Towards Scalable Flow and Context Sensitive Pointer Analysis

Jianwen Zhu
Department of Electrical and Computer Engineering
University of Toronto, Toronto, Ontario, Canada
jzhu@eecg.toronto.edu

ABSTRACT

Pointer analysis, a classic problem in software program analysis, has emerged as an important problem to solve in design automation, at a time when complex designs, specified in the form of C code, need to be synthesized or verified. However, precise pointer analysis algorithms that are both context and flow sensitive (FSCS), have not been shown to scale. In this paper, we report a new solution for FSCS analysis, which can evaluate the program states of all program points under billions of different calling paths. Our solution extends the recently proposed symbolic pointer analysis (SPA) technology, which exploits the efficiency of Binary Decision Diagrams (BDDs). With our new strategy of problem solving, called superposed symbolic computation, and its application on our generic pointer analysis framework, we are able to report the first result on all SPEC2000 benchmarks that completes context sensitive, flow insensitive analysis in seconds, and context sensitive, flow sensitive analysis in minutes.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.4 [**Programming Languages**]: Processors – compilers, optimization

General Terms

Algorithms, Languages, Experimentation

Keywords

Pointer analysis, binary decision diagrams, High-level synthesis

1. INTRODUCTION

An exciting implication of the systems-on-chip (SOC) technology is the convergence of hardware and software: we have seen not only an increasing migration of functionality from custom hardware implementation to software implementation for better flexibility, but also a shift of custom hardware development from hardware description language synthesis (such as VHDL/Verilog) to software language synthesis (such as C/C++) [17, 30, 18, 12, 24]. The latter trend, while compelling, cannot materialize unless efficient synthesis and verification methods are available for all constructs frequently used in software programs. Unfortunately, the most well understood computation model in the hardware community, the concurrent finite state machines (CFSM), does not match well with even a modestly sized C program, which is better modeled as a push-down state machine with potentially infinite states. An often-cited difficulty [8] is the pointer construct, which manifests in any C program with decent complexity, typically processing and manipulating a large amount of data using dynamically allocated memories. Pioneering efforts [21, 22, 19, 27] have shown that a prerequisite for the movement of behavioral synthesis from C/C++ languages is the solution of the pointer analysis problem, which conservatively estimates the runtime pointer values at compile time. It has also become apparent that for hardware synthesis to be worthwhile, the pointer analysis algorithm has to be precise to offer better performance than pure software implementation, and for hardware synthesis from C to be worthwhile, the pointer analysis algorithm has to scale to large programs to necessitate the departure from the mature HDL-based design methodology.

The quest for better precision and scalability for the pointer analysis problem has seen two decades of history in the software community. Numerous attempts have been made, the majority of which can be categorized according to two criteria: *flow sensitivity* and *context sensitivity*. A flow insensitive (FI) algorithm ignores the order of statements, whereas a flow sensitive (FS) algorithm takes control flow into account and performs the so-called *strong update*, or the killing of relevant old states whenever a scalar assignment occurs. A context insensitive (CI) algorithm does not distinguish the different calling paths of a procedure, whereas a context sensitive (CS) does so such that the spurious states generated by the mixture of actuals from different calling paths to a common procedure can be eliminated. Scalable solutions have been reported for the flow insensitive analysis [13], or partial context sensitive analysis [16, 11, 10, 5], but not flow sensitive analysis. The largest program reported by any sound FSCS algorithm, to the best of our knowledge, is limited to a few thousand lines of C code [9, 26][1]. The difficulty mainly stems from the fact that in principle, the program states at each program state of a procedure under potentially billions of different calling contexts have to be represented, tracked and computed; by far exceeding the capacity of conventional algorithms. The theoretical study reveals that the flow sensitive analysis problem for even a single procedure program can be a PSPACE-complete problem [4].

In this paper, we report a new FSCS analyzer, based on the symbolic pointer analysis technology recently proposed by Zhu [28], and Berndl et. al. [1]. The essence of this technology is to represent the point-to relation using its characteristic Boolean function, which in turn can be represented and manipulated using Bryant's Binary Decision Diagrams (BDDs) [2]. The technology has recently been shown to scale for FICS analysis [29, 25]. It has also been shown that it can alternatively be abstracted in the familiar paradigms of relational database [15], or datalog [25]. We extend this direction with several new contributions.

- **Superposed symbolic computation**: We propose a new problem solving strategy, which generalizes the technique responsible for the success of recently proposed BDD-based pointer analysis algorithms. This strategy is applied consciously to solve the FSCS analysis problem.
- **Flow and context sensitive symbolic analysis framework**: We establish a unique, elegant framework under which all pointer analysis algorithms with varying precisions can be implemented and compared. Employing the superposed symbolic computation paradigm, our framework can effectively perform context sensitive and/or flow sensitive pointer analysis the same way as context and flow insensitive analysis.
- **Symbolic flow analysis algorithm**: as an enabler for our framework to perform flow sensitive analysis, we devise a surprisingly simple algorithm to pre-calculate the whole program reaching definition in-

[1] The readers are referred to an detailed survey by Hind [14].

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

formation. This algorithm computes information impossible to compute and represent in the classic dataflow analysis framework, and avoid the repeated traversal of control flow graph during the fixed point iterations in the solving phase.

Our analyzer is the first symbolic pointer analysis algorithm capable of FSCS analysis. It is also the first analyzer, among all previously reported, that can perform FSCS analysis for all SPEC2000 benchmarks in minutes, and FICS analysis in seconds. Both results are reported with full context sensitivity, which for large benchmarks, the context count can exceed one hundred billions.

The rest of the paper is organized as follows. In Section 2, we introduce the superposed symbolic computation paradigm. In Section 3, we formulate the problem by showing how the relevant program information as well as the point-to information can be modeled symbolically. We describe our analysis framework in Section 4 and our flow analysis algorithm in Section 5. We give experimental results in Section 6 before we draw conclusions.

2. SUPERPOSED SYMBOLIC COMPUTATION

The key strategy advocated in this paper, called superposed symbolic computation, has first been used in [28] under the context of pointer analysis. In this section, we formalize it as a general strategy for solving combinatorial problems, in the hope that the same strategy can be extended to a broader scope of problems. The notation and foundation algorithms established in this section will be used throughout the rest of the paper.

Consider a discrete domain D. Since any discrete set is isomorphic to an integer set, without loss of generality, we limit our attention to the discrete domain $V = [0, n-1]$, where $n = |V|$ is the cardinality of V. It is well known that any set in a discrete domain can be mapped to a Boolean function under the binary number system.

DEFINITION 1. *Let $V = [0, n-1]$ be a discrete domain and $X = \{0,1\}^m$ be an m-dimensional Boolean space, where $m = \lceil log_2 n \rceil$. The* **characteristics function** *$\lambda_X^S : X \mapsto \{0,1\}$ of a set $S \subseteq V$ under X is defined as*

$$\lambda_X^S(\langle x_0, ..., x_{m-1}\rangle) = \begin{cases} 0, & \text{if } \sum_j x_j 2^j \notin V \\ 1, & \text{if } \sum_j x_j 2^j \in V. \end{cases}$$

The characteristics function is simply a set membership test. The importance of this construction is two-fold: First, traditionally manipulated as collections of values, sets can now be manipulated as Boolean functions. In fact, set union and intersection can be manipulated as Boolean disjunction and conjunction respectively. Second, when the characteristic functions are represented by Bryant's Binary Decision Diagrams (BDDs), the sizes of discrete sets are no longer proportional to their cardinality. Quite often, a large set corresponds to a small BDD, which can be exploited to develop scalable algorithms. Algorithms that manipulate BDDs are commonly referred to as symbolic algorithms, for their treatment of Boolean functions as "first-class values".

For convenience, we use the shorthand $\mathbf{i}_X$ to denote the special characteristics function $\lambda_X^{\{i\}}$, called **minterms**, for sets constructed from a single element. The set of all minterms form the basis of the family of characteristics functions: They are *complete* in the sense that any characteristics function can be represented as Boolean disjunction of a set of minterm; they are *orthogonal* in the sense that the Boolean conjunction of any distinct minterms are 0.

The immediate implication is that any graph $E \subseteq V \times V$, where V is the set of vertices, can be represented by its characteristics function, called **symbolic graph**, using two Boolean spaces V_0 and V_1, to encode the sources and sinks of the graph edges respectively.

EXAMPLE 1. *Consider a graph with vertex set $V = [0,5]$, and edge set $E = \{\langle 0,1\rangle, \langle 0,2\rangle, \langle 1,3\rangle, \langle 2,4\rangle, \langle 3,5\rangle, \langle 4,5\rangle\}$. The corresponding symbolic graph is*

$$\begin{aligned} \lambda_{V_0 \times V_1}^E &= \lambda_{V_0}^{\{0\}}\lambda_{V_1}^{\{1\}} \vee \lambda_{V_0}^{\{0\}}\lambda_{V_1}^{\{2\}} \vee \lambda_{V_0}^{\{1\}}\lambda_{V_1}^{\{3\}} \vee \\ &\quad \lambda_{V_0}^{\{2\}}\lambda_{V_1}^{\{4\}} \vee \lambda_{V_0}^{\{3\}}\lambda_{V_1}^{\{5\}} \vee \lambda_{V_0}^{\{4\}}\lambda_{V_1}^{\{5\}} \\ &= \mathbf{0}_{V_0}\mathbf{1}_{V_1} \vee \mathbf{0}_{V_0}\mathbf{2}_{V_1} \vee \mathbf{1}_{V_0}\mathbf{3}_{V_1} \vee \\ &\quad \mathbf{2}_{V_0}\mathbf{4}_{V_1} \vee \mathbf{3}_{V_0}\mathbf{5}_{V_1} \vee \mathbf{4}_{V_0}\mathbf{5}_{V_1}. \end{aligned}$$

□

With the symbolic graph representation, graph algorithms can be constructed exclusively using first-order logic operators, which include the following.

- $\neg x$: Boolean negation;
- $x \vee y$: Boolean disjunction;
- $x \wedge y$: Boolean conjunction;
- $\exists V_i . [x]$: Existential quantification, which eliminates variables in space V_i;
- $x|_{V_i \rightarrow V_j}$: Variable substitution, which substitutes variables in space V_i with the corresponding variable in space V_j.

Symbolic graph representation has been widely used in the CAD community, most notably in formal verification. Here the state transition graph is represented symbolically by a Boolean relation, called the transition relation. A breakthrough result [6, 3] was the efficient implementation of the **image computation operator**, which as shown in Line 8 of Algorithm 1, performs combined Boolean conjunction and existential quantification. Image computation is equivalent to finding the successors of a set of vertices, which when applied repeatedly, can lead to the efficient enumeration of all reachable vertices (states) from the starting vertex (state).

EXAMPLE 2. *Consider the execution of Algorithm 1 on the graph in Example 1. In the first iteration, the set of all successors of vertex 0 are enumerated by image computation, leading to $\mathbf{1}_{V_0} \vee \mathbf{2}_{V_0}$, which corresponds to set $\{1,2\}$. In the second iteration, the successors of set $\{1,2\}$ are enumerated, which gives $\{3,4\}$. Note that unlike conventional algorithms where successors of each vertex are enumerated one at a time, the successors of $\{1,2\}$ are enumerated collectively. This process continues and it takes 3 symbolic steps to enumerate 5 reachable vertices.* □

ALGORITHM 1.

```
reachable = func(                 1
  g : V0 × V1,                    2
  ) : V0 {                        3
  var r, i : V0;                  4
                                  5
  i = 0_V0;                       6
  do {                            7
    i = (∃V0.[g ∧ i])|V1→V0;      8
    r = r ∨ i;                    9
    i = i ∧ ¬r;                   10
    } while( i ≠ ∅ );             11
  return r ;                      12
  }                               13
```

ALGORITHM 2.

```
closure = func(                   14
  g : V0 × V1,                    15
  ) : V2 × V0 {                   16
  var r, i : V2 × V0;             17
                                  18
  i = equal(V2, V0);              19
  do {                            20
    i = (∃V0.[g ∧ i])|V1→V0;      21
    r = r ∨ i;                    22
    i = i ∧ ¬r;                   23
    } while( i ≠ ∅ );             24
  return r ;                      25
  }                               26
```

The application of this powerful concept has so far been primarily limited to the model checking of finite state machines. To generalize it as a strategy to solve a wider scope of combinatorial problems, we first note that any structured information is in essence a *relation*, or a Cartesian product of different sets. As such, the conventional data structures used to capture real world data can be canonically represented by BDDs, after the proper construction of discrete domains and their encodings. For better readability, all expressions in our presented algorithms are *typed* by relations, or products of named Boolean spaces (a set can be considered as a degenerate relation), which in practice are simply BDDs.

In addition, we employ the key concept of superposition, which is responsible for the theoretical efficiency of quantum computers. In quantum computing [20], different problems are encoded and superposed into a single physical entity, such as light, and processed collectively by quantum devices. It is a recurring pattern in practice that a large problem is solved by decomposing it into many instances of smaller problems: all problem instances could be solved by the same method, however the number of instances could be exponentially many. We propose the use of BDDs as the "physical entity" for superposition. We first represent each problem instance symbolically, for example, capturing a graph problem by the corresponding symbolic graph. We then introduce a domain of problem instances, as such all instances could be combined into a single Boolean function. In the end, we devise symbolic algorithms to solve the superposed problems collectively.

We illustrate this strategy by the graph transitive closure problem, which can be considered as many instances of graph reachability problems, solved by Algorithm 1. Here the number of problem instances coincides with the number of vertices, since in essence we try to find the set of reachable vertices for each vertex. It is instructive to see how Algorithm 1 is slightly modified into Algorithm 2. Here we use the Boolean space V_2 to encode the problem instances. As such, the result we are seeking, the set of reachable vertices with respect to each vertex, is superposed into a relation in $V_2 \times V_0$, which can also be visualized as a graph. This relation is initialized by a special relation in Line 19, which can be considered as a graph with each vertex pointing to itself. The superposed image computation operator in Line 21, is hardly changed from the original one: we simply leave the instance information untouched.

EXAMPLE 3. *Consider finding the transitive closure of the graph in Example 1 using the method in Algorithm 2. Initially, i represents a graph* $\{\langle 0,0\rangle, \langle 1,1\rangle, \langle 2,2\rangle, \langle 3,3\rangle, \langle 4,4\rangle, \langle 5,5\rangle\}$. *In the first iteration, all depth-one closure edges are enumerated using the superposed image computation operator, which gives* $\{\langle 0,1\rangle, \langle 0,2\rangle, \langle 1,3\rangle, \langle 2,4\rangle, \langle 3,5\rangle, \langle 4,5\rangle\}$, *corresponding to the original graph. In the second iteration, all depth-two closure edges are enumerated, which gives* $\{\langle 0,3\rangle, \langle 0,4\rangle, \langle 1,5\rangle, \langle 2,5\rangle\}$. *Finally, all depth-three closure edges are enumerated, which gives* $\{\langle 0,5\rangle\}$. *It is striking to note that comparing to Algorithm 1, this algorithm takes exactly the same number of steps to enumerate 11 closure edges.* □

The potential efficiency of superposed symbolic computation comes from the fact that different problem instances may share similar problem structure, which can be automatically identified by BDDs. As such, computation can be cached and shared across different instances at a large scale. As pointed out by [28], this follows the same principle of dynamic programming.

3. PROGRAM FORMULATION

To apply the superposed symbolic computation paradigm to the pointer analysis problem, we need to model the relevant program information, as well as the information to solve for, in symbolic relations. We first identify the relevant domains from which the useful relations can be constructed.

Table 1: Summary of the domains and spaces.

Domains	Spaces	Description
B	B_0, B_1, B_2, B_3	Block
F	F_0	Field
P	P_0, P_1	Path
R	R_0, R_1, R_3	Rank

In Table 3, we define several discrete domains, each of which is assigned with one or more Boolean spaces. Domain B corresponds to the set of all memory blocks, which includes either named blocks, such as globals, parameters, locals, or anonymous blocks, such as heap allocated objects, return values at the caller (denoted as rcallee), return values at the caller (denoted as rcaller), and ϕ blocks to handle intermediate representations in the static single assignment form (SSA) [7]. Domain F corresponds to the set of record fields, each of which is identified by a number representing the offset of the field relative to the origin. Domain P represents the set of all memory access paths through record fields in F. Domain R, which carries the set of program points, deserves special attention. For FICI analysis, the domain is empty. For FICS analysis, the domain corresponds to set of different contexts, each of which corresponds to a unique calling path in the call graph. For FSCS analysis, the domain corresponds to the set of all program points under different call paths. Note that each program statement with a side effect, or altering the program state, contributes one program point under one context. The major challenge of pointer analysis comes from the size of R. The structure of ranks will be discussed in more detail in Section 5.

The goal of pointer analysis is to compute the program state relevant to pointers, which can be represented as a graph, called the point-to graph. The vertex set of the point-to graph is simply B. An edge $\langle u, v\rangle$ in the point-to graph indicates that under the program state, u may point to v. In our analysis, we superpose program states at all program points together, which leads to the **superposed program state** $s : R_0 \times B_0 \times F_0 \times B_1$, stating that $\forall \langle r, u, f, v\rangle \in s$, memory block u may point to memory block v through field f at program point r.

EXAMPLE 4. *Consider the C program in Figure 1 (a), which is modified from [16]. Here* $B = \{g, a, h1, h2, p, q, r, t, f, h, rcallee\}$, *where* $g, a, h1, h2$ *are global blocks, and* p, q, r, t, f, h *are local blocks. The program states computed for the end of the program, are shown in Figure 1 (b), (c), (d) for different analyses respectively. It can be observed that FSCS leads to the smallest point-to graph, or the most precise result.* □

Unfortunately, the program state information cannot be directly derived from the program source. The difficulty comes from indirect memory dereferences. For example, an assignment in the form `p->f1->f2 = q->f3->f4` does not give any direct hint on its effect of program state, since the value of `p` and `q` are not known. To capture assignments with indirect dereferences, we model each assignment as a tuple in $(B_3 \times P_0) \times F_0 \times (B_4 \times P_1)$. By superposition, we can thus derive the **superposed transfer function** $t : R_0 \times (B_3 \times P_0) \times F_0 \times (B_4 \times P_1)$ from the source program, taking into account of both explicit assignments, as well as implicit assignments due to parameter passing and return value.

EXAMPLE 5. *Consider an assignment* `p->f1->f2 = q->f3->f4` *at program point r. Then* $\langle r, lr, lp, f, rr, rp\rangle \in t$, *where lr corresponds to block* `p`; *lp corresponds to the access path* `->f1`, *f corresponds to the field* `f2`, *rr corresponds to the block q, and rp corresponds to the access path* `f3->f4`. □

Each access path in P is a unique, acyclic sequence of field accesses. We use the path relation $l : P_0 \times F_0 \times P_1$ to precisely define the relation between different paths. It is important to note that an access path within a loop may lead to infinite sequences. In the past, this has been handled by limiting the length of the sequence, called K-limiting. Our method relies on the use of SSA representation: since each ϕ instruction becomes a distinct memory block, it effectively breaks the access cycle.

A central task of flow sensitive pointer analysis is to differentiate program state by statement execution order, and eliminate spurious state by performing strong update. For the former, the control flow graph for each procedure is needed. A control flow graph defines the precedence relation between program points. We can therefore represent the whole program **control flow graph** symbolically as $c : R_0 \times R_1$. For the latter, we can capture all known assignments to scalars as the kill information $k : B_0 \times R_1$.

The symbolic pointer analysis problem can then be formulated as the following.

PROBLEM 1. *Given the superposed transfer function* $t : R_0 \times (B_2 \times P_0) \times F_0 \times (B_3 \times P_1)$, *the access path relation* $l : P_0 \times F_0 \times P_1$, *the control flow graph* $c : R_0 \times R_1$, *the kill information* $k : B_0 \times R_1$, *find the superposed program state* $s : R_0 \times B_0 \times F_0 \times B_1$.

4. ANALYSIS FRAMEWORK

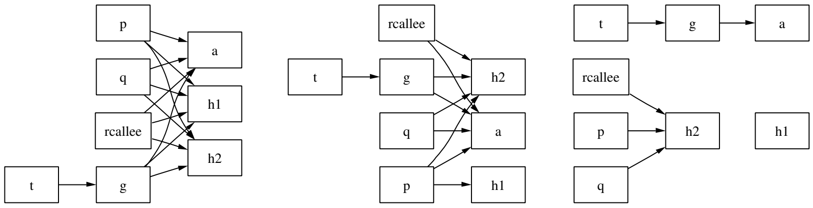
While both the control flow graph c and the kill information k are available and they are sufficient to help derive flow sensitive program states, we opt not to use them directly in our solver. The primary reason is that any solver for pointer analysis, including ours, may need many iterations to converge into a fixed point solution. In each such iteration, the control flow graph has to be traversed for the entire program, which by itself is a very expensive operation.

We instead propose to use the **superposed reaching dataflow** $f : B_0 \times (R_0 \times R_1)$, capturing the set of all reachable program points in R_1 for each new state generated at program point in R_0 for a memory block in B_0. Note that f, superposed with respect to each program point and each memory block, contains a rich set of information: It is interprocedural – the relation can be defined between any pairs of program points across different procedures. It is context sensitive – we believe it is important for flow sensitive analysis to be context sensitive as well, otherwise the correct statement ordering will quickly be corrupted. It can capture strong update – states generated at one program point x for a particular block b may not reach another program point y, even though it is reachable through the control flow graph, as long as it can be proved the state related to the block b will be killed on every path from x to y. We describe how to compute f in Section 5.

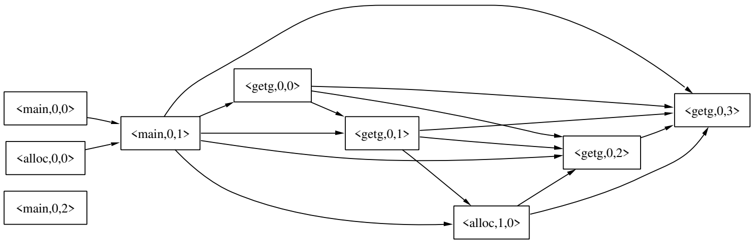
Given the superposed program state, we can derive the **superposed state query** $q : R_0 \times (B_3 \times P_0) \times B_0$, stating that $\forall \langle r, u, p, v\rangle \in q$, the access of block u via access path p may give the value of v at program point r. We can then compute the program state information by solving the following recurrence equations.

```
char    *g, a, h1, h2;
void main() {
   char   *p, *q;
S0: alloc( &p, &h1 );
S1: p = getg( &q );
S2: g = &a;
   }

char* getg( char** r ) {
   char  **t;
S0: t = &g;
   if( g == NULL )
S1:      alloc( t, &h2 );
S2: *r = *t;
S3: return *r;
   }

void alloc( char** f, char*h ) {
S0: *f = h;
   }
```

(a) C source code



(b) FICI result (c) FICS result (d) FSCS result



(e) Reaching dataflow with respect to block g.

Figure 1: A walk-through example.

$$s \;=\; apply(s, q, t, f) \qquad (1)$$

$$q \;=\; query(s, q, l) \qquad (2)$$

Given the current state query q, or the currently known memory dereference values, (1) computes the new state by applying the transfer function t, in other words, finding the effects of the assignments it captures. Given the current program state s, (2) derives the state query values for memory blocks via relevant access paths. Note that here each step is computed with the superposed symbolic computation paradigm: For (1), in contrast to the conventional algorithms where assignments and their effects on other program points are evaluated one at a time, we compute them collectively with Algorithm 3. For (2), in contrast to the conventional algorithm where the memory dereference values are found one at a time, and one program state at a time, we compute it collectively with Algorithm 4.

As shown in Lines 10–11 of Algorithm 3, superposed transfer function application can be accomplished by first symbolically composing t and q. In other words, for each occurrence of memory access $\langle b, p\rangle$ in t, we substitute it with the corresponding values defined in q. For flow sensitive analysis, the transfer function application only yields new states at the program points where they are generated. The new state needs to be propagated to all other reachable program points. As shown in Line 12 of Algorithm 3, this can be accomplished by composing the state generated earlier, and the precomputed reaching dataflow f.

As shown in Algorithm 4, superposed state query can be accomplished by a procedure similar to the transitive closure algorithm shown in Algorithm 2. The superposed image computation operator in Line 22 effectively implements the following inference rule: if $\langle r, b_3, p_0, b_0\rangle \in q$, if $\langle r, b_0, f, b_1\rangle \in s$, and if $\langle p_0, f, p_1\rangle \in l$, then $\langle r, b_3, p_1, b_1\rangle \in q$. This inference rule is applied repeatedly until a fixed point is reached.

5. SYMBOLIC FLOW ANALYSIS

For FSCS analysis, the distinguishable program points consist of three components: the procedure in which it is defined, the statement (with side-effect), or site at which it is defined, and the calling context under which it is defined. Let the set of all procedures be M, the set of all sites be S, and the set of all contexts be C, then $R = M \times C \times S$.

EXAMPLE 6. *Consider the program in Example 4. We have* $M = \{main, getg, alloc\}$*. We also have* $C = \{0, 1\}$*: procedures main and getg have only context 0, whereas procedure alloc has context 0, which corresponds to the calling path* $main \rightarrow alloc$*, and context 1, which corresponds to the calling path* $main \rightarrow getg \rightarrow alloc$*. For sites, we have* $S = \{0, 1, 2, 3\}$. □

ALGORITHM 3. *Superposed Transfer Function Application.*

1 *apply* = **func**(
2   $s : R_0 \times B_0 \times F_0 \times B_1$,
3   $q : R_0 \times (B_2 \times P_0) \times B_0$,
4   $t : R_0 \times (B_2 \times P_0) \times F_0 \times (B_3 \times P_1)$,
5   $f : R_0 \times B_0 \times R_1$
6   $) : R_0 \times B_0 \times F_0 \times B_1$ {
7   **var** $q' : R_0 \times (B_3 \times P_1) \times B_1$;
8   **var** $g, r : R_0 \times B_0 \times F_0 \times B_1$;
9
10   $q' = q|_{B_2 \rightarrow B_3, P_0 \rightarrow P_1, B_0 \rightarrow B_1}$;
11   $g = \exists B_2, P_0, B_3, P_1.[t \wedge q \wedge q']$;
12   $r = (\exists R_0.[g \wedge f])|_{R_1 \mapsto R_0}$;
13   **return** $s \vee g \vee r$;
14   }

ALGORITHM 4. *Superposed State Query.*

15 *query* = **func**(
16   $s : R_0 \times B_0 \times F_0 \times B_1$,
17   $q : R_0 \times (B_2 \times P_0) \times B_0$,
18   $l : P_0 \times F_0 \times P_1$,
19   $) : R_0 \times (B_2 \times P_0) \times B_0$ {
20   **var** $p : R_0 \times (B_2 \times P_0) \times B_0$;
21   **do** {
22     $p = (\exists B_0, F_0, P_0.[q \wedge s \wedge l])|_{B_1 \rightarrow B_0, P_1 \rightarrow P_0}$;
23     $q = q \vee p$;
24     $p = p \wedge \neg q$;
25   } **while**($p \neq \emptyset$) ;
26   **return** q ;
27   }

Both the intraprocedural control flow graph and the intraprocedural kill information can be trivially computed by the frontend. The context information, called the *symbolic invocation graph*, can be efficiently constructed in polynomial time by the method reported in [29]. Combining both information, it is easy to obtain the interprocedural control flow graph c and kill information k.

We now consider how to compute the reaching dataflow. While it is seemingly similar to the reaching definition analysis, the required information is extremely difficult to compute in the traditional dataflow framework. The difficulty comes from the fact that it is impossible to predict the potential blocks to be assigned at a program point where the target address is indirectly dereferenced (otherwise there would be no need for pointer analysis). A naive algorithm might have to enumerate all memory blocks, and perform dataflow analysis for each of them.

With the superposed symbolic computation paradigm, we can compute the required information collectively using a modified transitive closure algorithm in Algorithm 2. The modification is needed to take care of strong update due to the kill information k. We can accomplish this by filtering the new dataflow discovered in each iteration by the kill information. However, this procedure is not efficient for the following reason: For a large program, the depth of the control flow graph, d, which roughly corresponds to the length of the longest execution trace, can be extremely long. The procedure devised in Algorithm 2 expands one level of at a time. Therefore, the number of symbolic steps needed to complete the algorithm is exactly the depth the control flow graph. This can be unacceptably slow.

We instead devise a symbolic algorithm in Algorithm 5. At each iteration, we perform the superposed image computation against the current closure, instead of the original graph (Line 37). This way, if the depth of the graph already explored is k, then one iteration later the depth of the graph explored will be $2k$. As a result, we need at most $\lceil log_2 d \rceil$ number of symbolic steps to complete. This strategy proves to be much more effective for long programs.

ALGORITHM 5. *Superposed reaching dataflow analysis.*

```
28  flow = func(
29     c : R_0 × R1, k : B_0 × R_1
30     ) : B_0 × (R_0 × R_1) {
31     var i, f : B_0 × (R_0 × R_1);
32     var m : B_0 × (R_1 × R_2);
33
34     f = i = c;
35     do {
36        m = i|_{R_0→R_1, R_1→R_2};          // mirroring
37        i = (∃R_1.[m ∧ f])|_{R_2→R_1};     // superposed image computation
38        i = i ∧ ¬k;                          // filtering with k
39        i = i ∧ ¬f;                          // keep only the change
40        f = f ∨ i;
41        } while( i ≠ ∅ ) ;
42     return f ;
43     }
```

EXAMPLE 7. *The computed reaching dataflow with respect to the block g in the program in Example 4 is visualized in Figure 1 (e). An edge $\langle u, v\rangle$ in this graph indicates that if block g is assigned at program point u, then it can reach program point v. It is interesting to note that the program point $\langle main, 0, 2\rangle$ is dangling – no assignment to g can be propagated to here, precisely because g is assigned at this program point, or g is killed.* □

In practice, we refine Algorithm 5 with more engineering considerations to improve performance. One effective method is to first compute the intraprocedural reaching dataflow, then the interprocedural reach dataflow. Another effective method concerns the encoding of the rank. Efforts can be made on encoding that leads to smaller BDD sizes during the flow computation.

6. EXPERIMENTAL RESULTS

Our symbolic pointer analysis tool is implemented in C, and makes use of a compiler infrastructure to translate programs in C/Java/Verilog, into an intermediate representation (IR). The analysis is performed in several passes. In the *setup pass*, the IR is traversed to generate the whole program call graph, and the control flow graph of each procedure. The maximal strongly connected components (SCC) of the call graph, as results of recursive procedure calls, will also be identified. The procedures in the same SCCs are collapsed into a common node. In the *intra-procedural analysis pass*, The IR is traversed again, following the topological order of the now acyclic call graph. It is during this step that all relations needed for the analysis, are constructed with BDDs.

Table 2: Benchmark characteristics and BDD sizes.

Benchmark	#lines	#contexts	intra flow	inter flow	state	query
164.gzip	9074	3530	15271	166028	4370	8029
175.vpr	16984	179905	31008	418916	3719	15302
300.twolf	19756	5538	96231	1076776	2570	16338
255.vortex	67211	9.20E+10	58121	2307623	9687	102476
176.gcc	222183	1.18E+10	237681	13233784	315703	425278

Our empirical evaluation mainly concerns the scalability of the following passes, which work exclusively in the Boolean domain. In other words, they manipulate BDDs without referencing any information in the IR. They include an optional *flow analysis* pass, if the requested analysis is flow sensitive, and a *solving pass* where the program state is computed. We use Somenzi's publicly available CUDD package [23] for BDD implementation. The transfer functions for the C library functions are precomputed and applied as necessary.

6.1 Benchmark Characteristics

We conducted experiments on a wide range of benchmarks available to us. The experiment was performed on a Sun Blade 150 workstation with 550 MHz CPU and 128MB RAM, running on Solaris 8 Operating System.

In this paper, we report results only for large benchmarks in SPEC2000. The characteristics of the reported benchmarks in this paper are shown in the first two columns of Table 6.2. They ranges from 9K lines of C code for `gzip` to 200K lines of C code for `gcc`. The largest context size, is recorded for the `vortex` benchmark with $9E10$, which is close to 100 billion.

6.2 Space

The space usage is best illustrated by the BDD sizes, or the numbers of BDD nodes, of different relations. Column 4-7 list BDD sizes of four relations: the intraprocedural flow, the interprocedural flow, state and query, all collected from our FSCS analyzer. As can be expected, the BDD sizes in general grow with the complexity of the benchmark. In particular, the sizes of the interprocedural flow information do dominate. This has caused noticeable slow down of analysis speed as compared to flow insensitive analysis, suggesting future improvement might profit from reducing interprocedural flow size.

6.3 Runtime

With the common analysis framework described earlier, we report comparative results for FICI, FICS and FSCS analyses. All analyses are also field sensitive, in other words, all different elements of a record is distinguished. Figure 2 gives the combined runtime of the flow analysis pass and the solver pass, all in seconds.

We draw several observations from the runtime result. First, our FICS analyzer is extremely fast. In fact, all benchmarks can be analyzed in seconds. Most notably is the `gcc` benchmark, for which our solver completes in 10 seconds. This is in contrast with the state-of-the-art solvers [11, 10, 5], which completes in several minutes. Second, our FSCS analyzer is significantly slower. However, other than `gcc`, all benchmarks complete under one minute. The `gcc` benchmark completes in 453 seconds. To the best of our knowledge, it is the first time that FSCS analysis result on benchmarks of these sizes have been reported.

6.4 Precision

It has been established by many previous works that the context sensitivity and flow sensitivity contribute to the analysis precision. However, a comprehensive comparison has rarely been reported. In this section, we show the precision results of all types of analysis. Following the convention

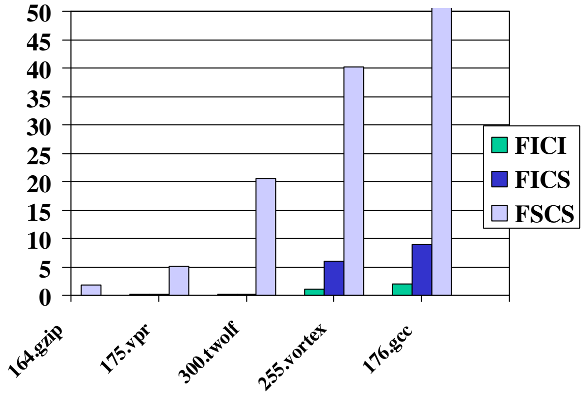


Figure 2: Run time for FICI, FICS, and FSCS analyses.

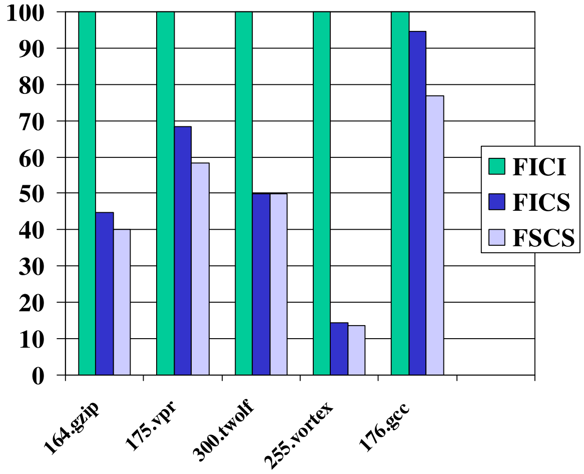


Figure 3: Precisions for FICI, FICS, and FSCS analyses.

of [9], the metric we use is the average point-to size of indirect references at each program point. For the purpose of comparison, the results are normalized against the point-to sizes reported by the FICI analysis, the most commonly used, but the least precise type of analysis. Consistent with results on smaller benchmarks published previously, the exact precision improvement varies from benchmark to benchmark, but in general flow sensitive analysis consistently improves precision.

7. CONCLUSION

In this paper, we demonstrate a technique of pointer analysis that leads to a new milestone of scalability for FSCS analysis. Based on our study, we conclude that the key strategy proposed in this paper, namely the pre-computation of reaching dataflow information, together with our symbolic pointer analysis framework, provides a valid path towards a scalable solution to flow and context sensitive pointer analysis. We believe this strategy can be useful for program analysis problems in general.

8. REFERENCES

[1] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Point-to analysis using BDD. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, June 2003.

[2] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computer*, C-35(8):677–691, August 1986.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, Washington, DC, 1990.

[4] V. T. Chakaravarthy. New results on the computability and complexity of points-to analysis. In *Proceedings of Priciple of Programming Languages (POPL'03)*, January 2003.

[5] B.-C. Cheng and W.-M. W. Hwu. Modular interprocedural pointer analysis using access paths: Design implementation and evaluation. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–69, Vancouver, British Columbia, Canada, June 2000.

[6] O. Coudert, C. Berthet, and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 126–129, November 1990.

[7] 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 Trans. Prog. Lang. Syst.*, 13(4), October 1991.

[8] S. A. Edwards. The challenges of hardware synthesis from C-like languages. In *Proceedings of the International Workshop of Logic and Synthesis (IWLS)*, June 2004.

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

[10] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 253–263, Vancouver, British Columbia, Canada, June 2000.

[11] J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Proceedings of Static Analysis Symposium*, pages 175–198, June 2000.

[12] D. Gajski, J. Zhu, D. Doemer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, Boston, March 2000.

[13] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, 2001.

[14] M. Hind. Pointer analysis: Haven't we solved this problem yet. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, June 2001.

[15] O. Lhoták and L. Hendren. Jedd: A BDD-based relational extension of Java. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.

[16] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *Proceedings of Static Analysis Symposium*, pages 279–298, 2001.

[17] S. Liao, S. Tjiang, and R. Gupta. An efficient implementation of reactivity for modeling hardware in the Scenic design environment. In *Proceeding of the 34th Design Automation Conference*, 1997.

[18] G. D. Micheli. Hardware synthesis from C/C++ models. In *Proceedings of the Design Automation and Test Conference in Europe*, March 1999.

[19] P. Panda, L. Semeria, and G. D. Micheli. Cache-efficient memory layout of aggregate data structures. In *Proceedings of the International Symposium on System Synthesis*, September 2001.

[20] E. Rieffel and W. Polak. An introduction to quantum computing for non-physicists. *ACM Computing Surveys*, (32(2)), September 1997.

[21] L. Semeria and G. D. Micheli. Resolution, optimization, and encoding of pointer variables for the behavioral synthesis from C. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, February 2001.

[22] L. Semeria, K. Sata, and G. D. Micheli. Synthesis of hardware models in C with pointers and complex data structures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, February 2001.

[23] F. Somenzi. CUDD: Binary decision diagram package release. `http://vlsi.Colorado.EDU/~fabio/CUDD/cuddIntro.html`, 1998.

[24] *SystemC Web Site*. `http://www.systemc.org`.

[25] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.

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

[27] J. Zhu. Static memory allocation by pointer analysis and coloring. In *Design Automation and Test in Europe*, March 2001.

[28] J. Zhu. Symbolic pointer analysis. In *Proceedings of the International Conference in Computer Aided Design*, San Jose, November 2002.

[29] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.

[30] J. Zhu, R. Doemer, and D. Gajski. Syntax and semantics of SpecC+ language. In *Proceedings of the Ninth Workshop on Synthesis and System Integration of Mixed Technologies*, Japan, December 1997.