Color Permutation: an Iterative Algorithm for Memory Packing

Jianwen Zhu

Technical Report 01-04-01
June 2001

10 King's College Road
Edward S. Rogers Sr.
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, M5S 1L1, Canada

jzhu@eecg.toronto.edu

Abstract

One of the bottlenecks in the recent movement of hardware synthesis from behavioral C programs is the difficulty in reasoning about runtime pointer values at compile time. The pointer analysis problem has been investigated in the compiler community for two decades, which has yielded efficient, polynomial time algorithms for context-insensitive analysis. However, at the accuracy level for which hardware synthesis is desired, namely context and flow sensitive analysis, the time and space complexity of the best algorithms reported grow exponentially with program size. In this paper, we present our first step towards a new analysis technology which potentially leads to almost-linear time complexity and sub-linear space complexity algorithm even for the most accurate analysis. The key idea that contributes to this efficiency is to implicitly encode the pointer-to relation in the Boolean domain, thereby capturing the procedure transfer function completely, compactly and canonically. This represents a wide departure from the traditional techniques, all of which explicitly capture pointer-to relation using variations of point-to graph, which have to be re-evaluated for different calling contexts. Experiments for our first flow-insensitive algorithm on common benchmarks show promising result.

Contents

1 Introduction

Today's system-on-chips are heterogeneous: while the majority of the functionality can be handled by software running on on-chip microprocessors, often times it is the set of on-chip dedicated cores that defines the performance, and therefore the competitiveness of the chip. The design of such cores, however, is known to be time consuming under the current commercial design methodology. The effort to automatically synthesize hardware from algorithm-level specification, known as behavioral (high-level) synthesis, has rarely gone beyond programs with simplistic data structures, such as scalars and arrays. This constraint seriously limits the applicability of behavioral synthesis technology, since any interesting applications will involve the use of complex data structures. This problem has become more apparent in the recent movement of behavioral synthesis from C/C++ programs, which encourage the use of pointers.

The presence of data pointers imposes challenges for temporal optimization, since flow-dependency between operations have to be conservatively approximated whenever pointer dereferences are involved, as well as spatial optimization, since the life times of memory blocks can not be accurately determined, and therefore their space cannot be aggressively shared. The presence of function pointer makes it difficult for whole-program analysis and therefore inter-procedural optimization, since even the call graph cannot be readily constructed. The degree at which these aggressive optimization can be carried out for pointer-intensive programs largely depends on the how well one can approximate the runtime pointer values at compile-time, which unfortunately, is an undecidable problem.

Luckily, the synthesis research community is not alone for the so-called *pointer analysis* problem, or in a slightly different formulation, the *alias analysis* problem. In fact, the optimizing compiler and programming language community has attacked this problem for two decades and a rich set of results have been yielded in the literature. The reported analysis algorithms vary with different accuracy speed tradeoff and can be categorized according to two criteria: *flow sensitivity* and *context sensitivity*. A flow-insensitive (FI) algorithm ignores the order of statements when it calculates pointer information, whereas a flow-sensitive (FS) algorithm takes control flow within a procedure into account. A context-insensitive (CI) algorithm does not distinguish the different calling contexts of a procedure, whereas a context-sensitive (CS) does.

Fast polynomial algorithms, such as derivatives of Steensgaard's [15] and Andersen's [1], have been developed for context-insensitive analysis. However, the accuracy of such analysis, especially Steensgaard's, is not acceptable for hardware synthesis aiming for aggressive optimization. On the other hand, the context-sensitive analysis algorithms developed in the literature suffer from a worst-case exponential time complexity. The fundamental reasons for such complexity is three fold: First, these algorithms cannot obtain the *complete transfer function* (CTF) of a procedure, in other words, summarize the effect of procedure on program state with respect to arbitrary initial state. Therefore, each procedure has to be re-analyzed for each calling context. This problem has been alleviated by the landmark work of Wilson and Lam [16], who developed the concept of *partial transfer function*. Nevertheless, this in-efficiency is not completely eliminated. Second, the cost of propagating the effect of callee to the caller is high, since the point-to relation of the caller has to be duplicated at each call site. This cost remains to be high even with the sparse representation [17]. Third, the data structure that these algorithms use to capture program state are inherently redundant, thereby causing in-efficiency in space. For flow-sensitive analysis, where program state for each program point has to be kept, the space in-efficiency will eventually limit the size of the program one can analyze.

In this paper, we describe an analysis technology, called symbolic pointer analysis, which starts a new direction in combating the three sources of in-efficiencies. The contributions of this paper is as follows: First, we map each program memory location of interest into a cube in a Boolean space, called the *domain space*, and a cube in another Boolean space, called the *range space*. This allows us to *implicitly* represent sets of memory locations using Boolean functions, thereby providing space efficiency and potentially speed efficiency. Second, instead of using set of alias pairs, or point-to graph, we propose a new approach to represent the program state using a Boolean function in the combined domain and range space. This allows us to perform state query using implicit image computation originally developed for reachability analysis in formal verification [3, 4], rather than the explicit graph traversal. Third, we introduce the *initial state blocks*, a concept similar but different from Wilson's extended parameters, as well as the concept of *scalar predicate*, which is necessary if accuracy due to "strong update" is not to be compromised. These two concepts allow us to capture the transfer function of a procedure *completely* as a Boolean function, which was otherwise impossible. Forth, we engineer a CSFI analysis algorithm based on the above concepts using Bryant's Binary Decision Diagram (BDD) [2] and illustrate why the compactness and canonicality of BDD can be exploited for efficiency.

The rest of the paper includes a detailed description of the proposed representation (Section **??**) and the fundamental algorithms (Section 4) under certain simplifying assumptions, which will later be relaxed. The engineering side of the algorithm is discussed in Section **??**. Lastly, while we are not able to perform a comprehensive and thorough empirical comparison with other works at this point due to the time and resource constraint, we present our preliminary result on standard benchmarks, which promisingly confirmed the expected efficiency.

In the text that follows, we use the *formal algorithm notation* (FAN) to state definitions and describe algorithms. Unlike pseudo-code based algorithm description, FAN relies on a type system, where each type is represented by a set, to present the algorithm in a formal, precise manner. For example, we use the notation $\langle\rangle^A$ to represent the power set of A, therefore any value of type $\langle\rangle^A$ must be a set of values of type A. Similarly, we use the notation $[\ ]^A$ to represent the set of all sequences over elements of A, therefore any value of type $[\ ]^A$ represents a sequence of values of type A. Readers are expected to find this notation very similar to any strongly-typed programming languages and hence straightforward to be translated into implementation, yet abstract enough to allow concise presentation.

2 Related Work

To quote a recent excellent survey by Hind [7], over seventy-five papers and nine PhD thesis haven be published on pointer analysis. So here we only review important milestones and recent works that are relevant to ours.

In the category of FICI pointer analysis, Steensgaard's work [15] stands out as the first equality-based method, which treats assignment as bidirectional and use a union-find data structure. His analyzer is extremely fast and has analyzed million lines of industrial code, however, the precision of equality-based approach degrades very fast in general, even with later improvement [18]. Andersen's [1] popular subset-based improves precision by treating assignment as a unidirectional flow of values, yet still run's in polynomial time.

Our algorithm primarily works in the research space of accurate FS/CS analysis. Wilson and Lam [17] proposed the concept of partial transfer function in an effort to reduce the number of times a procedure has to be analyzed. Along the same line, recently Liang and Harrold [8] has proposed to use the local and global auxiliary blocks, similar to our initial state block. However, their model cannot extend beyond FS analysis since it is impossible to detect strong updates.

The application of pointer analysis algorithm for behavioral synthesis is emerging. Semeria and De Micheli have researched on how to encode C pointers in hardware [11, 12, 13]. By a combination of runtime test and compile time analysis, they can produce efficient, customized pointer implementation. Panda et. al. [10] applied pointer analysis to optimally determine array memory layout for cache performance. Zhu [19] applied pointer analysis to the problem of memory block minimization.

3 Problem Formulation

In the text that follows, we use the *formal algorithm notation* (FAN) to state definitions and describe algorithms. Unlike pseudo-code based algorithm description, FAN relies on a type system, where each type is represented by a set, to present the algorithm in a formal, precise manner. Readers are expected to find this notation very similar to any strongly-typed programming languages and hence straightforward to be translated into implementation, yet abstract enough to allow concise presentation.

The input of the memory allocation problem is a set of memory blocks, as defined in Definition 1, as well as a *conflict* relation between these blocks, which indicate whether or not that any pair of the memory blocks can be shared, or having an overlapping memory address space. The memory block is characterized by its size, which can be any natural numbers. The conflict relation is derived by discovering the "life time" of the memory blocks using dataflow analysis, which is not the subject of this paper.

Definition 1 *A* **memory block** v : *Block is a member of*

$$
\begin{array}{ll}
\mathit{Block} = \textbf{tuple}\ \{ & \\
\quad \mathit{size} \qquad : \mathcal{N} & \\
\quad \} &
\end{array}
$$

An allocation, as defined by Definition 2 is then the assignment of address location, represented by an integer, to each of the memory block, such that the conflict relation is honored.

Definition 2 *Given a set of memory blocks*[1] V :

[1] Here we use the notation $\langle\rangle^A$ to represent a power set of A, and the notation $[\ ]^A$ to represent the set of all sequences over elements of A.

$\langle\rangle^{Block}$, *and a conflict relation* $E : \langle\rangle^{V\times V}$ *between the memory blocks, a* **memory allocation**, *or a* **memory packing**, *is a mapping* $\mathcal{A} : V \mapsto \mathcal{N}$, *such that* $\langle u, v\rangle \in E \rightarrow [\mathcal{A}(u), \mathcal{A}(u) + u.size] \cap [\mathcal{A}(v), \mathcal{A}(v) + v.size] = \oslash$.

Obviously, one allocation can be better or worse than another, depending on whether or not the total memory size occupied by all memory blocks is smaller. According to Definition 3, the allocation that results in the smallest total memory size is the optimal allocation.

Definition 3 *For an allocation* $\mathcal{A} : \mathcal{V} \mapsto \mathcal{N}$, *its memory size* $\|\mathcal{A}\|$ *is defined to be* $max_{v\in V}\mathcal{A}(v) + v.size$. *An allocation* $\mathcal{A}_0$ *is said to be optimal if* $\forall\mathcal{A}, \|\mathcal{A}\| \geq \|\mathcal{A}_0\|$.

4 Symbolic Pointer Analysis Algorithm

4.1 Program Modeling

We now describe our pointer analysis algorithm. In order to focus on the fundamentals, rather than the implementation details, we made a number of simplifying assumptions for the ease of presentation.

We assume a program is preprocessed so that it is available in the form defined in Definition 4, which contains a set of blocks. The blocks can be globals, locals, parameters, initial blocks and procedures, as discussed in Section **??**. All blocks are treated as scalars. The additional information for procedure blocks can be obtained from *procs*. The program also contains the Boolean environment necessary to define symbolic point-to relation, as discussed in Section **??**. Since a block correspond to a cube in the Boolean space, a Boolean function can be used to represent a set of blocks. The utility function *spaEnumerate* can be used to enumerate all blocks corresponding to a Boolean function.

Definition 4 *A program is a member of*

```
Spa = tuple {
    blks        : ⟨⟩^SpaBlock;
    procs       : SpaBlock ↦ SpaProc;
    domainVars  : [ ]^SpaDD;
    rangeVars   : [ ]^SpaDD;
    cache       : SpaDD × SpaDD × {bddAnd, bddOr, ...}↦ SpaDD;
    }
spaEnumerate = func( spa : Spa, l : SpaDD ) : ⟨⟩^SpaBlock;
```

A procedure contains a set of parameter, local, heap allocated blocks, and a series of instructions. Note that the formals contains a sequence of initial state blocks which are originally implicit in the source program.

Definition 5 *A procedure is a member of*

```
SpaProc = tuple {
    formals  : [ ]^SpaBlock;
    params   : ⟨⟩^SpaBlock;
    locals   : ⟨⟩^SpaBlock;
    heaps    : ⟨⟩^SpaBlock;
    instrns  : ⟨⟩^SpaInstrn;
    }
```

As shown in Figure 6, we consider only two types of instructions, assignment instruction (store), call instruction (call), and ignore others. The destination and sources of the instruction are in the form of access path.

Definition 6 *An instruction is a member of*

```
SpaInstrn = tuple {
    opcode  : {store, call, others};
    dst     : SpaAccessPath;
    srcs    : [ ]^SpaAcessPath;
    }
```

We assume an access path can be characterized by a block, which indicates the root of access, and level, which indicates the number of dereferences performed on the block. The C expression &b corresponds to level -1, b corresponds to level 0, and ***b corresponds to level 3.

Definition 7 *An access path is a member of*

```
SpaAccessPath = tuple {
    blk    : SpaBlock;
    level  : Z;
    }
```

We assume that the procedure does not contain recursive calls. Also we do not distinguish memory allocated at the same site under different calling contexts.

The program in Example **??** (a) is preprocessed into Example **??** (b). Note that an important difference from its source is that the formals refers to the initial state blocks, which are not explicit in the source code before. The corresponding actuals at the callsite, are also made explicit.

4.2 Pointer Analysis Algorithm

Algorithm 1 computes the Boolean CTF for a given procedure *root*. The parameter *tfs* keeps track of the CTFs of the procedures that have been analyzed. The algorithm starts by computing the initial points-to relation, which involves linking the blocks referenced in the procedures with their initial state blocks present in the formals. It then evaluates each instruction, after which it updates the current state *state*. Since this is a FI algorithm, the order of the instruction does not matter. When it encounters a store instruction, it calls *spaUpdateState* to compute the new state *nstate*. When a call instruction is encountered, the algorithm first calls *spaQueryState* to find out the set of procedures that the call instruction may call. This process is necessary since we may be accessing function pointers. The algorithm then iterate through all the potential callees, and call *spaApply* to apply the callsite point-to information to the CTF of the callee, thereby propagating the point-to information of the callee back to the call site. If the CTF is not available at *tfs*, it recurs and obtain the CTF of the callee before proceed. This process repeats until a fixed-point is reached, when there is no further update of the state function. Note that while this is an iterative process, each procedure needs to be analyzed only once. Before it returns the result, a call to *spaPrune* will be performed in order to remove from CTF the set of point-to edges emitting from local and parameter blocks, which, by the language scoping rule, are not accessible outside.

Algorithm 1 *Pointer analysis.*

```
spaAnalysisFICS = func(
    spa : Spa, root : SpaBlock, tfs : SpaBlock ↦ SpaDD
    ) : SpaDD {
    var state, nstate : SpaDD;
    var changed : {true, false};

    changed = true, state = spaInitState( spa, root );
    while( changed == true ) {
        changed = false;
        forall( instrn ∈ spa.procs(root).instrns ) {
            if( instrn.opcode == store ) {
                nstate = spaUpdateState(spa, state, instrn.dst,
instrn.srcs[0]);
                if( nstate ≠ state )
                    changed = true, state = nstate;
                }
            else if( instrn.opcode == call ) {
                forall( callee ∈ spaEnumerate(
                spaQueryState( spa, state, instrn.dst.blk.range,
instrn.dst.level )
                    ) ) {
                    if( tfs(callee) == ⊘ )
                        spaAnalysisFICS(spa, callee, tfs);
                    nstate = spaApply(spa, state, instrn.srcs,
callee, tfs(callee));
                    if( nstate ≠ state )
                        changed = true, state = nstate;
                    }
                }
            }
        }
    state = spaPrune( state, root.params ∪ root.locals );
    tfs(root) = state;
    return state ;
    }
```

The following results ensures that the algorithm terminates.

Lemma 1 *A set of Boolean functions defined on the Boolean space $\mathcal{B}^n$ forms a lattice, where the bottom element is constant 0 function, the top element is constant 1 function, and the meet operator is the Boolean or operator.*

Lemma 2 *The program state in the form of a Boolean function computed by Algorithm 1 at each iteration is monotonically increasing with respect to the Boolean or operator.*

Theorem 1 *Algorithm 1 terminates.*

4.3 State Query

Algorithm 2 is used to find out the set of all possible values an access path may have under a program state. It is needed to process every access path, be it a destination or source, of an instruction. On the point-to graph, it is equivalent to determining the envelop of the nth level breath-first search starting from a block, when n is the level of the access path. In our Boolean framework, the algorithm is similar to the image computation algorithm in reachability analysis. For level 1 access *b, where b is a block, one only needs to multiply the state function with the domain cube of the block. The resultant Boolean function captures the set of blocks in the range space. To access one more level, one needs to call *bddMirror* to isomorphically map the computed range back to the domain space, which can again be used to multiply with the state function. This process continues until we reach the desired level. Note that unlike algorithms based on point-to graph, the image computation can be performed implicitly, in other words, it can be performed purely logic operation without resolving to explicit graph traversal.

Algorithm 2 *State query.*

```
spaQueryState = func(
    spa : Spa, state : SpaDD, from : SpaDD, level : Z
    ) : SpaDD {
    if( level == 0 ) return from ;
    return bddAndAbstract(
        spa, state, bddMirror(
            spa, spaQueryState(spa, state, from, level-1)
            )
        ) ;
    }
```

4.4 Evaluating Stores

We use Algorithm 3 to compute the new state whenever a store instruction is encountered. Consider a simple case where the access paths for both the destination and the source of the instruction are simple blocks, which correspond to adding an edge in point-to graph based algorithms. Here we simply need to add a product term to the current state, which is the product of the domain cube of the destination and the range cube of the source. The operation holds equally well with complex access paths, whose net effect will still amount to adding a product of two Boolean functions. Note that this corresponds to collectively adding many point-to edges at the same time!

Algorithm 3 *State update.*

```
spaUpdateState = func(
    spa : Spa, state : SpaDD, dst : SpaAccessPath, src :
SpaAccessPath
    ) : SpaDD {
    return bddOr(
        state, bddAnd(
            bddMirror( spa,
             spaQueryState( spa, state, dst.blk.range, dst.level
)
                ),
          spaQueryState( spa, state, src.blk.range, dst.level+1
)
            )
        ) ;
    }
```

4.5 Evaluating Calls

The pointer information of the callee has to be propagated back to the caller by applying the program state at the call site to the transfer function of the callee. This is achieved by Boolean function composition, in other words, substituting the Boolean variables that construct the initial state blocks by a set of Boolean functions. Such projection map can be obtained from the binding of actuals to the formal. If we use a single variable for each initial state block, it can be computed trivially.

Algorithm 4 *Evaluate call.*

```
spaApply = func(
    spa : Spa, state : SpaDD, srcs : [ ]^SpaAccessPath,
    proc : SpaBlock, tf : SpaDD
    ) : SpaDD {
    var bind, proj : SpaDD ↦ SpaDD;
    var actual, formal : SpaDD;

    forall( i ∈ [0..‖srcs‖-1] ) {
        actual = spaQueryState(spa, state, srcs[i].blk.range,
srcs[i].level+1 );
        formal = proc.formals[i].range;
        bind(formal) = actual;
        bind(bddMirror(spa, formal)) = bddMirror(spa, actual);
        }
    proj = bddFindProjection(spa, bind);
    return bddOr(
        state, bddCompose( spa, tf, proj )
        ) ;
    }
```

4.6 State Pruning

Algorithm 5 first computes the Boolean function that corresponds to the set of all blocks need to be pruned. It then predicates the state with the inversion of the computed function.

Algorithm 5 *State pruning.*

```
spaPrune = func(
    spa : Spa, state : SpaDD, vars : ⟨⟩^SpaBlock
    ) : SpaDD {
    var sum : SpaDD;

    sum = bddFalse(spa);
    forall( v ∈ vars )
        sum = bddOr( sum, v.domain);
    return bddAnd( spa, state, bddNot( spa, sum ) ) ;
    }
```

4.7 Relaxing Assumptions

We now discuss how the basic algorithms described can be modified if the simplifying assumptions are relaxed.

C extensively uses aggregates such as arrays and structs. The notions of block as well as access path need to be refined so that different fields of a struct can be distinguished. Wilson [17] developed the concept of a *location set*, which characterizes a set of grouped location using a pair of normalized integers, called offset and stride. The use of location set won't change our algorithm, however, in order to maintain the invariant that blocks are disjoint, some block collapsing needs to be performed as the algorithm proceeds. Note that this tradeoff is also made in [17].

The basic algorithm does not handle *strong update*, which concerns the assignment of a unique location. Global and local scalar blocks are always unique. Dynamic blocks are never unique. When a strong update happens, instead of merging the new value with the original value as the new point-to set, as in the case of *weak update*, the original value stored in the location under current state needs to be removed. Under our symbolic framework, this removal can be easily carried out the same way as Algorithm 5. The real challenge, however, is that for initial state blocks, the knowledge of whether they are unique scalars is not available. This factor alone leads to the failure of true CTF in previous efforts [17, 9]. On the other hand, under our symbolic framework, it is just as simple as introducing an extra Boolean variable, called the *scalar predicate*, for each initial block.

Some extra care needs to be exercised to handle programs with recursive functions. We follows the usual path: whenever a calling cycle is hit, we use the transfer function computed so far as an approximation, and will iterate until a fixed-point is reached.

5 Experimental Result

We implemented an FICS pointer analysis tool based on the ideas outlined in this paper using a synthesis/compiler infrastructure we developed. Our infrastructure includes several frontends such as C, Java, Verilog as well as a C-based, objected oriented, polymorphic language developed specifically for System-on-chip. We also leverage Somenzi's CUDD package [14] for BDD manipulation.

Our first implementation wasn't intended for comprehensive comparison with what has been published in the literature, but rather a validation of a new idea.

Name	Source	LOC	#procs	density	Run Time (s)
01.qbsort	McGill	325	8	24.1%	0.10
06.matx	McGill	350	7	13.5%	0.13
15.trie	McGill	358	13	23.4%	0.21
04.bisect	McGill	463	9	9.7%	0.10
17.bintr	McGill	496	17	8.8%	0.13
05.eks	McGill	1202	30	4.0%	0.20
08.main	McGill	1206	41	20.9%	1.33
09.vor	McGill	1406	52	28.6%	5.54
allroots	Landi	227	7	1.3%	3.02
football	Landi	2354	58	1.8%	2.38
compiler	Landi	2360	40	5.1%	5.3
assembler	Landi	3446	52	16.6%	10.63
simulator	Landi	4639	111	6.3%	4.03

Table 1. Experimental results

We have important omissions: e.g., we do not yet distinguish field elements, and have not fully implement Wilson's location set concept. Our transfer functions for libraries are hardwired, and we still do not have a strategy to summarize library function without analyzing the library source code. We have not yet handle non-local control transfers such as setjmp/longjmp. All these complications have prevented us from attempting large industrial benchmarks. Also we are not even close to reaching the last miles of performance improvement using symbolic technology. Therefore, at this stage we are not ready for a fair, direct performance comparison with traditional techniques.

Nevertheless, we were able to apply our implementation on the benchmark set commonly used in the pointer analysis research community: one set from McGill university [5], and the other from Dr. Landi [6]. The statistics of the benchmarks are shown in Table 1, where the third column indicates the lines of code, the fourth column indicates the number of procedures (not including libraries), and the fifth column indicates the percentage of control flow nodes that have pointer assignment. The last column shows the runtime of our algorithm obtained on a Ultra-SPARC 5. The result is very encouraging: all experiments finish in seconds.

6 Conclusion

In this paper, we described an idea for accurate pointer analysis in a way significantly different from traditional approaches. We discuss the rationale of this approach by first analyzing sources of in-efficiency in traditional methods. We demonstrated our preliminary implementation and experiment, which confirmed our expectation. In the future, we will provide a thorough implementation of a comparative framework so that solid empirical data can be obtained for different methods. We will also apply pointer analysis techniques in high-level and system-level synthesis.

References

[1] O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Computer Science Department, University of Copenhagen, 1994.

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

[3] 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.

[4] O. Coudert and J. C. Madre. The implicit set paradigm: a new approach to finite state system verification. *Formal Methods in System Design*, 2(6):133–145, 2001.

[5] L. H. et al. Mccat compiler project. `http://www-acaps.cs.mcgill.ca/~benadmin/benchmarks/`.

[6] W. L. et al. Prolangs analysis framework. `http://www.prolangs.rutgers.edu`.

[7] 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.

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

[9] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insenstitive and context-insensitive points-to analyses for java. In *Workshop on Program Analysis For Software Tools and Engineering*, pages 73–79, 2001.

[10] 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.

[11] L. Semeria and G. D. Micheli. Spc: Synthesis of pointers in C: Application of pointer analysis to the behavioral synthesis from C. In *Proceedings of the International Conference on Computer-Aided Design*, pages 321–326, November 1998.

[12] 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.

[13] 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.

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

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

[16] 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.

[17] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1995.

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

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