

The Value Evolution Graph and its Use in Memory Reference Analysis *

Silvius Rus
Texas A&M University
silviusr@cs.tamu.edu

Dongmin Zhang
Texas A&M University
dzhang@cs.tamu.edu

Lawrence Rauchwerger
Texas A&M University
rwerger@cs.tamu.edu

Abstract

We introduce a framework for the analysis of memory reference sets addressed by induction variables without closed forms. This framework relies on a new data structure, the value evolution graph (VEG), which models the global flow of values taken by induction variable with and without closed forms. We describe the application of our framework to array data-flow analysis, privatization, and dependence analysis. This results in the automatic parallelization of loops that contain arrays addressed by induction variables without closed forms. We implemented this framework in the Polaris research compiler. We present experimental results on a set of codes from the PERFECT, SPEC, and NCSA benchmark suites.

1. Motivation

The analysis of memory reference sets is crucial to important optimization techniques such as automatic parallelization and locality enhancement. This analysis gives information about data dependences within or across iterations of loops, about potential aliasing of variable names and, most importantly about the flow of values stored in program memory. The analysis of memory references reduces to an analysis of the addresses used by the program, or, more specifically in the case of Fortran programs, an analysis of the arrays indices. A large body of research has been devoted to this field yielding significant achievements. When index functions are relatively simple expressions of the loop induction variables and the array references are not masked by a complex control flow, then the analysis is relatively straight forward. For example, if in a loop an array is indexed through an affine function of the loop induction variable and the references are control flow insensitive then the

data dependence analysis can be performed accurately and, if possible and profitable, the loop can be parallelized.

Unfortunately, arrays are not always referenced in such a simple manner. Sometimes the values of the addresses used are not known during compilation, e.g., when the values of the addresses are read from an input file or computed within the program (use of indirection arrays). In other situations although the addresses are expressed as a simple function of the loop induction variable, the control flow that masks the actual references makes it impossible to compute a closed form of the index variable and thus very difficult to perform any meaningful analysis.

Some of these difficulties have been addressed in the recent past by using run-time analysis and speculative optimizations for loops that cannot be analyzed statically, e.g., loops with input dependent reference patterns [22].

Recently, Hybrid Analysis [25] has improved the accuracy and performance of optimizations by bridging the gap between static and run-time analysis. In this paradigm, the partial results of static, compile-time analysis can be saved and used during a dynamic analysis phase, when all statically unknown values are available.

However, despite recent progress, memory reference analysis and subsequent loop parallelization, cannot be performed with sufficient accuracy when arrays are indexed by subscripts that cannot be expressed as a closed form of the loop induction variable. Arrays cannot be proved independent because their indices cannot be analyzed with classical data dependence techniques and indices of arrays (addresses) cannot be computed independently by each iteration (or processor). In this paper we propose the *Value Evolution Graph (VEG)* as a novel representation for the value flow of induction variables that cannot be expressed as a simple algebraic function of their loop index. We show how this technique can improve the accuracy of data dependence analysis, privatization and the recognition of certain classes of memory reference patterns, such as *pushback* sequences. We show how these improved techniques can lead to the automatic parallelization of a larger number of codes than ever before.

* Research supported in part by NSF CAREER Award CCR-9734471, NSF Grant ACI-9872126, NSF Grant EIA-0103742, NSF Grant ACI-0326350, NSF Grant ACI-0113971, DOE ASCI ASAP Level 2 Grant B347886,

1.1. Background and a Motivating Example

Recurrences with closed forms are those in which the i -th term can be written as an algebraic formula of i . In recurrences with closed forms most relations between values are proved using symbolic calculus. For example, references to arrays using recurrences with closed forms, can be meaningfully expressed using systems of in-equations [27, 21, 14] or triplet-based notations [15, 10] containing the closed form terms and other symbolic values such as loop bounds. We will not address such recurrences in this paper. When a recurrence with no closed form is used to index an array, the corresponding memory reference set cannot be summarized using an algebraic formula. For example, the algebraic expressions for the index of array A at line 8 in Fig. 1(a) for iterations k and $k+1$ are identical, p , but their values always differ. Hence, we need to develop alternative analysis techniques that can deal with such cases. There are various uses for information about recurrence values. In the example in Fig. 1, we can find array A independent in the loop at line 3 if we show that $q < p$ and that the values of p are different in any two iterations that write to A . We can propagate the values stored in array A in the loop at line 3 to where they are used at line 13 if we know that the set of the definition indices covers the set of use indices. We can propagate the values in B if we know that $p \leq 2 * old$ (the value of p at statement 12).

1.2. Our Solution: The Value Evolution Graph

To solve the problems presented above we propose to model the value flow of the recurrences without closed form with the *Value Evolution Graph* (VEG) and use it to obtain sufficient information to allow parallelization.

The reference pattern on array B in Fig. 1 (a) uses a recurrence with closed form $q(i) = i$, which was substituted in Fig. 1(b). It is easy to prove that there are no loop carried dependences on B because the index of B is expressed as an analytical function of the loop index. However, there is no such formula for the index of A because it is indexed by p , which is defined by a recurrence without a closed form (due to conditional incrementation). Fortunately, data dependence analysis does not require us to have closed form solutions, but rather to prove relations between the index sets corresponding to different iterations. In order to prove array A independent, we first need to show that statements 6 and 8 are independent. Note that at statement 6 we read from A at offsets between $[1:old]$, and at 8 we write based on all the values of the recurrence on p . We can do it by finding all the values of the recurrence – its *image* – and prove that they do not intersect $[1:old]$. We also need to prove that statement 8 does not cause cross-iteration dependences by

<pre> 1 old = p 2 q = 0 3 DO i = 1, old 4 q = q+1 5 B(q) = 1 6 IF (A(q).GT.0) 7 p = p+1 8 A(p) = 0 9 ENDIF 10 ENDDO 11 sum = 0 12 DO i = old+1, p 13 sum = sum+A(i) 14 +B(i-old) 14 ENDDO </pre>	<pre> 1 old = p0 3 DO i = 1, old 4 p1 = μ(p0, p3) 5 B(i) = 1 6 IF (A(i).GT.0) 7 p2 = p1+1 8 A(p2) = 0 9 ENDIF 10 ENDDO 11 sum = 0 12 DO i = old+1, p4 13 sum = sum+A(i) 14 +B(i-old) 14 ENDDO </pre>
--	--

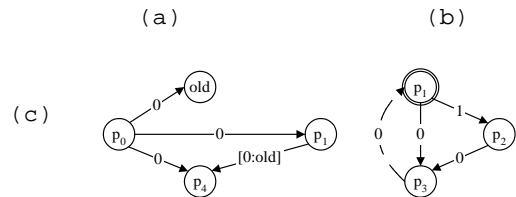


Figure 1. (a) Code sample, (b) in GSA after closed form substitution, (c) Value Evolution Graphs.

itself. We can do it by proving that the value of the index at line 8 always takes a positive *step*.

The *Value Evolution Graph* shown at the bottom of Fig. 1(c) translate the problems of computing the *step*, *image*, and *last value* of the recurrence within the first loop into graph problems. Although the idea of value flow in the program is not new [24, 29, 30], the VEG offers unique features and functionality needed by various analyses (Sec. 2). We have integrated the VEG into a generic memory reference analysis framework that can solve multiple classes of optimization problems in the presence of recurrences without closed forms.

1.3. Our Contribution

We believe that this paper makes the following original contributions:

- We propose the *Value Evolution Graph* that can represent the data flow in recurrences used as array indices which have no closed form solutions. The graphs are pruned based on control dependence predicates and produce tighter value ranges than abstract interpretation methods.
- Unlike the previous efforts of looking for patterns in the code text, we can analyze partially aggregated and classified memory descriptors. This single generic approach both extends and unifies in a single framework

most cases which were previously solved using various, different, pattern matching techniques. It allows for the parallelization of important classes of memory reference patterns, e.g., pushbacks.

- The presented technology is implemented and fully functional as a module in the Polaris compiler and was crucial in further parallelizing a larger number of benchmark codes.

Moreover, by integrating the VEG in our memory classification analysis we have been able to accurately classify memory access sequences and use them to prove or disprove relations between memory sets: increasing, contiguous and consecutive. We have further used the VEG to improve the coverage of other important analysis techniques, e.g., data dependence analysis, privatization

In the following section we will formally introduce the Value Evolution Graph (VEG), its use in the Memory Classification Analysis. Then we will show how we have used it to perform more accurate dependence analysis, privatization and finally parallelization. Experimental results and a comparison to previous work will conclude the paper.

2. The Value Evolution Graph (VEG)

Finite recurrences are usually described by an initial value, a function to compute an element based on the previous one¹ (an *evolution* function), and a limiting condition. Depending on the evolution function’s formula, in certain cases we can evaluate important characteristics even for recurrences without closed forms: the *distance* between two consecutive elements, the *image* of the recurrence, i.e. the set of all values it may take, and the *last element* in the sequence.

We introduce the *Value Evolution Graph (VEG)*, a compiler representation for the flow of values across arbitrarily large and complex program sections, including, but not limited to, recurrences without closed forms. Consider the loop at line 3 in Fig. 1. It performs a repeated conditional push to a stack array A . The stack pointer is stored in variable p . Due to the fact that p is incremented conditionally, there is no closed form for the recurrence that defines its value. We represent values as *Gated Static Single Assignment (GSA)* [2] names. In GSA, there are three types of ϕ -nodes. γ nodes merge two values on different forward control flow paths. μ nodes merge a loop back value with a loop incoming value. η nodes merge the outcome of a loop with the value before the loop. While this helps to discern between the values of p on the left and right hand side of the assignment at line 7 respectively, it does not differentiate between the value of p at line 8 in successive iterations. However, it makes it easy

to determine that the stack array is written only at position p_2 , and that p_2 is always the result of an addition of 1 to p_1 . The subgraph consisting of $\{p_1, p_2, p_3\}$ (in Fig. 1(c)) represents the value flow between different GSA names for p in a single iteration of the loop. Each edge label represents the value added to its source to obtain its destination. The dashed edge carries values across iterations, but is not part of the VEG as it does not contribute to the flow of values within an iteration. We can employ well-known graph algorithms to prove that the distance between two consecutive values of p_2 is always 1, which makes the write to $A(p_2)$ be a stack push operation.

We will show how we construct the VEG in general, and how we run queries on it to compute recurrence characteristics over complex program constructs, such as loop nests, complex control flow, and subprogram calls.

2.1. Formal Definition

We define a *value scope* to be either a loop body (without inner loops), or a whole subprogram (without any loops). Immediately inner loops and call sites are seen as simple statements. We treat arrays as scalars and assume that programs have been restructured such that control dependence graph contains no cycles other than self-loops at loop headers. We have implemented such a restructuring pass in our research compiler.

Definition. Given a value scope, the Value Evolution Graph is defined as a directed acyclic graph in which the nodes are all the GSA names defined in the value scope and the edges represent the flow of values between the nodes.

Nodes. In addition to the nodes defined in the value scope, we add, for every immediately inner loop, the set of GSA names that carry values outside the inner loop. An example is p_1 in Fig. 1.

Such nodes appear both in their current value scope graph as well as in the immediately outside value scope graph. They are called μ nodes in the context of the graph corresponding to the inner value scope and are displayed as double circles. Nodes representing variables assigned values defined outside their scope are called *input* nodes and are labeled with the assigned value (they are displayed as rectangles). The μ and *input* nodes are the only places where values can flow into a VEG. Values can flow out of the VEG through μ nodes only.

Edges. An edge between two variables p and q represents the *evolution* from p to q , defined as the function f , where $q = f(p)$. The evolution belongs to a scope if p and q are defined within the scope, and all symbolic terms in f are defined outside it. We represent four types of evolutions, additive and multiplicative for integer values and *or* and *and* for logical values. We represent an evolution by its type and the value of the free term. Certain evolutions can be com-

¹ We only address first order recurrences in this paper.

Statement	Edge	Ev. Type	Label
$b_1 = a + \text{exp}$	$a \rightarrow b_1$	+	exp
$b_1 = a .\text{OR. exp}$	$a \rightarrow b_1$	\vee	exp
$b_1 = a * \text{exp}$	$a \rightarrow b_1$	*	exp
$b_1 = a .\text{AND. exp}$	$a \rightarrow b_1$	\wedge	exp
$b_1 = a$	$a \rightarrow b_1$	Default	Identity
$b_1 = \text{exp}$	no edge, mark <i>input</i> node		
$b_2 = \gamma(b_0, b_1)$	$b_1 \rightarrow b_2$	Default	Identity
	$b_0 \rightarrow b_2$	Default	Identity
$b_2 = \mu(b_0, b_1)$	no edge, mark μ node		
$b_2 = \eta(b_0, b_1)$	$b_1 \rightarrow b_2$	Default	Loop effect
	$b_0 \rightarrow b_2$	Default	Identity
CALL $\text{sub}(b_1 \rightarrow b_2)$	$b_1 \rightarrow b_2$	Default	<i>sub</i> effect

Table 1. Extracting evolutions from the program.

posed along a path symbolically. For instance, the evolution along path $p_1 \rightarrow p_2 \rightarrow p_3$ is an additive evolution with value $I + O = I$. Instead of keeping a single value for an evolution, we store a range of possible values. This allows us to define an aggregated evolution from a node p to a node q as the union of the evolutions along all paths from p to q . For example, the aggregated evolution from p_1 to p_3 is $[0:1]$ which represents the union of the evolution $[0:0]$ along path $p_1 \rightarrow p_3$ and the evolution $[1:1]$ along path $p_1 \rightarrow p_2 \rightarrow p_3$.

Complexity. VEGs are as scalable as the GSA representation of the program since the number of nodes in all VEGs is at most twice the number of GSA names in the program and every node corresponding to a ϕ definition has the same number of incoming edges as the number of ϕ arguments. All other nodes have at most one incoming edge.

2.2. Value Evolution Graph Construction

Table 1 shows how we create edges from their corresponding statements. For now, we support only one evolution type per VEG. This evolution type is given by the first evolution we encounter, and is called the default type of the graph. If a value is computed in a way different from the ones shown in the table, we conservatively transform it into an *input* node and label it with $[-\infty : +\infty]$ (or $[\text{FALSE}::\text{TRUE}]$). If it is computed in an assignment statement, then we try to find a closer range for the right hand side of the statement. We compute the aggregated evolution of an entire recurrence as the aggregated evolution, over all iterations, from the μ node to all nodes that may carry evolutions to the next iteration. We draw an edge from the value of the μ node to the corresponding value on the left hand side of the corresponding η definition, and we label it with the aggregated evolution of the inner recurrence. Fig. 1(c) shows such an edge between p_1 (a μ node in the inner recurrence $\{p_1, p_2, p_3\}$) and p_4 . The range $[0:old]$ is a re-

sult of multiplying the range of the aggregated evolution from p_1 to p_3 , $[0:1]$, with the iteration count of the loop, *old*. When values are obtained as a result of a subprogram call, we add edges to represent the aggregated value evolutions of the *OUT* actual arguments (and global variables) as functions of *IN* actual arguments (and global variables). In the last line in Table 1, b_2 and b_1 are the *OUT* and *IN* arguments respectively.

The VEGs are built in a single bottom-up traversal of the whole program. The call graphs and the loop nest graphs of each program are traversed in reverse topological order. Within each scope we identify all definitions, build edges and associate input values. We use aggregated information from inner loops and called subprograms as shown in Table 1. We compute the aggregated value evolution for all the recurrences associated with the loops using shortest/longest path algorithms that are linear in the size of the graph (number of edges + number of nodes). We compute the shortest and longest paths between every μ and *input* node and every other node. If every node is reachable from exactly one μ node and there are no *input* nodes, the complexity of the algorithm is linear in the number of GSA names + the number of arguments in all the ϕ nodes in the program. If more than one μ node can reach one same other node (coupled recurrences), the complexity may increase by a factor of at most the number of coupled recurrences.

2.3. Queries on Value Evolution Graphs

We obtain needed information about the values taken by induction variables by querying the VEG. All the queries we support are implemented using shortest path algorithms. Since all the VEGs are acyclic, these algorithms have linear complexity.

Distance between two values in two consecutive iterations of a loop. Given two GSA variables (possibly identical) and a loop, we can compute the range of possible values for the difference between the value of the second variable in some iteration $i+1$, and the value of the first variable in iteration i . For recurrences without closed forms, this computes the *distance* between two consecutive elements. In the example in Fig. 1, the distance between p_2 in iteration i and p_2 in iteration $i+1$ is exactly I . This information can be used to prove that the write pattern on array A at statement 8 cannot cause any cross-iteration dependences. The value of the distance between a source node and a destination node across two consecutive iterations of a loop can be used for comparisons only if the destination node is not reachable from an *input* node.

Range of a variable over an arbitrarily complex loop nest. Given a GSA variable and a loop, we can compute the range of values that the variable may take over the iteration space of the loop. For recurrences without closed

```

1 A(p1) = ...
2 f1 = 0
3 IF (cond)
4   f2 = 1
5   p2 = p1+1
6 ENDIF
  p3 = γ(p1, p2,
         cond)
  f3 = γ(f1, f2,
         cond)
7 IF (f3.GT.0)
8   p4 = p3-1
9 ENDIF
  p5 = γ(p3, p4,
         f3.GT.0)
10 IF (f3.EQ.1)
11   ... = A(p5)
12 ENDIF

```

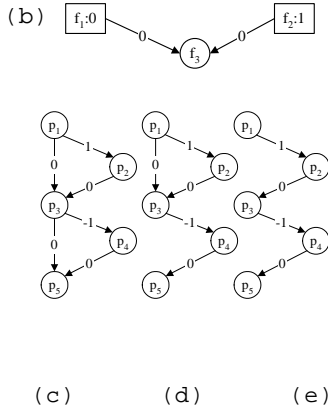


Figure 2. (a) Sample code in GSA, (b) VEG for f_1, f_2, f_3 ; VEG for p_1, p_2, p_3, p_4, p_5 – (c) before pruning, (d) after pruning based on GSA Paths, and (e) based on range tracing.

forms, this computes their *image* and can be used to evaluate the *last element*. In the example in Fig. 1, the range for variable p_2 over the loop is $[p_0+1:p_0+old]$. This information is crucial for proving that the write pattern on array A at statement 8 cannot have cross-iteration dependences with the read pattern at statement 6 (they are contained in disjoint ranges $[p_0+1:p_0+old]$ and $[1:p_0]$ respectively). This information is computed in $O(d)$ time, where d is the depth of the loop nest between the given loop and the definition site of the given variable.

Global value comparison. Given two GSA variables in the same subprogram, we can compare their values even if they are not in the same value scope, by comparing their ranges in a larger common scope. This information can be used to prove either an order between their values or their equality and which in turn can be used in many compiler analyses.

2.4. VEG Conditional Pruning

We can prune a VEG by removing certain edges that cannot be taken when based on the truth value of a condition. The shortest path algorithms used to compute aggregated evolutions will then produce tighter ranges. Consider the code shown in Fig. 2 (a). Because we do not know anything about the value of *cond*, we cannot compare the values of p_1 and p_5 , information that is needed to determine if the memory read at offset p_5 in array A is always covered by the write at offset p_1 . Based on its corresponding VEG (Fig. 2 (c)), we can only infer that $p_5 \in [p_1-1:p_1+1]$.

The GSA path technique [28] describes how control dependence relations can be used to disambiguate the flow of values at γ gates. The GSA path technique can infer that at line 11 condition $f_3.EQ.1$ holds true, which implies also $f_3.GT.0$ holds true. To the VEG, this means that value p_5 comes from p_4 and not directly from p_3 . With the VEG pruned using this information (Fig. 2 (d)), we have $p_5 \in [p_1-1:p_1]$.

We have improved on [28] by using the VEG to trace back ranges extracted from given control dependence predicates. The read from array A at line 11 is guarded by condition $f_3.EQ.1$. This implies $f_3.EQ.1$ holds true. From this predicate, we extract the range $[1:1]$ for f_3 . In Fig. 2 (b), we trace this range for f_3 backward to see where it could have come from. Since the initial value for *input* node f_1 is 0, and the edge $f_1 \rightarrow f_3$ has weight 0, the only range that can be produced on the path $f_1 \rightarrow f_3$ is $0+0=0$. The GSA gate $f_3 = \gamma(f_1, f_2, cond)$, associates the pair (f_1, f_3) with condition *NOT.cond*. Since f_3 cannot come from f_1 , *NOT.cond* must be false, thus *cond* must be true. The same predicate, *cond*, controls the other gate, $p_3 = \gamma(p_1, p_2, cond)$. Since *cond* holds true, p_3 must have come from p_2 , and not from p_1 . So the edge $p_1 \rightarrow p_3$ cannot be taken. This leads to the graph in Fig. 2 (e). On the pruned graph in Fig. 2 (e), $p_5 = p_1+1+0-1+0 = p_1$, which proves the read at line 11 covered by the write at line 1.

This method improves on [28], leads to more accurate ranges than the abstract interpretation method used in [4], and can solve classes of problems that [29] cannot. One use of VEG conditional pruning is presented in Sec. 4.

3. VEG-based Memory Reference Analysis

3.1. Memory Reference Classification

Memory Classification Analysis (MCA) [15] is a general dataflow technique used to perform data dependence tests and privatization analysis, but which is also usable in any optimization that requires dataflow information, such as constant propagation. For a given program context – a statement, loop, or subroutine body – MCA classifies all memory locations accessed within the context in *Read Only (RO)*, *Write First (WF)* and *Read Write (RW)*. The *RO* set records all memory locations that are only read (never written); the *WF* set records all memory locations that are first written, then possibly read and/or written again; the *RW* set includes all other memory locations. We perform MCA in a single bottom-up traversal of the program by aggregating and classify memory locations across larger and larger program contexts. For instance, if a variable is *RO* in a statement and *WF* in the following statement, it is classified as *RW* for the block of two statements.

```

CALL classify-loop () → WFi, ROi, RWi
CALL contiguous-write(WFi, ROi, RWi) → CW
CALL expand(WFi, ROi, RWi) → WF, RO, RW
CALL update(WF, RO, RW, CW) → WF, RO, RW
END

SUB update(WF, RO, RW, CW) → WF, RO, RW
  RO = RO - CW
  RW = RW - CW
  WF = WF ∪ CW
END

```

Figure 3.

```

PROGRAM main
1 DO k = 1, 100
  q1 = μ(q0, q2)
2 old = q1
3 CALL build(q1→q2)
4 ... = A(old : q2-1)
5 ENDDO
...

FUNCTION ten(b, e)
1 DO j = b, e-1
2 IF (A(j)...)
3 RETURN .F.
4 ENDIF
5 ENDDO
6 RETURN .T.
7 END

SUB build(p0→p5)
1 old = p0
2 DO i = 1, 100
  p1 = μ(p0, p4)
3 IF (ten(old, p1))
4 DO j = p1, p1+9
5 A(j) = ...
6 ENDDO
7 p2 = p1+10
8 ELSE
9 A(p1) = ...
10 p3 = p1+1
11 ENDF
12 p4 = γ(p2, p3)
13 ENDDO
14 p5 = η(p0, p1)
15 END

```

Figure 4. Code extracted from EXTEND_do400.

The most important memory classification process takes place at loop level (Fig. 3). The *expand()* operation generates the access pattern of the whole loop based on the access pattern within an iteration. For instance, in the example in Fig. 1, the *WF* pattern for array *B* within an iteration of the loop at line 3 is $\{i\}$. Across the entire loop, it is $\bigcup_{i=1}^{old} \{i\} = [1:old]$. When the recurrence has no closed form, these operations cannot be performed symbolically. However, we can use the VEG to detect contiguous sequences of memory locations indexed by recurrences without closed forms. These sequences, found by subroutine *contiguous-write* are used to adjust the results of *expand*, as shown in subroutine *update*.

Consider the example in Fig. 4. Conceptually, the loop in program *main* performs a repeated pushback on array *A*, based on index *q*. The stack array *A* is also read at line 4 in program *main* and at line 2 in function *ten*. Both reads are to elements that have been pushed within the same iteration of the loop in program *main*, thus they are covered by writes. Consequently, array *A* is privatizable.

Traditional analysis fails because of the conditional incrementation of the index *p* by either *10* or *1*. Recent work [18, 30, 31] focused on statement-level pattern matching of recurrence expressions. These approaches fail to relate the

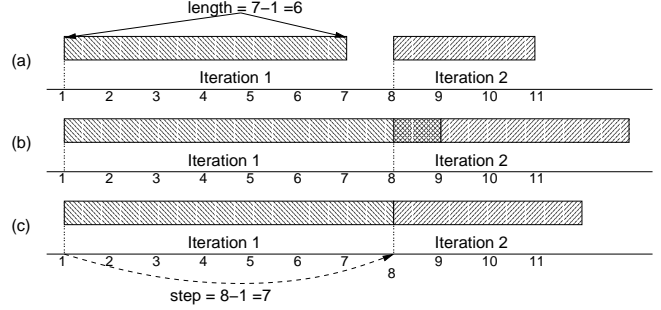


Figure 5. Increasing (a), contiguous (b), and consecutive (c) reference patterns in a loop.

write to $A(j)$ at line 5 in subroutine *build* to p . Also, they cannot handle the presence of *read* memory references and recurrences over multiple variables (q , old , p).

Our approach is to aggregate memory references symbolically using VEG information. Our addition to MCA does not require new data structures, as the new information is used to refine the existent *RO*, *WF*, and *RW* descriptors. We aggregate the reference pattern over the loop at line 4 in subroutine *build* into $[p_1:p_1+9]$. We cannot aggregate the reference pattern over the outer loop in subroutine *build* because the recurrence on p_1 has no closed form. Regardless of the value returned by function *ten*, we can see that the write pattern is *contiguous*, i.e. it has no gaps between any two successive iterations. The write access pattern can be aggregated across the whole loop as $[p_0:p_5-1]$. At the beginning of any iteration i , the extent of the contiguously written section in previous iterations is $[p_0:p_1-1]$. The read from *A* at line 2 in function *ten* is always within $[old:p_1-1]$. We can prove it is covered by previous writes, since $old = p_0$. Also, we can find that the extent of the contiguous write for the whole loop is $[p_0:p_5-1]$. At the call site in program *main*, this translates into $[q_1:q_2-1]$, which covers the successive reads completely within every iteration. We solved this MCA problem not based on the closed form of the index but rather on the information about the recurrence exposed by the VEG.

In order to parallelize the loop in program *main*, we still have to prove that there are no cross-iteration output dependencies. We do it by proving that the per-iteration descriptor, $[q_1:q_2-1]$, is *increasing*, i.e. it has no overlaps. A VEG query is used to evaluate the step from q_2-1 to q_1 across two successive iterations and to prove it is positive.

3.2. Memory Reference Sequence Classification

A memory reference sequence is *increasing in a loop* if every access index in iteration $i + 1$ is strictly larger than any index in iterations 1 to i (Fig. 5(a)). It is *contiguous in a*

	Sequence Class	Context	Benefit
1	Contiguous	Inner	Privatization
2	Increasing	Outer	Independence
3	Contiguous	Outer	Efficient parallel code

Table 2. Uses of memory reference sequence classification for the parallelization the outer loop of a doubly nested loop.

loop if it is contiguous within every iteration and, for any iteration i , its image over all iterations up to i is contiguous (Fig. 5(b)). It is *consecutive in a loop* if it is both contiguous and increasing in the loop (Fig. 5(c)). These definitions can be extended to strided memory access. These properties have to be proved true across all control paths.

We use VEG information to measure and compare the extent of memory reference sets and recurrence steps. This analysis is control-flow sensitive. In order to prove a sequence contiguous, we show that on all paths, and under the same or implied conditions the step of induction variable (obtained from the VEG) is smaller or equal to the span of the memory reference, at the loop level.

3.3. Classic Compiler Optimizations

Let us assume that we want to parallelize the outer loop of the nested loops *Outer* and *Inner*. Table 2 presents the overall use of memory reference sequence classification in privatization and data dependence analysis.

Dataflow Analysis. We can use the WF , RO , and RW sets to prove general dataflow relations. For instance, a WF followed by a RO represents a def-use edge with weight $WF \cap RO$. This information can be used in transformations such as constant propagation. In the example in Fig. 1, we can prove that there is a def-use edge between lines 8 and 13 on array A , with weight $[old+1:p_4]$. We can thus propagate all constant array values at offsets within this range.

Privatization. The privatization transformation benefits from memory reference sequence classification indirectly. The refined WF , RO , and RW sets for *Inner* will result in refined RO_i , WF_i , and RW_i sets for *Outer*. leading to more opportunities for privatization. This corresponds to edge 6 in Fig. 6, and to row 1 in Table 2.

Dependence Analysis. Let us assume that we have the descriptors RO_i , WF_i , and RW_i for *Outer*. If we can find a memory reference sequence d that includes them and is increasing in *Outer*, then there can exist no cross iteration data dependences. This corresponds to edges 3 and 5 in Fig. 6, and to row 2 in Table 2.

3.4. Recognition of Pushbacks and Other Parallelizable Prefix Computations

Many programs access arrays in loops according to patterns that are determined by loop induction variables. Even though induction variables are computed by recurrences, there are many important cases in which such loops can be executed in parallel. First, necessary conditions are that (i) there should be no data dependences between iterations of the loop except those involving the induction variable, (ii) there is no dependence cycle between the induction variable used as an address and the data computation, and (iii) it must be possible to compute the values taken on by the induction variable in parallel. Two cases in which the induction values can be computed in parallel are when the induction recurrence has a closed form solution or when it is associative; in the former case parallelization is trivial and in the latter case it can be done using a parallel prefix type computation [16]. Parallel prefix typically consists of three stages: (i) compute the local prefix sums of the associative induction variable, (ii) compute the prefix sums of the induction variable across processors, (iii) use the results of the cross-processor phase to compute the corresponding global values of the local indices, and then copy out the contents of the local arrays to their corresponding offset in the global array.

[19] addresses loops that contain the pattern $p = p+I$; $A(p) = \dots$ and where p does not appear anywhere else in the loop body, and parallelize them using a technique named “array-splitting,” which is essentially a prefix computation.

In this work, we use the VEG to extend the applicability of the parallel prefix parallelization to more general types of loops that cannot be analyzed using pattern matching techniques alone. In particular, for loops with induction variables with no closed form solution, we impose the condition that the induction variable can only be used as an address into an array, i.e., it does not contribute to the global data and/or control flow of the loop. In other words, if the induction variable is assigned to a shared variable or controls the execution of the program (e.g., used as an absolute inner loop bound) we will take the conservative approach and not parallelize it. An exception is made for the case when the value of the recurrence is used to test loop termination.

Pushback sequences. We first consider loops which compute so-called *pushback sequences* that are generally defined as a sequence of consecutive write-first (WF) reference sets. In the following, we describe how we have used information provided by the VEG to extend the applicability of parallel prefix parallelization to pushback sequences. For illustration we use the code example in Fig. 4 which effectively performs a pushback on array A .

a) References to the pushback array have to be WF only. This implies that read accesses, to the array covered by the WF are allowed in any order. The WF set is computed accu-

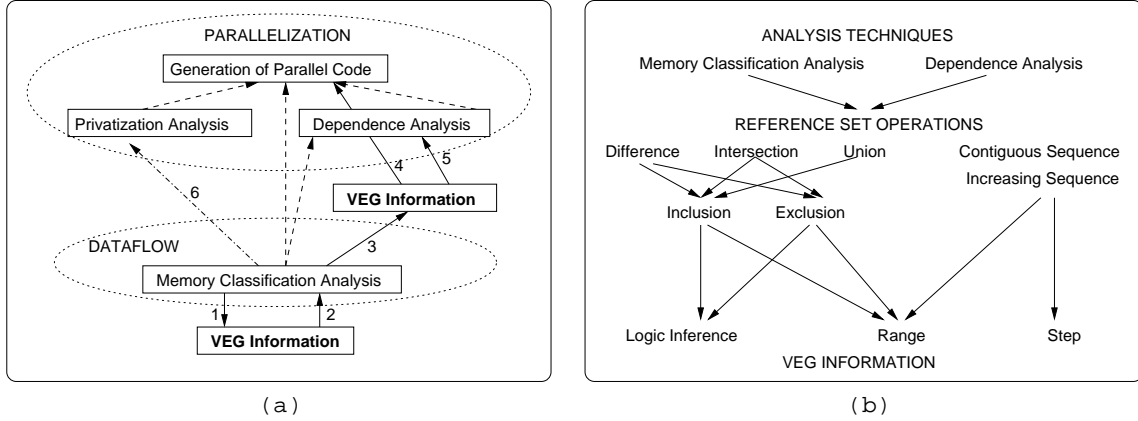


Figure 6. (a) Integration in the Hybrid Analysis framework and (b) details on how the VEG information is used.

rately by the VEG improved MCA and thus qualifies more loops for parallelization. In the example in Fig. 4 we can see that the read at line 2 in function *ten* is always covered by a write in a previous iteration of the loop at line 2 in subroutine *build*, but within the same iteration of the loop in program *main*.

b) Most previous techniques analyze the patterns in which the induction variable appears, and from that try to infer which array addresses are used; this only works if there is a very simple (e.g., identity) relation between induction values and array indices. We can qualify more loops as pushbacks since we can use VEG enhanced MCA to analyze more complex functions of the induction variable and determine if the resulting index sequence satisfies the $step \leq length^2$ condition.

c) Information provided by the VEG can help us identify cases in which the induction variable does not contribute to the control flow, even if it would appear that it does using pattern matching techniques. For instance, in Fig. 4 the reference to *A* at line 2 in function *ten* is through *j*, which is the index of the loop at line 1. This loop has recurrence values as bounds. The looping statement can be normalized as $DO k = 1, e-b$. We use the VEG and evaluate the (e-b) loop bound and find that it does not depend on the induction variable of the outer loop, i.e., that we do not use the value of the induction variables of the loop we are considering (we use a local value).

d) We can parallelize loops where the recurrence value is also used as an early termination condition. Such cases are common for error checks such as stack overflow which usually result in premature loop exits. We execute the parallel prefix speculatively [23], compute the final value of the recurrence variable (before the termination) and then use it to copy out only the section of the private arrays that fits in the

correct bounds.

Other Sequences. Using VEG enhanced MCA we can parallelize additional sequences with parallel prefix. Here are some interesting sequences we can recognize:

a) A sequence whose index is generated by a simple associative recurrence with *any* positive step such that $step > length$. In this case, the copy out phase will require that the computation of the indices into the global array be done in a more complex manner than for a pushback. Instead of using ranges of global addresses we have to compute them individually.

b) Sequences whose index is generated by a more complex associative induction of some form $v = f(v, k)$ where *f* is an associative operator. In this case, VEG enhanced MCA can be used to guide the application of the set operations. (Although it is true the set operations themselves will be more complex, that is a symbolic manipulation problem that is beyond the scope of this paper.)

It is interesting to remark that when we do not deal with a simple pushback sequence, the parallel prefix computation of the recurrence value and the actual computation of the loop must be done, conceptually, in separate stages. Sometimes it is beneficial to perform in the local stage only the computation of the recurrence values and leave the remainder of the loop computation for the third phase of the parallel prefix. Other times, when the distribution of the recurrence computation implies a large amount of work duplication, it is beneficial to compute everything in the first phase in private storage and leave the actual address computation and copy out for the third phase. The compiler can use a simple work evaluation model to decide between the two alternatives.

2 The *step* of the recurrence versus the *length* of the memory access.

Program	Loop	Seq. %	Description
TRACK	EXTEND_do400	15-65	CP-SLU, P-CW
	FPTRAK_do300	4-50	CP-SL
	GETDAT_do300	1-5	CP-SLU, P-CW
P3M	PP_do100	52	P-CW, P-VEG
	SUBPP_do140	9	P-CW
BDNA	ACTFOR_do240	29	P-VEG
MDLJDP2	JLOOPB_do20	12	CP
ADM	DKZMH_do60	6	P-CW
QCD	QQQLPS_do21	< 1	CP
DYFESM	SETCOL_do1	< 1	CP
HYDRO2D	WNFLE_do10	< 1	CP

Table 3. Loops parallelized. CP = Conditional Pushback, SL(U) = Stack Lookup (and Update), P-CW = Privatization based on Contiguous Writes, P-VEG = Privatization using the VEG directly.

4. Experimental Results

Hybrid Analysis [25] integrates compile-time and run-time analysis of memory reference patterns. Its static part consists mainly of a framework for aggregation of memory references using the compact RTLMAD memory location set representation. This framework is used to perform Memory Classification Analysis which is used for automatic parallelization. We have integrated the information produced by VEGs in this framework – Fig. 6(a). Fig. 6(b) shows the relation between three levels of abstraction in the analysis process. High-level routines such as dependence analysis relies on memory reference set operations (such as intersection) and on the recognition of increasing memory reference sequences. These operations rely heavily on VEG information, such as step, range, or logical inferences. We implemented the VEG and integrated it with the Hybrid Analysis pass in Polaris.

Table 3 presents our results over codes *TRACK*, *BDNA*, *QCD*, *ADM* and *DYFESM* from the *PERFECT* benchmark suite, *P3M* from the *NCSA* suite, and *HYDRO2D* and *MDLJDP2* are from *SPEC92*. The third column shows the percentage of the total sequential execution of the program spent in the loop. The parallelization of these loops is crucial to the overall performance improvement in *TRACK*, *BDNA*, *ADM*, *P3M*, and *MDLJDP2*. Although our new techniques can parallelize a larger number of loops, we only display results in addition to the ones obtained using traditional analysis techniques.

Seven out of the eleven parallelized loops were *conditional pushbacks*. The cases in *TRACK* are the most difficult as the arrays are not used as a stack at statement level, but only at the whole loop body level. We are not aware of any

```

1 DO i = 1, ny
2 DO j = 1, nx
3 DO k = 1, nz
4   p = k
5   CALL sr(A(p), inc)
6   IF (A(p).GT.0)
7     ENDDO
8     p2 = η(p0, p1)
9     p3 = p2 + inc3
10    DO k = p, nz
11      A(k) = ...
12    ENDDO
13    DO k = 1, nz
14      ... = A(k)
15    ENDDO
16 ENDDO
3 DO k = 1, nz
4   p1 = k
5   CALL sr(A(p1),
6     inc3)
7   IF (A(p1).GT.0)
8     GOTO 8
9   p2 = η(p0, p1)
10  p3 = p2 + inc3
11  DO k = p3, nz
12    A(k) = ...
13  ENDDO
14  ...
15  SUB sr(a, inc)
16  inc1 = 0
17  a = ...
18  IF (...) inc2 = 1
19  inc3 = γ(inc1, inc2)
20 END

```

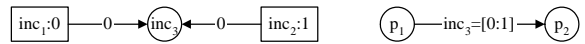


Figure 7. Code extracted from *DKZMH_do60*.

other static analysis that can parallelize any of these three loops. Six out of eleven loops required privatization analysis based on either *contiguous writes* or VEG information directly (value ranges).

Loop *BDNA/ACTFOR_do240* contains an inner loop that fills an index array *ind* with values within range $[1:i]$, where *i* is the index of the outer loop. These values are then used to index a read operation on an array *xdt*. Since array *xdt* is first written in every iteration of the outer loop from 1 to *i*, this write covers all successive reads from *xdt(ind(:))*. The read pattern *ind(:)* is found to be completely contained in $[1:i]$ based on the VEG range approximation for *ind*, which proves *xdt* privatizable. This pattern also appears on some arrays in *P3M/PP_do100*.

We also ran the analysis on the Barnes-Hut code *TREE* from the University of Hawaii in order to compare our results to previous work reported in [18]. This is an interesting case of an array that is used as a stack (push and pop operations) within an iteration of a loop, and is thus privatizable. However, the loop cannot contain cross-iteration dependences because the stack array is a local variable in a subroutine *treewalk* which is called from within the loop. Even if the code were inlined, our VEG-enhanced MCA would find the array privatizable in the outer loop.

ADM/DKZMH_do60:

The loops at line 1 and 2 in Fig. 7 can be parallelized if we can show that array *A* is privatizable. We show that *A* has no exposed reads for the context between lines 3-14.

At lines 3-11, $WF = [1:p_2] \cup [p_3:nz]$. The distance between p_2 and p_3 is the value range for variable *inc3*. This range was found by the VEG for subroutine *sr* to be $[0:1]$. This implies $p_2+1 \geq p_3$, so $WF = [1:nz]$. At lines 3-14, $RO = RO - WF = [1:nz] - [1:nz] = \emptyset$. $WF = [1:nz]$.

```

1  $flag_1 = 0$ 
2 DO  $j = old, p$ 
3   IF ( $A(j) \dots$ )
4      $flag_2 = 1$ 
5      $same_1 = j$ 
6     GOTO 9
7   ENDIF
8 ENDDO
    $flag_3 = \eta(flag_1, flag_2)$ 
    $same_2 = \eta(same_0, same_1)$ 
9 IF ( $flag_3.EQ.1$ )
10   $A(same_2) = \dots$ 
11 ENDIF

```

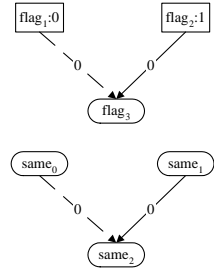


Figure 8. Code extracted from *EXTEND_do400*.

TRACK/EXTEND_do400:

This loop is our most complex case, and was presented in detail as the loop in program *main* in Fig. 4. It consists of pushbacks performed in an inner loop. Another loop, inner to both of them, reads backwards the elements that were pushed within the same iteration of the outermost loop and, based on some condition, may modify some of these locations. It is crucial to prove that the access within an iteration of the outer loop is confined to locations that were pushed within the same iteration. In addition to the problems discussed with relation to Fig. 4, this loop presents another problem that can only be solved using a conditionally pruned VEG.

This innermost loop is shown in Fig. 8. The range of the elements that have been pushed back within the current iteration of the outermost loop is $[old:p]$. The write reference at line 10 is at offset $same_2$. We must prove that $same_2$ is within $[old:p]$. A simple query on the range of values for $same_2$ on the VEG returns a range of $[1:p]$ because it has to take into account the possibility that $same_2$ was not defined in the loop at line 2. The value could have come on edge $same_0 \rightarrow same_2$. Since $same_0$ could have carried a value from a previous iteration of the outermost loop, $same_2$ might not be confined to what was pushed in the current iteration. However, this edge is removed during the pruning of the VEG based on condition $flag_3.EQ.1$. Since a value for $flag_3$ could have come only on edge $flag_2 \rightarrow flag_3$, and since the pair $(flag_2, flag_3)$ corresponds to the same control flow edge as $(same_1, same_2)$ (based on GSA γ predicates), we can remove edge $same_0 \rightarrow same_2$. The VEG now evaluates the range of $same_2$ to $[old:p]$ for the use at the statement at line 10.

5. Related Work

Recurrence Recognition, Classification, and Parallelization. [1, 26, 29, 8] present the automatic recognition and classification of general recurrences. The idea of a value graph was introduced in [24]. Although similar to the SSA

graph [29], the VEG adds more power and functionality to the representation: closure for the meet-over-paths operator using ranges and accuracy by pruning based on conditionals. [6] discusses the parallelization of linear recurrences. [5] and [7] present the recognition and parallelization of certain classes of recurrences but do not address cases when memory is referenced using the values of the recurrence. We express recurrence functions as paths in VEGs. Although in theory these techniques could cover more cases, in practice they are limited to recurrence formulae consisting of linear algebraic expressions coupled with conditionals. Since some of the edges in the VEG represent algebraic relations and others represent conditional execution, we can also represent this mix of linear functions and conditionals.

Analysis of Memory Referenced by Recurrences without Closed Forms. [13] and [20] found more closed forms for classes of recurrences that had not been commonly recognized/substituted by compilers. [9] presents the parallelization of loop nests that may contain recurrences by flattening the nests into single loops and pre-computing the recurrences in inspector loops. This method may not be feasible when the recurrence depends on computation within the loop itself. [12] presents the use of monotonicity in reducing the number of bound checks for arrays referenced using a recurrence without closed form.

Let us follow (by row in Table 4) a comparison between our framework and the most recent work on the parallelization of loops that reference memory through recurrences without closed forms [11, 18, 30, 31].

1, 2. [18] presents three algorithm recognition techniques that can be used for privatization and dependence analysis. [30, 31] used the concept of monotonic evolutions for data dependence tests. We introduce a single technique that covers all the problems solved by [18, 30, 31], has wider applicability, and, additionally, builds generic array dataflow information that can be used by other transformations (such as constant propagation). [11] uses monotonic information to improve memory reference set operation accuracy in a generic way, but does not recognize contiguous sequences. [30, 31] do not address privatization and [18] does it only based on specific algorithm recognition. Our analysis is generic and was used uniformly to the parallelization of loops *EXTEND_do400*, *FPTRAK_do300*, *GETDAT_do300* from *TRACK*, *DKZMH_do60*, *PP_do100*, *SUBPP_do140* and *ACTFOR_do240*. [18] cannot solve the privatization problems of the first four, and it solves the last three using two algorithm recognition methods.

3, 4, 5, 6. [30, 31] introduced the idea of evolution and a recurrence model that produces distance ranges. [11] extracts ranges from array indices as well as from predicates based on affine expressions. We believe that the VEG graph representation makes it easier to express aggregated evo-

		Gupta et al [11] PACT'99	Lin, Padua [18] CC'00	Wu, Padua [30, 31] ICS'01-LCPC'01	Our Framework
1	Problems Solved	Privatization, Data Dependence	Privatization, Some Data Dependence	Data Dependence	Privatization, Data Dependence, Dataflow
2	Method	Memory Reference Analysis	Algorithm recognition (3)	Monotonic evolution	Memory Reference Sequence Classification
3	Recurrence Model	Implicit	Implicit: DDG	Explicit: evolution	Explicit: evolution graph
4	Multi-variable	Not specified	No	No	Yes
5	Distance Ranges	Yes	No	Yes	Yes
6	Conditional Ranges	Range extraction	No	No	Range extraction and tracing
7	Mem. Ref. Type	Generic	Single indexed	Not defined	Generic
8	Interprocedural	Yes	No	No	Yes
9	Pushback Parallelization	No	Yes (restrictive)	No	Yes (more general)

Table 4. Comparison to recent work on memory referenced through recurrences without closed forms.

lutions by associating them with graph paths. These paths contain explicit evolution and control information (by using GSA). The VEG can model recurrences defined using multiple variables, unlike previous representations that rely on the statement-level pattern $i = i + exp$. The VEGs are pruned based on ranges extracted from conditional values, which leads to closer value ranges. The static parallelization of loop *EXTEND_do400* can only be decided on this pruned graph, and has not been reported before.

7, 8. [18, 30, 31] require that arrays be unidimensional and that the index expression consist of exactly the recurrence variable. The recurrence variable cannot appear in the loop text except for the recurrence statements and as an array index. Our framework is more flexible: we analyze partially aggregated generic memory descriptors that represent the reference pattern in a single statement, an inner loops or a whole subprogram uniformly. Loop *DKZMH_do60*, and loops *EXTEND_do400* and *FPTRAK_do300* reference memory in inner loops and via subroutines; some arrays are two-dimensional; in one case array elements are seen as scalars inside a called subprogram; in a few cases, the recurrence variable appears in the bounds of an inner loop, while the actual array index expression is the loop index.

9. We present the parallelization of *pushback sequences* in a more general case than the one presented in [18] where the important loops *EXTEND_do400*, *FPTRAK_do300* and *GETDAT_do300* from *TRACK* could not be parallelized.

Our work also led to improvements to the range techniques presented by [3, 4], and to the GSA path technique presented by [28].

6. Limitations and Future Work

At this point we only support one evolution type within a single graph. This allows us to compose evolutions along paths by performing simple range arithmetic. We are planning to investigate the need for an evolution graph that contains multiple types of evolutions and the algorithmic complications involved.

For now, we treat arrays as scalars. We are planning to investigate the use of array dataflow information produced by MCA to create more expressive value evolution graphs.

We are also looking into further applications of value evolution graphs to the GSA path technique. Preliminary results show that, with minor improvement, we could solve more complex problems such as the parallelization of loop *INTERF_do1000* in code *MDG* from the *PERFECT* suite.

References

- [1] Z. Amarguella and W. L. Harrison, III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *ACM SIGPLAN '90 Conf. on Prog. Language Design and Implementation*, 1990.
- [2] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The Program Dependence Web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *ACM SIGPLAN '90 Conf. on Prog. Language Design and Implementation*, pp. 257–271, White Plains, N.Y., June 1990.
- [3] W. Blume and R. Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. *Proc. of Supercomputing '94, Nov. 1994*.
- [4] W. Blume and R. Eigenmann. Symbolic Range Propagation. TR 1381, Univ. of Illinois at Urbana-Champaign, Cntr for Supercomputing Res & Dev, Oct 1994.

- [5] D. Callahan. Recognizing and parallelizing bounded recurrences. In *1991 Workshop on Languages and Compilers for Parallel Computing*, number 589 in LNCS, pp. 169–185, Springer Verlag, Berlin.
- [6] S.-C. Chen and D. J. Kuck. Time and parallel processor bounds for linear recurrence systems. *IEEE Trans. on Computers*, 24(7):701–717, 1975.
- [7] A. L. Fisher and A. M. Ghuloum. Parallelizing complex scans and reductions. In *Proc. of the ACM SIGPLAN 1994 Conf. on Prog. Language Design and Implementation*, pp. 135–146. ACM Press, 1994.
- [8] M. P. Gerlek, E. Stolz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Trans. on Prog. Languages and Systems*, 17(1):85–122, 1995.
- [9] A. M. Ghuloum and A. L. Fisher. Flattening and parallelizing irregular, recurrent loop nests. In *Proc. of the 5-th ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog.*, pp. 58–67. ACM Press, 1995.
- [10] J. Gu, Z. Li, and G. Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Conf. on Supercomputing*, 1995.
- [11] M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. *Int. J. of Parallel Prog.*, 28(6):537–562, 2000.
- [12] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Prog. Lang. and Systems*, 2(1–4), 1993.
- [13] M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Trans. on Prog. Lang. and Systems*, 18(4):477–518, 1996.
- [14] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84–89, Dec. 1996.
- [15] J. Hoeffinger. *Interprocedural Parallelization Using Memory Classification Analysis*. PhD thesis, University of Illinois, Urbana-Champaign, Aug., 1998.
- [16] J. JàJa. *An Introduction Parallel Algorithms*. Addison-Wesley, Reading, MA, 1992.
- [17] S.-W. Liao, A. Diwan, R. P. B. Jr., A. M. Ghuloum, and M. S. Lam. SUIF explorer: An interactive and interprocedural parallelizer. In *Principles & Pract. of Parallel Prog.*, 1999.
- [18] Y. Lin and D. Padua. Analysis of irregular single-indexed array accesses and its application in compiler optimizations. In *Int. Conf. on Compiler Construction*, 2000.
- [19] Y. Lin and D. A. Padua. On the automatic parallelization of sparse and irregular fortran programs. In *1998 Workshop on Languages and Compilers for Parallel Computing*.
- [20] W. Pottenger and R. Eigenmann. Parallelization in the presence of generalized induction and reduction variables. TR 1396, Univ. of Illinois at Urbana Champaign, Cntr. for Supercomputing Res. & Dev., Jan. 1995.
- [21] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91*, pp. 4–13, Albuquerque, N.M., Nov. 1991.
- [22] L. Rauchwerger and D. A. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Trans. on Parallel and Distributed Systems*, 10(2), 1999.
- [23] L. Rauchwerger and D. A. Padua. Parallelizing WHILE Loops for Multiprocessor Systems. In *Proc. of 9th Int. Parallel Processing Symp.*, April 1995.
- [24] J. R. Reif and H. R. Lewis. Symbolic evaluation and the global value graph. In *4th ACM Symp. on Principles of Prog. Languages*, pp. 104–118, 1977.
- [25] S. Rus, J. Hoeffinger, and L. Rauchwerger. Hybrid analysis: static & dynamic memory reference analysis. *Int. J. of Parallel Prog.*, 31(3):251–283, 2003.
- [26] M. Spezialetti and R. Gupta. Loop monotonic statements. *IEEE Trans. on Software Engineering*, 21(6), 1995.
- [27] R. Triolet, F. Irigoien, and P. Feautrier. Direct parallelization of Call statements. In *ACM SIGPLAN '86 Symp. on Compiler Construction*, Palo Alto, CA, June 1986.
- [28] P. Tu and D. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proc. of the 9th ACM Int. Conf. on Supercomputing, Barcelona, Spain*, pp. 414–423, January 1995.
- [29] M. Wolfe. Beyond induction variables. In *ACM SIGPLAN '92 Conf. on Prog. Language Design and Implementation*, San Francisco, CA, June 1992.
- [30] P. Wu, A. Cohen, J. Hoeffinger, and D. Padua. Monotonic evolution: An alternative to induction variable substitution for dependence analysis. In *Proc. of the 2001 Int. Conf. on Supercomputing*, June 16–21, 2001, Sorrento, Napoli, Italy. ACM, 2001.
- [31] P. Wu, A. Cohen, and D. Padua. Induction variable analysis without idiom recognition: Beyond monotonicity. In *2001 Workshop on Lang and Compilers for Parallel Computing*, number 2624 in LNCS, pp. 427–441, Springer Verlag, Berlin.