Scalable Conditional Induction Variables (CIV) Analysis

Cosmin E. Oancea
Department of Computer Science
University of Copenhagen
cosmin.oancea@diku.dk

Lawrence Rauchwerger
Department of Computer Science and Engineering
Texas A & M University
rwerger@cse.tamu.edu

Abstract

Subscripts using induction variables that cannot be expressed as a formula in terms of the enclosing-loop indices appear in the low-level implementation of common programming abstractions such as filter, or stack operations and pose significant challenges to automatic parallelization. Because the complexity of such induction variables is often due to their conditional evaluation across the iteration space of loops we name them *Conditional Induction Variables* (CIV).

This paper presents a flow-sensitive technique that summarizes both such CIV-based and affine subscripts to program level, using the same representation. Our technique requires no modifications of our dependence tests, which is agnostic to the original shape of the subscripts, and is more powerful than previously reported dependence tests that rely on the pairwise disambiguation of read-write references.

We have implemented the CIV analysis in our parallelizing compiler and evaluated its impact on five Fortran benchmarks. We have found that that there are many important loops using CIV subscripts and that our analysis can lead to their scalable parallelization. This in turn has led to the parallelization of the benchmark programs they appear in.

1. Introduction

An important step in the automatic parallelization of loops is the analysis of induction variables and their transformation into a form that allows their parallel evaluation. In the case of "well behaved" induction variables, that take monotonic values with a constant stride, their generating sequential recurrence can be substituted with the evaluation of closed form expression of the loop indices. This transformation enables the parallel evaluation of the induction variables. When these variables are used to form addresses of shared data structures, the memory references can be statically analyzed and possibly parallelized. Thus we can conclude that the analysis of induction-variables use is crucial to loop dependence analysis [1, 7, 18] and their subsequent parallelization.

For example, the loop in Figure 1(a) increments by two the value of k (produced by the previous iteration), and updates the kth element of array a. The uniform incrementation of k allows to substitute this recurrence on k with

```
k = k0      Ind. k = k0             DO i = 1, N
DO i =1,N   Var. DO i = 1, N         IF(cond(b(i)))THEN
  k = k+2    ⇒     a(k0+2*i)=..         civ = civ+1 ⇒?
  a(k)=..   Sub. ENDDO                  a(civ) = ...
ENDDO            k=k0+MAX(2N,0)  ENDIF ENDDO
   (a)                (b)                (c)
```

Figure 1. Loops with affine and CIV array accesses.

its closed-form equivalent k0+2*i, which enables its independent evaluation by all iterations. More importantly, the resulted code, shown in Figure 1(b), allows the compiler to verify that the set of points written by any distinct iterations *do not overlap*, a.k.a., output independence: $\texttt{k0+2*i}_1 = \texttt{k0+2*i}_2 \Rightarrow \texttt{i}_1\texttt{=i}_2$. It follows that the loop in Figure 1(b) can be safely parallelized.

A known difficulty in performing this analysis arises when subscripts use scalars that do not form a uniform recurrence, i.e., their stride (increment) is not constant across iteration space of the analyzed loop. We name such variables *conditional induction variables* or CIV because they are typically updated (only) on some of the possible control-flow paths of an iteration. For example, the loops in Figures 1(a) and (c) are similar, except that in (c) both civ and the array update are performed only when condition cond(b(i)) holds. Although the recurrence computing civ values can still be parallelized via a scan [2] (prefix sum) precomputation, neither CIV nor the subscript can be summarized as affine expressions of loop index i, and hence the previous technique would fail.

This paper presents a novel induction-variable analysis that allows both affine and CIV based subscripts to be summarized at program level, using *one common representation*.

Current solutions [3, 8, 10, 23, 26] use a two-step approach: First, CIV scalars are recognized and their properties, such as monotonicity and the way they evolve inside and across iterations, are inferred. Intuitively, in our example, this corresponds to determining that the values of civ are increasing within the loop with step at most 1. This paper does not contribute to this stage, but exploits previously developed techniques [23]. Second and more relevant, each pair of read-write subscripts is disambiguated, by means of *specialized* dependency tests that exploit the CIV's properties. In our example, the cross-iteration monotonicity of the

1-4799-8161-8/15/$31.00 ©2015 IEEE

CIV values dictates that the CIV value, named $\texttt{civ}_2$ after being incremented in some iteration $\texttt{i}_2$ is strictly greater than the value, named $\texttt{civ}_1$ of any previous iteration $\texttt{i}_1$. It follows by induction that the update of `a(civ)` cannot result in cross-iteration (output) dependencies, i.e., the system $\texttt{civ}_2\texttt{-civ}_1 \geq 1$ and $\texttt{civ}_2 = \texttt{civ}_1$ has no solution.

Other summarization techniques [9, 14, 16] aggregate array references across control-flow constructs, e.g., branches, loops, and model dependency testing as an equation on the resulted abstract sets. In practice, they were found to scale better than previously developed analysis based on pairwise accesses [9], but they do not support CIVs.

This paper proposes an extension to summary-based analysis that allows CIV based subscripts to use the same representation as the affine subscripts. This enables scalable memory reference analysis without modification of the previously developed dependency tests. The gist of our technique is to aggregate symbolically the CIV references on every control-flow path of the analyzed loop, in terms of the CIV values at the entry and end of each iteration or loop. The analysis succeeds if (i) the symbolic-summary results are identical on all paths and (ii) they can be aggregated across iterations in the interval domain.

We demonstrate the technique on the loop in Figure 1(c), where we use $\texttt{civ}^i_\mu$ and $\texttt{civ}^i_b$ to denote the values of `civ` at the entry and end of iteration `i`, respectively. On the `THEN` path, i.e., `cond(b(i))` holds, the write set of array `a` is interval $W_i = [\texttt{civ}^i_\mu\texttt{+1},\texttt{civ}^i_b]$, i.e., point $\{\texttt{civ}^i_\mu\texttt{+1}\}$, because the incremented value of `civ` is live at the iteration end. On the `ELSE` path, the write set of `a` is the empty set. For uniformity of representation we express this empty set as an interval with its lower bound greater than its upper bound. In particular, the interval of the `THEN` path matches, because `civ` is not updated and so $\texttt{civ}^i_\mu\texttt{+1>}\texttt{civ}^i_b$.

Aggregating W_k across the first `i-1` iterations results in $\cup_{k=1}^{i-1} W_k = [\texttt{civ}^1_\mu\texttt{+1},\texttt{civ}^i_\mu]$, where we have used `civ` values' monotonicity and the implicit invariant $\texttt{civ}^{k-1}_b \equiv \texttt{civ}^k_\mu$, i.e., the `civ` value at an iteration end is equal to the `civ` value at the entry of the next iteration. As a last step, program invariants are modeled as abstract-set equations that do not depend on the original-subscript shape, i.e., CIV-based or affine. In our example, the output independence equation: $(\cup_{k=1}^{i-1} W_k) \cap W_i = [\texttt{civ}^1_\mu\texttt{+1},\texttt{civ}^i_\mu] \cap [\texttt{civ}^i_\mu\texttt{+1},\texttt{civ}^i_b] = \emptyset, \forall \texttt{i}$, verifies that the write set of any iteration `i` does not overlap the write set of all previous iterations, and, by induction, that no distinct iterations write a common element of `a`.

Our technique is well integrated into the PARASOL branch of the Polaris [4] `Fortran` compiler. We use *hybrid analysis* to verify dataset-sensitive invariants at runtime by evaluating a sequence of sufficient-conditions of increasing complexity. These conditions (predicates) are statically synthesized from the summary equations that model the relevant invariants Our existing hybrid analysis did not require modifications but has benefited from our technique: First, our analysis simplifies CIV-summary expressions enabling successful verification of the relevant invariants. Second, our analysis extracts the slice of the loop that computes the CIV values and evaluates it in parallel before loop execution. This allows: (i) runtime verification of CIV monotonicity whenever this cannot be statically established, (ii) sufficient conditions for safe parallelization to be expressed in terms of the CIV values at loop's entry, end, and anywhere in between, and (iii) safe parallelization in the simple case when CIV are not used for indexing. The latter is not reported in related work. In summary, main contributions are:

- A flow-sensitive analysis that extends program level summarization to support CIV-based indices and leads to the parallelization of previously unreported loops,
- A non-trivial code generation that separates the program slice that performs the parallel-prefix-sum evaluation of the CIV values from the main (parallel) computation in which they are inserted.
- An evaluation of five difficult to parallelize benchmarks with important CIV-subscripted loops, which measures all runtime overheads and shows application level speedups as high as $7.1\times$ and on average $4.33\times$ on 8 cores.

Speedups of two of this paper benchmarks shown were also reported in [14]. They were obtained in part by using the CIV technique that is *presented here for the first time*.

2. An Intuitive Demonstration

Figure 2(a) shows a simplified version of loop `CORREC_do401` from BDNA benchmark, as an example of non-trivial loop that uses both CIV and affine-based subscripts. Variable `civ` uses (gated) single-static-assignment (SSA) notation [25]: For example, statement `civ@2=`γ`(i.EQ.M,civ@1,civ@4)` has the semantics that variable `civ@2` takes, depending on the evaluation of `i.EQ.M`, either the value of `civ@1` for the first iteration of the loop that starts at `M`, or the value of `civ@4` for all other iterations. For simplicity we omit the gates in the figure and use only read and write reference sets (but our implementation uses read-only, write-first, read-write sets).

Parallelizing loop `CORREC_do401` in Figure 2(a) requires verifying two invariants: The first refers to disproving cross-iteration true and anti dependencies. For such dependencies to occur it is necessary that there exists a memory location that is both read and written, and we plan to disprove this by verifying that the overestimates of the read and write sets (of subscripts) of `X`, aggregated at the outer-loop level, do not overlap. The second invariant refers to disproving output dependencies, which reduces to showing that the write-set overestimates of any two iterations do not overlap.

Matters are simple for summarizing the affine read access `X(i)`: since `i` is the loop index ranging from `M` to `N`, the read set across the loop is $\lceil R \rceil = \cup_{i=M}^{N} \{i-1\} = [\texttt{M-1},\texttt{N-1}]$, where we have used the convention that array indices start from 0 (rather than 1 as in the introductory example).

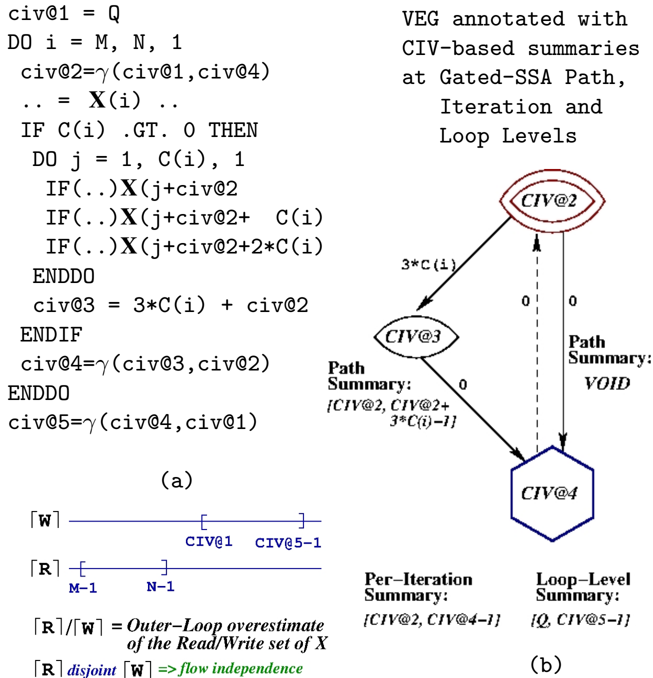


Figure 2. (a) Loop CORREC_do401 and (b) its Value Evolution Graph (VEG), (from BDNA, PERFECT-CLUB).

Computation of the write-set *overestimate* starts by summarizing the inner loop: the `if` branches are conservatively assumed as taken, and the three affine inner-loop updates are summarized to interval `[civ@2,civ@2+3*C(i)-1]` by expanding index `j` to its range `[1,C(i)]`, e.g., access `X(j+civ@2)` generates interval `[civ@2,civ@2+C(i)-1]`.

Similar to the introductory example, outer-loop aggregation analyzes each path of an iteration: On the path on which `C(i).GT.0`, we have `civ@4=civ@3=civ@2+3*C(i)`, and the path summary is rewritten as `[civ@2,civ@4-1]`, i.e., in terms of the `civ` variables that start and end an iteration. The other path neither updates `X` nor increments `civ`, hence it can use *the same symbolic summary* `[civ@2,civ@4-1]`$=\emptyset$, where `civ@2=civ@4>civ@4-1`. Note that an empty interval has its lower bound greater than its upper one. It follows that the outer-loop iteration summary is W_i = [civ@2,civ@4-1].

Since `civ` is updated only when `C(i)` is positive, its values are monotonically increasing within the loop with the lower and upper bounds being `civ@1=Q` and `civ@5`. It follows that the write-set overestimate of the outer loop is interval $\lceil W \rceil$=$\lceil \cup_{i=M}^{N} W_i \rceil$=`[Q,civ@5-1]`. Flow independence requires $\lceil W \rceil \cap \lceil R \rceil \equiv [\mathtt{M}-1, \mathtt{N}-1] \cap [\mathtt{Q}, \mathtt{civ@5}-1] = \emptyset$, and a sufficient condition for this equation to hold is (easily) extracted: $\mathtt{Q} \geq \mathtt{N} \vee \mathtt{M} > \mathtt{civ@5}$. Finally, output independence is proven statically, similarly to the CIV loop of Figure 1.

We conclude this section with *two high-level observations*: *First*, our analysis summarizes CIV and affine subscripts much in the same way, and enables a dependence test that is agnostic to the kind of subscripts that were used. These properties do not seem to hold for related approaches: For example, techniques that disambiguate consecutively written, single indexed subscripts [10, 23] cannot solve CORREC_do401 because the accesses do not follow such pattern. Similarly, a technique [26] that disambiguates pairs of accesses of shape {`X(civ)`,`X(civ+CT)`} may prove the output, but *not* the flow independence of CORREC_do401.

Second, summary approximation has been key to successful analysis. For example in the code in Figure 2(a), overestimating the write set of the inner loop by assuming that all branches are taken allowed the CIV subscripts to be aggregated as an interval. More importantly, the separation of the read and write sets cannot be verified statically: it requires runtime information. It follows that the accurate write-first summary at the outer-loop level cannot be statically simplified to an interval, but its overestimate, consisting of only the CIV term, can and is sufficient for proving parallelization.

Finally, even if an affine (or different-CIV) *write access* is added to the code, the write overestimate is still computable by separately aggregating each term and uniting the results.

3. Preliminary Concepts

Our analysis of CIV subscripts builds on four main techniques: First, a baseline analysis summarizes array indices into read-only RO, read-write RW and write-first WF abstract sets, using a representation named unified set reference [22] (USR). Second, loop independence is modeled as an equation in the USR domain. Third, whenever this equation cannot be verified statically, a predicate is extracted from it, and its evaluation validates independence at runtime [14]. Fourth, the value evolution graph [23] (VEG) is used to model the flow of CIV values in a loop, and to query scalars' ranges. The remaining of the section establishes a uniform notation and formulates the problem to be solved by CIV analysis.

USR	::=	$LMAD$	(strided intervals)
	|	$USR \cup USR$	(set union)
	|	$USR \cap USR$	(intersection)
	|	$USR - USR$	(difference)
	|	$Exp^{bool} \# USR$	(gated USR)
	|	$\imath \cup_{i=1}^{N} USR$	(full recurrence)
	|	$\imath \cup_{k=1}^{i-1} USR$	(partial recurrence)
	|	$CallSite \bowtie USR$	(callsite translation)

Figure 3. Unified Set Reference (USR) Grammar.

USR Construction [22]. Summaries use the representation shown in Figure 3, and can be seen as a DAG in which leaves correspond, for simplicity, to strided (multi-dimensional) intervals, named linear memory access descriptors [16] (LMAD) in the literature. USR's internal nodes represent operations whose results are not accurately representable in the LMAD domain: (i) irreducible set operations, such as union, intersection, subtraction ($\cup$, $\cap$, $-$), or (ii) control flow: gates predicating summary existence (prefixed by

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

(a)

OUTPUT INDEPENDENCE EQ:
$\{\cup_{i=1}^{N}(WF_i \cap (\cup_{k=1}^{i-1} WF_k))\} = \emptyset$

FLOW/ANTI INDEPENDEP EQ:
$\{(\cup_{i=1}^{N} WF_i) \cap (\cup_{i=1}^{N} RO_i)\} \cup$
$\{(\cup_{i=1}^{N} WF_i) \cap (\cup_{i=1}^{N} RW_i)\} \cup$
$\{(\cup_{i=1}^{N} RO_i) \cap (\cup_{i=1}^{N} RW_i)\} \cup$
$\{\cup_{i=1}^{N}(RW_i \cap (\cup_{k=1}^{i-1} RW_k))\} = \emptyset$

(b)

Figure 4. (a) Loop Memory Reference Summary and (b) Loop Independence with Set Equations.

#) or total ($_l\cup_{i=1}^{N}$) and partial ($_l\cup_{k=1}^{i-1}$) unions corresponding to a loop l of index `i=1,N`. We note that while USRs are complex and accurate, one can always under/over-estimate an USR in the simpler strided-interval domain, albeit this may prove very conservative, i.e., $\emptyset$, or $[0, \infty]$.

USRs are built during a bottom-up traversal of a control-flow reducible program. In this pass, data-flow equations dictate how summaries are initialized at statement level, merged across branches, translated across call sites, composed between consecutive regions, and aggregated across loops. For example, the aggregation equation for a loop l of index `i=1,N` is shown in Figure 4(a): The loop-aggregated write-first set WF is obtained by (i) subtracting from the write-first set of iteration i, i.e., WF_i, the reads of any iteration preceding i, and (ii) by uniting the per iteration sets.

Independence Equations. Figure 4(b) shows the USR equations, of form $S = \emptyset$, that model loop independence. The output independence equation states that if for any i, the write-first set of iteration i does not overlap with the write-first set of any iteration preceding i, then, by induction, no two iterations write the same location. Similarly, for flow/anti independence it is checked (i) the disjointness of the total unions of the per-iteration WF, RO, and RW set, and (ii) that the RW sets of any two iterations do not overlap.

Synthesizing Independence Predicates [14, 15]. When the independence equation is not statically satisfied, we use a translation scheme $\mathcal{F}$, from the USR language to a language of predicates, named PDAG, to extract a sufficient-independence condition: $\mathcal{F}$: USR $\rightarrow$ PDAG, $\mathcal{F}(S) \Rightarrow S = \emptyset$. The result is then separated into a cascade of predicates that are tested at runtime in the order of their complexities. If output dependencies cannot be disproved then privatization is necessary. If WF_i is loop invariant then only the last iteration writes to non-private storage (static last value).

Value-Evolution Graph [23] (VEG) is a DAG that represents the flow of values between gated-SSA [25] names of a scalar variable. VEGs are constructed on demand at loop and subroutine levels. For example Figure 2(a) and (b) shows our running example and the VEG describing the evolution of variable CIV in the outer loop: The entry node, e.g., merging cross-iteration values, is named the μ node and is shown in Figure 2(b) as a double ellipse. The recurrence, named the *back edge*, is shown as a dotted line, and the *back node* is drawn as a hexagon. Regular nodes, drawn as ellipses, correspond to reduction-like statements. The VEG shows that our loop uses a variable named `civ` that is incremented on some paths with `3*C(i)`, and is unmodified on other paths.

Note that branch conditions are not explicit in the VEG, but they are easily found from the gated-SSA definition of γ-merged CIVs. Nodes corresponding to arbitrary assignments, i.e., not a reduction, are named input nodes and are drawn in rectangles. A conditional induction variable (CIV) corresponds to a particular VEG, in which: (i) the μ node is unique, dominates all other VEG nodes, i.e., is reachable from within the VEG only through its unique back edge, (ii) there are no input nodes, and (iii) the μ-node values are provably monotonic. Our running example complies with these rules: For example, evolution `3*C(i)` in Figure 2(b) is proven positive at the point of use, because path `civ@2` $\rightarrow$ `civ@3` $\rightarrow$ `civ@4` is guarded by condition `C(i).GT.0`, which, while not shown, is part of the gated-SSA definition of γ-node `civ@4`. If this is not possible statically, analysis can optimistically assume monotonicity, and verify it at runtime in the slice that pre-computes the CIV values.

Problem Statement. We denote by $\mathcal{L}$ a normalized loop of index $i \in \{1..N\}$ that exhibits subscripts using CIV variables. We denote by U_i one of the write-first (WF), read-write (RW) or read-only (RO) set-expression summaries of some array `X` corresponding to iteration i of loop $\mathcal{L}$, where the USR's leaves, i.e., LMADs, may use CIV variables. For simplicity, we consider LMADs to be strided intervals, i.e., $[l, u]^s = \{l, l+s, l+2*s, .., u\}$. If $s = 1$, we omit writing s; if the interval is a point p, we may write it $\{p\}$.

The goal is to compute symbolic under/over-estimates in the interval domain, denoted by $\lfloor A \rfloor / \lceil A \rceil$, for $U = \cup_{i=1}^{N} U_i$. In the same way in which loop aggregation of an affine subscript eliminates the loop index i from the result summary, we declare CIV-*aggregation* across $\mathcal{L}$ *successful* if $\lfloor A \rfloor$ ($\lceil A \rceil$) does not depend on any loop-variant symbols, e.g., any symbol in CIV_μ's VEG. For example, in Figure 2(a), the affine subscript of `X(i)`, expressed as $\{$`i-1`$\}$, is aggregated across the outer loop as `[M-1,N-1]`, which is independent on `i`. Similarly, we would like to overestimate the write accesses across the loop as interval `[civ@1,civ@5-1]` which does not depend on loop-variant symbols, such as `civ@3`.

In essence this would allow to treat *uniformly* and *compositionally* CIV-based and affine summaries: For example, the compiler can now establish flow independence by comparing the read and write summaries, which have the same representation, albeit the read and write sets correspond to affine and CIV-based subscripts, respectively. The other example refers to the difficult benchmark `track`, in which the output of the CIV-summarized (inner) loop `EXTEND_do500` becomes the input to CIV-based summarization of the outer loop `EXTEND_do400` (hence composable summarization).

4. Monotonic CIV Summaries

Our analysis is implemented as an extension of baseline summarization for the special cases of loop and iteration aggregation (and call-site translation), because at these levels the CIV-value properties, summarized by VEG, can be effectively used. To compute the symbolic summary across all paths, the analysis needs to combine the control-flow of the summary, e.g., encoded in USR's gates, with the control flow of the CIV, available in VEG. The key idea here is to conservatively associate summary terms with VEG nodes and to merge across all VEG paths. It follows that our analysis computes under and overestimate summaries, but accuracy is recovered when the under and overestimate are identical.

Over/under estimation also allows (separate) aggregation of the affine and CIV-based subscripts, which are typically still accurate enough to verify the desired invariant. As with our running example, in practice, important loops use both affine and CIV-based subscripts on the same array, hence terms of both kinds may appear in the set expression of the accurate summary, which cannot be simplified to an interval.

The remaining of this section is organized as follows: *Section 4.1* presents the basic flow-sensitive analysis technique for summarizing over and underestimates, and demonstrates it on the running example depicted in Figure 2. *Section 4.2* shows several enhancements to the basic technique that solve more difficult loops, such as `EXTEND_do400` from benchmark `track`. Our analysis also extends (with some modifications) to a stack-like access pattern in which the CIV values are only piecewise monotonic; this is discussed in Section 4.3. Finally, *Section 4.4* details on the overall implementation, and focuses on how to (pre)compute safely, in parallel the CIV_μ values associated to each iteration.

4.1 Basic Flow-Sensitive-Analysis Techinque

The algorithm implementing the basic analysis is depicted in Figure 5 and has four main stages: First, under and overestimates of input USR U_i are computed under the form of a union of gated-interval pairs. Second, the intervals that use CIV variables are associated with nodes on the CIV's VEG graph. Third, each VEG path is summarized via an interval expressed in terms of $\texttt{CIV}_\mu$ and $\texttt{CIV}_b$ nodes.
Finally, path intervals are merged across all paths, according to the under/over-estimate semantics, to yield the iteration-level interval A_i. Total and partial-recurrence intervals, A and $A_{k=1}^{i-1}$, are computed by suitably substituting $\texttt{CIV}_\mu$ and $\texttt{CIV}_b$ in A_i with suitable (VEG) bounds, which is safe due to the cross-iteration monotonicity of CIV values. The remaining of this section details and demonstrates each algorithmic stage on our running-example loop introduced in Figure 2.

1. Approximating an USR. For brevity, we do not present the algorithm for over/under-estimating an USR via *a union of gated-interval* $g\#L$ *pairs*, where L is an interval and g is a condition predicating L's existence. We note that it is always possible to build an interval under/overestimate, by

CIV-SUMMARIZATION (U_i : USR)

// ***Output:*** (A_i, A, $A_{k=1}^{i-1}$) or **Fail**s, i.e.,
// the under/over-estimate intervals of (U_i,$\cup_{i=1}^{N} U_i$, $\cup_{k=1}^{i-1} U_i$).

1. OVER/UNDER-ESTIMATE U_i BY A UNION OF GATED INTERVALS: $\cup_k (g_k \# L_k) \leftarrow U_i$

2. ASSOCIATE EACH L_k, TO A VEG NODE `CIV@q`, WHICH IS THE CIV-IMMEDIATE (POST) DOMINATOR OF THE PROGRAM POINT WHERE L_k WAS SUMMARIZED

3. For Each VEG PATH, SYMBOLICALLY UNITE (EXACTLY) THE L_k'S ON THAT PATH INTO ONE INTERVAL L_{path}.

If L_{path} CANNOT BE WRITTEN IN TERMS OF ONLY CIV_μ AND CIV_b LOOP-VARIANT SYMBOLS **Then Fail**.

If IN UNDERESTIMATE CASE THEN CHECK THAT THE VEG-$path$ CONDITION IMPLIES g_k. **Else** $L_k \notin path$.

4. If ALL $path$S HAVE IDENTICAL L_{path}
Then $A_i = L_{path}$, AND COMPUTE A AND $A_{k=1}^{i-1}$ BY SUBSTITUTING CIV_μ AND CIV_b WITH THEIR BOUNDS.
Else Fail.

Figure 5. CIV-Based-Summarization Pseudocode.

recursively pattern matching the USR's shape [15]. We have enhanced this algorithm to also gather gate information, and we have not encountered any significant problems.

With the loop in Figure 2(a), the write-first underestimate $\lfloor WF_i \rfloor$ of array `X` is the empty set, because `X` is conditionally updated in the inner loop, and the gated overestimate is $\lceil WF_i \rceil = (\texttt{C(i).GT.0})\#[\texttt{civ@2}, \texttt{civ@2} + 3 * \texttt{C(i)} - 1]$, because the CIV accesses occur inside the inner loop, which belongs to the `THEN` target of statement `IF(C(i).GT.0)`.

2. Associating Summaries with VEG Nodes. Our analysis uses the VEG to approximate the control-flow of the loop, because (i) VEG describes the evolution of the CIVs used in subscripts, and (ii) it also retains the key control-flow information necessary for summarization.

As such, we associate each gated-interval pair $g\#L$ containing a CIV with a node in the CIV's VEG[1]. This corresponds to identifying the CIV node (in VEG) that most-accurately describes the program point $\texttt{PP}_L$ where interval L was summarized: We compute both the immediate CIV-node dominator and post-dominator of $\texttt{PP}_L$, denoted by $\texttt{CIV}_d$ and $\texttt{CIV}_{pd}$, respectively, and chose $\texttt{CIV}_{pd}$ as the associated node if $\texttt{CIV}_d$ dominates $\texttt{CIV}_{pd}$, and $\texttt{CIV}_d$ otherwise. This ensures that the associated CIV is the "closest" node that belongs to any path passing through $\texttt{PP}_L$ (the reverse does not hold). For our running example, the $\lceil WF_i \rceil$ overestimate of `X`,

[1] In principle, we may have "regular" gated-interval pairs, and also different pairs may correspond to CIVs from different VEGs. These are treated independently and the result is the union of partial results. Analysis fails if one interval exhibits two CIVs belonging to different VEGs.

namely [civ@2,civ@2+3*C(i)-1], is projected to node civ@3, as shown in the annotated VEG in Figure 2(b).

For underestimate computation it is *not enough* to associate an interval L to a CIV node in the manner presented before, because it is not guaranteed that any path that passes through the CIV node would also pass through PP_L. At this point we use the gate g associated with L, which subsumes all the conditions guarding the accesses summarized by L: In the computation of an underestimate, a gated-interval $g\#L$ is considered part of a VEG path *iff the condition of the* VEG *path*, i.e., the conjunction of all the γ-node gates on that path, *implies* g, i.e., the existence condition of L. This guarantees that the summary belongs to any control-flow path that includes the considered VEG path[2].

3. VEG-Path Summarization. To compute the result on *one* path, all intervals are rewritten in terms of only the CIV_μ node, by using the symbolic formulas of the path's evolution. Then, all intervals belonging to the VEG path are united (unioned). If this succeeds, i.e., the result is one interval, then the resulting-interval upper bound is rewritten in terms of the back node, CIV_b. This is possible because each path has a known evolution from CIV_μ to CIV_b. If the result is free of loop-variant symbols other than CIV_μ and CIV_b, such as 3*C(i), then path-level analysis succeeds, otherwise it fails.

For example, the path civ@2 $\rightarrow$ civ@3 $\rightarrow$ civ@4 of the VEG in Figure 2(b), exhibits loop-variant evolution 3*C(i), and results in interval [civ@2,civ@2+3*C(i)-1]. However, rewriting the upper bound in terms of the back CIV node, results in [civ@2, civ@4 $-$ 1], i.e., we have performed substitution civ@2$\leftarrow$civ@4-3*C(i) derived from that path's evolution. It follows that aggregation succeeds.

The other VEG path exhibits empty summaries and 0 evolution, i.e., civ@2=civ@4. It follows that [civ@2,civ@4-1] $\equiv$ $\emptyset$ describes it correctly as well, i.e., an interval in which the upper bound is smaller than the lower bound is empty. It follows that all paths share the same per-iteration result, in which the only loop-variant terms are the μ and back CIV nodes, and analysis succeeds.

4. Merge Across All Paths. The last stage of the analysis is to compute the merge-over-all-paths result (interval). In our implementation the merge succeeds only when all path results are identical, which holds on both examples, e.g., $\lceil WF_i \rceil$ = [civ@2, civ@4 $-$ 1]. In the general case, one can compute the intersection/union over all paths as the under/overestimate result, respectively. For simplicity, in the following, we assume that CIV values are monotonically increasing, and that the result's upper bound increases with the iteration number, i.e., positive stride.

We compute the loop result as if we aggregate an affine access across a loop whose lower and upper bounds are equal to the value of the CIV at the loop entry and exit, respectively. In our case this corresponds to replacing civ@2 and civ@4

[2] Establishing this invariant strictly from the CFG is more conservative, e.g., our gates are translated/simplified across call sites, hence more accurate.

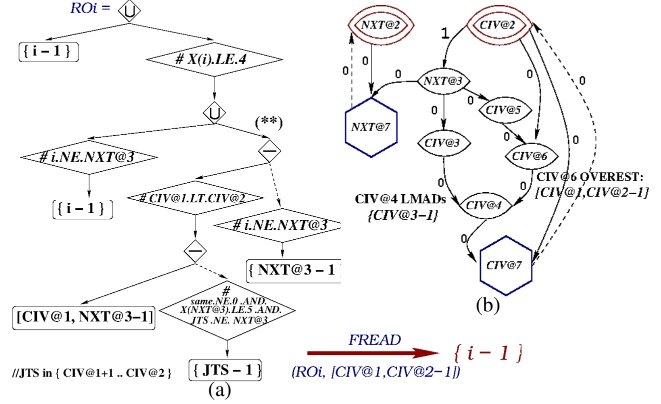


Figure 6. (a) RO_i USR & (b) civ's VEG for EXTEND_do500.

with civ@1 and civ@5 (the CIV values at the loop entry and exit), which results in [civ@1, civ@5-1]. If the sign of the CIV factor in the affine-CIV expression is positive, then CIV's monotonicity ensures that the overestimate is correct.

For the underestimate, correctness requires checking that the per-iteration result A_i is contiguous (or overlapping) between any two consecutive iterations. If A_i =$[\texttt{lb}_i,\texttt{ub}_i]^s$, this corresponds to checking that $\texttt{ub}_{i-1}$+s = $\texttt{lb}_i$, i.e., the upper bound of iteration i-1 plus the stride equals ($\leq$) the lower bound of iteration i. The check uses the invariant that the CIV value at the end of an iteration, i.e., the back node, equals the CIV value at the entry of the next iteration, i.e., the μ node. If $\lfloor WF_i \rfloor$ would be [civ@2,civ@4-1][1], this would correspond to replacing civ@4 with civ@2 in the upper bound and checking civ@2-1+1=civ@2, which verifies statically. Note that if the per-iteration result is a point, then the stride is not set yet, and in this case the stride is set to the value that verifies the invariant, i.e., $\texttt{lb}_i$-$\texttt{ub}_{i-1}$.

Similarly, to compute the partial-loop-aggregation result, i.e., $\cup_{k=1}^{i-1}$, we replace the CIV μ with the CIV node at the entry of the loop, and the back node with the CIV μ node of iteration i, which results in interval [civ@1,civ@2-1].

Finally, if the over and underestimate results are identical then the result is exact. The computed summaries allow now to prove the flow and output independence of the running example, as we have already seen in Section 2.

4.2 Enhancements to the Basic Technique

This section presents several refinements of the basic analysis, that allow to parallelize several difficult loops, e.g., track. One such loop is EXTEND_do400 whose body consists of inner loop EXTEND_do500. Since even the simplified code is too complex for paper presentation, Figure 6(b) presents the VEG of the inner loop, which sheds significantly more insights than the code would do.

1. External Variables Allowed in VEG. Sometimes two different variables are semantically the same CIV, for example in the sequence nxt=civ;...nxt=nxt+1;...civ=nxt,

USR **FREAD**(R : USR, WF^p : LMAD)

// ***Output:*** R filtered-out of WF^p terms.

Case R of:

LMAD L: **IF** $\mathcal{F}(L - WF^p)$ **THEN** $\emptyset$ **ELSE** L

$g\#A$: $g\#$**FREAD**(A,WF^p)

$A \cup B$: **FREAD**(A,WF^p) $\cup$ **FREAD**(B,WF^p)

$A \cap B$: **FREAD**(A,WF^p) $\cap$ **FREAD**(B,WF^p)

$A - B$: **FREAD**(A,$\lfloor B \rfloor \cup WF^p$)

LOOP OR CALLSITE :

IF($\mathcal{F}(\lceil R \rceil - WF^p)$ THEN $\emptyset$ ELSE R

Figure 7. Read-Set Filtering Algorithm.

shown in the VEG graph of Figure 6(b). It follows that we allow (external) variables such as `nxt@3` to belong to the VEG graph of `civ` because (i) they semantically contribute to the flow of `civ` values and (ii) the extended VEG still complies with the definition of a CIV variable (see Section 3), and because (iii) subscripts may contain both variables.

The analysis treats `nxt@3` as any other member of the `civ` family of variables, except that summaries *cannot* be associated with external nodes, because the latter do not necessarily encapsulate the correct control flow, i.e., the γ nodes of variable `nxt` are not part of `civ`'s VEG graph.

2. Filtering The Read Set. The RO (USR) summary corresponding to iteration `i` of loop `EXTEND_do500` is shown in Figure 6(a), where $\cup$, $-$ and $\#$ internal nodes correspond to set union, subtraction and gates, respectively. One can observe that RO_i is already nontrivial, and using it as input to outer-level summarization would result in complex set expressions, which would be ill-suited for computing independence. We use instead a filtering technique based on the observation that the contribution of RO_i to the loop-aggregated RO set cannot possibly belong to $\cup_{k=1}^{i-1}\text{WF}_k$. The same property holds for the read-write set.

Figure 7 shows the recursively-defined operator `FREAD` that filters out from the RO_i (RW_i) set, the terms that are included into an interval underestimate of $\lfloor \cup_{k=1}^{i-1}\text{WF}_k \rfloor$, denoted WF^p. The result summary replaces RO_i in Figure 4(a)'s equations. The algorithm pattern-matches the shape of the read-only summary: If the current node is an interval L then we extract a sufficient predicate for $L - WF^p = \emptyset$ by using the USR-to-predicate translation $\mathcal{F}$ mentioned in Section 3. If the predicate is statically `true` then L is filtered out, otherwise L (or $L - WF^p$) is kept.

Similarly, if the current node, named R, is a loop or callsite node, then we check using $\mathcal{F}$ whether an interval overestimate of R is included in WF^p; if so then R is filtered out, otherwise it is kept. If R is a gate or union or intersection node, then each term is filtered and the results are composed back. If R is a subtraction node $A - B$ then A is filtered with the union of WF^p and an interval underestimate of B.

Filtering the read-only set depicted in Figure 6(a) with the computed $\lfloor \cup_{k=1}^{i-1} WF_k \rfloor$ =`[civ@1,civ@2-1]` results in the simple $RO'_i = \{\texttt{i} - \texttt{1}\}$, where the problematic subtraction node, denoted by `(**)` in Figure, was simplified[3] to $\emptyset$.

3. Output-Dependence Pattern. When a CIV subscript is written on a 0-evolution path, then cross-iteration dependencies are likely to occur because the next writing iteration is likely to overwrite the same subscripts. We will now present a technique to remove at compile time such dependencies. To make the presentation clearer we demonstrate it on the loop `EXTEND_do400`, which contains the loop `EXTEND_do500`, whose VEG is shown in Figure 6. Assume the loop has per-iteration and partial-recurrence WF sets:

$\lceil WF_i \rceil$=`[civ@1,civ@8 ]`, $\lceil \cup_{k=1}^{i-1} WF_k \rceil$=`[civ,civ@1]`,
$\lfloor WF_i \rfloor$=`[civ@1,civ@8-1]`, $\lfloor \cup_{k=1}^{i-1} WF_k \rfloor$=`[civ,civ@1-1]`

where `civ@1` and `civ@8` are the μ and back nodes of the VEG associated to loop `EXTEND_do400` and variable `civ`. One can observe that output-dependencies may exist, since the output-independence equation of Figure 4(b) is not satisfied: $\lceil WF_i \rceil \cap \lceil \cup_{k=1}^{i-1} WF_k \rceil \equiv \{\texttt{civ@1}\} \neq \emptyset$.

Still, dependencies exhibit a well-structured pattern that our implementation exploits. The key observations are:

- Indices belonging to $\lfloor WF_j \rfloor$ underestimate do not result in cross dependencies: $\lfloor WF_i \rfloor \cap \cup_{k=1}^{i-1} \lfloor WF_k \rfloor = \emptyset$.
- All remaining indices, are overwritten by the next CIV-increasing iteration named `j`: $\lceil WF_i \rceil - \lfloor WF_i \rfloor \subseteq \lfloor WF_j \rfloor$. In our case we have $\lceil WF_i \rceil - \lfloor WF_i \rfloor$=`{civ@8}`, and rewriting $\texttt{civ@8}_i$ of iteration `i` as $\texttt{civ@1}_j$, i.e., the μ node of the next increasing iteration `j`, leads to the satisfied equation $\{\texttt{civ@1}_j\} \subseteq \lfloor WF_j \rfloor$=$[\texttt{civ@1}_j, \texttt{civ@8}_j\texttt{-1}]$.
- Last iteration increases the CIV value; this holds for `EXTEND_do400` because CIV is updated in an inner loop.

If all properties hold, then the output dependence is removed by privatizing the array (updates) and by copying out, at iteration's end, the indices belonging to the $\lfloor WF_i \rfloor$ *underestimate*, i.e., `[civ@1,civ@8-1]`. The last iteration copies in and out the overestimate (or updates directly), and because it increases CIV, it is guaranteed to overwrite the uncommitted part of previous (non-increasing) iterations. The last property is verified during the CIV-value (pre)computation. The first two are derived statically for the current loop, but in general, they may also use runtime verification.

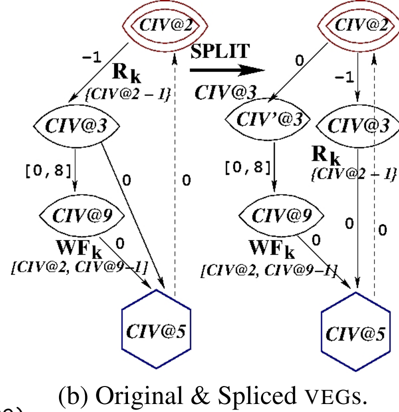
4.3 Privatizing Stack-Like Accesses

Stack like reference pattern is an important operation on arrays, and represent a non-monotonic evolution of the top of the stack CIV index. We will model such a reference

[3] The algorithm has used the VEG-derived properties `civ@1<jts≤civ@2` and `M≤civ@1`, where `civ@1` is the CIV value just before entering the loop, and `M` is the loop upper bound. Since `civ`-values are monotonically increasing, it also follows: $\texttt{i} \leq \texttt{M} \leq \texttt{civ@1} \leq \texttt{civ@2} < \texttt{nxt@3}$.

```
DO i = 1, N, 1
 civ = 1
 X(civ) =...
 WHILE ( civ.GT.0 )
  civ@2=γ(civ,civ@5)
  q =  X(civ@2)
  civ@3 = civ@2 - 1
  IF (q.LE.5) THEN
   res = res +..q..
  ELSE
   DO j = 1, 8, 1
    IF (civ.LT.SIZE)
      civ = civ + 1
      X(civ)=...
   ENDIF ENDDO
   civ@9=γ(civ@3,..)
  ENDIF
  civ@5=γ(civ@3,civ@9)
ENDWHILE ENDDO
```

(a) Loop ACCEL_do10.



(b) Original & Spliced VEGs.

Figure 8. a) Stack-like access in benchmark tree and (b) the VEG graphs of the while loop.

pattern as a sequence of monotonic evolutions, and reduce the problem to the one discussed in previous sections.

For example, the code in Figure 8(a) is a simplified version of loop ACCEL_do10 from tree benchmark. Each iteration of the outer loop maintains its own stack, denoted X, that is used in the while loop to compute a result res: The while loop is exited when the stack is empty, and each iteration pops one element and, later on, pushes up-to-eight new elements from/to the stack. The goal is to prove that the stack X is privatizable in the context of the outer loop.

The plan is to devise an analysis capable of verifying that the aggregated *read* and *write-first* sets at the while-loop level start from a cut-off point determined by the first iteration and expand monotonically, but in opposite directions, such that they do not overlap. In our case, the read set should be $\lceil$R$\rceil$=$\lceil$RO$\rceil\cup\lceil$RW$\rceil$=[0,civ-1] and the write-first set should be $\lceil$WF$\rceil$=[civ,SS], where SS denotes the stack size and civ=1 is the value before entering the while loop.

One can now check whether the read set of the while loop is included in the WF set of the region just before the while loop. In our case this reduces to checking $\{0\} \subseteq \{0\}$ because X(1) is written before the while loop. If this inequality holds then the read set of each iteration of the outer loop is the empty set, and X can be safely privatized.

The original VEG of the while loop, depicted on the left side of Figure 8(b), exhibits a non-monotonic CIV_μ evolution, i.e., it can be anywhere between minus one and seven. The intuition is to use an inductive summarization for the read and WF sets: *The base case* corresponds to analyzing the first iteration to determine a split point, denoted SP, from which, without loss of generality, the write-first set increases in the upper-bound direction, i.e., WF=[SP,...], and the read set increases in the lower-bound direction, i.e., R=[...,SP-1], where we have assumed for simplicity the stride to be 1. (The other case is treated in a similar fashion.)

With our example, WF_k =(q.GT.5)#[civ@2,civ@9-1] is associated with VEG node civ@9, and R_k ={civ@2-1}, as before, with civ@3, where k denotes an arbitrary iteration of the while loop. The split point is SP=civ=1, since, for the first iteration, civ@2 has the value at while's entry.

The second step is to attempt a semantics-preserving VEG transformation such that (i) each CIV-decreasing/increasing path holds only intervals belonging to the read/WF set, respectively, and (ii) a 0-path holds only empty intervals.

The semantics to be preserved is that the aggregated-read/WF summaries of any valid execution of the original VEG should be preserved by (at least) one valid execution of the transformed VEG. The approach is to search for a path where, for example, the read set occurs before a WF set. One can then splice a convenient node CIV_{sp} between the two, and introduce a 0-evolution path from CIV_{sp} to the back node, if none already exist, and a 0-evolution path from CIV_μ to the spliced node CIV^{cl}_{sp}. The original path is translated with the original path up to CIV_{sp}, then directly to the back node, followed by the 0-evolution path to CIV^{cl}_{sp}, and the rest of the original path. The process is repeated to a fix-point.

For example, the right side of Figure 8(b) shows the result of splitting node civ@3 of the original VEG: a new 0-evolution edge now connects civ@2 and civ'@3. In addition, the WF set has been adjusted to the new CIV_μ value, but has also been conservatively extended with the result of the read-set of the previous (transformed) iteration, resulting in the same WF_k interval as in the original VEG.

Finally, the third step is to aggregate the WF and read sets only on the monotonically increasing and decreasing (including 0) paths, respectively. If the read and WF results are exact, i.e., the under and overestimate are identical, and have shapes [...,SP-1] and [SP,...], respectively, then one can prove that the result of this piecewise monotonic aggregation matches the result of the original-while aggregation: a read-set overestimate is [0,SP-1], and privatization was proven.

4.4 The CIV Slice

The independence results discussed in previous sections have referred so far to disambiguating array accesses that exhibit CIV subscripts, but not to the computation of the CIV values. When the CIV variable is not privatisable in the context of the target loop, e.g., in Figure 2(a), the CIV is the source of cross-iteration flow dependencies: it is always updated in reduction-like statements, e.g., civ=civ+3*C(i), but it is read in arbitrary statements, e.g., in a subscript.

While related approaches [10, 23] avoid this, we always pre-compute in parallel, prior to loop execution, the CIV_μ values at the beginning of each (chunk of) iteration(s). Reasons are threefold: *First*, in most cases the runtime overhead is small. *Second*, the CIV values may dictate whether the loop is independent or not: For example,

the flow-independence predicate of the loop in Figure 2(a), $\texttt{N} \leq \texttt{Q} \vee \texttt{civ@5} < \texttt{M}$, depends on civ@5, and resolving statically the output dependencies of EXTEND_do400, depends, as shown in Section 4.2, on establishing whether the last iteration has increased the CIV value. *Third*, CIV values may flow in the computation of data (rather than only subscripts), e.g., the recurrent formula of the Sobol random number generator, used in benchmark Pricing of Section 5.

CIV_μ values are (pre)computed by extracting the loop-slice that contains the transitive closure of all statements that are necessary to compute the CIV variable (we use the control-dependency graph). We privatize all arrays and scalars, including CIV, that are not read-only inside the slice, where each iteration copies-in the indices of its $RO_i \cup RW_i$ set. Finally, an iteration-header statement is inserted to initialize CIV to the value just before the loop (Q), and similarly, the end-of-iteration value minus Q is saved into array CIVS.

Figure 9(a) shows the CIV slice of the loop in Figure 2(a), where scalars i and civ were privatized. The slice computes the per-iteration CIV increments and records them in CIVS. Then SCAN computes in parallel the prefix sum of the CIVS values, which are used each by one iteration of the parallelized loop. Correctness requirements are *twofold*:

First, all symbols that appear in the slice have been already proven to not introduce cross-iteration dependencies in the original loop; otherwise the computation of the slice might violate the semantics of the sequential execution.

Second, CIV variables may appear in the slice only in reduction-like statements, such as civ=civ+1. If this holds than the final CIV value would correspond to an (additive) reduction, and hence, the intermediate CIV_μ values are safely computed via the parallel prefix sum of the values in CIVS.

If the latter does not hold, then we have a cycle between CIV-value computation and their use. We address such a case via a fixed-point implementation, which (i) optimistically computes the CIV values as before, and then (ii) logically re-executes the slice on the computed-CIV values and checks (in parallel) whether the resulted value matches the input-value of the next iteration. If the check succeeds across all iterations, one can prove correctness by induction: The first iteration is always correct. If the value at the end of the first iteration coincides with the input of the second, then the second iteration is correct, etc.

Finally, CIV_μ input values are plugged in the parallel execution of the original loop: The arrays that were proven flow and output independent are shared, i.e., not privatized. For cases such as EXTEND_do400, that exhibit the special pattern of output dependencies, we adopt a strategy similar to the one used for CIV computation: arrays such as X are privatized, their read-set $\text{RO}_i \cup \text{RW}_i$ is copied in, and the per-iteration WF underestimate is copied-out at the iteration's end. Our technique is well integrated in the underlying compiler framework: predicates derived from equations on memory-reference sets are used in the computation of CIV summaries, and vice-versa.

5. Experimental Evaluation

While we have analyzed [14] more than two thousand loops from about thirty benchmarks (SPEC, PERFECTCLUB and more recent ones [12]), this section reports only five benchmarks in which our CIV analysis was crucial for efficient benchmark-level parallelization: we consider only CIV loops that have a significant global coverage and belong to the most beneficial schedule of parallelized loops. However, the number of loops exhibiting CIV subscripts is much larger, and we believe it is significant that 17% (5 out of 30) of the analyzed benchmarks contain essential CIV loops.

Figure 9(b) characterizes several representative loops, named in the third column, and their corresponding benchmarks, named in the first column. The reported parallel runtime were measured on P=8 cores.

The second column shows (i) the parallel and sequential runtime, $\text{T}_{P/S}$, measured in seconds, (ii) the percentage, SC, of the sequential runtime that has been parallelized, and (iii) the overhead of the associated runtime tests, if any, which is represented as percentage of the total-parallel runtime. With our benchmarks, the only runtime test that introduces non-negligible overhead is the (pre)computation of CIV values, denoted $\text{CIV}_{\texttt{COMP}}$, and presented in Section 4.4.

The third column shows the names of the most important loops, the fourth shows their sequential coverage, LSC, and the fifth shows each loop's parallel and sequential runtime, $\text{T}^L_{P/S}$, in seconds. We note that benchmarks bdna and nasa7 have poor scalability, e.g., very small performance gain from four to eight-core execution, because they exhibit small (historical) datasets: low-granularity important loops that limits the profitability of parallelization. In addition, (i) loop GMTTST_do120 is parallel but has loop-count three, and (ii) loop RESTAR_do15, which covers 9.3% of the sequential runtime, uses IO operations, and was run sequentially.

The sixth column shows *how* the loop was classified parallel: FI O(1) means that a predicate of runtime complexity O(1) has validated loop's flow independence at runtime. $\text{CIV}_{\texttt{AGG}}$ indicates that CIV-based summarization was instrumental in proving independence statically, and $\text{CIV}_{\texttt{COMP}}$ means that, in addition to $\text{CIV}_{\texttt{AGG}}$, the CIV_μ values were precomputed at runtime, i.e., CIV was not privatizable. ST-PAR indicates static parallelization of a "regular" loop.

Our Test Suite consists of benchmarks bdna and track from PERFECT-CLUB suite: (i) ACTFOR_do240 is a CIV loop that uses a vector in which elements are inserted at the end, and both the CIV and the vector array can be privatized, (ii) loop FPTRAK_do300 from track is similar to EXTEND_do400, and loop CORREC_do401 is not shown because its sequential coverage is under 1%. Benchmark nasa7 is part of SPEC2000 suite, and its loop EMIT_do5 is similar to ACTFOR_do240 except that output and flow independence is proven with runtime predicates, and it also requires precomputation of CIV values. Benchmark tree [5] is an implementation of the Barnes-Hut algorithm. Its main

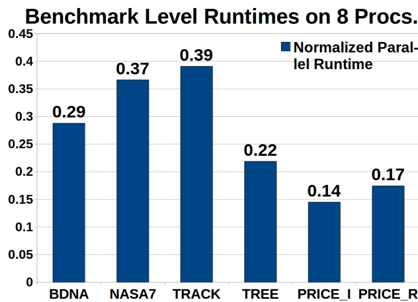
```
//slice for computing partial civ values
DOALL i = M, N, 1        $PRIVATIZED(civ,i)
  civ = Q
  IF ( C(i) .GT. 0 ) civ = civ + 3*C(i)
  CIVS(i-M+1) = civ - Q
ENDDOALL

//SCAN(op+,e,n,X) ≡{e,e+X(1),..,e+X(1)+..+X(n)}
SCAN(op +, Q, N-M+1, CIVS)

//civ values are plugged in the loop
DOALL i = M, N, 1        $PRIVATIZED(civ,i)
  civ = CIVS(i-M+1)
  ... rest of the loop code
ENDDOALL
```

(a)

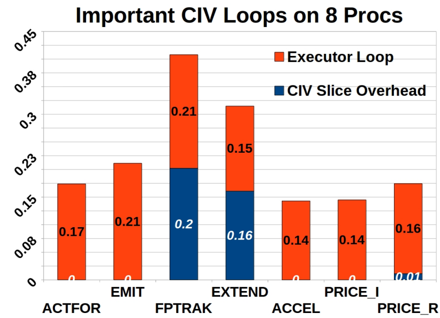
Properties of Benchmarks Exhibiting Important Loops That Use CIVs					
BENCH	PROPERTIES	DO LOOP	LSC%	$T^L_{P/S}$(s)	TYPE
BDNA	$T_{P/S}$=.19/.65 s	ACTFOR_500	47.8	.05/.31	ST-PAR
P=8	SC=87%,OV=0%	ACTFOR_240	35.6	.04/.23	CIV_{AGG}
NASA7	$T_{P/S}$=1.14/3.1 s	GMTTST_120	17.4	.27/.54	FI O(1)
P=8	SC=98%,OV=0%	EMIT_5	13.6	.09/.42	CIV_{COMP}
					OI O(N)
		BTRTST_120	10.1	.05/.31	FI O(1)
TRACK	$T_{P/S}$=6.6/16.8 s	FPTRAK_300	52.8	3.6/8.9	CIV_{COMP}
P=8	SC=97%,OV=45%	EXTEND_400	43.9	2.3/7.4	CIV_{COMP}
TREE	$T_{P/S}$=12.8/59 s	ACCEL_10	91.2	7.6/54	CIV_{AGG}
P=8	SC=91%,OV=0%				
PRICE_I	$T_{P/S}$=.29/2.0 s	PRICE_I_10	99	.29/2.0	CIV_{AGG}
P=8	SC=99%,OV=0%				
PRICE_R	$T_{P/S}$=.17/.98 s	PRICE_R_10	99	.17/.98	CIV_{COMP}
P=8	SC=99%,OV=6.4%				

(b)

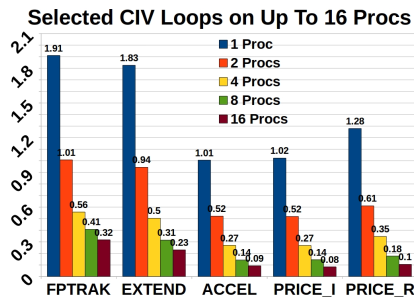
Figure 9. (a) Parallel Computation of CIV_μ Values. (b) Characterization of Important CIV Loops.



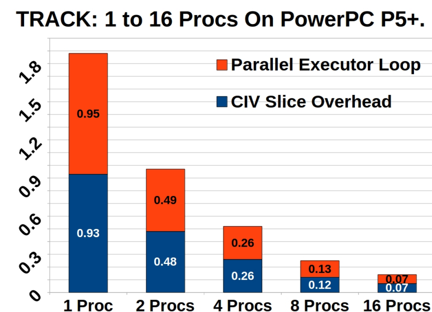
(a)



(b)



(c)



(d)

Figure 10. Benchmark and Loop-Level Normalized (Total) Parallel Runtime. Sequential Runtime is 1.

loop, `ACCEL_do10`, exhibits the stack-access pattern of Figure 8(a), and accounts for 91% of the sequential runtime. The remaining 9% corresponds to IO operations and cannot be parallelized. Finally, `price_i/r` is a (simplified) kernel of a real-world application that computes the price of a financial contract [12]: The difference between the two versions is that (i) `price_i` uses an independent Sobol-random-number-generator algorithm, i.e., computing the ith random number requires only the value of i, and exhibits a CIV pattern similar with `ACTFOR_do240`, while (ii) `price_r` uses a faster, recurrent formula that computes the ith random number based on the previous $(i-1)^{th}$ one. This means that in `price_r` the CIV is not used for indexing but directly in the computation of Sobol numbers, i.e., a parallel prefix scan with exclusive-or operator `xor`. Parallelization of `price_r` consists of (pre)computing in parallel these CIV_μ values, in the manner of Section 4.4. Because our implementation does not support `xor`, we have manually turned the `xor` into addition before compilation and then back to `xor` after compilation.

Experimental Methodology. Our source-to-source compiler receives as input a sequential Fortran77 program and automatically analysis loop-level parallelism and produces OpenMP code. The sequential and parallel code were compiled with gfortran, option -O3, and were run on a 16-core AMD Opteron(TM) 6274 system with 128GB memory.

Results. Figure 10(a) and (b) show the normalized-parallel runtime for each entire benchmark and CIV loop, respectively, where the CIV-computation overhead is included in each bar. Figure 9(b) has shown that the CIV-computation slice (i) represents a small fraction 6% of the execution of the pric_r benchmark, but (ii) it accounts for 45% of the parallel execution time for track. The latter case is not surprising since the slice contains almost all statements of the original loop, i.e., EXTEND_do400 and FPTRACK_do300, and both the loop and the slice are executed in parallel. Overall, for track we obtain a speedup of (only) 2.5x on eight cores.

The less-than-optimal runtime for bdna, nasa7, and tree was already explained, i.e., significant time spent in IO, small data sets. However, their loops show better results, e.g., ACCEL_do10 shows a 7.1x speedup on eight cores.

We have also tried to run scalability tests on a different architecture; an eight dual-core POWER 5+@1.9GHz with 32GB memory. The obtained results, depicted in Figure 10(d) show significantly-improved scalability up to sixteen cores for the total runtime. The CIV-computation overheads, also depicted, scale equally well.

In summary, on eight processors we report an average benchmark-level speedup of 4.33×, and an average CIV-loop speedup of 5.12×. The highest observed speedup, corresponding to price_i, is 12.5x on sixteen cores.

6. Related Work

Classical loop analysis examines each pair of read/write accesses, and models dependencies into linear systems of (in)equations that are solved statically via Gaussian-like elimination [1, 7]. Such analysis can drive powerful code transformations to optimize parallelism [17], albeit in the narrow(er) domain in which subscripts, loop bounds, if conditions are affine expressions of loop indices. At the opposite end of the spectrum are entirely-dynamic techniques [13, 20], which parallelize aggressively, but at the cost of significant runtime overhead.

Several static techniques have been proposed to handle irregular subscripts that have the shape of closed-form formulas in the loop indices. For example, The Range Test [3] uses symbolic ranges to disambiguate a class of quadratic and exponential subscripts by exploiting the monotonicity of the read-write pair of subscripts. A class of indirect-array subscripts is solved with an idiom-recognition method [10], where interprocedural analysis verifies the (assumed) monotonicity of the values stored in the index array. Furthermore, Presburger arithmetic was extended to support uninterpreted functions [19], and the irreducible-result formula is executed at runtime to disambiguate a class of irregular accesses.

A different type of approach, found more effective in solving larger loops, is to encode loop independence into one equation on abstract sets, for each array. The abstract set models the memory references of the corresponding array and is computed via interprocedural summarization of accesses [9, 11, 15, 16]. While each of these methods covers some irregular accesses, none of them handles CIV-subscripted loops, such as the ones analyzed in this paper.

A related body of work presents (i) symbolic-algebra systems that, for example, compute upper and lower bounds of nonlinear expressions [6], and (ii) techniques to characterizes scalar-value monotonicity within a loop [23, 24].

This solves only half of the problem, which corresponds to establishing the monotonic properties of CIVs. This paper addresses the second challenge: how to build array summaries by using the CIV-related invariants, where the immediate application is verifying loop independence. Related solutions analyze special cases of accesses: For example, a pair of accesses of form {X(CIV),X(CIV+d)}, where d is a constant, can be disambiguated by reasoning in terms of the (range of the) cross-iteration evolution of the corresponding CIV [26]. However, this does not address the case when a subscripts is affine and the other uses CIVs, i.e., it might prove output, but not flow independence for the loop in Figure 2(a), and it will not solve TRACK.

Enhancing the idiom-recognition support [10] allows to relate better the CIV properties with the dependency test: For example, when the written CIV subscript matches the pattern of a consecutively-written single-index (CW-SI) array, it is often possible to apply privatization, e.g., loop ACTFOR_do240 in bdna and the stack access in Figure 8(a). However, loops such as the one in Figure 2(a), or TRACK, are unanalyzable because the subscripts are neither single indexed, nor consecutively written.

In comparison, over/under-estimate summarization allows us, intuitively, to reduce the problem to a known idiom, albeit the code does not fall, strictly speaking, within that idiom. Furthermore, other than VEG, our analysis does not rely on pattern matching, but discovers non-trivial invariants that enable the CIV-agnostic test either (i) to prove loop independence, or (ii) to be tuned in a manner that resolves a specific pattern of dependencies, e.g., TRACK.

Finally, the value-evolution graph has been applied in the context of auto-parallelization [23]. The difference is that there, subscripts are separated early into CIV-based and affine and they do not mix: they are summarized under different representations, and disambiguated via specialized dependency tests. Analysis builds only accurate summaries, which may restrict the effectiveness of dependency tests. More specifically, there are not reported: (i) symbolic, non-constant CIV evolutions, e.g., Figure 2(b), (ii) the complex output-dependency pattern of TRACK, (iii) extraction of runtime predicates for CIV-dependence tests, (iv) prefix-sum

precomputation of CIV values, in the cases when they are used as data, rather than for indexing, (v) privatization of stack-like accesses, (vi) runtime and scalability results.

7. Conclusions

This paper has presented an analysis that summarizes both affine and CIV-based subscripts under the same representation. The result summaries are CIV agnostic and can be used for various purposes, e.g., dependence analysis or array SSA [21]. Our analysis seems less conservative than related approaches in that the algebra of under/over-estimates can exploit reference patterns that are similar, but do not fit exactly known programming idioms, such as push-back operations on vectors. We have reported an automatic solution that is well integrated in our compiler that combines static and dynamic techniques to aggressively parallelize loops. We have demonstrated the viability of the approach by evaluating it on five real-world applications.

Acknowledgments

This work was supported in part by the Danish Council for Strategic Research under contract number 10-092299 (HIPERFIT), and by NSF awards CCF 0702765, CNS-0551685, CCF-0833199, CCF-1439145, CCF-1423111, CCF-0830753, IIS-0917266, by DOE awards DE-AC02-06CH11357, DE-NA0002376, B575363, by Samsung, IBM, Intel, and by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST).

References

[1] U. Banerjee. Speedup of Ordinary Programs. *Ph.D. Thesis, Dept. of Comp. Sci. Univ. of Illinois at Urbana-Champaign*, Report No. 79-989, 1988.

[2] G. E. Blelloch. Scans as Primitive Parallel Operations. *Computers, IEEE Transactions*, 38(11):1526–1538, 1989.

[3] W. Blume and R. Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-Linear Expressions. In *Procs. Int. Conf. on Supercomp*, pages 528–537, 1994.

[4] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, and P. Tu. Automatic detection of parallelism: A grand challenge for high-performance computing. *IEEE Parallel and Distributed Technology*, 2, 1994.

[5] J. Edward. http://www.ifa.hawaii.edu/ barnes/ftp/treecode/. Technical report.

[6] T. Fahringer. Efficient symbolic analysis for parallelizing compilers and performance estimators. *Journal of Supercomputing*, 12:227–252, 1997.

[7] P. Feautrier. Dataflow Analysis of Array and Scalar References. *Int. Journal of Parallel Program.*, 20(1):23–54, 1991.

[8] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-Driven SSA Form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17:85–122, 1995.

[9] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Interprocedural Parallelization Analysis in SUIF. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):662–731, 2005.

[10] Y. Lin and D. Padua. Compiler Analysis of Irregular Memory Accesses. In *Int. Prg. Lang. Design Implem. (PLDI)*, pages 157–168, 2000.

[11] S. Moon and M. W. Hall. Evaluation of Predicated Array Data-Flow Analysis for Automatic Parallelization. In *Princ. Pract. of Par. Prog. PPoPP*, pages 84–95, 1999.

[12] C. Oancea, C. Andreetta, J. Berthold, A. Frisch, and F. Henglein. Financial Software on GPUs: between Haskell and Fortran. In *Funct. High-Perf. Comp. (FHPC)*, pages 61–72, 2012.

[13] C. E. Oancea and A. Mycroft. Set-Congruence Dynamic Analysis for Software Thread-Level Speculation. In *Lang. Comp. Par. Comp. (LCPC)*, pages 156–171. Springer, 2008.

[14] C. E. Oancea and L. Rauchwerger. Logical Inference Techniques for Loop Parallelization. In *ACM Int. Conf. Prog. Lang. Design and Implem. (PLDI)*, pages 509–520, 2012.

[15] C. E. Oancea and L. Rauchwerger. A Hybrid Approach to Proving Memory Reference Monotonicity. In *Int. Lang. Comp. Par. Comp. (LCPC)*, pages 61–75. Springer, 2013.

[16] Y. Paek, J. Hoeflinger, and D. Padua. Efficient and Precise Array Access Analysis. *ACM Transactions on Programming Languages and Systems*, 24(1):65–109, 2002.

[17] L.-N. Pouchet and et al. Loop Transformations: Convexity, Pruning and Optimization. In *Int. Conf. Princ. of Prog. Lang. (POPL)*, pages 549–562, 2011.

[18] W. Pugh. The Omega Test: a Fast and Practical Integer Programming Algorithm for Dependence Analysis. *Communications of the ACM*, 8:4–13, 1992.

[19] W. Pugh and D. Wonnacott. Constraint-Based Array Dependence Analysis. *Trans. on Prog. Lang. and Sys. (TOPLAS)*, 20(3):635–678, 1998.

[20] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Trans. on Parallel and Distributed System*, 10(2):160–199, Feb 1999.

[21] S. Rus, G. He, C. Alias, and L. Rauchwerger. Region Array SSA. *Int. Par. Arch. Comp. Tech. (PACT)*, pages 43–52, 2006.

[22] S. Rus, J. Hoeflinger, and L. Rauchwerger. Hybrid analysis: Static & dynamic memory reference analysis. *Int. Journal of Parallel Programming*, 31(3):251–283, 2003.

[23] S. Rus, D. Zhang, and L. Rauchwerger. Value Evolution Graph and its Use in Memory Reference Analysis. In *Int. Par. Arch. Comp. Tech. (PACT)*, pages 243–254, 2004.

[24] M. Spezialetti and R. Gupta. Loop Monotonic Statements. *IEEE Trans. on Software Engineering*, 21(6):497–505, 1995.

[25] P. Tu and D. Padua. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. In *Procs. Int. Conf. on Supercomputing*, pages 414–423, 1995.

[26] P. Wu, A. Cohen, and D. Padua. Induction Variable Analysis Without Idiom Recognition: Beyond Monotonicity. In *Int. Lang. Comp. Par. Comp. (LCPC)*, pages 427–441, 2001.