Memory Classification Analysis for Recursive C Structures

Naftali Schwartz*

February 7, 1999

Abstract

The long-time quest of the parallelizing compiler community for effective aggregate summarization techniques has led to increasingly sophisticated array section representations. In this paper, we show how the latest of these can be used for nested C structure summarization. We then show how this summarization notation can be used to make Shape Analysis precise on arbitrarily low-level code. Combining these techniques, we show that an appropriate generalization of Memory Classification Analysis, originally presented for Fortran programs, provides a flow dependence summarization technique for C code as well, while avoiding code normalization compared with previous techniques. In so doing, we break down perhaps the final conceptual barriers in the construction of practical programmer-friendly C parallelizing compilers.

1 Introduction

Although the theoretical foundation for program parallelization, data dependence analysis, has existed for decades[Ban76], researchers are still struggling with how the basic principles can be effectively applied to real-world programs. While dependence testing is trivial in the case of unaliased variables, traditional data dependence analysis addresses the aliasing introduced by loop-indexed array references. It provides an equational framework whose solution reveals whether any access to an array by one reference will touch any elements accessed by another. If the references generate identical accesses, we say that a data dependence of some sort exists between the references.

The difficulty in applying this technique has been that the cost of this analysis is quadratic in the number of references, which naturally grows with the program size. Therefore, access summarization techniques have long been held to provide the only hope of obtaining pragmatic implementations. However, early summarization techniques were either too simplified to avoid approximation or too general to avoid expensive testing algorithms in the face of complex subscript expressions. The approximate techniques offered no better than flow-insensitive or may-access information, which cannot be used to compute data-flow[CI97a]. The general ones introduced their own impracticalities[AI91]. Indeed, the lack of any clearly preferable representation has led to the generalization of existing frameworks to work with many different ones. For example, the PIPS analysis framework has adopted a generic framework supporting intervals, lists of convex polyhedra and Presburger formulae side by side with the original single convex polyhedron representation[CI97b], with the attendant rise in implementation complexity.

*Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer St., New York, NY 10012-1185, nschwart@cs.nyu.edu.

1.1 LMADs in Fortran

However, the recent introduction of the Access Region Descriptor[PHP98] (ARD) as a summarization technique coupled with Memory Classification Analysis[Hoe98] (MCA) for the Polaris Fortran compiler[PEH+93] has been shown to be a firm foundation on which to build a practical array-level data-flow framework. The ARD is a complex data structure whose centerpiece consists of the Linear Memory Access Descriptor (LMAD) describing all details of a memory access. The LMAD achieves maximum precision by describing accesses uniformly as a (recursive) sequence of equidistant element references. An arbitrarily complex multidimensional traversal can be uniquely represented with only the base offset and a series of stride/span pairs (access dimensions). The size in bytes of each element then completes the exact memory map for the access.

Because the LMAD uses array declarations in the same manner they are used in a compiler, namely, to calculate actual element offsets, it avoids the complexity of continually relating subscript expressions to array bounds expressions. By explicitly linearizing array accesses, the representation directly supports programmer linearization, and offers Fortran-style array reshape transparency. Furthermore, it avoids the so-called in-bounds assumption often employed in other constrained triplet notations[Gu97] but which is so uncharacteristic of real-world programs.

1.2 Problems in C

Several issues arise in employing this representation in a C compiler. One concern is that widespread casting found in ordinary C code can render otherwise comparable ARDs to be incompatible by virtue of differing element sizes, resulting in conservative estimations of access conflicts.

Furthermore, it's not immediately obvious how this scheme can be generalized to work with structure accesses. An ARD with a single stride/span pair of 1/1 and appropriate base and element size, to be sure, can be used to represent a single member access, but when arrays are nested together with structures, it's not immediately clear how to construct an LMAD to precisely represent all accesses.

The rest of this paper is organized as follows. In Section 2 we show how the general LMAD construction can be generalized to represent arbitrarily nested C structure accesses. In Section 3 we recall the manner in which MCA effectively summarizes all program dependences. In Section 4 reviews a recent Shape Analysis algorithm for simple recursive structures. Section 5 describes how we can extend this algorithm to deal with low-level C code by combining it with our LMAD representation. Then in Section 6 we show how an enhanced MCA based on the above two techniques provides efficient and effective flow dependence summarization for recursive C structures without resort to any code normalization. Section 7 reports on related work, and Section 8 concludes.

2 General LMAD Construction

LMAD construction for an array access starts with the formation of a linearized access expression which is the sum of the products of each dimension's access expression and the size of each element of that dimension, and in which each index variable has been normalized. Then this symbolic expression is repeatedly expanded through successive index variables, starting with the innermost. As each index variable is substituted out, a stride/span pair is inserted into the LMAD. The stride is set to the increase in base expression due to the increment by one of the normalized index variable, and the span is set to the difference between the base offset expression with the normalized index variable set to its minimum and maximum values.

Because of the differences between the language conventions of C and Fortran we present in Figure 1 a (corrected) version of their LMAD construction example[Hoe98] as it would be written in C. The steps of

```
int A[*][14];
for ( int i = 0; i < 2; i++ )
    for ( int j = 0; j < 2; j++ )
        for ( int k = 0; k < 10; k += 3 )
            ...  A[j][k+26*i] ...
```

$$(\mathcal{A}^{3,\,14,\,26}_{9,\,14,\,26} + 0)$$
$$(\mathcal{A}^{3,\,14}_{9,\,14} + 26i)$$
$$(\mathcal{A}^{3}_{9} + 14j + 26i)$$
$$(\mathcal{A} + 3k + 14j + 26i)$$

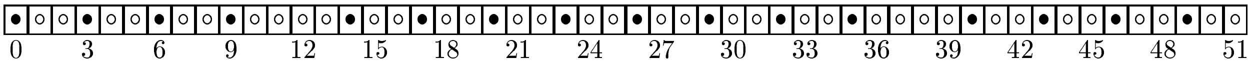


Figure 1: C language version of Hoeflinger's Example Showing Expansion Through Indices

LMAD expansion from the reference out through each index variable is also displayed in the figure, as well as the access memory map.

2.1 LMAD Construction for C Memory Accesses

The problems outlined in the introduction can be uniformly addressed by modeling all accesses at the byte level. The element size is dropped from the ARD, and one additional access dimension of stride one is inserted whose span is the byte size of each unit access. This provides the necessary casting transparency during analysis, albeit at the expense of introducing machine-dependencies into the analysis. However, these dependencies will only affect correctness of the analysis where unsafe casting is actually used, and under these circumstances we can do no better.

Furthermore, as shown in Figure 2, this generalizes well even to nested structures and arrays. However, in this case we must also introduce a member offset term into LMAD to account for the nature of generalized aggregate access, as shown in the figure. The computational framework Hoeflinger describes for the ARD will also automatically coalesce two contiguous structure members when the structures have no holes. Structures which do contain padding, however, can easily be handled by "upgrading" the (small) datatype generating the hole to a larger one which fills the gap. These gaps are labeled with an overbar in the memory access map of Figure 2. (Of course, unions as well as structures are supported with this scheme.)

With a C-style adaptation of the LMAD in hand, we now consider Memory Classification Analysis for C programs. We first give a general overview of the MCA process for scalars and aggregates without pointers, and then use our LMAD interpretation for the analysis of recursive C structures.

```
struct X { int a; char b; };
struct Y { char c; X d[2]; char e; } A[*];
for ( int i = 0; i < 2; i++ )
    for ( int j = 0; j < 2; j++ )
        ...  A[i].d[j].a ...
```

$$(\mathcal{A}^{1,\,8,\,24}_{4,\,8,\,24} + \texttt{\&A[0].d[0].a} - \texttt{(int*)\&A})$$
$$(\mathcal{A}^{1,\,8}_{4,\,8} + 24i + \texttt{\&A[0].d[0].a} - \texttt{(int*)\&A})$$
$$(\mathcal{A}^{1}_{4} + 8j + 24i + \texttt{\&A[0].d[0].a} - \texttt{(int*)\&A})$$

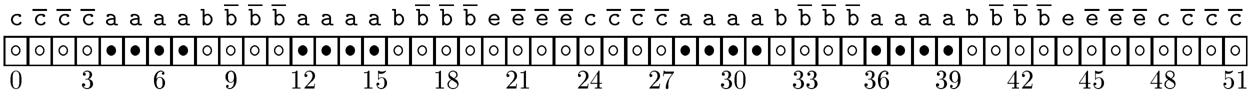


Figure 2: Nested Array/Structure Example Using Unitary Element Size

`x = NULL`
`x->sel = NULL`
`x = malloc( ... )`
`x = y`
`x->sel = y`
`x = y->sel`

Table 1: Basic Shape Operators

3 Memory Classification Analysis

MCA is a methodology of assigning memory accesses to access classes to provide precise program dependence dataflow between dependence grains, which are areas of code which will execute on a single processor, and therefore within which internal dependences can be safely ignored.

In this scheme, every dependence grain is associated with three memory access sets: Read-Only, Write-First and Read/Write. As each program statement is processed, it generates new statement-level sets of these three types, which are then merged with the old sets according to a list of equations involving set union, intersection and difference. Any guarded statements generate descriptors with an appropriate execution predicate. Loop summaries are expanded through the variables of the respective loop indices. Dependence testing in this framework involves computing the various intersections between these three sets of different access types.

The MCA framework also incorporates conditional array access, by attaching a (true by default) predicate to each ARD. The framework is kept efficient by the aggressive combining of related ARDs. For example, a series of writes to successive array indices under the same condition will be incorporated into the appropriate write set as only a single ARD. By keeping the three sets as small as possible, maximum efficiency of dependence testing is assured.

By extending the LMAD and thus the ARD to work with arbitrary C structures, we have shown that MCA may be performed on C programs without pointers. We now show that combining this with sophisticated shape analysis we can further extend MCA to operate on arbitrary C programs.

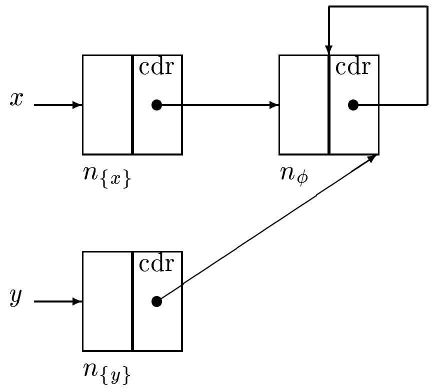
4 Shape Analysis

Recent work by Sagiv, *et al.*[SRW96, SRW98] has shown how a dataflow analysis may be conducted on code which manipulates heap allocated data structures to efficiently compute a good static estimate of the stable "shape" of the heap. In the fixed-point solution, each statement is associated with a "static shape graph" (SSG), whose nodes are the union of set of variables pointing into the heap and a set of "shape nodes", which are named by the set of variables pointing to them. Edges of the shape graph are either a (variable node, shape node) pair, or a (shape node, selector, shape node) triple.

In their paper, they define an abstract interpretation for the operators listed in Table 1 (in C notation). These operators are to be used on "normalized" programs – in C terms, this means (1) allocation statements may only assign to simple variables, (2) in each assignment, the same variable does not occur on both the left- and right-hand sides, (3) each assignment from a non-NULL value is immediately preceded by a NULL assignment to the same left-hand side and (4) all temporary variables introduced by the normalization are assigned NULL at the program termination. The semantics of address assignment, although not explicitly mentioned, will produce a shape node which need not be named transiently by variables which point to it, but is permanently associated with the addressed variable.

```
/* x points to an unshared,
    acyclic, singly linked list */
y = NULL;
while ( x != NULL )
{
    t = y;
    y = x;
    x = x->cdr;
    y->cdr = t;
}
t = NULL;
```

(a) Program Text



(b) Stable Shape Graph of Final Node

Figure 3: Sagiv, *et al.*'s Shape Analysis Example

Their example code and final shape graph are reproduced in Figure 3. In this graph, we have made their selector labels of each selector edge explicit. The final shape graph tells us, among other things, that both x and y end up pointing to unshared lists. While this scheme is sufficient for code without casting or unions, a much more powerful labeling scheme will be required for real C code. This scheme would need to consider that selector labels are not necessarily representative of particular structure offsets. In fact, truly low-level code may not even mention any selector label at all. For example, Figure 4(b) contains a semantically equivalent (under any reasonable machine model) version of the higher-level code in Figure 4(a). The standard UNIX `offsetof` macro is defined as:

```
#define offsetof(type, field) ((long) &((type *)0)->field)
```

The low-level code version shows how the compiler has to often deal with lumps of memory devoid of any form, where any reliance on field names is doomed to fail. Since the memory semantics in the first case is clear and in the second case equivalent, we need a framework which supports either of these equally well. The extended LMAD we introduced for nested structure analysis is a perfect fit here as well. In the next section, we show how using this form leads to MCA for recursive structures.

```
struct { int *a; char *b[10]; } *A;
int *P;
char *Q;
.
.
A->a = P;
for ( int i = 0; i < 10; i++ )
    A->b[i] = Q;
```

(a) High-level version

```
struct { int *a; char *b[10]; } *A;
int *P;
char *Q;
.
.
*(int**)((char*)A+offsetof(X,a)) = P;
for ( int i = 0; i < 10; i++ )
    ((char**)((char*)A+offsetof(X,b)))[i] = Q;
```

(b) Low-level version

Figure 4: Different Code Abstraction Levels

5 Shape Analysis for Real Code

As we showed in the last section, a shape graph based on selector labels cannot in general describe the types of structure accesses found in real C programs. Any truly general representation needs rather to be based on the actual offset and size of each access. The previous caveat about introducing machine dependencies applied here as well, but again, this seems to be the best we can do.

In place of selector labels in the SSG, we use the ARD based on the extended LMAD specifying precisely the memory locations involved in the access. Unlike ordinary ARDs, though, two neighboring ARDs which are unique selector labels cannot be merged unless they also point to the same shape node.

The ARD representation alone obviously cannot represent complex pointer relationships. On the other hand, shape analysis alone does not effectively deal with pointer arrays or complex addressing. Their combination, however, produces an extremely powerful analysis. This representation captures the semantics of casting, unions and even arbitrary offset addressing for recursive structures of arbitrary complexity. Furthermore, through the mechanism of expansion through loop variables, it provides an efficient way of classifying all memory accesses.

6 Memory Classification Analysis

To formulate the full MCA, we need some notion of an invariant node id. Every ARD in MCA is associated with a particular variable; in our generalization, there is no particular variable *per se*, but we can formulate some notion of invariant shape node identity. In Figure 5(a), we recall the data dependence example of [RS98], transcribed, as ever, into C, and in Figure 5(a) we show the SSGs at the various program statements.

Just as MCA creates an ARD for every direct memory access, MCA creates one for every indirect access. Figure 5(c) shows the LMADs that each program statement would create. These classification sets would lead directly to the flow dependences they report for this code, but more importantly, they can be used directly in the MCA to compute summarizations. Furthermore, indirectly accessed arrays can be subjected to a similar expansion through loop index technique when used within indexed loops.

To see how flow dependences are precisely characterized, observe that all dependences which are associated with a variable are trivially computed. For flow dependences associated with a heap location, of which there are six in Figure 5(c), we must introduce a special "tracing" naming scheme during shape analysis, which we sketch briefly here. Specifically, note that because the $(\{head\}_4^1)$ shape node written at `s2` is later identified as $(\{head, tail\}_4^1)$ at `s4`, all of `s5`, `s11` and `s12`, which read a shape node with identity $(\{tail, temp\}_4^1)$ or $(\{head\}_4^1)$ must be considered dependent. In contrast, the $(\{temp\}_4^1)$ node written at `s7`, which has never been directly associated with the $(\{head\}_4^1)$ node, only has dependences `s5` and `s12`, the statements which read the $(\{tail, temp\}_4^1)$ node.

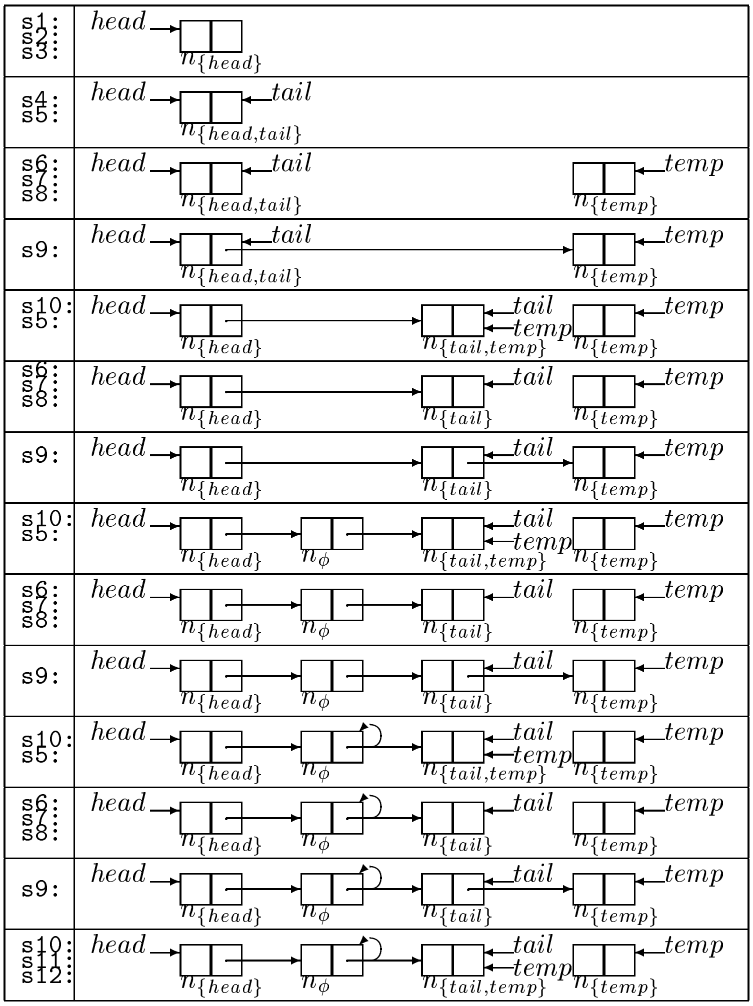
Therefore, the invariant shape node identifier we seek is the set of names with which a particular shape node has been associated through the course of shape analysis. By using these names for ARDs associated with heap locations we can develop a framework for efficient and accurate access summarization for heap locations using the identical machinery used for variables.

7 Related Work

The connection between alias analysis and program dependences was investigated by Ross and Sagiv[RS98], whose dependence example we have used in Section 6. They demonstrate a flow dependence reduction which equates the problem with may-alias analysis. However, they do not provide an access summarization scheme,

```
void DestructiveAppend()
{
     struct N { char car; N *cdr; };
     N *head, *tail, *temp;
s1:  head = malloc( sizeof N );
s2:  cin >> head->car;
s3:  head->cdr = NULL;
s4:  tail = head;
s5:  while ( tail->car != 'x' )
     {
s6:      temp = malloc( sizeof N );
s7:      cin >> temp->car;
s8:      temp->cdr = NULL;
s9:      tail->cdr = temp;
s10:     tail = tail->cdr;
     }
s11:  cout << head->car;
s12:  cout << tail->car;
}
```

(a) Example code



(b) Shape graphs

Statement	Read Set	Write-First Set	Read/Write Set	Variable	Heap
s1:		$(head_4^1)$		s2, s3, s4, s11	
s2:	$(head_4^1)$	$(\{head\}_4^1)$			s5, s11, s12
s3:	$(head_4^1)$	$(\{head\}_4^1+4)$			
s4:	$(head_4^1)$	$(tail_4^1)$		s5, s9, s10, s12	
s5:	$(tail_4^1)$ $(\{tail,temp\}_4^1)$				
s6:		$(temp_4^1)$		s7, s8, s9	
s7:	$(temp_4^1)$	$(\{temp\}_4^1)$			s5, s12
s8:	$(temp_4^1)$	$(\{temp\}_4^1+4)$			
s9:	$(tail_4^1)$ $(temp_4^1)$	$(\{tail\}_4^1+4)$			s10
s10:	$(\{tail\}_4^1+4)$		$(tail_4^1)$	s5, s9, s10, s12	
s11:	$(head_4^1)$ $(\{head\}_4^1)$				
s12:	$(tail_4^1)$ $(\{tail,temp\}_4^1)$				

(c) Memory Classification Analysis Sets and Flow Dependences

Figure 5: Ross & Sagiv's Data Dependence Analysis Example

essential for parallelization analysis of large codes. Furthermore, although the flow dependence reduction they provide is efficient, it may be inappropriate for, *e.g.*, automatic parallelization systems which seek to analyze code at the source level for the purpose of promoting high levels of programmer recognition of the transformed code.

Ghiya, Hendren and Zhu addressed the goals of the current research in a recent paper[GHZ98]. Their work is based on the family of pointer analyses supported by the McCAT compiler for stack- and heap-allocated objects. Because the scope of their work is the detection of common parallelism patterns, by design it does not generalize well to whole program analysis in the same manner as our interpretation of Memory Classification Analysis. What's more, they do not provide a unified framework for general program dependence calculation, but rather a collection of techniques which may be applied in various situations.

Another completely different approach to generalized dependence testing over C structures was taken by Kennell and Eigenmann[KE98]. Ironically enough, this work was also carried out in the context of the Polaris compiler. They set out to determine what form a C program could be transcribed into to leverage the elaborate existing Fortran Polaris analysis framework. While they do report some important analysis results, the awkwardness of their approach seems undeniable.

Other recent work in pointer analysis for programs with structures and casting is reported by Yong, *et al.*[YHR99], but was unavailable at the time of this writing.

8 Conclusion

We have described how appropriate generalizations of Linear Memory Access Descriptors[Hoe98] and Shape Analysis[SRW98] can be combined to produce a general Memory Classification Analysis for C programs with recursive structures, nested structures with arrays, typecasting and unions. Furthermore, the orthogonality of this combination provides immediate generalization to interprocedural analysis by simply combining the respective interprocedural adaptations outlined for each individual technique. Although we have introduced platform dependencies in our algorithm, this is the unavoidable cost for precise analysis of low-level C codes, which today seem to be in the majority.

It must be noted in this context that in this opinion we diverge from the view of, *e.g.*, Morgenthaler[Mor97], who opts to introduce imprecision into the analysis to avoid any platform assumptions. In our view, it seems obvious that when heavy-duty program analysis is called for, most programmers would gladly port the analyzer over to the appropriate platform in order to extract the maximum mileage from their programs.

The techniques described in this paper together with those of a companion paper[Sch98] are two steps in the goal of bringing us closer to the realistic application of sophisticated Fortran analysis techniques to ordinary C programs, ultimately discrediting the efficiency argument to which some diehard Fortran programmers cling so tenaciously. The other paper describes a technique for effectively analyzing the code at the complex C source level while avoiding normalization of one form, and this work describes a heap dependence analysis technique which avoids normalization of another form. We attempt to bring the state of the art past normalization, for the ultimate goal of promoting programmer-friendliness in program transformation tools.

9 Acknowledgements

This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320; by the National Science Foundation under grant number CCR-94-11590; and by Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

References

[AI91] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[Ban76] U. Banarjee. Data Dependence in Ordinary Programs. Master's thesis, University of Illinois at Urbana-Champaign, 1976.

[CI97a] B. Creusillet and F. Irigoin. Exact versus approximate array region analyses. *Lecture Notes in Computer Science*, 1239:86–??, 1997.

[CI97b] Batrice Creusillet and Franois Irigoin. Interprocedural Analyses of Fortran Programs. *Journal on Parallel Computing*, 24(3-4):629–648, June 1997.

[GHZ98] Rakesh Ghiya, Laurie J. Hendren, and Yingchun Zhu. Detecting Parallelism in C Programs with Recursive Data Structures. In *Proceedings of the International Confrence on Compiler Construction, CC '98*. Springer-Verlag, LNCS 1383, 1998.

[Gu97] J. Gu. *Interprocedural Array Data-Flow Analysis*. PhD thesis, University of Minnesota, December 1997.

[Hoe98] J. Hoeflinger. *Interprocedural Parallelization Using Memory Classificaiton Analysis*. PhD thesis, University of Illinois at Urbana-Champaign, 1998.

[KE98] R. L. Kennell and R. Eigenmann. Automatic Parallelization of C by Means of Language Transcription. In *Advances in Languages and Compilers for Parallel Computing*, LNCS. Springer-Verlag, 1998.

[Mor97] J. Morgenthaler. *Static Analysis for a Software Transformation Tool*. PhD thesis, University of California, San Diego, 1997.

[PEH$^+$93] David Padua, Rudolf Eigenmann, Jay Hoeflinger, Paul Petersen, Peng Tu, Stephen Weatherford, and Keith Faigin. Polaris: A New-Generation Parallelizing Compiler for MPP's. Technical Report 1306, Center for Supercomputing Research and Development, June 1993.

[PHP98] Yunhueng Paek, Jay Hoeflinger, and David Padua. Simplification of array access patterns for compiler optimizations. *ACM SIGPLAN Notices*, 33(5):60–71, May 1998.

[RS98] J. Ross and Mooly Sagiv. Building a Bridge between Pointer Aliases and Program Dependences. *Nordic Journal of Computing*, 8:361–386, 1998.

[Sch98] Naftali D. Schwartz. Steering Clear of Triples: Deriving the Control Flow Graph Directly from the Abstract Syntax Tree in C Programs. Technical Report TR1998-766, New York University, Department of Computer Science, June 1998.

[SRW96] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–31, St. Petersburg Beach, Florida, 21–24 January 1996.

[SRW98] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[YHR99] S.H. Yong, S. Horwitz, , and T. Reps. Pointer Analysis for Programs with Structures and Casting. In *Programming Language Design and Implementation*, May 1999.