A Hybrid Approach to Proving Memory Reference Monotonicity

Cosmin E. Oancea and Lawrence Rauchwerger

PARASOL Lab, Texas A & M University, College Station, 77843, USA
coancea@cse.tamu.edu and rwerger@cse.tamu.edu

Abstract. Array references indexed by non-linear expressions or subscript arrays represent a major obstacle to compiler analysis and to automatic parallelization. Most previous proposed solutions either enhance the static analysis repertoire to recognize more patterns, to infer array-value properties, and to refine the mathematical support, or apply expensive run time analysis of memory reference traces to disambiguate these accesses. This paper presents an automated solution based on static construction of access summaries, in which the reference non-linearity problem can be solved for a large number of reference patterns by extracting arbitrarily-shaped predicates that can (in)validate the *reference monotonicity* property and thus (dis)prove loop independence. Experiments on six benchmarks show that our general technique for dynamic validation of the monotonicity property can cover a large class of codes, incurs minimal run-time overhead and obtains good speedups.

Keywords: monotonicity, access summary, autoparallelization

1 Introduction

The emergence of multi-core systems as mainstream technology has brought automatic program parallelization back to the forefront. Classical dependence analysis, based on distance/direction vectors [1, 2], or on algorithms to solve exactly a system of integer (in)equations [11, 20], has achieved somewhat limited success for the class of small loop nests with linear access patterns.

For larger codes, several studies [4, 16, 19] have outlined the necessity of interprocedural data-flow analysis that: (i) summarizes array accesses to overcome array reshaping at call sites and to reduce the number of dependency tests, (ii) exploits control flow to either improve summary precision or to predicate optimistic results of statically undecidable access summaries, and that (iii) is capable of disambiguating non-linear array accesses, which fall outside Presburger arithmetic [10]. These issues were first investigated in Fortran codes, however the solutions that have been developed also apply to other languages, such as C++ and Java. In this paper we will present in some detail the issue of *non-linear accesses*, such as those using nonlinear indexing or subscript (index) arrays. These typically appear in programs with sparse data structures using index arrays and/or are an artifact of compiler transformations, such as induction variable substitution and/or multi-dimensional array reshaping at call sites, where merging summaries requires flattening the original array. Historically there have been two main approaches to the analysis of such non-linear, irregular memory reference patterns:

(a) *Static analysis* methods which attempt to prove loop independence at compile time. Solutions either (i) extend the library of recognizable access patterns and apply interprocedural inference of index-array value properties [14], or (ii) enhance the mathematical support with more refined symbolic ranges [6, 9], or more encompassing algebras, such as chains of recurrence [8], or extensions of Presburger arithmetic [21]. If the analysis fails, then the user may be asked to examine parallelism based on the irreducible result of the corresponding algebra [16, 21]. Static techniques are efficient (no overhead) but conservative and often ineffective in detecting parallelism.

(b) *Run-time Analysis* techniques which analyze the code memory references during program execution and decide if an optimization (e.g., parallelization) can be applied. Notable examples are the TLS (thread-level speculation) [22] and inspector/executor [23] techniques, which analyze dynamically memory reference traces to detect data dependencies. Run-time techniques are effective because they can extract most available parallelism, but exhibit significant overhead.

In this paper we present an automated solution rooted at a mid-point between the two classical directions: *Static* interprocedural analysis builds memory access summaries and extracts arbitrarily-shaped predicates, which can qualify loops as independent. When these predicates are too complex for the current symbolic analysis available to us then their evaluation is deferred until program *execution-time* where results are exact but overhead may be costly. Our technique represents a unified framework to optimize the tradeoff between static and dynamic analysis. We always prefer compile time analysis but we will complement it with minimal run-time analysis in order to successfully parallelize programs.

We have developed several techniques that can specialize the parallelization predicates in the most efficient manner. Instead of trying to extract exact conditions (necessary and sufficient) conditions for loop independence we generate predicates that represent only sufficient conditions for parallelization. The main contribution of this paper is a technique to extract predicates that can validate the assertion that memory references take strictly monotonic values (and are therefore independent). The motivation behind it is that in practice this specialized condition is easier to formulate than the general independence assertion and that in practice monotonic references are quite frequent.

For example, with the loop nest:

```
DO i = 1, N, 1
  DO j = 1, i, 1
    DO l = 1, 1 + i - j, 1
       XIJRSO(j + l - 1 + (i^2 - i)*M) = ...
 ENDDO ENDDO ENDDO
```

$$\textit{Output Independence} : \cup_{i=1}^{M}(S_i \cap \cup_{k=1}^{i-1} S_k) = \emptyset$$
$$\textit{where } S_i = [M*(i^2-i)+1,\ M*(i^2-i)+i]$$

an overestimate of the set of written locations of array XIJRSO in iteration i is interval $S_i = [M*(i^2-i)+1,\ M*(i^2-i)+i]$. Loop output independence requires that the writes of iteration i do not overlap with the writes of any iteration preceding i, which is formalized via equation $\cup_{i=1}^{N}(S_i \cap \cup_{k=1}^{i-1} S_k) = \emptyset$, where $\cup$ and $\cap$ denote set union and intersection, respectively. One can observe that the above (data-flow) equation is satisfied if S_i is strictly monotonic – i.e. the upper bound of S_i is less than the lower bound of the iteration $i+1$ summary S_{i+1}, for any i. This leads to the output independence predicate: $2 * M \geq 1$, which is relatively easier to formulate and verify at run-time.

The extracted predicates, including those testing monotonicity, are then (i) optimized via common mathematical support, (ii) factored into a sequence of sufficient loop-independence conditions ordered by their estimated run-time complexity, and (iii) evaluated at run time until one (possibly) succeeds. In practice we were able to often extract predicates of complexity $O(1)$ and $O(N)$ for polynomial and array indexing, respectively, which compares quite favorably (relative to overhead) to previous run-time techniques.

An evaluation on six PERFECT Club benchmarks [3] validates our statements by showing negligible run-time overhead for the predicate evaluation, as well as speed-ups between 1.54x and 3.72x, with an average of 2.24x on a commodity four-core system.

2 Preliminary Concepts

Our framework for analyzing non-linear accesses has two main stages: First, interprocedural data-flow analysis constructs an *exact* summary, named unified set reference (USR), of the read-only (RO), read-write (RW) and write-first (WF) array accesses. Loop independence is reduced to proving that an USR is empty. This stage has been named *hybrid analysis* [24], since run-time USR evaluation can be seen as a continuation of static analysis.

The second stage applies a top-down $USR = \emptyset$ factoring algorithm to extract sufficient predicates for parallelization, which are tested at run-time in the order of their complexity. If they all fail, an exact answer can be obtained via thread-level speculation – the LRPD test [7, 22] – or via USR run-time evaluation. This part has been named *sensitivity analysis* [25] because it models how parallelism depends on statically unavailable input parameters. Sections 2.1 and 2.2 give the gist of the two stages and the information needed to understand this paper.

2.1 Hybrid Analysis

Hybrid Analysis builds on *linear-memory access descriptors* (LMAD) [19], which are denoted $[\delta_1, .., \delta_P] \vee [\sigma_1, .., \sigma_P] + \tau$ and represent the set of points: $\{\tau + i_1 * \delta_1 + .. + i_P * \delta_P \mid 0 \leq i_k * \delta_k \leq \sigma_k, \forall k \in 1..P\}$. Being uni-dimensional, LMADs allow reshaping at call sites where the dimensions of the formal and actual parameter differ, but strides δ_k and spans σ_k also model "virtual" multi-dimensional accesses. Summaries are constructed by traversing the call and control dependency (CDG) graphs in reverse topological order, while within a CDG region nodes are traversed in program order. During this bottom-up parse, data-flow equations dictate how summaries are initialized at statement level, merged across branches,

```
SUMMARIZE($REG_1$; $REG_2$)
  $(WF_1, RO_1, RW_1) \leftarrow REG_1$
  $(WF_2, RO_2, RW_2) \leftarrow REG_2$
  $WF = WF_1 \cup (WF_2 - (RO_1 \cup RW_1))$
  $RO = (RO_1 - (WF_2 \cup RW_2)) \cup$
        $(RO_2 - (WF_1 \cup RW_1))$
  $RW = RW_1 \cup (RW_2 - WF_1) \cup$
        $(RO_1 \cap WF_2)$
  RETURN $(WF, RO, RW)$
```

(a) Consecutive Region Composition

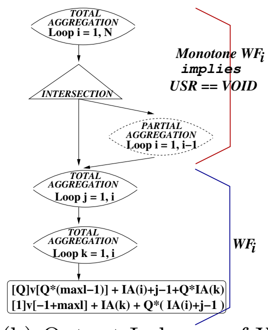
```
SUMMARIZE($REG_i$, $i = 1, N$)
  $(WF_i, RO_i, RW_i) \leftarrow REG_i$
  $WF = \bigcup_{i=1}^{N}(WF_i - \bigcup_{k=1}^{i-1}(RO_k \cup RW_k))$
  $RO = \bigcup_{i=1}^{N} RO_i - \bigcup_{i=1}^{N}(WF_i \cup RW_i)$
  $RW = \bigcup_{i=1}^{N}(RO_i \cup RW_i) - (WF \cup RO)$
  RETURN $(WF, RO, RW)$
```

(b) Loop Summary Aggregation

Fig. 1. Building WF, RO and RW summaries across consecutive regions and loops.

```
REAL X(Q, *)
DO i = 1, N, 1
   DO j = 1, i, 1
      a = 1.0D+00/(i+j)
      DO k = 1, i, 1
         maxl = k
         IF (k+(-i).EQ.0) maxl = j
         DO l = 1, maxl, 1
            X(IA(i)+j, IA(k)+l) = a + 1.0/(k+l)
            X(IA(k)+l, IA(i)+j) = a + 1.0/(k+l)
         ENDDO
      ENDDO ENDDO ENDDO
```

(a) Simplified INTGRL_do140 Code


TOTAL AGGREGATION Loop i = 1, N
INTERSECTION
PARTIAL AGGREGATION Loop i = 1, i–1
TOTAL AGGREGATION Loop j = 1, i
TOTAL AGGREGATION Loop k = 1, i
[Q]v[Q*(maxl–1)] + IA(i)+j–1+Q*IA(k)
[1]v[–1+maxl] + IA(k) + Q*(IA(i)+j–1)
Monotone WF_i implies USR == VOID
WF_i

(b) Output-Ind. USR of X

Fig. 2. Code and Output-Independence USR for `TRFD`'s Loop `INTGRL_do140`.

translated across call sites, composed between consecutive regions, and aggregated across loops. The latter two cases are illustrated in Figures 1(a) and 1(b). For example, the composition of a read-only region S_1 with a write-first region S_2 gives $RO = S_1 - S_2$, $RW = S1 \cap S2$ and $WF = S_2 - S_1$.

We employ an *exact* (DAG) representation, named USR, in which leafs are sets of LMADs and internal nodes are operations that cannot be precisely expressed in the LMAD domain. Internal nodes can represent unsimplifiable *set operations* ($\cup$, $\cap$, $-$), or control flow: *gates* predicating LMAD's existence, UN-translatable *call sites*, or *loops* that fail exact aggregation. As such, USRs represent memory references at a program level, in a scoped and closed-under-composition language. A loop of iteration summary (WF_i, RO_i, RW_i) has no flow/anti dependencies *iff*:

$$\{(\cup_{i=1}^{N} WF_i) \cap (\cup_{i=1}^{N} RO_i)\} \ \cup \ \{(\cup_{i=1}^{N} WF_i) \cap (\cup_{i=1}^{N} RW_i)\} \ \cup$$
$$\{(\cup_{i=1}^{N} RO_i) \cap (\cup_{i=1}^{N} RW_i)\} \ \cup \ \{\cup_{i=1}^{N}(RW_i \cap (\cup_{k=1}^{i-1} RW_k))\} = \emptyset.$$

Similarly, output independence can be modeled via equation:

$$\{\cup_{i=1}^{N}(WF_i \cap (\cup_{k=1}^{i-1} WF_k))\} = \emptyset.$$

The privatization and reduction parallelization transformations are also supported but are outside the scope of this paper.

Figure 2(a) and 2(b) show loop `INTGRL_do140` of the `trfd` benchmark and the associated output independence USR for array `X`. The WF_i USR is shown in the bottom part of the figure – loop nodes are introduced because exact LMAD

```
SG FACTOR($D$ : USR)
  // Output: $P$ s.t. $P \Rightarrow (D = \emptyset)$
  Case $D$ of:
    $q\#A$:      $P = \overline{q} \vee$ FACTOR($A$)
    $A \cup B$:  $P =$ FACTOR($A$) $\wedge$ FACTOR($B$)
    $A - B$:     $P =$ FACTOR($A$) $\vee$ INCLUDED($A$,$B$)
    $A \cap B$:  $P =$ FACTOR($A$) $\vee$ FACTOR($B$)
                     $\vee$ DISJOINT($A$,$B$)
    $\bigcup_{i=1}^{N}(A_i)$: $P = \bigwedge_{i=1}^{N}$ FACTOR($A_i$)
    $A \bowtie CallSite$: $P =$FACTOR($A$)$\bowtie CallSite$

SG DISJOINT_APPROX($C$ : USR, $D$ : USR)
  $(P_C, \lceil C \rceil) \leftarrow$ conditional LMAD overestimate of$A$
  $(P_D, \lceil D \rceil) \leftarrow$ conditional LMAD overestimate of$D$
  $P = P_C \vee P_D \vee$ DISJOINT_LMAD($\lceil C \rceil$, $\lceil D \rceil$)
```

(a) Extracting Predicates from $USR = \emptyset$

```
SG DISJOINT($C$ : USR, $D$ : USR)
  // Output: $P$ s.t. $P \Rightarrow (C \cap D = \emptyset)$
  $P =$ DISJOINT_HALF($C$, $D$) $\vee$
       DISJOINT_HALF($D$, $C$) $\vee$
       DISJOINT_APPROX($C$, $D$)

SG DISJOINT_HALF($C$:USR,$D$:USR)
  Case $C$ of:
    $q\#A$:     $P = \overline{q} \vee$ DISJOINT($A$, $D$)
    $A \cup B$: $P =$ DISJOINT($A$, $D$) $\wedge$
                    DISJOINT($B$, $D$)
    $A - B$:    $P =$ DISJOINT($A$, $D$) $\vee$
                    INCLUDED($D$,$B$)
    $A \cap B$: $P =$ DISJOINT($A$, $D$) $\vee$
                    DISJOINT($B$, $D$)
```

(b) Functions used by FACTOR

Fig. 3. Construction of an Arbitrarily-Shaped Independence Predicate.

aggregation fails over loop `k` due to the non-linear use of `IA(k)`. The output independence pattern is explicit in the upper part of the figure, where partial aggregation corresponds to the $\cup_{k=1}^{i-1} WF_k$ term. Even if LMAD aggregation would have succeeded for loop `k`, it would still fail partial aggregation due to `IA(i)`.

2.2 Sensitivity Analysis

Figure 3(a) and 3(b) give the gist of the top-down `FACTOR` algorithm. Inference on set-algebra properties guides a recursive construction of an arbitrary complex predicate. Its representation, named *sensitivity graph* (SG), similar to USR, is a DAG in which leafs contain predicate expression, while nodes represent either (i) logical operators – *or* ($\vee$) , *and* ($\wedge$) – or (ii) control flow – call sites, loops – across which predicates cannot be summarized.

For example, from $q\#A = \emptyset$, where q denotes the branch condition under which A is defined, we extract the predicate $\overline{q} \vee$ `FACTOR`(A). Sufficient conditions for $A \cap B = \emptyset$ are $A = \emptyset$ or $B = \emptyset$. Additional predicates for this equation are inferred in `DISJOINT_HALF` by examining the shape of A and B.

Whenever we reach LMAD leafs or when we encounter call site or loop nodes (that end the `DISJOINT`-based logical inference), `DISJOINT_APPROX` conservatively flattens the problem to the LMAD domain. We have found most useful to represent an overestimate of USR D as a pair $(P_D, \lceil D \rceil)$, where P_D is a predicate under which D is empty, while $\lceil D \rceil$ is an LMAD-overestimate of D. Building on the LMAD intersection and subtraction algorithms [19], `DISJOINT_LMAD` extracts the conditions for (i) valid projection on number-of-strides dimensions and for (ii) the corresponding pairs of resulting 1D-LMADs to be disjoint.

The result predicate is brought to disjunctive normal form in which terms are sorted in increasing order of their estimated complexity. Code is automatically generated to cascade these tests, and to implement conditional loop parallelization. Redundancy is optimized by hoisting calls to these tests inter-procedurally at the highest dominator point where all the input values are available.

3 Extracting LMAD-Monotonicity Predicates

Section 3.1 answers the question “where and when is monotonicity tested?” Since our solution to non-linear accesses is intrinsically related to LMAD summarization, Section 3.2 establishes a uniform notation and discusses how LMADs are aggregated across loops. Our strategy applies an incremental effort to proving monotonicity: Section 3.3 presents a simple test to handle one quasi-linear LMAD, and Sections 3.4 and 3.5 treat the uni and multi-dimensional cases of non-linear LMADs with polynomial and array indexing, respectively. Finally, Section 3.6 discusses the overall design of the monotonicity test, and possible extensions.

3.1 When To Apply Monotonicity Tests?

We try to extract monotonicity predicates *wherever* the `FACTOR` algorithm encounters the equation $\cup_{i=1}^{N}(A_i \cap (\cup_{k=1}^{i-1} A_k)) = \emptyset$, for an arbitrary USR A_i. One can observe that if an overestimate of A_i, named $\lceil A_i \rceil$, is strictly monotonous, under some definition of set order ”$>$”, then the above equation holds, i.e. if $\lceil A_i \rceil > \lceil A_{i-1} \rceil$ for any $2 \leq i \leq N$, then by induction $\lceil A_i \rceil > \lceil A_j \rceil$, for any $1 \leq j \leq i-1$, and hence the intersection $A_i \cap (\cup_{k=1}^{i-1} A_k)$ is empty for any i. We also check the monotonicity of $RO_i \cup WF_i \cup RW_i$ since this is a sufficient condition for the independence of the loop of index `i`. The answer to the *where* question is thus pattern matching at USR level; this is very different from solutions based on code pattern recognition in that our matching is less a consequence of the problem and more an artifact of the proof, i.e. irreducible data-flow equations.

We try to extract monotonicity predicates *whenever* an LMAD overestimate of A_i, named L_i, can be computed, but the aggregation of L_i over the loop of index i is either inexact or fails in the LMAD domain. The latter is the sign of an irregular access, either non-linear: L_i exhibits subscripted or polynomial or exponential indexing, or quasi-linear: the same index appears in two LMAD dimensions, or division operations occlude linearity. Either way, `FACTOR` is likely to fail. For example, assuming an LMAD overestimate L_i exists for WF_i in Figure 2(b), L_i would fail partial aggregation over the outermost loop of index `i` due to the indirect access `IA(i)`. `FACTOR` can only try to prove that WF_i is empty but since this is not the case the result will be the predicate `false`.

3.2 Problem Statement, Notation and LMAD Loop Aggregation

The problem addressed is extracting predicates that are sufficient conditions for the satisfiability of the USR equation $X = \cup_{i=1}^{N}(D_i \cap (\cup_{k=1}^{i-1} D_k)) = \emptyset$, where $i = 1, N$ is the range of normalized loop L. We denote with $A_i = \{A_i^1, A_i^2, ..., A_i^M\}$ a list of LMADs that overestimates D_i (as a set of points). We recall that:
$E = [\delta_1, \delta_2, .., \delta_P] \vee [\sigma_1, \sigma_2, .., \sigma_P] + \tau$ denotes a P-dimensional LMAD representing the set of points: $\{i_1 * \delta_1 + i_2 * \delta_2 + ... + i_P * \delta_P + \tau \mid 0 \leq i_k * \delta_k \leq \sigma_k, 1 \leq k \leq P\}$, where δ_k and σ_k are called the stride and span of dimension k, respectively.

If for any k, $\delta_k > 0$, then interval $[\tau, \tau + \sum_{j=1}^{P}(\sigma_j)]$ overestimates E. A_{i+1}^k, τ_{i+1} are obtained by replacing i with $i+1$ in A_i^k and τ_i.

To compute an LMAD overestimate $\lceil X \rceil$ of an USR X we apply a recursively defined operator on the USR domain. In the *top down* parse the operator disregards node B in terms such as $C - B$, $C \cap B$, while in the return (*bottom-up*) parse it translates, aggregates and adds (union) the encountered LMAD leafs over call site, loop and $\cup$ nodes, respectively. At the LMAD level, aggregation over loop L succeeds as long as the resulting strides/span/offset are loop invariant (since the USR language is scoped). The remainder of this section demonstrates at an intuitive level how LMAD's *overestimate* and *exact* aggregation works.

```
     DO i = 1, N                          DO i = 1, N
       DO j = 1, N                             DO j = 1, i
$S_1$ :    Y(j+i*N) = ...                         Y(j+i*N) = ...
     ENDDO ENDDO                          ENDDO ENDDO
```

With both loop nests above, the LMAD describing the write access of Y at statement S_1 is the LMAD point $(j-1)+N*(i-1)$. Aggregation over the *left-hand side loop* of index j creates a new dimension of stride $\delta_{new} = \tau_{j+1} - \tau_j = 1$ and span $\sigma_{new} = \tau_{j \leftarrow N} - \tau_{j \leftarrow 1} = N-1$, since $j \in [1, N]$, giving LMAD $[1] \vee [N-1] + N*(i-1)$. Aggregation over loop i similarly creates a new dimension of stride N and span $N^2 - N$, giving $[1, N] \vee [N-1, N^2 - N] + 0$. The left-hand side loop nest aggregation has been exact. Considering the *right-hand side loop* nest, aggregation over loop j gives $[1] \vee [i-1] + N * (i-1)$, since $j \in [1, i]$, and over loop i gives $[1, N] \vee [i-1, N^2 - N] + 0$. Since the USR language is scoped, exact aggregation over loop i fails, because loop-variant symbol i still appears in the LMAD. However, an overestimate can be computed by replacing i with its upper bound N, which gives LMAD $[1, N] \vee [N-1, N^2 - N] + 0$.

Intuitively, the strategy for testing monotonicity is to overestimate each LMAD dimension via intervals, and to formulate several kinds of monotonicity which all reduce to studying interval monotonicity; for example increasing monotonicity would correspond to the upper bound of the interval corresponding to iteration i being less than the lower bound of the interval corresponding to iteration $i+1$.

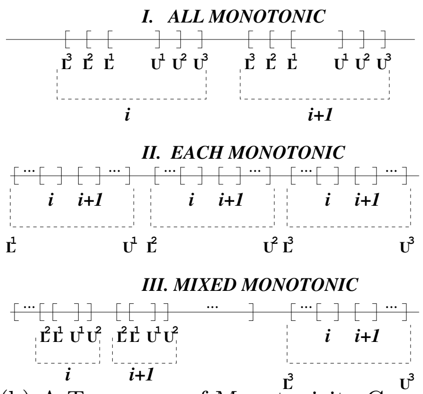
3.3 Quasi-Linear Case

We consider the case when the overestimate of USR D_i is *one* LMAD A_i. Observe that if overestimate L-aggregation of A_i succeeds and creates a new dimension, then necessarily the new stride is L invariant, and hence the access is strictly monotonic in the new dimension. Denoting with A the aggregated LMAD, a sufficient condition for the data-flow equation $X = \emptyset$ to hold is that A has non-zero strides and its "virtual" dimensions do not overlap. Checking non-overlap requires normalizing A – all strides/spans are made positive and dimensions are sorted according to strides – and checking that $\sum_{j=1}^{k-1} \sigma_j < \delta_k$, for any $2 \leq k \leq P$. When the values of δ_k, σ_k are not all known statically, an non-overlap predicate of complexity $O(1)$ is extracted and evaluated at run time.

While this test primarily handles quasi-linear access in which i appears in two dimensions, it also covers some non-linear cases. For example access $Y(j+N^2*i)$ in a two-level loops nest of indexes $i = 1, N$ and $j = 1, i^2$ gives after j-aggregation $[1] \vee [i^2 - 1] + N^2 * (i-1)$. Note that σ_1 is quadratic in i, but this does not

```
SE REDUCE_GT_0( expr )
  //Input:   an int-type expression
  //Output: P s.t. P ⇒ (expr > 0)
    (a, b, i, L, U, err) =
          FIND_SYMBOL(expr);
    // expr = a*i+b, L ≤ i ≤ U, i ∉ b
// P = (a ≥ 0 ∧ a*L+b>0) ∨ (a < 0 ∧ a*U+b>0)
    IF ( err ) THEN RETURN (expr > 0);
    ELSE RETURN
              [ REDUCE_GT_0(a+1) ∧
                REDUCE_GT_0(a*L+b) ] ∨
              [ REDUCE_GT_0(−a)  ∧
                REDUCE_GT_0(a*U+b) ];
```

(a) Symbolic Fourier-Motzkin Elimination



(b) A Taxonomy of Monotonicity Cases

Fig. 4.

impede i-aggregation because $\tau = (i-1) * N^2$ is linear in i and an upper bound for i^2 is known (N^2) leading to $[1, N^2] \vee [N^2 - 1, N^3 - N^2]$ which never overlaps.

Loop FTRVMT_do109 of the ocean benchmark was solved via such an $O(1)$ non-overlap test, where the L-aggregated LMAD has four symbolic dimensions.

3.4 Nonlinear, Unidimensional (1D) LMAD Case

We consider the case when the per-iteration summary is overestimated via a list of 1D LMADs $A_i = \{A_i^1 = [\delta_i^1] \vee [\sigma_i^1] + \tau_i^1, .., A_i^M = [\delta_i^M] \vee [\sigma_i^M] + \tau_i^M\}$, where all M strides and spans are positive. The intuitive idea is to overestimate each A_i^k to an interval $F_i^k = [\tau_i^k, \tau_i^k + \sigma_i^k]$ and to study monotonicity at interval level, where we denote the lower and upper bounds of F_i^k by $L_i^k = \tau_i^k$ and $U_i^k = \tau_i^k + \sigma_i^k$.

Figure 4(b) depicts three main cases: In the *all monotonic case*, overestimating the whole per-iteration summary A_i to an interval still forms a strictly monotonic sequence of intervals in i. This guarantees that the accesses of iteration i do not overlap with those of iteration $i+1$, for any i, and by induction, with any of an iteration other than i. More formally, predicate P_{all_mon} =

$$[\wedge_{i=1}^{N-1}(\mathtt{MAX}_{k=1}^{M}(U_i^k) < \mathtt{MIN}_{k=1}^{M}(L_{i+1}^k))] \quad \vee \quad [\wedge_{i=1}^{N-1}(\mathtt{MAX}_{k=1}^{M}(U_{i+1}^k) < \mathtt{MIN}_{k=1}^{M}(L_i^k))]$$

satisfies our data-flow equation $X = \emptyset$, where the first and second terms are the conditions for strictly increasing and decreasing monotonicity, respectively.

In the *each monotonic case*, each LMAD in list A_i has the property that its accesses form a strictly monotonic, hence non-overlapping, sequence in the iteration space: i.e. the interval sequence F_i^k is strictly monotonic in i. Additionally, maximizing the M intervals over the range of i gives M pairwise disjoint intervals F^k. This is a sufficient condition that satisfies equation $X = \emptyset$. Formally, the predicate for the first condition is $PRED_{each\_mon}$ =

$$[\wedge_{i=1}^{N-1}(U_i^1 < L_{i+1}^1) \quad \vee \quad \wedge_{i=1}^{N-1}(U_{i+1}^1 < L_i^1)] \wedge ... \wedge$$
$$[\wedge_{i=1}^{N-1}(U_i^M < L_{i+1}^M) \quad \vee \quad \wedge_{i=1}^{N-1}(U_{i+1}^M < L_i^M)].$$

The "additional" part requires computing the lower and upper bounds for each interval F^k. We use the assumed monotonicity of F_i^k to exactly express the lower and upper bounds of F^k: $L^k = \text{MIN}(L_1^k, L_N^k)$ and $U^k = \text{MAX}(U_1^k, U_N^k)$. We create a predicate $PRED_{sc}$ as a sensitivity graph node, named *sorted check*, whose run-time semantic is: sort the M intervals $[L^k, U^k]$ in increasing order of L^k and return $\wedge_{k=1}^{M-1}(U^k < L^{k+1})$. Thus $PRED_{each\_mon} \wedge PRED_{sc}$ is another sufficient condition that satisfies the loop-independence data-flow equation $X = \emptyset$.

Finally, in the *mixed monotonic* case, any combination of the previously discussed cases would satisfy $X = \emptyset$. This leads to too many possible solutions and hence we restrict our implementation to the first two "extreme" patterns.

While the predicates extracted so far will prove monotonicity at run-time, they can be significantly optimized via symbolic Fourier-Motzkin elimination of loop-variant symbols of known bounds, as depicted in Figure 4(a). In the *each monotonic* case, we apply it to eliminate L-variant symbols such as i from the $U_i^k < L_{i+1}^k$ and $U_{i+1}^k < L_i^k$ inequations. In the *all monotonic* case, we try to solve $O(M^2)$ inequations, the resulting predicate being:

$$P_{all\_mon\_opt} = \wedge_{1 \le k,j \le M}(U_i^j < L_{i+1}^k) \ \vee \wedge_{1 \le k,j \le M}(U_{i+1}^j < L_i^k).$$

Successful elimination gives a loop-invariant predicate of $O(1)$ run-time complexity for both cases. Otherwise, in the *all monotonic* case, we use the non-optimized predicate, which has $O(N * M)$, rather than $O(N * M^2)$ complexity.

Array `XIJRS0` from loop `OLDA_do300` of `trfd` benchmark exhibits a quadratic per-iteration summary formed by three LMADs ($M = 3$), that gives an $O(1)$ independence predicate equivalent to[1]: $\texttt{morb}+\texttt{morb}^2+(-1)*\texttt{num}+(-1)*\texttt{num}^2 \le 0$.

3.5 Nonlinear Multi-Dimensional LMAD Case with Index Arrays

When A_i contains P-dimensional LMADs of similar shape – i.e. all strides match across $A_i^k, k \in [1, M]$, then we study monotonicity dimension wise. This requires to split τ across dimensions, and to extract a well-formedness predicate P_{wf}.

For example, LMAD $[1, Q] \vee [i-1, Q*(i-1)] + i + Q*(i+1)$ can be interpreted as a 2D array access that covers intervals $[i, 2*i-1]$ and $[i+1, 2*i]$ on the first and second direction, respectively. The decomposition is valid as long as dimensions do not overlap, hence P_{wf} is $2*i-1 < Q$, which, since $i \in [1, N]$, becomes $2*N-1 < Q$ (by Fourier Motzkin). τ splitting is implemented via a heuristic that we do not discuss here, but which may fail when there is not enough confidence that the guess is correct. When splitting fails we approximate an LMAD with the interval $[\tau, \tau+\sum_{j=1}^{P}\sigma_j]$ and apply the technique of Section 3.4.

Intuitively, with the *each monotonic case*, each LMAD monotonicity is established by showing monotonicity in *one* dimension. The *all monotonic case* is more complex because the generation of a sufficient condition for increasing monotonicity, requires that, for any LMAD, there exist at least one dimension in which its lower bound for iteration $i+1$ is greater than the iteration-i upper bounds of *all* LMADs. Formally, denoting by $[L_i^{t,j}, U_i^{t,j}]$ the interval overestimate

[1] Constant propagation could determine $morb = num$ and the decision can be static.

of dimension t of A_i^j, and

$$\begin{aligned} P_{each\_mon} &= \wedge_{j=1}^{M}\{\ \vee_{t=1}^{P}[\ \wedge_{i=1}^{N-1}(U_i^{t,j} < L_{i+1}^{t,j})\ \vee\ \wedge_{i=1}^{N-1}(U_{i+1}^{t,j} < L_i^{t,j})\ ]\ \} \\ P_{all\_mon} &= \wedge_{j=1}^{M}\{\ \vee_{t=1}^{P}[\ \wedge_{i=1}^{N-1}(U_i^{t,1} < L_{i+1}^{t,j}\ \wedge \ldots \wedge\ U_i^{t,M} < L_{i+1}^{t,j})\ \vee \\ &\qquad\qquad \wedge_{i=1}^{N-1}(U_{i+1}^{t,1} < L_i^{t,j}\ \wedge \ldots \wedge\ U_{i+1}^{t,M} < L_i^{t,j})\qquad]\ \}, \end{aligned}$$

the *all monotonic* predicate is $P_{all\_mon} \wedge P_{wf}$ and the *each monotonic* one is $P_{each\_mon} \wedge P_{sc} \wedge P_{wf}$, where we optimize by Fourier-Motzkin all inequations.

To handle complex index-array cases, we refine implementation to allow conditional LMAD aggregation based on assumed monotonicity of the index array that would otherwise hinder aggregation.

We demonstrate the approach on loop `INTGRL_do140` of `trfd` benchmark, where, for clarity, we show only the monotonically *increasing* case. The code and $\texttt{WF}_i$ USR are shown in Figures 2(a) and 2(b). To compute A_i we need to aggregate: $B_k^j = \{\ [\texttt{Q}] \vee [\texttt{Q} * (\texttt{maxl} - 1)] + (\texttt{IA}(i) + j - 1) + \texttt{Q} * \texttt{IA}(k),$

$$[1] \vee [\texttt{maxl} - 1] \quad + \texttt{Q} * (\texttt{IA}(i) + j - 1) + \texttt{IA}(k) \qquad \}$$

over loops of indexes $k \in [1, i]$ and $j \in [1, i]$. We assume that `IA(k)` is monotonic in k, i.e. $P_{IAmon}^{i,j} = \wedge_{k=1}^{i-1}\texttt{IA}(k) < \texttt{IA}(k+1)$, and extend this predicate over the range of i, giving $P_{IAmon} = \wedge_{i=1}^{N-1}\texttt{IA}(i) < \texttt{IA}(i+1)$. Using `IA`'s assumed monotonicity, the range of `IA(k)` in loop k becomes $[\texttt{IA}(1), \texttt{IA}(i)]$, and the range of `maxl` is $[1, i]$. Thus, overestimate aggregation over loops k and j succeeds:

$$\begin{aligned} A_i = \{\ & [1, \texttt{Q}] \vee [i-1, \texttt{Q} * (i - 1 + \texttt{IA}(i) - \texttt{IA}(1))] + \texttt{IA}(i) + \texttt{Q} * \texttt{IA}(1), \\ & [1, \texttt{Q}] \vee [i - 1 + \texttt{IA}(i) - \texttt{IA}(1)\ , \texttt{Q} * (i-1)] + \texttt{Q} * \texttt{IA}(i) + \texttt{IA}(1)\}. \end{aligned}$$

Projection on the first dimension gives intervals: $[\texttt{IA}(i), \texttt{IA}(i) + i - 1]$, and $[\texttt{IA}(1), \texttt{IA}(i) + i - 1]$; the second dimension is similar, but reversed. After simplification, the *all monotonicity* formula gives: $P_{all\_mon} = \wedge_{i=1}^{N}[\ \texttt{IA}(i+1) \geq \texttt{IA}(i) + i]$.

Finally, the well-formedness predicate that guarantees dimensions do not overlap is $P_{wf} = \texttt{IA}(N) + N - 1 < Q$, and implication-based reductions, such as $\texttt{IA}(i+1) \geq \texttt{IA}(i) + i \Rightarrow \texttt{IA}(i+1) > \texttt{IA}(i)$, optimizes away term P_{IAmon}. Overall, we obtain the output-independence predicate $P_{all\_mon} \wedge P_{wf}$, whose $O(N)$ runtime overhead is negligible against the $O(N^4)$ complexity of loop `INTGRL_do140`.

3.6 Overall Design and Possible Extensions

We try first the non-overlap test presented in Section 3.3 since it is the cheapest in terms of both compile and run time complexity – $O(1)$. If a predicate cannot be extracted this way, we study monotonicity in the more comprehensive form of Sections 3.4 and 3.5: If A_i contains LMADs of similar shape and dimension-wise projection succeeds, we apply the multi-dimensional test, otherwise LMADs are flattened and predicates are extracted via the $1D$ test. In practice we encountered only instances of the *all monotonic* cases.

Our solution evolves naturally: *First*, non-linearity is discovered as a consequence of exact aggregation failing on LMADs. *Second*, predicate extraction models a general form of monotonicity at interval level, and as such is transparent to the non-linearity shape, i.e., indirect access or polynomial indexing.

Finally, the extracted predicates are aggressively optimized: (i) symbolic Fourier-Motzkin elimination is applied on inequations, (ii) the assumed monotonicity refines ranges and (iii) implication-based invariants effectively filter the sensitivity graph SG, e.g., if $A \Rightarrow B$ then $A \wedge B \equiv A$. Denoting by N the number of iterations of the outermost loop, we typically extract predicates of run-time complexity $O(1)$ for polynomial and $O(N)$ for array indexing, where the $O(N)$ predicate is evaluated in parallel, giving *scalable* performance.

4 Related Work

Hoeflinger's ART test [13, 19] summarizes accesses via non-linear LMADs that allow strides to contain loop-variant symbols of the aggregated loop, such as the loop index. The LMAD corresponding to the independence test is aggregated across the to-be-analyzed loop and independence is tested by checking that LMAD dimensions do not overlap. We found that such aggressive aggregation is often too conservative. We diverge early in our design by keeping LMADs quasi linear, i.e., loop-invariant strides, and using USRs to allow exact summarization, and `FACTOR` to extract independence predicates. The test of Section 3.3 is similar with Hoeflinger's non-overlap test, the difference being that our LMAD algebra requires checking that aggregation creates a new dimension. While manual application of ART reported success on codes with exponential indexing, such as `tfft2`, that we do not cover, ART probably cannot handle the codes exhibiting index arrays and even quadratic indexing, such as `INTGRL_do140` and `OLDA_do300`. This would probably require an analysis similar to ours to prove loop-variant strides positive.

The test we described in Section 3.4 intuitively resembles Blume and Eigenmann's Range Test [5], which uses symbolic-range propagation and the monotonicity of a read/write pair of accesses to disprove non-independent direction vectors. The Range Test covers polynomial and some exponential indexing, but cannot handle index arrays (e.g., `INTGRL_do109`). Loop index appearing in different dimensions may also be a hindrance in some cases.

Lin and Padua give a code *pattern-matching* extension of the Range Test, named Offset-Length Test [14], that handles index-array accesses such as the ones in `INTGRL_do140`. The solution uses interprocedural analysis to verify invariants on the values stored in the index array and as such, to derive independence statically. While a static result is always desired, array properties could be impossible to establish statically, e.g., array read from a file. In contrast we extract a $O(N)$ predicate, whose run-time overhead is negligible given that the original loop has $O(N^4)$ complexity, and furthermore our predicate stems naturally from LMAD monotonicity rather than relying on code pattern matching.

Pugh and Wonnacott extend Presburger arithmetic with support for uninterpreted functions [21] to analyze dependencies of a pair of non-linear accesses (some control flow included as well). Wherever Presburger-like algebra cannot prove independence, the irreducible formula is presented to the user for validation. Several refinements, such as inductive simplification, extract simpler formulas that are sufficient independence conditions. The more complex

Loops with Non-Linear USRs from Several Perfect Benchmarks					
Benchmark	Bench Properties	Non-Linear Loop	Successful Pred Type	Weight	Pred OV
	SC = 98.5%	OLDA_do100	O-IND, $O(1)$, i^2	68.2%	0.003%
TRFD	RT-IW, PRIV,	OLDA_do300	F/O-IND, $O(1)$, i^2	26.8%	0.015%
	SLV, DLV	INTGRL_do140	O-IND, $O(N)$, IA_{mon}	3.4%	0.510%
	SC = 96.5%	MXMULT_do10	F/O-IND, $O(N)$, $\texttt{IA}_{mon}$	62.6%	0.076%
DYFESM	RT-IW, PRIV,	SOLXDD_do10	O-IND, $O(N)$, $\texttt{IA}_{mon}$	12.6%	0.001%
	RED, RT-RED	SOLVH_do20	F/O-IND, $O(N)$, $\texttt{IA}_{mon}$	12.1%	0.016%
		FORMR_do20	F/O-IND, $O(N)$, $\texttt{IA}_{mon}$	8.1%	0.692%
ARC2D	SC = 96.7%	YPENT2_do11	F-IND, $O(1)$	10.2%	0.084%
	PRIV, SLV	*et al.*	Non-Overlap		
	SC = 99.3%	INTERF_do1000	STATIC, NONE	91.7%	0%
MDG	PRIV, RED,	POTENG_do2000	STATIC, NONE	7.4%	0%
SPEC77	SC = 91.3%	SICDKD_do1000	F-IND, $O(1)$	4.1%	0.007%
	PRIV,RED,SLV		Non-Overlap		
OCEAN	SC = 88.8%	FTRVMT_do109	F-IND, $O(1)$	63.2%	0.075%
	PRIV,RED,SLV		Non-Overlap		

Table 1. Loop and Benchmark Level Properties. The last three columns describe (i) the type of non-linearity: polynomial indexing (i^2), indirect array ($\texttt{IA}_{mon}$), or index spanning multiple dimensions (Non-Overlap), (ii) the complexity of the predicate that proves flow/output independence (F/O-IND), (iii) the loop contribution to sequential coverage (*weight*) and (iv) the predicate overhead as percentage of the parallel loop run time.

index-array loop `INTGRL_do140` is not reported, and the quadratic indexed loops `olda_do100/300` are (only) observed to form monotonic indexing before induction variable substitution. Our test extracts parallelism from all reported loops.

Hall *et al.* use interprocedural summarization [12, 16] to give a comprehensive study of parallelism for the PERFECT Club and SPEC2000 benchmarks. Branch conditions are exploited to enhance summary precision and to predicate optimistic data-flow results. If independence cannot be decided statically, user tools test an independence predicate. While SUIF does not seem to target non-linear accesses, it handles loops `olda_100/300` in a way similar to Pugh's.

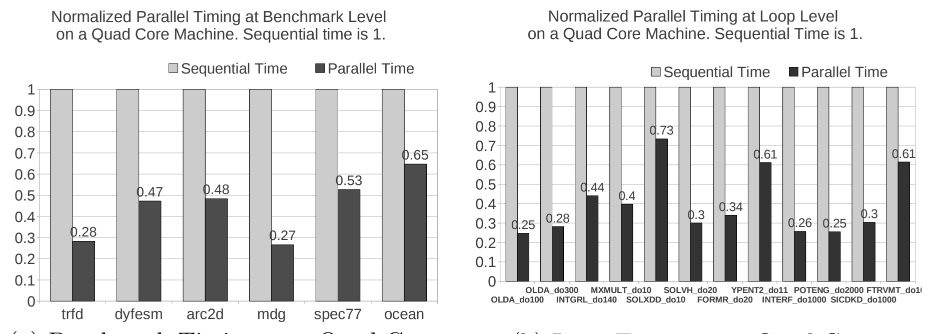
Another body of important work includes formulation of powerful symbolic algebra systems such as the one of Fahringer [9] that among others computes upper and lower bounds of nonlinear expressions, and the one of Engelen that uses a more general form of induction variables via recurrence chains [8].

Other work presents pattern matching solutions for code with conditional induction variables such as Lin's and Rus' stack [15] and pushback [26] arrays. While we handle cases as difficult as `TRACK`'s `extend_do400`, we do not discuss them here because there we rely on improving LMAD aggregation, while this paper handles non-linearity that fails (any) aggregation.

Finally, the other main direction of solving irregular accesses has been to analyze memory references at run-time [22, 17, 18]. These techniques have overhead proportional to the number of the original-loop accesses, and hence we use them only as a last resort, once all the lighter independence predicates have failed.

5 Experimental Evaluation

We evaluate our technique on a number of benchmarks known to be rich in non-linear accesses. Table 1 characterizes several such irregular loops, named in *column* 3, and their corresponding benchmark, named in *column* 1.



(a) Benchmark Timing on a Quad Core.

(b) Loop Timing on a Quad-Core.

Fig. 5. Parallel Normalized Timing on a Quad-Core Machine.

Column 2 shows the percentage of the sequential run-time coverage (SC) that has been parallelized, and the enabling transformations: privatization (PRIV), static/dynamic last value (SLV/DLV), reduction (RED). RT-IW and RT-RED correspond to predicates that may eliminate the need for privatization and reduction.

The *fourth column* classifies non-linear accesses: indirect arrays ($\mathtt{IA}_{mon}$), polynomial indexing (i^2), and loop index spanning multiple dimensions, which corresponds to the quasi-linear test that checks that dimensions do *not overlap*. Predicates typically prove flow/output independence (F/O-IND) in $O(N)$, for $\mathtt{IA}_{mon}$, and $O(1)$ run-time complexity for the other two. The *fifth column* shows the *weight* of the loop, as percentage of the sequential run time of the entire program. The *sixth column* demonstrates that the overhead of the extracted predicates, seen as a percentage of the concurrent loop run time, is negligible.

Figures 5(a) and 5(b) show the normalized timing for entire benchmarks and for selected loops (sequential time is 1). The tests and loops are shown in the order they appear in Table 1. We used IFORT version 11.1 to compile the source file generated by our compiler on a commodity INTEL quad-core Q9550@2.83GHz machine. Since the data-sets of these tests are very small, we used the `O0` level. Compilation under `O2` gives comparable speed-up for `trfd`, `mdg` and `spec77` benchmarks, some speed-up for `arc2D` and significant slowdown for `dyfesm` and `ocean`. The reason for the slowdown is that all important loops in the latter two benchmarks, while executed many times, have granularity in the range of tens of microseconds, which is too small to amortize the thread spawning overhead.

We have already discussed `TRFD`'s irregularities. All important loops from `DYFESM`: `MXMULT_do10`, `FORMR_do20`, `SOLXDD_do4/10/30/50`, `HOP_do20`, are proved flow/output independent via $O(N)$ monotonicity predicates that involve index arrays. We have found loop `SOLVH_do20` to be particularly interesting in that it requires predicates extracted from control flow *and* from both linear and non-linear LMADs. For example, flow independence of arrays `XE` and `HE` is verified with the $5*\mathtt{NNPED} \geq \mathtt{NDFE} \wedge \mathtt{NSYM} \neq 0$ and $8*\mathtt{NNPES} \geq \mathtt{NSFE}$ predicates, respectively, while output independence of `HE` is verified via the $O(N)$ predicate:
$\wedge_{\mathtt{iss}=1}^{\mathtt{NSS}-1}[-33+\mathtt{NSFE}+32*(\mathtt{IDBEGS(iss)}+\mathtt{NEPSS(iss)}) < 32*\mathtt{IDEBGS(iss+1)}]$.

Benchmarks `ARC2D`, `SPEC77` and `OCEAN` exhibit instances of the non-overlap, $O(1)$ test. `ARC2D`'s loops `YPENTA_do11`, `XPENT2_do11/15`, `YPENT2_do11/13` together have weight 10.2%, while `OCEAN`'s `FTRVMT_do109` shows a robust weight of 63.2%. Finally, while monotonicity tests are used on some unimportant loops of `MDG`, we included results here mainly because it has been reported non-linear [21]. Both important loops, `INTERF_do1000`, and `POTENG_do2000` are proved statically in our framework: the control-flow-based implication necessary to privatize `rl` is found *true* very early at USR aggregation level (otherwise a branch predicate would be extracted). We also apply run-time reduction and privatization.

In summary, on a four-core commodity system, we get an average speed-up on entire benchmarks of 2.24x and 2.52x, with maximal values of 3.70x and 3.97x, and minimal values of 1.54x and 1.40x, even when small data sets with (too) small granularity are used. The overhead of the monotonicity predicate is on average a negligible 0.15% of the loop concurrent run time.

6 Conclusions

In this paper we have presented a static analysis technique that can extract arbitrary predicates that can (in)validate the monotonicity property of memory reference summaries. These predicates are then simplified using symbolic elimination algorithms and implications of the assumed monotonicity. Finally, the predicates are evaluated at run time, where they can (in)validate loop independence. We have implemented the technique in our compiler and applied it on six benchmarks with difficult to analyze non-linear array references. The experimental results show that the dynamic evaluation of predicates represents a negligible overhead, especially when compared to previous run-time parallelization techniques. Our technology enabled the parallelization of the benchmarks with full coverage and produced scalable speedups.

Acknowledgment. This research was supported in part by NSF awards CRI-0551685, CCF-0833199, CCF-0830753, IIS-096053, IIS-0917266, NSF/DNDO award 2008-DN-077-ARI018-02, by the DOE NNSA under the Predictive Science Academic Alliances Program by grant DE-FC52-08NA28616, by by Chevron, IBM, Intel, Oracle/Sun and by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST).

References

1. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann (2002)
2. Banerjee, U.: Speedup of Ordinary Programs. Ph.D. Thesis, Dept. of Comp. Sci. Univ. of Illinois at Urbana-Champaign Report No. 79-989 (1988)
3. Berry, M., *et al.*, The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers. Int. J. of Supercomputer Applications 3, 5–40 (1988)

4. Blume, W., Eigenmann, R.: Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. IEEE Trans. Par. Distr. Sys. 3, 643–656 (1992)
5. Blume, W., Eigenmann, R.: The Range Test: A Dependence Test for Symbolic, Non-Linear Expressions. In: Procs. Int. Conf. on Supercomp., 528–537 (1994)
6. Blume, W., Eigenmann, R.: Demand-Driven, Symbolic Range Propagation. In: Procs. Intl. Lang. Comp. Parallel Comp., 141–160. Springer Verlag (1995)
7. Dang, F., Yu, H., Rauchwerger, L.: The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In: Procs. of Int. Parallel and Distributed Processing Symp., 20–29 (2002)
8. Engelen, R.A.V.: A unified framework for nonlinear dependence testing and symbolic analysis. In: Procs. Int. Conf. on Supercomputing., 106–115 (2004)
9. Fahringer, T.: Efficient symbolic analysis for parallelizing compilers and performance estimators. J. of Supercomputing 12, 227–252 (1997)
10. Feautrier, P.: Parametric Integer Programming. Operations Research 22(3), 243–268 (1988)
11. Feautrier, P.: Dataflow Analysis of Array and Scalar References. Int. J. of Parallel Programming 20(1), 23–54 (1991)
12. Hall, M.W., *et al.*: Interprocedural parallelization analysis in SUIF. ACM Trans. on Programming Languages and Systems 27(4), 662–731 (2005)
13. Hoeflinger, J., Paek, Y., Yi, K.: Unified Interprocedural Parallelism Detection. Int. J. of Parallel Programming 29(2), 185–215 (2001)
14. Lin, Y., Padua, D.: Demand-Driven Interprocedural Array Property Analysis. In: Procs. Int. Lang. Comp. Parallel Comp., 303–317 (1999)
15. Lin, Y., Padua, D.: Analysis of Irregular Single-Indexed Arrays and its Applications in Compiler Optimizations. In: Procs. Int. Conf. Comp. Constr., 202–218 (2000)
16. Moon, S., Hall, M.W.: Evaluation of predicated array data-flow analysis for automatic parallelization. In: Procs. Int. Princ. Pract. of Par. Prog., 84–95 (1999)
17. Oancea, C.E., Mycroft, A.: Set-Congruence Dynamic Analysis for Software Thread-Level Speculation. In: Procs. Int. Lang. Comp. Parallel Comp., 156–171 (2008)
18. Oancea, C.E., Mycroft, A., Harris, T.: A Lightweight, In-Place Model for Software Thread-Level Speculation. In: Procs. Symp. Paral. Alg. Arch., 223–232 (2009)
19. Paek, Y., Hoeflinger, J., Padua, D.: Efficient and Precise Array Access Analysis. ACM Trans. on Programming Languages and Systems 24(1), 65–109 (2002)
20. Pugh, W.: The Omega Test: a Fast and Practical Integer Programming Algorithm for Dependence Analysis. Communications of the ACM 8, 4–13 (1992)
21. Pugh, W., Wonnacott, D.: Nonlinear Array Dependence Analysis. In: Proc. of Workshop on Lang. Comp. and Run-Time Support for Scallable Systems (1995)
22. Rauchwerger, L., Padua, D.: The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. IEEE Trans. Par. Distr. Sys. 10(2), 160–199 (Feb 1999)
23. Rauchwerger, L., Amato, N.M., Padua, D.A.: A scalable method for run-time loop parallelization. Int. J. of Parallel Programming 26, 26–6 (1995)
24. Rus, S., Hoeflinger, J., Rauchwerger, L.: Hybrid analysis: Static & dynamic memory reference analysis. Int. J. of Parallel Programming 31(3), 251–283 (2003)
25. Rus, S., Pennings, M., Rauchwerger, L.: Sensitivity Analysis for Automatic Parallelization on Multi-Cores. In: Procs. Int. Conf. on Supercomp. 263–273 (2007)
26. Rus, S., Zhang, D., Rauchwerger, L.: The Value Evolution Graph and its Use in Memory Reference Analysis. In: Procs. of Int. Conf. on Parallel Architectures and Compilation Techniques, 243–254 (2004)
27. Yu, H., Rauchwerger, L.: Techniques for Reducing the Overhead of Run-Time Parallelization. In: Procs. Int. Conf. Comp. Constr., 232–248 (2000)