁ = ref(S₁ → (τ₄ * λ₁))
2.   b = &y              b:   τ₂ = ref(S₁ → (τ₅ * λ₂))
3.   if P then           c:   τ₃ = ref(S₁ → (τ₅ * λ₃))
4.     y = &z            x:   τ₄ = ref(S₄ → α₁)
5.   else                y:   τ₅ = ref(S₅ → (τ₄ * λ₄))
6.     y = &x            z:   τ₄
7.   fi                  P:   τ₆ = ref(S₆ → α₂)
8.   c = &y
9.   x = z
```

$$
\begin{array}{ll}
1.\quad \texttt{a = \&x} & \texttt{a:}\quad \tau_1 = \texttt{ref}(S_1 \rightarrow (\tau_4 * \lambda_1)) \\
2.\quad \texttt{b = \&y} & \texttt{b:}\quad \tau_2 = \texttt{ref}(S_1 \rightarrow (\tau_5 * \lambda_2)) \\
3.\quad \texttt{if P then} & \texttt{c:}\quad \tau_3 = \texttt{ref}(S_1 \rightarrow (\tau_5 * \lambda_3)) \\
4.\quad \texttt{y = \&z} & \texttt{x:}\quad \tau_4 = \texttt{ref}(S_4 \rightarrow \alpha_1) \\
5.\quad \texttt{else} & \texttt{y:}\quad \tau_5 = \texttt{ref}(S_5 \rightarrow (\tau_4 * \lambda_4)) \\
6.\quad \texttt{y = \&x} & \texttt{z:}\quad \tau_4 \\
7.\quad \texttt{fi} & \texttt{P:}\quad \tau_6 = \texttt{ref}(S_6 \rightarrow \alpha_2) \\
8.\quad \texttt{c = \&y} & \\
9.\quad \texttt{x = z} &
\end{array}
$$

End of Example 5

From the above example and the inferred location types, we can derive the following SSG – $[(\tau_1 \rightarrow \tau_4), (\tau_2 \rightarrow \tau_5), (\tau_3 \rightarrow \tau_5), (\tau_4), (\tau_5 \rightarrow \tau_4), (\tau_6)]$, using which we get the point-to set – $[a \rightarrow \{x, z\}, b \rightarrow \{y\}, c \rightarrow \{y\}, y \rightarrow \{x, z\}]$. The fact that the analysis is path-insensitive causes the safe points-to set for y, namely $\{x, z\}$. The algorithm is flow-insensitive which is not immediately obvious from the above example. If the program is modified so that downstream x and z are made distinct, then the algorithm will produce a superset of the same points-to sets, since a join is never undone based on context. A subsequent reaching definition of the variable a will have less precise points-to information. Since processing a statement can have an impact on the types inferred for locations already processed, the join is recursively applied. Since the algorithm works on the location types database, the impact is global and hence flow-insensitive.

The complexity of Steensgaard's algorithm is almost linear in the number of statements N. Since, the processing of each statement may involve recursive join operations on encountered types, the overall complexity is not strictly linear but has been determined $O(N\alpha(N, N))$, where $\alpha(N, N)$ is a slowly increasing inverse Ackermann's function.

TODO: Read the BDD Paper

6 Pointer analysis using BDDs

This is based on the paper "Points-to Analysis using BDDs" [3] from the SABLE group at McGill. This is the first attempt, as the authors write, at using BDDs for pointer analysis. This approach addresses the issue of scalability in pointer analysis. This technique while being flow-insensitive and path-insensitive like the analysis based on type inference, addresses one of the principle limitations of the type based approach, namely handling fields of structures. The types based analysis provides for abstract memory locations to be aggregates, but does not analyze points-to sets at the granularity of fields, like Andersen's inclusion based approach does [1]. Andersen's approach uses a constraint solver over inclusion constraints derived from the

handles fields of structs

statements of a C program. The solving process iteratively traverses dereferences and field accesses till a fixpoint is reached.

In this approach BDDs are employed to represent sets over Boolean variables. A particular assignment of Boolean values for the variables will be represented as a string of zeroes and ones, for a given variable order. A relation will then be a subset of the 2^n possible strings over n ordered variables.

The analysis attempts to create a points-to set that maps each variable in the program with the abstract memory locations that it can point to. This is different from the richer Steensgaard's analysis. The work done using BDDs can be extended for generality. This technique is explained using the following example,

```
Example 6

1.   a = allocate(x)
2.   b = allocate(x)
3.   c = allocate(x)
4.   a = b
5.   c = b

End of Example 6
```

We would like to extract the points-to set {(a, A), (a, B), (b, B), (c, C), (c, B)}, where A, B and C are abstract memory locations that correspond to the allocate calls. In this scheme, variables form one domain and abstract memory locations another. Field accesses are modeled as relationships between abstract memory locations. Therefore, the points-to sets will have the form, {(v, h) | v ∈ V ∧ h ∈ H} ∪ {(h_1, h_2) | h_1 ∈ H ∧ h_2 ∈ H}. V is the variables domain and H is the abstract memory locations domain.

Each variable is assigned a unique bit string. Each abstract memory location (including field access) is assigned a unique bit string. For a set of n variables and m abstract memory locations, we need log(n)+log(m) bits in the final representation. For Example 6, the variables a, b and c can have strings 00, 01 and 10 representing them. The abstract memory locations A, B and C can have the same 00, 01 and 10 encoding. For the allocate statements we have the following strings that represent valid points-to relations {(0000), (0101), (1010)}, which is the same as {(a, A), (b, B), (c, C)}. The assignment statements on lines 4 and 5 contribute the following strings purely in the variables domain {(0100), (0110)}, which is the same as {(b, a), (b, c)}. The relations in the variables domain for the above example can be read as b reaches a and b reaches c.

The task of computing the final points-to set now amounts to set operations on the BDDs that store the encoded allocations and assignments.
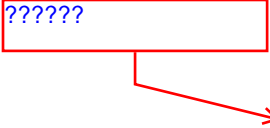
Note that in using BDDs to represent inclusion relationships over the variables domain, we need to replicate all variables to store relations such as a → b, since b cannot be the same as the b in the domain of a in order to instantiate valid BDDs, as BDDs are acyclic graphs by definition. The variables in a program contribute two domains V1 and V2, that have the same string encodings and are replicas of each other.

The set operations relProd, replace and union are then used to compute the final points-to set. The relProd relation is defined as, $relProd(X, Y, V1) = \{(v_2, h) \mid \exists v_1 ((v_1, v_2) \in X \land (v_1, h) \in Y)\}$. The replace set operation will replace variables in one domain with variables from another domain and union has the usual meaning.

Given these relationships, applying relProd on the sets X = {(a, A), (b, B), (c, C)} and Y = {(b, a), (b, c)} over the domain V1, gives us the set {(a, B), (c, B)}. The variables a and b in the resulting set belong to the domain V2. A replace operation will replace all variables in domain V2 with variables in domain V1. A subsequent union of the set X that represents the points-to relationship from variables to abstract locations and the result of relProd will yield the final points-to set, {(a, A), (a, B), (b, B), (c, C), (c, B)}.

This technique extends to structure fields in a similar fashion, with sets that contain relations between abstract memory locations. The paper also talks about variable ordering which is beyond the scope of this document. In short the implementation issues attempt to address the eternal problem of variable ordering in BDDs to produce the most concise representations, improving runtime efficiency. The complexity is related to the complexity of BDD operations, that are in general linear in the size of the BDDs.

7   A practical application of pointer analysis

There is extensive literature on pointer analyses that compute points-to sets directly by using the CFG of a given program. This has been the "traditional" approach to pointer analysis with the types based or BDD based analyses being relatively recent. The earliest reference to alias analysis I found was to a paper by Aho in 1986. For a discussion of these techniques the paper "Tracking Pointers with Path and Context Sensitivity for Bug Detection in C programs" [4] from the SUIF group at Stanford, is fairly representative and is also recent.

The technique employed is to construct a variant of the SSA form called IPSSA form (Inter-Procedural SSA form). The IPSSA form tracks accurately intra as well as inter-procedural def-use relationships. The assumption made for procedure parameters and global variables is that there are no aliases between them, while processing a particular call instance. This assumption, though unsound, enables a faster algorithm. It is sound from

a practical programmer perspective in that most functions are written to factor in possible aliases amongst parameters by defensive programmers. If not, the fact that functions are usually written to work in the presence of aliases renders the approach relatively safe.

From the Background section above, we have a definition of SSA. We introduced a ϕ function that captures unconditionally, possible reaching definitions at the confluence of many edges in a CFG. A variant of SSA called gated SSA replaces ϕ functions with γ functions that capture control flow. In Example 3 above, the ϕ function will be replaced by γ(P, u_0, ¬P, u_1), where the value of the predicate P determines which of the two definitions reach subsequent uses of variable u. This renders the representation path sensitizable. In reality, since most predicates are dynamic and rarely statically determinable, the analyses are typically of the "what-if" nature, exhaustively examining def-use pairs for all possible values of the gating predicates. This is in fact the source for an exponential increase in processing times.

IPSSA extends gated SSA to handle inter-procedural def-use analysis. It introduces two new functions ι functions and ρ functions. The former take call site, definition pairs for each parameter and the latter take function instance and return value definition pairs. Each call site is uniquely numbered. Each call instance calls a particular variant of a given function. The function ι(c1, d_0, c2, e_0) associates the definition d_0 with call site c1 and definition e_0 with call site c2. Similarly, the function ρ(f1, RET_0^{f1}, f2, RET_0^{f2}) associates return definitions with function call instances f1 and f2. This helps track all possible paths through function calls and return statements to compute very accurate def-use chains.

The procedure defines the notion of *hot-locations*. All global variables, procedure parameters, local variables and locations accessed using the notion of *simple access paths* are *hot-locations*. Access paths that are one level of dereference or one level of field access for record types are considered *simple access paths*. The analysis that builds the IPSSA does not process non-simple access paths, which are paths that have iterated dereferences (pointers to pointers) or iterated field accesses. Yet, the IPSSA form is flow-sensitive and path-sensitive through the use of the gating γ functions. It is inter-procedural through the use of ι and ρ functions.

Steengaard's flow-insensitive and path-insensitive analysis is used to get a first cut approximate of points-to sets. The IPSSA structure is then built from the call graph by iterating over each function bottom-up, replacing indirect accesses and definitions by direct accesses while creating ρ functions and ι functions at call sites and return statements.

Once we have the IPSSA form built for a program, analyzing points-to relationships and code vulnerabilities is fairly straight-forward. Code vulnerabilities are analyzed over def-use chains computed on the IPSSA

form. As an example, consider buffer overruns. In a given def-use chain, if the definition is a statically allocated buffer (known static size) and the eventual use (or sink) is a write into that buffer, then this is a potential vulnerability if the user supplied string overruns the static buffer. If the sizes of the sources and targets can be accurately captured statically, then this would generate an error. If they cannot all be determined statically or if there are gating predicates, the analyzer can warn the user with an assignment of predicates (if any) along the def-use chain that will lead to the potentially bad sink.

As precise analysis of all def-use chains will cause an exponential increase in runtime, the analysis is limited to paths or procedures that are considered interesting. For the remainder we have the less accurate results computed using Steensgaard's analysis.

8 Conclusion

We have seen the various flavors of pointer analysis in this document, from flow and path insensitive to flow and path sensitive. We have also seen how Steengaard's formal type system or the SABLE group's BDD based approach can be used to infer points-to sets. We have seen a practical application of pointer analysis through a more accurate representation prior to analysis by the SUIF group at Stanford.

While Steengaard's analysis is not as accurate as some applications demand, it is always a first cut approximate, that will suffice for many applications. The fact that fields are not handled is a handicap for languages such as C, but then the ideas are sound and can be easily extended to handle everything except transformations that don't preserve types. Even in such cases, sub domains that are type preserving could still be analyzed. It is fast and appears to be an initial step in more accurate pointer analyses as is evident from the construction of the IPSSA form. The strength of Steengaard's analysis is that aliases identified are maximal. The method is sound. References mapped to different abstract memory locations (implies different assigned types) are guaranteed to be un-aliased, leaving only the refinement of the aliases mapped to the same abstract memory location as the problem for any subsequent analysis.

The BDD based approach is nascent but promising. BDDs tend to increase in size and are very sensitive to variable orders. Even with the best order, the number of variables in an industrial application can be very large. So, scalability is always an issue. Again, Steengaard's analysis to identify an initial points-to set before refining each of them using BDDs may scale very well. This approach handles fields of structures that are absent in Steengaard's analysis. The advantage of this approach is that BDD packages have been tweaked for performance in the hardware verification domain for many

years and are well maintained. More sophisticated language constructs can be added to this approach to handle more cases. Procedure cloning that produces big increases in the number of variables can also be tackled using this approach.

The analysis over IPSSA forms of programs is by far the most accurate. It does not handle iterated pointer dereferencing or field accesses from a performance perspective. For such cases, a flow-insensitive first approximate is still available. They demonstrate the usefulness of their system in finding many vulnerabilities and only one false positive in their paper [4].

Pointer analysis has existed for more than two decades. It finds application in many areas related to processing programs, from optimizations in compilers to bug detection. In the work that we are doing in the Design Verification Lab under Prof. Luca de Alfaro, we need pointer analysis to capture def-use chains of resources such as mutexes or counting semaphores to ensure fairness in the static schedules.

Pointer analysis is an area that has seen significant development and will continue to be active as long as we have popular languages that support pointer types. Even after such languages are replaced with languages similar to ML, there will still be legacy applications that will require such analysis. While static techniques are not fully decidable, the reason they are extremely useful is vested in every programmers desire to get her or his program correct.

References

[1] L. Andersen. *A Program Analysis and Specialization for the C Programming Language*. Ph.D. thesis, DIKU, 1994.

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

[3] M. Berndl, O. Lhoták, F. Qian, L. Hendren and N. Umanee. *Points-to Analysis using BDDs*. PLDI 2003, June 2003.

[4] V. B. Livshits and M. S. Lam. *Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs*. In Proceedings of the 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-11), September 2003.

[5] J. Whaley and M. S. Lam. *An Efficient Inclusion-Based Points-to Analysis for Strictly-Typed Languages*. In Proceedings of the $9^{t}h$ International Static Analysis Symposium, Madrid, Spain, September 2002.