

# Automatic Parallelization using the Value Evolution Graph <sup>\*</sup>

Silvius Rus, Dongmin Zhang, and Lawrence Rauchwerger  
{silviusr, dzhang, rwerger}@cs.tamu.edu

Parasol Laboratory  
Department of Computer Science  
Texas A&M University

**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 scalar and array values within a program. 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 indexed 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 Introduction

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 indices used to reference arrays. 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. Current compilers can easily answer questions such as “what is the array region written first in a loop”.

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

---

<sup>\*</sup> 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,

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 on loops that cannot be analyzed statically, e.g., loops with input dependent reference patterns [21]. Recently, Hybrid Analysis [22] has improved the accuracy and performance of optimizations by bridging the gap between static and run-time analysis (compile-time analysis partial results can be saved and used at run-time, 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 the arrays are indexed by functions that cannot be expressed as a closed form of the primary loop induction variable.

```

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   ENDF
10 ENDDO
11 sum = 0
12 DO i = old+1, p
13   sum = sum+
14     A(i)+B(i-old)
15 ENDDO

```

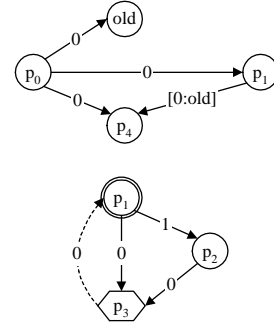
(a)

```

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   ENDF
10 ENDDO
11 p3 = γ(p1, p2)
12 p4 = η(p0, p1)
13 sum = 0
14 DO i = old+1, p4
15   sum = sum+
16     A(i)+B(i-old)
17 ENDDO

```

(b)



(c)

**Fig. 1.** (a) Arrays  $A$  and  $B$  are indexed with values from recurrences without and with closed forms respectively. (b) The same code after closed form substitution and in GSA. (c) Value Evolution Graphs for the recurrence on  $p$

Recurrences with closed forms are those in which the  $i$ -th term can be written as an algebraic formula of  $i$ . References to arrays using recurrences with closed forms can be meaningfully expressed using systems of inequations [25, 20, 13] or triplet-based notations [14, 9] containing the closed form terms and other symbolic values such as loop bounds. When a recurrence that has 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 an alternative analysis

technique because we cannot use the symbolic calculus algebra we use with closed form recurrences.

We propose to model the data flow of the recurrences without closed form using a new data structure, the *Value Evolution Graph* (VEG) and use it to extract parallelization information. Array  $B$  in Fig. 1 (a) is accessed through 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 it is indexed by an analytical function of the loop index. However, there is no such function for  $p$  (the index of  $A$ ), which is defined by a recurrence without a closed form. Fortunately, data dependence analysis does not require the existence of closed form solutions, but rather proofs of relations between the index sets corresponding to different iterations. For instance, in order to prove array  $A$  independent, we need to show that statements 6 and 8 are independent. 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 prove that the set of all the values of the recurrences does not intersect  $[1 : old]$  by means of graph search operations on the VEG displayed in Fig. 1(c).

### Our Contribution

(a) We propose the *Value Evolution Graph* that can represent the flow of scalar and array values within a program. The Value Evolution Graph is pruned based on control dependence and predicate extraction, and produces tighter value ranges than abstract interpretation methods.

(b) 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.

(c) By integrating the VEG in our memory classification analysis we have been able to accurately classify memory access sequences and use them to improve the coverage of important analysis techniques, e.g., data dependence analysis, privatization.

(d) 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.

## 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 one<sup>1</sup> (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

---

<sup>1</sup> We only address first order recurrences in this paper.

*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  $\phi$ -nodes.  $\gamma$  nodes merge two values on different forward control flow paths.  $\mu$  nodes merge a loop back value with the loop incoming value.  $\eta$  nodes merge the outcome of a loop with the value before the loop. We extended the GSA representation to interprocedural contexts in a way similar to [16]. 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  $\mu$  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  $\mu$  and *input* nodes are the only places where values can flow into a VEG. Values can flow out of the VEG through  $\mu$  nodes only. *Back* nodes represent the last value within a VEG (shown as hexagons). They are used to solve recurrences.

**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 composed along a path symbolically. For instance, the evolution along path  $p_1 \rightarrow p_2 \rightarrow p_3$  is an additive evolution with value  $1 + 0 = 1$ . 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  $p_1 \rightarrow p_2 \rightarrow p_3$ . Each edge is also labeled with a predicate extracted from the associated GSA gate. For instance, edge  $p_1 \rightarrow p_2$  is labeled with predicate  $A(i).GT.0$ . If no GSA gate is associated with an edge, we label it *.TRUE.*. Predicates are not displayed in to improve the clarity of the presentation.

**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  $\phi$  definition has the same number of incoming edges as the number of  $\phi$  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 *.FALSE. : .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  $\mu$  node to all nodes that may carry evolutions to the next iteration. We draw an edge from the value of the  $\mu$  node to the corresponding value on the left hand side of the corresponding  $\eta$  definition, and we label it with the aggregated evolution of the inner recurrence. Fig. 1(c) shows such an edge between  $p_1$  (a  $\mu$  node in the inner recurrence  $\{p_1, p_2, p_3\}$ ) and  $p_4$ . The range  $[0 : old]$  is a result 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

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 $\mu$ node		
$b_2 = \eta(b_0, b_1)$	$b_1 \rightarrow b_2$	Default	Loop effect
	$b_0 \rightarrow b_2$	Default	Identity
CALL sub( $b_1 \rightarrow b_2$ )	$b_1 \rightarrow b_2$	Default	<i>sub</i> effect

**Table 1.** Extracting evolutions from the program.

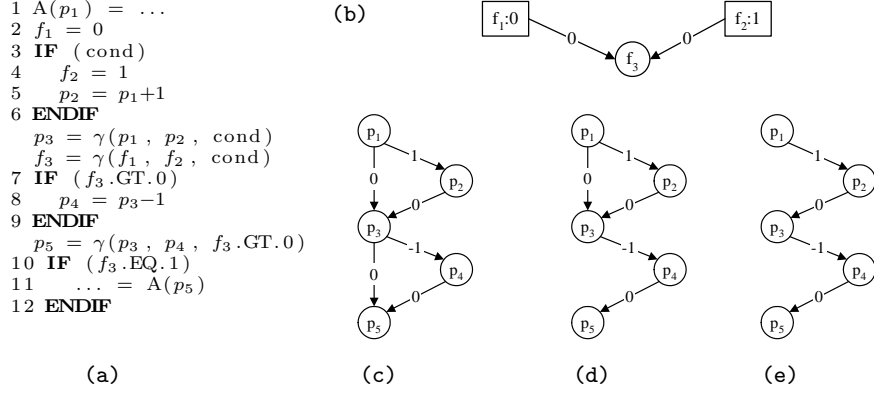
evolutions of the *OUT* actual arguments (and global variables) as functions of *IN* actual arguments (and global variables). In the last line in Table 1,  $b_2$  and  $b_1$  are the *OUT* and *IN* arguments respectively.

The VEGs are built in a single bottom-up traversal of the whole program. We precompute the shortest and longest paths between every  $\mu$  and *input* node and every other node and use these measures to solve queries such as recurrence *step* in  $O(1)$  time. If every node is reachable from exactly one  $\mu$  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  $\phi$  nodes in the program. If more than one  $\mu$  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 our queries are implemented using shortest path algorithms. Since all 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 1. This information can be used to prove that the write pattern on array  $A$  at statement  $\delta$  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.



**Fig. 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.

**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 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 analysis techniques.

## 2.4 VEG Logic Inference and Conditional Pruning

We can prune a VEG by removing certain edges that cannot be taken 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 [26] describes how control dependence relations can be used to disambiguate the flow of values at  $\gamma$  gates. It 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 [26] by using the VEG to trace back ranges extracted from 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 [26], leads to more accurate ranges than the abstract interpretation method used in [3], and can solve classes of problems that [27] cannot.

### 3 VEG Enabled Memory References Analysis

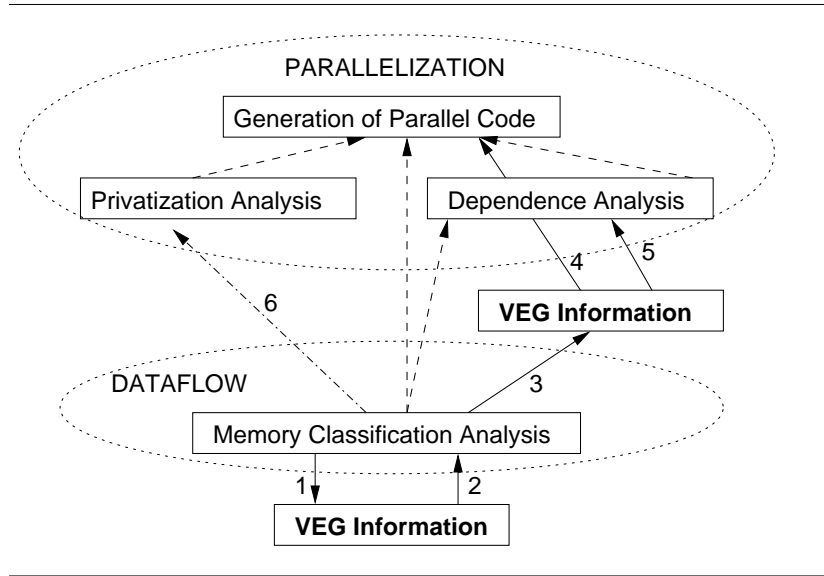
**Memory Reference Classification.** Memory Classification Analysis (MCA) [14] is a general dataflow technique used to perform data dependence tests and privatization analysis, but which is also usable in any optimization that requires dataflow information, such as constant propagation. For a given program 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)*.<sup>2</sup> We perform MCA in a single bottom-up traversal of the program by applying simple rules that aggregate 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. The symbolic manipulation within MCA relies on the existence of closed forms for induction variables. We will show that in certain cases we can perform MCA even in the absence of closed forms.

**Memory Reference Sequences.** The write to  $A$  in Fig 1, although indexed by a recurrence without closed forms, can be described by  $[1 : p_4]$  since it is written contiguously within these bounds. A memory reference sequence is *contiguous in a loop* if it is contiguous within every iteration and, for any iteration  $i$ , its image over iterations  $1 - i$  is contiguous. It is *increasing in a loop* if

<sup>2</sup> 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.

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

**Table 2.** Uses of sequence classification to the parallelization at the outer level of a nest of two loops.



**Fig. 3.** Integration of the information produced by the VEG in a parallelizing compiler.

every access index in iteration  $i + 1$  is strictly larger than any index in iterations  $1 - i$ . It is *consecutive in a loop* if it is both contiguous and increasing in the loop. These definitions can be extended to strided memory access. To prove a sequence contiguous, we show that *on all control flow paths within each iteration*, the step of induction variable is smaller or equal to the span of the memory reference. To prove a sequence increasing, we show that *on all control flow paths within each iteration*, the step of induction variable is larger or equal to the span of the memory reference.

### 3.1 Applications: Compiler Optimizations

Fig. 3 presents a general view of the use of VEG information to various analysis crucial to effective automatic parallelization. Table 2 presents the use of memory reference sequence classification to the parallelization at the outer level of a loop nest containing two loops.

**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 therefore 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**. This leads to more opportunities for removing memory related dependences through privatization. This corresponds to edges 2 and 6 in Fig. 3, and to row 1 in Fig. 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 sequence  $d$  that includes them and is increasing in **Outer**, then no cross iteration data dependences can exist. This corresponds to edges 3 and 5 in Fig. 3, and to row 2 in Fig. 2.

**Recognition and Parallelization of Pushbacks.** When a loop contains array references through induction variables, its parallelization is conditioned by: (i) there should be no data dependences between iterations of the loop except those involving the induction variable, (ii) 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 [15]. [18] presents the parallelization of loops that contain the pattern  $p = p+1; A(p) = \dots$  and where  $p$  and  $A$  do not appear anywhere else in the loop body, using the “array-splitting” technique.

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. We define a *pushback sequence* as a sequence of consecutive  $WF$  reference sets.  $WF$  is more general than *write* (covered reads are allowed). The  $WF$  set is computed accurately by the VEG improved MCA and thus qualifies more loops for parallelization. By working on aggregated reference sets, we can qualify more loops as pushbacks through the VEG-enhanced MCA, while previous approaches are bound to statement-level pattern recognition.

## 4 Implementation Details and Experimental Results

**Integration with Hybrid Analysis in Polaris.** The Hybrid Analysis framework [22] integrates compile-time and run-time analysis of memory reference patterns. It performs dataflow analysis by implementing the MCA algorithm using partially aggregated RT\_LMAD memory reference descriptors, and uses its results to parallelize loops (Fig. 4). Privatization, data dependence, and parallel code generation problems such as copy-in and last value assignment are mapped into equations containing the  $RO$ ,  $WF$ , and  $RW$  per-iteration descriptors (as RT\_LMADs). These operations rely heavily on VEG information, such as step,

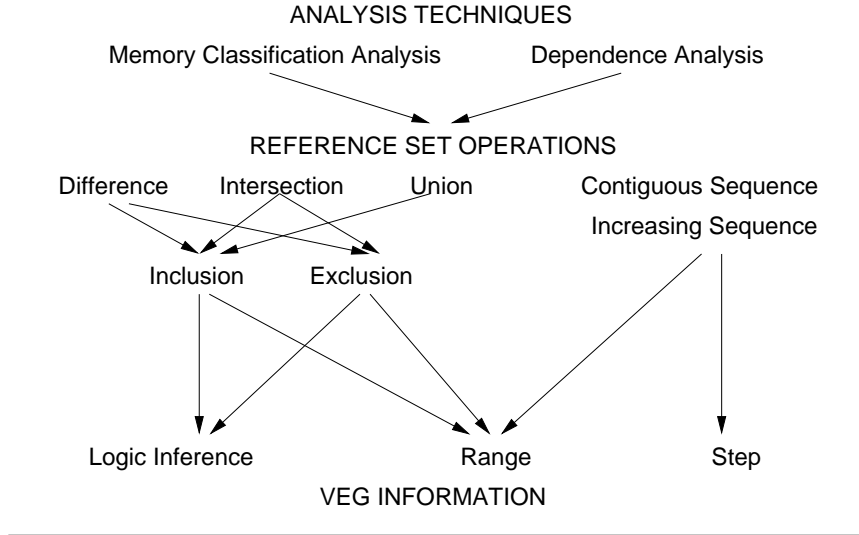


Fig. 4. Implementation of memory reference analysis using the VEG.

range, or logical inferences. The RT\_LMAD operations:  $\cap$ ,  $-$  and  $\cup$  (Fig. 4) rely on two relational operators, set inclusion and set exclusion. In addition to the range information, the more accurate GSA paths produced while pruning the VEG are used to prove predicate implication, which is used extensively in the inclusion and exclusion operators. For instance, in the case in Fig.1(b), in order to classify the references on array  $A$  across statements 6–8, we use the VEG to intersect the  $RO$  descriptor  $i$  with the  $WF$  descriptor  $p_2$ . The VEG produces the range  $[1 : old]$  for  $i$  and  $[old + 1 : 2 * old]$  for  $p_2$ . Since these ranges are found disjoint (exclusive), the intersection ( $RW$ ) is empty.

Table 3 presents a set of loops parallelized using VEG-based memory reference analysis. 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 VEG-based analysis can also parallelize a large number of loops in which memory is referenced based on recurrences with closed forms, we only show here the loops that contain recurrences without closed forms and thus could not be solved previously.

Seven out of the eleven parallelized loops are *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 [23]. We are not aware of any 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).

Program	Loop	Seq. %	Description
TRACK	EXTEND_do400	15-65	CP, CW, stack lookup/update in inner loop
	FPTRAK_do300	4-50	CP, stack lookup in inner loop
	GETDAT_do300	1-5	CP, CW, stack lookup/update in inner loop
P3M	PP_do100	52	CW in inner loops – privatization
	SUBPP_do140	9	CW in inner loops – privatization
BDNA	ACTFOR_do240	29	Index array range using VEG – privatization
MDLJDP2	JLOOPB_do20	12	CP
ADM	DKZMH_do60	6	CW in inner loops – privatization
QCD	QQQLPS_do21	< 1	CP
DYFESM	SETCOL_do1	< 1	CP
HYDRO2D	WNFLE_do10	< 1	CP

**Table 3.** Loops parallelized using the VEG. ADM, BDNA, DYFESM, QCD, and TRACK are from the PERFECT benchmark suite, MDLJDP2 and HYDRO2D from SPEC92, and P3M from NCSA. CP = Conditional Pushback, CW = Contiguous Write.

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. Then, these values 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 in *P3M/PP\_do100*.

## 5 Related Work

Automatic recognition and classification of general recurrences was discussed extensively in [1, 24, 27, 7], and their parallelization was presented in [5, 4, 6]. [12, 19, 8, 11] extend the analysis of memory indexed by irregular recurrences, but do not address most of the cases we focus on. Let us follow (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 [10, 17, 28, 29].

**1, 2.** We introduce a single technique that covers all the problems solved by [17, 28, 29], has wider applicability, and, additionally, builds generic array dataflow information. [10] uses monotonic information to improve memory reference set operation accuracy in a generic way, but does not recognize contiguous sequences. [28, 29] do not address privatization and [17] does it only based on specific algorithm recognition.

**3, 4, 5, 6.** [10] extracts ranges from array indices as well as from predicates based on affine expressions. [28, 29] introduced the idea of evolution and a recurrence model that produces distance ranges. We believe that the VEG paths represent the evolution and control information more explicitly. The VEG can model recurrences defined using multiple variables, unlike previous representa-

	Gupta et al [10] PACT'99	Lin, Padua [17] CC'00
	Wu, Padua [28, 29] ICS'01-LCPC'01	Our Framework
1 <b>Problems Solved</b>	Privatization, Data Dependence	Privatization, Some Data Dependence
	Data Dependence	Privatization, Data Dependence, Dataflow
2 <b>Method</b>	Memory Reference Analysis	Algorithm recognition (3)
	Monotonic evolution	Memory Reference Sequence Classification
3 <b>Recurrence Model</b>	Implicit	Implicit: DDG
	Explicit: evolution	Explicit: evolution graph
4 <b>Multi-variable</b>	Not specified	No
	No	Yes
5 <b>Distance Ranges</b>	Yes	No
	Yes	Yes
6 <b>Conditional Ranges</b>	Range extraction	No
	No	Range extraction and tracing
7 <b>Mem. Ref. Type</b>	Generic	Single indexed
	Not defined	Generic
8 <b>Interprocedural</b>	Yes	No
	No	Yes
9 <b>Pushback Parallelization</b>	No	Yes (restrictive)
	No	Yes (more general)

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

tions that rely on the statement-level pattern  $i = i + exp$ . The VEG conditional pruning is a new feature.

**7, 8.** Previous approaches generally 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.

**9.** We can parallelize *pushback sequences* in a more general case than [17]. None of the other techniques could parallelize the loops from *TRACK* in Table 3.

## 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 arith-

metic. 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 Conference on Programming Language Design and Implementation*, pages 283–295, White Plains, N.Y., June 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 Conference on Programming Language Design and Implementation*, pages 257–271, White Plains, N.Y., June 1990.
3. W. Blume and R. Eigenmann. Symbolic Range Propagation. Technical Report 1381, Univ of Illinois at Urbana-Champaign, Cntr for Supercomputing R&D, 1994.
4. D. Callahan. Recognizing and parallelizing bounded recurrences. In *1991 Workshop on Languages and Compilers for Parallel Computing*, number 589 in Lecture Notes in Computer Science, pages 169–185, Santa Clara, Calif., Aug. 1991. Berlin: Springer Verlag.
5. S.-C. Chen and D. J. Kuck. Time and parallel processor bounds for linear recurrence systems. *IEEE Transactions on Computers*, 24(7):701–717, 1975.
6. A. L. Fisher and A. M. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 135–146. ACM Press, 1994.
7. M. P. Gerlek, E. Stolz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, 1995.
8. A. M. Ghuloum and A. L. Fisher. Flattening and parallelizing irregular, recurrent loop nests. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 58–67, Santa Barbara, CA, 1995. ACM Press.
9. J. Gu, Z. Li, and G. Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 47. ACM Press, 1995.
10. M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. *International Journal of Parallel Programming*, 28(6):537–562, 2000.
11. R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1–4):135–150, 1993.
12. M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 18(4):477–518, 1996.

13. 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, December 1996.
14. J. Hoeflinger. *Interprocedural Parallelization Using Memory Classification Analysis*. PhD thesis, University of Illinois, Urbana-Champaign, August, 1998.
15. J. JàJà. *An Introduction to Parallel Algorithms*. Addison–Wesley, Reading, Massachusetts, 1992.
16. 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 Practice of Parallel Programming*, pages 37–48, 1999.
17. Y. Lin and D. Padua. Analysis of irregular single-indexed array accesses and its application in compiler optimizations. In *International Conference on Compiler Construction*, pages 202–218, 2000.
18. Y. Lin and D. A. Padua. On the automatic parallelization of sparse and irregular fortran programs. In *Pocceedings of the 1998 Workshop on Languages, Compilers, and Run-TimeSystems for Scalable Computers (LCR98)*, pages 41–56, 1998.
19. W. Pottenger and R. Eigenmann. Parallelization in the presence of generalized induction and reduction variables. Technical Report 1396, Univ. of Illinois at UrbanaChampaign, Center for Supercomp. R&D, January 1995.
20. W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91*, pages 4–13, Albuquerque, N.M., Nov. 1991.
21. L. Rauchwerger and D. A. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, 1999.
22. S. Rus, J. Hoeflinger, and L. Rauchwerger. Hybrid analysis: static & dynamic memory reference analysis. *International Journal of Parallel Programming*, 31(3):251–283, 2003.
23. S. Rus, D. Zhang, and L. Rauchwerger. The value evolution graph and its use in memory reference analysis. In *13th Conference on Parallel Architecture and Compilation Techniques*, pages 243–254. IEEE Computer Society, 2004.
24. M. Spezialetti and R. Gupta. Loop monotonic statements. *IEEE Transactions on Software Engineering*, 21(6):497–505, 1995.
25. R. Triolet, F. Irigoien, and P. Feautrier. Direct parallelization of Call statements. In *ACM '86 Symp. on Comp. Constr.*, pages 175–185, Palo Alto, CA., June 1986.
26. P. Tu and D. Padua. Gated SSA–based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 9th ACM International Conference on Supercomputing, Barcelona, Spain*, pages 414–423, January 1995.
27. M. Wolfe. Beyond induction variables. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 162–174, San Francisco, Calif., June 1992.
28. P. Wu, A. Cohen, J. Hoeflinger, and D. Padua. Monotonic evolution: An alternative to induction variable substitution for dependence analysis. In *2001 ACM International Conference on Supercomputing*, pages 78–91, Sorrento, Italy.
29. P. Wu, A. Cohen, and D. Padua. Induction variable analysis without idiom recognition: Beyond monotonicity. In *2001 Workshop on Lang. and Compilers for Par. Computing*, pages 427–441, Cumberland Falls, KY.