Logical Inference Techniques for Loop Parallelization

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

Lawrence Rauchwerger
Parasol Lab, Texas A&M University
rwerger@cse.tamu.edu

Abstract

This paper presents a fully automatic approach to loop parallelization that integrates the use of static and run-time analysis and thus overcomes many known difficulties such as nonlinear and indirect array indexing and complex control flow. Our hybrid analysis framework validates the parallelization transformation by verifying the independence of the loop's memory references. To this end it represents array references using the USR (uniform set representation) language and expresses the independence condition as an equation, $S = \emptyset$, where S is a set expression representing array indexes. Using a language instead of an array-abstraction representation for S results in a smaller number of conservative approximations but exhibits a potentially-high runtime cost. To alleviate this cost we introduce a language translation $\mathcal{F}$ from the USR set-expression language to an equally rich language of predicates ($\mathcal{F}(S) \Rightarrow S = \emptyset$). Loop parallelization is then validated using a novel logic inference algorithm that factorizes the obtained complex predicates ($\mathcal{F}(S)$) into a sequence of sufficient-independence conditions that are evaluated first statically and, when needed, dynamically, in increasing order of their estimated complexities. We evaluate our automated solution on 26 benchmarks from PERFECT-CLUB and SPEC suites and show that our approach is effective in parallelizing large, complex loops and obtains much better full program speedups than the Intel and IBM Fortran compilers.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel Programming; D.3.4 [*Processors*]: Compiler

General Terms Performance, Design, Algorithms

Keywords auto-parallelization, USR, independence predicates.

1. Introduction

Automatic loop parallelization requires the analysis of memory references for the purpose of establishing their data independence. For array references, compilers have used data dependence analysis which has historically taken two distinct directions: Static (compile time) analysis and run-time (dynamic) analysis.

Static dependence analysis, first proposed by [2, 4, 12, 22], analyzes an entire loop nest by modeling the data dependencies between any pair of read/write accesses. While this technique can also drive powerful code transformations such as loop interchange, skewing, etc., they are typically limited to the affine array subscript domain and their effectiveness [21] is limited to relatively-small loop nests. However, in the more general case, there are many obstacles to loop parallelization such as difficult to analyze symbolic constants, complex control flow, array-reshaping at call sites, quadratic array indexing, induction variables with no closed-form solutions [6, 13, 20]. Various techniques have been proposed to partially address some of these difficulties. For example, a class of index-array and stack-like accesses may be disambiguated via access pattern-matching techniques [15, 16] and some monotonic accesses (e.g., quadratic indexing) can be disambiguated via non-affine dependency tests [7]. Similarly, Presburger arithmetic can be extended to the non-affine domain [24] to solve a class of irregular accesses and control-flow. The next step has been to extend analysis to program level by summarizing accesses interprocedurally, where an array abstraction is used to represent a (regular) set of memory references either via systems of affine constraints [13], or linear-memory-access descriptors [20], respectively. Loop independence has been modeled typically via an equation on summaries of shape $S = \emptyset$. To improve precision, summaries are paired with predicates [14, 18] that guard (otherwise unsafe) simplifications in the array-abstraction domain or predicate the summary existence (i.e., control-flow conditions).

Run-time analysis is necessary and useful when static analysis alone cannot decide whether a loop is independent or not and thus needs to make the conservative choice, i.e., not parallel. Run-time analysis can always resolve static ambiguities because it can use instances of symbols and thus produce accurate results. Historically, dynamic dependence analysis has been performed by tracing and analyzing a loop's relevant memory references by executing an inspector loop (inspector/executor model [26]), or by tracing and analyzing a speculative parallel execution of the loop as it is done in thread-level speculation [25]. Such approaches typically extract maximal parallelism, at the cost of significant overhead, usually proportional to the number of traced dynamic memory references.

In order to attain our goal of effectively parallelizing a large number of codes automatically we have devised a hybrid compiler technology that can extract maximum static information so that the overhead of the dynamic analysis is reduced to a minimum.

1.1 Static Analysis with Light Weight Dynamic Complement

Summary based static analysis based on the array abstraction has been shown by previous research to be more scalable and useful for loop parallelization. However, from our experience we have found that its main source of inaccuracy lies in the fact that the array abstraction does not form a closed algebra under the required set operations: (recurrence) union, intersection, subtraction, gates, call sites. This shortcoming results in the necessity of conservative approximations during early construction stages, both at the array-abstraction and its associated-predicate levels. Thus either fewer loops are qualified as parallel or more dynamic analysis is needed. To mitigate this lack of scalability we have adopted a more expressive, higher level intermediate representation language. We are using the USR(Uniform Set Representation) [28], a composable language that subsumes both the array abstraction as well as the

PLDI'12, June 11–16, 2012, Beijing, China.

```
SUBROUTINE solvh(HE,XE,IA,IB)
  DIMENSION  HE(32, *), XE(*)
  READ(*,*) SYM, NS, NP, N
  CCC    SOLVH_do20
  DO i = 1, N, 1
    DO k = 1, IA(i), 1
      id = IB(i) + k - 1
      CALL geteu    (XE, SYM,  NP)
      CALL matmult(HE(1,id),XE,NS)
      CALL solvhe   (HE(1,id), NP)
    ENDDO
ENDDO END

SUBROUTINE matmult(HE,XE,NS)
  DIMENSION HE(*), XE(*)

  DO j = 1, NS, 1
    HE(j) = XE(j)
    XE(j) = ...
ENDDO END
```

```
SUBROUTINE geteu(XE,SYM,NP)
  DIMENSION  XE(16,*)

  IF (SYM .NE. 1) THEN
    DO i = 1, NP, 1
      DO j = 1, 16, 1
        XE(j, i) = ...
      ENDDO
    ENDDO
  ENDIF
END

SUBROUTINE solvhe(HE,NP)
  DIMENSION  HE(8, *)

  DO j = 1, 3, 1
    DO i = 1, NP, 1
      HE(j, i)=HE(j, i)+..
    ENDDO
ENDDO END
```

Figure 1. Simplified Loop SOLVH_DO20 from DYFESM Bench.

control flow. It includes in the language results of operations previously deemed outside the array-abstraction domain. Thus we can express the loop data independence summary equation as a the set equation $S = \emptyset$. Sometimes this equation is easy to prove statically. However, usually for real codes, the USR which needs to proven empty is very complex. Furthermore, for outer loops, the set expressions become very long and cumbersome to deal with during compilation.

To deal with this problem in a scalable manner, i.e., for outer, program level loops, we define an equally-rich language of predicates (PDAG) and an effective rule-based translation between the two languages, such that the result predicate p is a sufficient condition for loop independence: $p \Rightarrow \{S = \emptyset\}$. This transformation shifts the effort of proving loop independence from manipulating set expressions to that of handling logical expressions. Our translation scheme is general, allows (later) expansion and builds a less-constrained *predicate program* with fewer conservative approximations than those of related approaches (e.g., p's input symbols need not be read-only). Finally, we have developed a powerful and extensible logical inference based factorization algorithm that generates a set of sufficient conditions for parallelism. Some of these factors can be disambiguated statically and others need to be evaluated dynamically if aggressive parallelization is to be achieved. The generated sufficient run-time tests are ordered based on their estimated complexity and/or probability of success. Depending on the level of risk desired, the run-time complexity of the dynamic tests can be upper bounded during compilation.

1.2 A Simple Code Example

Figure 1 shows the simplified version of loop SOLVH_DO20 from the `dyfesm` benchmark. We will now analyze the references to arrays `XE` and `HE` to establish loop independence. If we consider `XE` and `HE` as unidimensional arrays, we can observe that `XE` is written in subroutine `geteu` on all indexes belonging to interval `[1,16*NP]` whenever `SYM.NE.1` holds, and is read in `matmult` on indexes in `[1,NS]`. Similarly, `HE` is written in `matmult` on all indexes in interval `[`τ`+1,`τ`+NS]`, and is read and written in `solvhe` on a subset of indexes in interval `[`τ`+1,`τ`+8*NP-5]`, where τ`=32*(id-1)` is the array offset of parameter `HE(1,id)`.

Flow independence of the outermost loop is established by showing that the per-iteration read sets of `XE` and `HE` are covered by their respective per-iteration write sets. This corresponds to solving equations `[1,NS]`$\subseteq$`[1,16*NP]` and `[`τ`+1,`τ`+8*NP-5]`$\subseteq$`[`τ`+1,`τ`+NS]` that result in the independence predicates `SYM.NE.1` $\wedge$ `NS`$\leq$`16*NP` and `8*NP`$<$`NS+6` for arrays `XE` and `HE`, respectively.

Examining *output independence*, we observe that the per-iteration write set of array `XE` is invariant to the outermost loop, hence `XE` can be privatized and updated with the values written by the last iteration (i.e., static-last value SLV). As discussed in Section 3.3, a successful predicate that proves that the per-iteration writes of `HE` do not overlap across iterations, hence `HE`'s output independence, is: $\wedge_{i=1}^{N-1}$`NS`$\leq$`32*(IB(i+1)-IA(i)-IB(i)+1)`.

Finally, proving `solveh_do20` independent requires predicates derived from both summary equations, such as NS$\leq$16*NP, and control flow, such as `SYM.NE.1`. Also `HE`'s output independence predicate requires a recurrence-based formula. The overhead represented by the dynamic predicate evaluation is negligible: O(1) and O(N), respectively, compared with the loop's O(N^2) complexity.

Our set to predicate translation and predicate factorization approach can be applied beyond parallelization to optimize problems such as last-value assignment and reduction parallelization. Furthermore, our design is open ended because it can readily accept more rules for translation and factorization. The monotonicity-based techniques in Section 3.3 represent such an extension example which has been well integrated in our framework.

1.3 Main Contributions

The main contributions of this paper are:

- A compiler framework that integrates a language translation $\mathcal{F}$ from the USR set-expression language to a rich language of predicates ($\mathcal{F}(S) \Rightarrow S = \emptyset$).
- A novel and extensible logic inference algorithm that factorizes complex predicates ($\mathcal{F}(S)$) into a sequence of sufficient-independence conditions which can be evaluated statically and, when necessary, dynamically.
- an experimental evaluation on twenty six `Fortran` benchmarks from PERFECT-CLUB, SPEC92, 2000, and 2006 suites.

The experimental evaluation demonstrates that: (i) the extracted predicates are successful (accurate) in most cases, (ii) the runtime overhead of these predicates is negligible, i.e., under 1% of the parallel runtime, while the other parallelism-enabling techniques show scalable speedup, and that (iii) we obtain speedups as high as 4.5x and 8.4x and on average 2.4x and 5.4x on four and eight processors, respectively, which are superior to those obtained by INTEL's `ifort` and IBM's `xlf_r` commercial compilers.

Compared with results reported in the literature, the evaluation shows that (i) our novel factorization scheme parallelized a number of previously unreported benchmarks via light-weight predicates, (ii) our unified framework solves a class of non-affine loops that have been previously analyzed with a number of different techniques, and that (iii) we match previous results obtained by SUIF [13] and POLARIS [5] on the statically-analyzable codes.

2. Preliminary Concepts: Summary Construction

Our solution to automatic, run-time parallelization builds on Rus, Pennings and Rauchwerger's hybrid analysis [28] which we briefly review. The main idea is to summarize accesses into read-only (RO), write-first (WF), and read-write (RW) abstract sets [14]. This is achieved by constructing summaries via interprocedural, structural data-flow analysis and representing them accurately in a scoped, closed-under-composition language, named unified set reference (USR). In this setting, the loop independence is derived from examining whether an summaries equation ($S = \emptyset$) holds.

2.1 Unified Set Reference (USR) Construction

Summaries are constructed during a bottom-up parse of the call and control dependency graphs (CDG) of a structured program in SSA representation, where within a CDG region nodes are traversed in

COMPOSE $(REG_1; REG_2)$

$(WF_1, RO_1, RW_1) \leftarrow REG_1$

$(WF_2, RO_2, RW_2) \leftarrow REG_2$

$WF = WF_1 \cup (WF_2 - (RO_1 \cup RW_1))$

$RO = (RO_1 - (WF_2 \cup RW_2)) \cup (RO_2 - (WF_1 \cup RW_1))$

$RW = RW_1 \cup (RW_2 - WF_1) \cup (RO_1 \cap WF_2)$

RETURN (WF, RO, RW)

(a) Consecutive Region Composition

AGGREGATE $(REG_i, i = 1, N)$

$(WF_i, RO_i, RW_i) \leftarrow REG_i$

$WF = \bigcup_{i=1}^{N}(WF_i - \bigcup_{k=1}^{i-1}(RO_k \cup RW_k))$

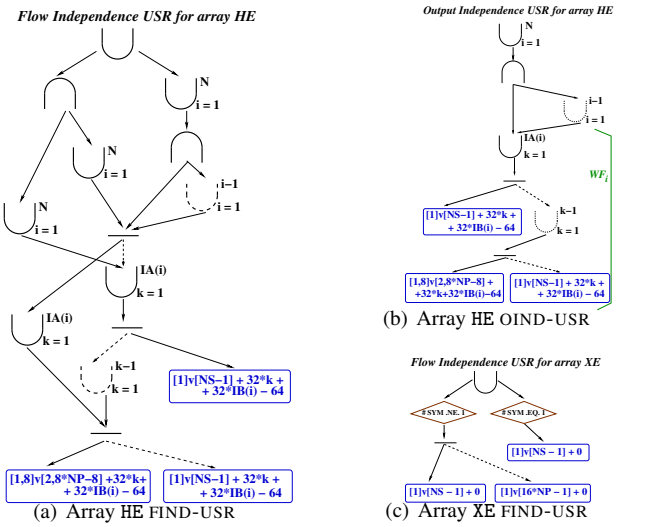
$RO = \bigcup_{i=1}^{N} RO_i - \bigcup_{i=1}^{N}(WF_i \cup RW_i)$

$RW = \bigcup_{i=1}^{N}(RO_i \cup RW_i) - (WF \cup RO)$

RETURN (WF, RO, RW)

(b) Loop Aggregation

Figure 2. Data-Flow Equations Used in USR Construction.



(a) Array HE FIND-USR (b) Array HE OIND-USR (c) Array XE FIND-USR

Figure 3. Flow/Output Independence USRs for HE, XE in Figure 1. Corresponding F/O-IND Predicates are: (c) SYM.NE.1$\wedge$NS$\leq$16NP, (a) 8*NP<NS+6, (b) $\wedge_{i=1}^{N-1}$NS$\leq$32*(IB(i+1)-IA(i)-IB(i)+1). Dotted line points to the subtracted part; dotted $\cup_{i=1}^{i-1}$ denotes partial recurrence under a fresh variable that ranges from 1 to $i-1$.

program order. 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. The latter two cases are illustrated in Figures 2(a) and 2(b). For example, the composition of a read-only region S_1 with a write-first region S_2 gives $RO = S_1 - S_2$ (i.e., S_2 cannot contribute to the RO result since it corresponds to a write access), and similarly, $WF = S_2 - S_1$ and $RW = S1 \cap S2$.

Summaries use a directed-acyclic-graph (DAG) representation, named USR, in which *leafs* are sets of linear memory access descriptors [20] (LMAD). LMADs define an algebra for aggregating index sets over quasi-affine patterns. For example a sequence of array accesses of stride δ with offset τ and bounded by (span) σ leads to the index set $\{\tau + i*\delta | \ 0 \leq i*\delta \leq \sigma\}$. In a straightforward generalization, an LMAD describes an arbitrary number of such access sequences into an unidimensional (unified) set of indexes:

$$\{\tau + i_1 * \delta_1 + ... + i_M * \delta_M \mid 0 \leq i_k * \delta_k \leq \sigma_k, k \in [1, M]\} \quad (1)$$

denoted by $[\delta_1, ..., \delta_M] \vee [\sigma_1, ..., \sigma_M] + \tau$, where strides δ_k and spans σ_k model "virtual" multi-dimensional accesses.

For example, the WF summaries for the write access to A at each level of the loop nest below are presented on the right-hand side:

```
Lo  DO i = 1, N, 1     [k,N]∨[k(M-1),N(N-1)] + k-1
Li    DO j = 1, M, 1     [k]∨[k(M-1)] + (i-1)N+k-1
St      A[i*N+j*k] = ...       []∨[] + (i-1)*N+jk-1
      ENDDO ENDDO
```

At statement St the summary is a point; aggregating the accesses over loop L_i, creates an 1D-LMAD of stride $\delta_{L_i} = \tau_{j \leftarrow j+1} - \tau =$ k and span $\sigma_{L_i} = \tau_{j \leftarrow M} - \tau_{j \leftarrow 1} =$ k(M − 1), etc. Note that an LMAD is transparent to the dimensionality of its corresponding array, and it does not guarantee that its dimensions do not overlap; this can be verified [14] via $\sum_{j=1}^{k-1} \sigma_j < \delta_k$ (i.e., N>k(M-1) in our example).

We found LMADs well suited for our representation because: (i) they support transparent array reshaping at call sites, as they are by definition a set of unidimensional points, and (ii) they provide better symbolic support, e.g., symbolic (constant) strides, are not affine constraints in Presburger arithmetic.

USR's *internal nodes* represent operations that cannot be accurately expressed in the LMAD domain: (i) irreducible set operations, such as union, intersection, subtraction ($\cup$, $\cap$, $-$), or (ii) control flow: gates predicating summary's existence, call sites across which summaries cannot be translated, or total ($\cup_{i=1}^{N}$) and partial ($\cup_{k=1}^{i-1}$) recurrences that fail exact aggregation. For example, with the USR in Figure 3(c), the set subtraction between LMADs $[1] \vee [\text{NS} - 1] + 0$ and $[1] \vee [16 * \text{NP} - 1] + 0$ cannot be represented in the symbolic LMAD domain, hence a subtraction USR node ($-$) was introduced. Moreover, the resulting (subtraction) set is part of the result *iff* gate SYM.NE.1 holds, etc. We note that resorting to conservative approximation instead of introducing the subtraction node, would *miss* the condition under which the loop is provably *independent*. Similar arguments can be made for the USRs in Figures 3(a) and 3(c).

2.2 Loop Independence as USR Equations (IND-USR)

Having summarized accesses at loop level, answering loop independence reduces to establishing the satisfiability of an independence equation in the USR domain. Denoting by (WF_i, RO_i, RW_i) the per-iteration write-first, read-only and read-write summaries of array X in loop L, where L's iterations range from 1 to N, the output independence of array X in L is represented via equation:

$$\{\cup_{i=1}^{N}(WF_i \cap (\cup_{k=1}^{i-1} WF_k))\} = \emptyset \quad (2)$$

Equation 2 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 no two different iterations write the same location, hence no cross-iteration dependency exists. We name the left-hand side of equation 2 the output-independence USR (OIND-USR) of X in L.

Similarly, flow-anti independence is established via equation:

$$\{(\cup_{i=1}^{N} WF_i) \cap (\cup_{i=1}^{N} RO_i)\} \cup \{(\cup_{i=1}^{N} WF_i) \cap (\cup_{i=1}^{N} RW_i)\} \cup \{(\cup_{i=1}^{N} RO_i) \cap (\cup_{i=1}^{N} RW_i)\} \cup \{\cup_{i=1}^{N}(RW_i \cap (\cup_{k=1}^{i-1} RW_k))\} = \emptyset \quad (3)$$

where we denote the left-hand side via FIND-USR. Figure3 depicts the independence USRs of our running example, where LMAD leafs were seen as intervals in Section 1.2.

When the independence USR (IND-USR) is definitely empty or non-empty the loop can be classified statically as independent or dependent, respectively. In numerous cases, however, independence is either statically undecidable, for example because certain variables are input dependent, or too complex to analyze with the current compiler infrastructure. In principle, a solution would be to directly evaluate IND-USR at run-time, and to implement conditional loop parallelization based on the independence result.

This technique works well in some special instances: (i) when IND-USR has $O(1)$ runtime complexity, as with the USR in Figure 3(c), or (ii) when its evaluation is amortized over many executions of the loop, i.e., USR evaluation has been safely hoisted out-

$g_1 = \texttt{SYM.NE.1}$ $S_1 = \texttt{[0,NS-1]} - \texttt{[0,16*NP-1]}$ $A = g_1 \# S_1$
$g_2 = \texttt{SYM.EQ.1}$ $S_2 = \texttt{[0,NS-1]}$ $B = g_2 \# S_2$

Translate $A \cup B$ i.e., $A \cup B = \emptyset \Leftarrow \mathcal{F}(A \cup B)$

$A \cup B = \emptyset \Leftarrow \mathcal{F}(A) \wedge \mathcal{F}(B)$

$A = \emptyset \Leftarrow \overline{g_1} \vee \mathcal{F}(S_1) = \texttt{SYM.EQ.1} \vee \mathcal{F}(S_1)$
$S_1 = \emptyset \Leftarrow \texttt{[0,NS-1]} \subseteq \texttt{[0,16*NP-1]} \Leftarrow \texttt{NS} \leq \texttt{16} * \texttt{NP}$
Hence $\mathcal{F}(A) = \texttt{SYM.EQ.1} \vee \texttt{NS} \leq \texttt{16} * \texttt{NP}$

$B = \emptyset \Leftarrow \overline{g_2} \vee \mathcal{F}(S_2) = \texttt{SYM.NE.1} \vee \mathcal{F}(S_2)$
$S_2 = \emptyset \Leftarrow \texttt{[0, NS-1]} \subseteq \emptyset \Leftarrow false$
Hence $\mathcal{F}(B) = \texttt{SYM.NE.1}$

Finally, $\mathcal{F}(A \cup B) = \texttt{NS} \leq \texttt{16} * \texttt{NP} \wedge \texttt{SYM.NE.1}$

Figure 4. Deriving the predicate program corresponding to the simple F-IND summary in Figure 3(c) via translation scheme $\mathcal{F}$.

side at least one outer loop. In the general case, however, we have found that runtime USR evaluation generates very high overhead, when compared with our technique.

3. Summary to Predicate Language Translation

IND-USR runtime evaluation is an expensive technique because it computes all the memory locations involved in potential cross-iteration dependencies, and as such, solves a more difficult problem than loop independence, which only requires classifying IND-USR as empty or non-empty. In contrast, our approach is to define an effective translation scheme $\mathcal{F}$ from USR to an equally expressive, closed-under-composition language of predicates, named PDAG: $\mathcal{F} : \text{USR} \rightarrow \text{PDAG}, \quad \mathcal{F}(S) \Rightarrow S = \emptyset$.

While the *predicate program* is just a sufficient condition for loop independence, it is less constrained than the flattened predicates of related approaches, e.g., it's input symbols need not be read-only, and it is also less conservative as a consequence of being constructed from non-trivial inference rules that pattern match the shape of an accurate independence summary (IND-USR). Finally, redundancy is removed by hoisting invariant terms outside loop-conjunction nodes, and the simplified predicate is factored into a sequence of sufficient conditions for loop independence, which are tested at runtime in the order of their estimated complexity.

This section is organized as follows: Section 3.1 presents at a high-level the language-translation scheme, implemented via a factorization algorithm. Section 3.2 describes how leaf predicates are extracted from operations in the LMAD domain. For completeness, Section 3.3 briefly explains more complex translation rules that naturally extend and are well integrated in our framework. Section 3.4 discusses two USR-reshaping transformations that enhance the resulting predicate accuracy. Finally, Section 3.5 explains how the result program is separated into a cascade of increasingly-complex predicates, and Section 3.6 discusses the asymptotic compile and run time complexity and the limitations of our framework.

3.1 Factorization Algorithm

Figure 4 demonstrates the gist of the language translation $\mathcal{F}$ on the simple IND-USR of Figure 3(c): A sufficient condition for the input USR, which is a union of two terms, to be empty is that each term is empty. Recursively, term A corresponds to a gated node, which exhibits a boolean expression g_1 under which summary S_1 exists. A sufficient condition for A to be empty is thus that either g_1 does not hold, or S_1 is empty. Finally, S_1 is the difference between two LMADS, seen for simplicity as intervals. If the first interval is included in the second then the difference is empty. The predicate corresponding to A is thus: $\texttt{SYM.EQ.1} \vee \texttt{NS} \leq \texttt{16*NP}$, and B is derived in the same manner. We note that, similarly to USR, the

PDAG **FACTOR**(S : USR)
*// **Output:** P s.t. $P \Rightarrow (S = \emptyset)$*
P = .FALSE.
***CASE** S **OF**:*
$q \# S_1$: $P = \overline{q} \vee$ **FACTOR**(S_1)
$S_1 \cup S_2$: $P =$ **FACTOR**(S_1) $\wedge$ **FACTOR**(S_2)
$S_1 - S_2$: $P =$ **FACTOR**(S_1) $\vee$ **INCLUDED**(S_1,S_2)
$S_1 \cap S_2$: $P =$ **FACTOR**(S_1) $\vee$ **FACTOR**(S_2) $\vee$ **DISJOINT**(S_1,S_2)
$\bigcup_{i=1}^{N}(S_i)$: $P = \bigwedge_{i=1}^{N}$ **FACTOR**(S_i)
$S_1 \bowtie$ *CallSite*: $P =$**FACTOR**(S_1)$\bowtie$*CallSite*

PDAG **DISJOINT**(S_1, S_2 : USR)
*// **Output:** $P \Rightarrow (S_1 \cap S_2 = \emptyset)$*
***IF** ($S_1 = \bigcup_{i=1}^{N}(S_i^1)$ **and** $S_2 = \bigcup_{i=1}^{N}(S_i^2)$)*
$(S_1^{\text{inv}}, S_2^{\text{inv}}) \leftarrow$ invariant overestimates of (S_i^1, S_i^2)
(1) $P =$**DISJOINT**(S_1^{inv}, S_2^{inv})
ELSE
$P =$ **DISJOINT_H**(S_1, S_2) $\vee$ **DISJOINT_H**(S_2, S_1) $\vee$ **DISJOINT_APP**(S_1, S_2)

PDAG **DISJOINT_H**(U, S : USR)
*// **Output:** $P \Rightarrow (U \cap S = \emptyset)$*
***CASE** U **OF**:*
$q \# S_1$: $P = \overline{q} \vee$ **DISJOINT**(S_1, S)
$S_1 \cup S_2$: $P =$ **DISJOINT**(S_1, S) $\wedge$ **DISJOINT**(S_2, S)
(2) $S_1 - S_2$: $P =$ **DISJOINT**(S_1, S) $\vee$ **INCLUDED**(S, S_2)
$S_1 \cap S_2$: $P =$ **DISJOINT**(S_1, S) $\vee$ **DISJOINT**(S_2, S)

(a) Factorizing IND-USR.

PDAG **INCLUDED**(S_1, S_2 : USR)
*// **Output:** P s.t. $P \Rightarrow (S_1 \subseteq S_2)$*
***IF** (S_1=$\bigcup_{i=1}^{N} S_i^1$ **and** S_2=$\bigcup_{i=1}^{N} S_i^2$)*
(3) $P_1 = \wedge_{i=1}^{N}$ **INCLUDED**(S_i^1, S_i^2)
ELSE** $P_1 =$INCLUDED_H**(S_1, S_2)*
$P = P_1 \vee$ **INCLUDED_APP**(S_1, S_2)

PDAG **INCLUDED_H**(S, U : USR)
*// **Output:** P s.t. $P \Rightarrow (S \subseteq U)$*
$P = P_1 = P_2 =$ .FALSE.
***CASE** U **OF**:*
$q \# S_1$: $P_1 = q \wedge$ **INCLUDED**(S, S_1)
$S_1 \cup S_2$: $P_1 =$ **INCLUDED**(S, S_1) $\vee$ **INCLUDED**(S, S_2)
(4) $S_1 - S_2$: $P_1 =$ **INCLUDED**(S, S_1) $\wedge$ **DISJOINT**(S,S_2)
$S_1 \cap S_2$: $P_1 =$ **INCLUDED**(S, S_1) $\wedge$ **INCLUDED**(S, S_2)
(5) LMAD(L): $P_1 =$ **FILLS_ARR**(L)
***CASE** S **OF**:*
$q \# S_1$: $P_2 = \overline{q} \vee$ **INCLUDED**(S_1, U)
$S_1 \cup S_2$: $P_2 =$ **INCLUDED**(S_1, U) $\wedge$ **INCLUDED**(S_2, U)
$S_1 - S_2$: $P_2 =$ **INCLUDED**(S_1, U)
$S_1 \cap S_2$: $P_2 =$ **INCLUDED**(S_1, U) $\vee$ **INCLUDED**(S_2, U)
$P = P_1 \vee P_2$

PDAG **INCLUDED_APP**(C, D : USR)
$((P_C, \lceil C \rceil), (P_D, \lfloor D \rfloor)) \leftarrow$ condit. LMAD over/under-estim. of C, D
$P = P_C \vee (P_D \wedge$ **INCLUDED_LMADS**($\lceil C \rceil$, $\lfloor D \rfloor$))

PDAG **DISJOINT_APP**(C, D : USR)
$((P_C, \lceil C \rceil), (P_D, \lceil D \rceil)) \leftarrow$ condit. cond. LMAD overestim. of C, D
$P = P_C \vee P_D \vee$ **DISJOINT_LMADS**($\lceil C \rceil$, $\lceil D \rceil$)

(b) Helper Functions.

Figure 5. Logical Inference Rules of The Factorization Algorithm.

predicate language has a DAG representation in which leaves are boolean expressions, while internal nodes represent either logical `and`, `or` operators ($\wedge$, $\vee$), or control-flow: untranslatable call sites ($\bowtie$*Call Site*), or irreducible loop-level conjunctions ($\wedge_{i=1}^{N} P_i$).

Figure 5 presents the *factorization algorithm* (FACTOR), which implements the translation scheme $\mathcal{F}$. Inference on set-algebra properties guides a recursive construction of a predicate program via a top-down traversal of the input summary. For example, a subtraction between two summaries is empty if the first operand is either empty or is included in the second operand, and similarly an intersection is empty if any operand is empty or the two operands are disjoint. In such cases INCLUDED and DISJOINT are called, respectively, to add add more specialized factors to the result:

A summary S is *included* in the difference of other two summaries $S_1 - S_2$ if, as in rule (4), S is included in S_1 and disjoint with S_2: $S \subseteq S_1 - S_2 \Leftarrow \mathcal{F}(S - S_1) \wedge \mathcal{F}(S \cap S_2)$. Two recurrence-union summaries over the same loop are in an include relation if, as in rule (3), the iteration-based summaries are in an include relation: $(\cup_{i=1}^{N} S_i^1) \subseteq (\cup_{i=1}^{N} S_i^2) \Leftarrow \wedge_{i=1}^{N} \mathcal{F}(S_i^1 - S_i^2)$. Finally, rule (5) extracts via FILLS_ARR the predicate under which an LMAD L fully covers the maximal (declared) dimension of its corresponding array; this predicate guarantees that any summary is included in L.

Similarly, a summary S is *disjoint* from the difference between two summaries $S_1 - S_2$ if, as in rule (2), either S is disjoint with S_1 or is included in S_2, since in the latter case S cannot be part of the subtraction result: $S \cap (S_1 - S_2) = \emptyset \Leftarrow \mathcal{F}(S \cap S_1) \vee \mathcal{F}(S - S_2)$.

However, disjointness of per-iteration summaries of the same loop does not imply that the recurrence-union summaries are disjoint: $(\cup_{i=1}^{N} S_i^1) \cap (\cup_{i=1}^{N} S_i^2) = \emptyset \not\Leftarrow \wedge_{i=1}^{N} \mathcal{F}(S_i^1 \cap S_i^2)$. In this case, rule (1) of Figure 5 attempts to find loop-invariant overestimates for S_i^1 and S_i^2, denote them S_{inv}^1 and S_{inv}^2, for example by filtering out loop-variant gates. The disjointness of the overestimates is now a sufficient condition for the disjointness of the two recurrence-union summaries[1]: $(\cup_{i=1}^{N} S_i^1) \cap (\cup_{i=1}^{N} S_i^2) = \emptyset \Leftarrow \mathcal{F}(S_{inv}^1 \cap S_{inv}^2)$.

A powerful inference rule that solves a large class of nonlinear accesses refers to the pattern $\cup_{i=1}^{N}(S_i \cap \cup_{k=1}^{i-1} S_k) = \emptyset$, which is satisfied under predicate $\wedge_{i=1}^{N} \texttt{MONOTON}(S_i)$. While this rather complex rule [19] is briefly demonstrated in Section 3.3, the intuition is that S_i monotonicity, e.g., the maximal element of S_i being smaller than the minimal element of S_{i+1} is a sufficient condition for the targeted summary equation to hold.

3.2 Extracting Predicates from LMAD Operations

When the factorization algorithm reaches LMAD leafs or encounters summaries of shapes that are not covered by any rules, INCLUDED_APP and DISJOINT_APP conservatively flatten the problem to the LMAD domain. We first show how flattening is achieved and then describe an algebra under which leaf predicates are extracted from comparing LMADs for inclusion and disjointness; the generalization to sets of LMADs being straightforward[2].

We have found most useful to represent an overestimate of summary C as a pair $(P_C, \lceil C \rceil)$, where P_C is a predicate under which C is empty, while $\lceil C \rceil$ is a set of LMADs that overestimates C. The latter is computed via a recursively defined operator on the USR domain, which, (i) on the top-down parse disregards node B in terms such as $C - B$, $C \cap B$, and (ii) on the bottom-up parse it translates, aggregates and unions the encountered LMAD leafs over call site, recurrence, and $\cup$ nodes, respectively. Similarly, D is underestimated via $(P_D, \lfloor D \rfloor)$, where $\lfloor D \rfloor$ is a (maximal) LMAD-set-underestimate of D when predicate P_D holds.

We recall from Section 2.1 that an LMAD is denoted by $[\delta_1, ..., \delta_M] \vee [\sigma_1, ..., \sigma_M] + \tau$ and represents the set of indexes $\{\tau + i_1 * \delta_1 + ... + i_M * \delta_M \mid 0 \leq i_k * \delta_k \leq \sigma_k, k \in [1, M]\}$. Also, under the simplifying assumption that all LMAD strides are positive ($\delta_k > 0$), one can observe that the interval $[\tau, \tau + \sigma_1 + .. + \sigma_M]$ overestimates its corresponding LMAD.

We first treat the case of *uni-dimensional* LMADs: Two 1D-LMADs can be proved disjoint in two scenarios: (i) they either correspond to an interleaved, but non-overlapping sequence of accesses, e.g., LMADs $A_1 = [2] \vee [99] + 0 \equiv \{0, 2, .., 98\}$ and $A_2 = [2] \vee [99] + 1 \equiv \{1, 3, .., 99\}$ are disjoint, or (ii) they can be overestimated by disjoint intervals, e.g., $A_1 = [2] \vee [49] + 0 \subseteq [0, 49]$ and $A_2 = [2] \vee [49] + 50 \subseteq [50, 99]$.

Formally, a sufficient predicate for $A_1 = [\delta_1]\mathrm{v}[\sigma_1] + \tau_1$ and $A_2 = [\delta_2]\mathrm{v}[\sigma_2] + \tau_2$ to be *disjoint* is: $gcd(\delta_1, \delta_2) \not| (\tau_1 - \tau_2) \vee (\tau_1 > \tau_2 + \sigma_2 \vee \tau_2 > \tau_1 + \sigma_1)$, where the first and second term satisfy the interleaved-access and disjoint-intervals scenarios, respectively. Similarly, using the same notations, one can observe that a sufficient predicate for A_1 to be *included* in A_2 is: $(\delta_2 \mid \delta_1) \wedge (\delta_2 \mid \tau_1 - \tau_2) \wedge (\tau_1 \geq \tau_2) \wedge (\tau_1 + \sigma_1 \leq \tau_2 + \sigma_2)$.

For example, in loop CORREC_DO711 from the bdna benchmark, loop independence requires to establish that [1]∨[0]+IX(2)+i-2 and [1]∨[i-2]+IX(1)-1 are disjoint (the loop index is i∈[1,NOP]).

PDAG **DISJOINT_LMAD**(A, B : LMAD)
*//**Input***: P s.t. $P \Rightarrow (A \cap B = \emptyset)$
IF(A and B unidimensional LMADs)
P = **DISJOINT_LMAD_1D**(A,B)
ELSE
(A^{1d}, B^{1d}) ← **FLATTEN_LMADS**(A,B)
P_{flat} = **DISJOINT_LMAD_1D**(A^{1d},B^{1d})
(C, D) ← **UNIFY_LMAD_DIMS**(A,B)
$(P_C^{wf}, C^{in}, C^{out})$←**PROJ_OUTER_DIM**($C$)
$(P_D^{wf}, D^{in}, D^{out})$←**PROJ_OUTER_DIM**($D$)
P_d^{out} = **DISJOINT_LMAD_1D**(C^{out}, D^{out})
P_d^{in} = **DISJOINT_LMAD** (C^{in}, D^{in})
$P = P_{flat} \vee (P_C^{wf} \wedge P_D^{wf} \wedge (P_d^{out} \vee P_d^{in}))$
(a) Predicate for Disjoint LMADs.

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) ***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)];
(b) Symbolic Fourier-Motzkin

Figure 6. Algorithm for Characterizing LMAD Disjointness.

Noting that the interleaved-access term evaluates to false (because 1 divides everything), the rules above yield predicate: IX(1)+1-IX(2)-i>0 ∨ IX(1)≤IX(2).

To eliminate loop index i from term IX(1)+1-IX(2)-i>0, we use a Fourier-Motzkin-like algorithm, depicted in Figure 6(b). The algorithm receives a symbolic expression expr and returns a sufficient predicate for expr > 0 to hold. First, a scalar symbol i of known upper and lower bounds is picked from expr, and expr is re-written as a*i+b, where i does not appear in b. If no suitable i is found, the result is expr>0. Finally, when a≥0 or a<0, i is replaced with its lower or upper bound, respectively, to give sufficient conditions for the inequation expr>0. Note that this leads to solving four subproblems of necessarily smaller exponent of i, which ensures that the recursion eventually terminates (in exponential time). In our example, the elimination of i yields term IX(2)+NOP≤IX(1), and the overall $O(1)$ predicate for loop CORREC_DO711 becomes IX(2)+NOP≤IX(1) ∨ IX(1)≤IX(2).

In general, *multi-dimensional* LMADs present two difficulties: first, an LMAD dimensions may overlap, and second, since there is no relation between the LMAD dimensionality and that of its corresponding array, we may have to compare LMADs that exhibit a different number of dimensions. We address this via a heuristic that unifies the LMADs dimensions, projects and compares a dimension at a time, and joins the result. In addition, predicates are extracted to guard the safety of the projection (i.e., the non-overlap invariant).

We demonstrate the approach, sketched in Figure 6(a), on bdna's loop CORREC_DO900, which requires to prove LMADs [M]∨[2*M]+j-1+2*M and [1,M]∨[j-2,2*M]+2*M disjoint, where the loop index j is in $1..N$. The first step is to flatten the input to 1D-LMADs and to extract a predicate P_{flat} as discussed before. The second step is to unify LMAD dimensions, e.g., the 1D-LMAD is padded with an empty dimension of stride 1 and span 0 giving: [1,M]∨[0,2*M]+j-1+2*M. Next, if both LMADs have equal outer strides, PROJ_OUTER_DIM(C) separates the outer dimension of C, returning a well-formedness predicate P_C^{wf}, together with LMADs C^{out} and C^{in} that correspond to the last and remaining dimensions.

With our example, C=[1,M]∨[0,2*M]+j-1+2*M splits into C^{in}=[1]∨[0]+j-1 and C^{out}=[M]∨[2*M]+2*M. P_C^{wf}=(j-1<M) verifies that the separated dimensions do not overlap – i.e. the range of the inner LMAD is less than the outer stride. Fourier-Motzkin simplification gives P_C^{wf}=(N≤M). Similarly, D=[1,M]∨[j-2,2*M]+2*M is split into D^{in}=[1]∨[j-2]+0, and D^{out}=[M]∨[2*M]+2*M, where P_D^{wf}=(N-1≤M). Recursively testing disjointness of the inner and outer LMADs gives predicates: P_d^{out}=false, since C^{out}=D^{out}, and P_d^{in}=true, since [j-1,j-1]∩[0,j-2]=∅. The independence predicate for CORREC_DO900 is thus $P_{flat} \vee (N \leq M)$.

[1] This rule solves several important loops from zeusmp, see Figure 9(b).

[2] The set of LMADs S_1 is disjoint from (included in) the set of LMADs S_2 if any LMAD in S_1 is disjoint to any (included in at least one) LMAD in S_2.

```
// Estimate ∪_{i=1}^{NRI}(WF_i)=
//    ∪_{i=1}^{NRI}([1]∨[2]+3*SHIFT(n))
(lower, upper) ← (⌈INT⌉,⌊INT⌋)
REDUCTION(MAX:upper)
REDUCTION(MIN:lower)
PRIVATE(tmp_lub,n)
DOALL n=1,NRI
  tmp_lub(1)=1+3*SHIFT(n)
  tmp_lub(2)=3+3*SHIFT(n)
  lower = MIN(tmp_lub(1), lower)
  upper = MAX(tmp_lub(2), upper)
ENDDOALL
```

(a) Estimating the size of `FSHIFT` in `gromacs`'s loop INL_DO1130.

```
CIV@1 = Q
DO i = 1, N, 1
  CIV@2 = γ(i.EQ.1, CIV@1,CIV@4)
  cond = X(i+M).NE.1 .AND. NSP(i).GT.0
  IF (cond) THEN
    DO j = 1, NSP(i), 1
      IF(0.NE.IA(1,i)) X(j+CIV@2 )=..
    ENDDO
    CIV@3 = NSP(i) + CIV@2
  ENDIF
  CIV@4 = γ(cond, CIV@3, CIV@2)
ENDDO
CIV@5 = γ(N.GE.1, CIV@4, CIV@1)
```

(b) CIV_{agg} example for loop CORREC_DO401 of `bdna`.

Figure 7. (a) Optimizing Reduction and (b) CIV Aggregation

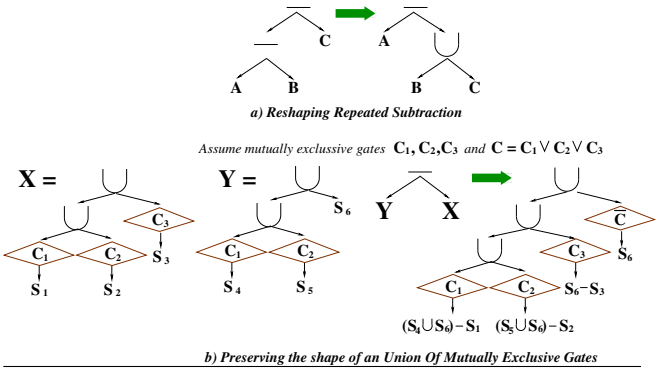


Figure 8. USR Reshaping Transformations.

3.3 Proving Non-Linear Access Independent

For completeness, this section sketches how our framework handles a class of non-linear accesses, such as those exhibiting array indexing or induction variables without closed-form solutions.

The first difficulty corresponds to quadratic or array indexing, which either appears directly in the code, e.g., sparse-matrix implementation uses index arrays, or as artifacts of transformations such as induction-variable substitution, e.g., quadratic indexes.

The intuition is that solving such accesses corresponds to a rather-complex rule, described in detail elsewhere [19], that translates equations of shape: $\cup_{i=1}^{N}(S_i \cap \cup_{k=1}^{i-1}(S_k))=\emptyset$ to the predicate domain. If S_i are seen as intervals, we can observe that if they form a monotonic sequence, e.g., the lower bound of the interval corresponding to iteration $i+1$ is always greater than the upper bound of that of iteration i – then the summaries S_i of any two consecutive iterations do not overlap, and by induction, any two summaries of distinct iterations do not overlap, hence the above equation is satisfied. Figure 3(b) shows such an example, where S_i =`WF`$_i$ is overestimated to interval `[32*(IB(i)-1),32*(IB(i)+IA(i)-2)+NS-1]`. Imposing the above strictly-increasing-monotonicity assumption results in predicate $\wedge_{i=1}^{N-1}$`NS≤32*(IB(i+1)-IA(i)-IB(i)+1)` that verifies output independence under $O(N)$-runtime complexity.

The second difficulty corresponds to the use of induction variables that are conditionally incremented (CIV), such as `CIV` in Figure 7(b). The solution is to devise a flow-sensitive USR-aggregation technique that succeeds in summarizing CIV accesses at iteration and loop level. For example, on the CFG path that takes the `if` branch (and contains the inner loop), an overestimate of the per-iteration write access to `X`, denoted W_i, is interval `[CIV@2+1,CIV@2+NSP(i)]`, which can be rewritten as `[CIV@2+1,CIV@4]` (since `CIV@4=CIV@3=CIV@2+NSP(i)`).

On the other path, W_i is the empty set, which can also be written as `[CIV@2+1,CIV@4]`, since the interval's upper bound `CIV@4=CIV@2` is less then its lower bound `CIV@2+1`. Hence the interval overestimate $\lceil W_i \rceil$=`[CIV@2+1,CIV@4]` holds on all paths. One can compute overestimates: $\lceil \cup_{k=1}^{i-1} W_k \rceil$=`[Q+1,CIV@2]`, and $\lceil \cup_{i=1}^{N} W_i \rceil$=`[Q+1,CIV@5]` in a similar fashion. Output independence is proven statically since `[Q+1,CIV@2]`∩`[CIV@2+1,CIV@4]` gives the empty interval. Observing that the read access to `X` can be overestimated via interval `[M,M+N]`, the flow independence predicate `Q≥M+N ∨ M>CIV@5` is extracted from the requirement that the loop summaries for the write and read accesses are disjoint (i.e., `[M,M+N]`∩`[Q+1,CIV@5]`=∅). Finally, predicate evaluation and parallel execution of the loop requires that the CIV values at the beginning of each iteration are precomputed via a loop slice that is also executed in parallel to ensure scalable speedup.

3.4 Enabling USR Transformation for Predicate Extraction

Predicates are constructed by pattern matching the shape of the input summary, and as such, semantically equivalent summaries may translate to predicates of varying accuracy. Figure 8 depicts two high-level USR reshaping rules that we have found to improve predicates' quality in a significant number of (important) loops.

The first rule says that a repeated (irreducible) subtraction from a summary should be reorganized as one subtraction between that summary and the union of the subtracted terms[3]. In Figure 8 we show intuitively that when A is included in neither B nor C, perhaps the union of B and C can simplify to a larger set in the array-abstraction domain which includes A, thus enabling the extraction of a more meaningful predicate. Note that related approaches would likely miss this opportunity because in the absence of a language, subtractions are performed in order, and the irreducible $A-B$ would have already been treated conservatively.

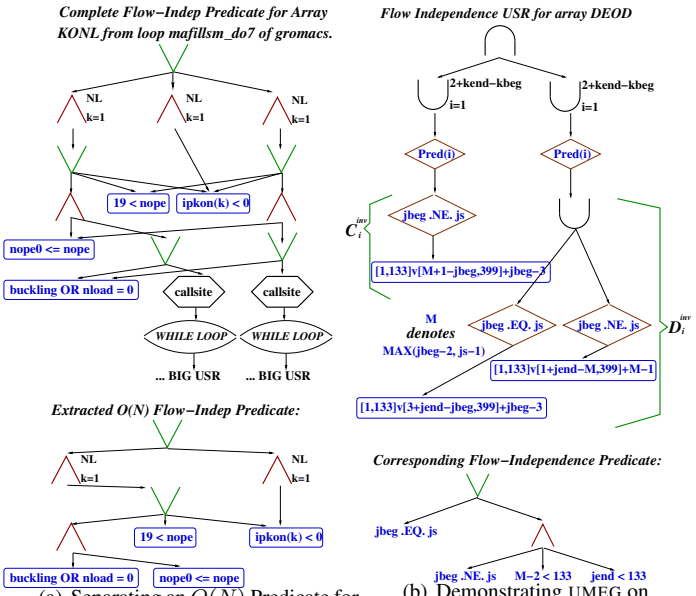
The second rule refers to preserving the shape of a *union of mutually exclusive gates* (UMEG) when subtracting, intersecting and uniting summaries of compatible-UMEG shapes, i.e, to distribute the operation inside each mutually exclusive gate, where by compatible shapes we mean that the gates of one summary are either a subset or match those of the other summary. The motivation is similar to the one for the first rule; the missing step in the figure being that, before the rule fires, Y is (semantically) normalized to:
$(C_1\#(S_4 \cup S_6)) \cup (C_2\#(S_5 \cup S_6)) \cup (C_3\#S_6) \cup (\overline{C}\#S_6)$.

The first transformation was useful in numerous loops, while the second was instrumental in parallelizing the larger SPEC2006 benchmarks ZEUSMP and CALCULIX. Figure 9(b) shows the *simplest* FIND-USR obtained via UMEG-preserving transformations for ZEUSMP's loop TRANX2_DO2100. Since C_i^{inv} and D_i^{inv} are loop-invariant overestimates for the two same-loop nodes of index k, `FACTOR` calls `DISJOINT`$(C_i^{\text{inv}}, D_i^{\text{inv}})$ which derives the independence predicate that succeeds at runtime:
`(jbeg.EQ.js) ∨ (jbeg.NE.js ∧ jend<133 ∧ M<135)`,
where the last two terms represent well-formedness predicates corresponding to separating the outer dimensions when comparing multi-dimensional LMADs (see Section 3.2). Without UMEG-preserving transformations, the FIND-USR is too large to be presented and none of the extracting predicates succeed at runtime.

3.5 Predicate Simplifications

The predicate program built under the language-translation scheme can be significantly optimized by (i) hoisting the loop-invariant terms outside loop nodes, and (i) by removing redundancy. The

[3] This resembles strength-reduction optimization, where addition is preferred to the more inexact subtraction operation.



(a) Separating an $O(N)$ Predicate for loop MAFILLSM_DO7 of `calculix`.

(b) Demonstrating UMEG on `zeusmp`'s TRANX2_DO2100.

Figure 9. Predicate-Separation & USR-Transformation Examples.

former transformation is essential in improving the accuracy of each term of the cascade of (partial) independence predicates.

For example *invariant hoisting* identifies loop-invariant children of a n-ary $\vee/\wedge$ node and hoists them outside the loop node, e.g., $\wedge_{i=1}^{N}(\vee(A_1^{\texttt{inv}},..,A_r^{\texttt{inv}},B_1^{\texttt{var}},..,B_p^{\texttt{var}})) \rightarrow \vee(A_1^{\texttt{inv}},..,A_r^{\texttt{inv}}) \vee \wedge_{i=1}^{N}(\vee(B_1^{\texttt{var}},..,B_p^{\texttt{var}}))$. Hoisting is mainly enabled by two transformations: First, $\vee/\wedge$ flattening merges repeated compositions of the same $\vee$ or $\wedge$ operator into one n-ary $\vee$ or $\wedge$ node, e.g., $(A_1 \vee A_2) \vee (A_3 \vee A_4) \rightarrow \vee(A_1, A_2, A_3, A_4)$, Second, common factor extraction, e.g., $\wedge(B_1 \vee A, .., B_p \vee A) \rightarrow \wedge(B_1, .., B_p) \vee A$, beside reducing redundancy, would allow now the loop-invariant predicate A to be hoisted outside the hypothetical loop node.

For example, running our algorithm on FIND-USR of Figure 3(a) identifies that a sufficient independence condition is that the bottom node[4] - $\texttt{([1,8]}\vee\texttt{[2,8*NP-8]+}\tau\texttt{)}-\texttt{([1]}\vee\texttt{[NS-1]+}\tau\texttt{)}$ - is empty, where τ=`32*(k-2+IB(i))`. This reduces to satisfying the interval inclusion $\texttt{[}\tau\texttt{,}\tau\texttt{+8*NP-6]} \subseteq \texttt{[}\tau\texttt{,}\tau\texttt{+NS-1]}$, which gives predicate $P_{\texttt{leaf}}$ = `8*NP<NS+6`. However, even though $P_{\texttt{leaf}}$ is invariant to loops of indexes `k` and `i`, the algorithm bottom-up pass still wraps it inside loop nodes, giving $\wedge_{i=1}^{N}(\wedge_{k=1}^{\texttt{IA(i)}}\texttt{8*NP<NS+6})$. Applying the above simplifications moves $P_{\texttt{leaf}}$ outside both loop nodes and removes the now empty loops, giving the $O(1)$ predicate `8*NP<NS+6`.

Finally, the predicate is factored into terms of increasing runtime complexity, typically $O(1)$ and $O(N)$, where code is generated to implement these tests and to cascade them to implement conditional parallelization, privatization, etc. Separating $O(1)$ predicates applies more aggressive rules to extract invariant factors, e.g., $\wedge_{i=1}^{N}(\wedge(A_1^{\texttt{inv}} \vee B_1^{\texttt{var}},..,A_p^{\texttt{inv}} \vee B_p^{\texttt{var}})) \rightarrow \wedge(A_1^{\texttt{inv}},..,A_p^{\texttt{inv}})$.

Separating $O(N)$ predicates is obtained by replacing any inner-loop node (i.e., nest depth > 1) in the original predicate via `false` and simplifying the result. Figure 9(a) demonstrates this technique: removing the two `while`-loop nodes in the complete predicate results in the much simpler $O(N)$ counterpart that succeeds at runtime, where scalars `nope` and `nope0` are loop variant.

[4] Note that a conservative approximation at that point in the summary construction would miss on extracting the relevant independence predicate.

3.6 Asymptotic Complexity and Limitations Discussion

With respect to the compile-time complexity, we note that the Fourier-Motzkin-like elimination is (only) exponential in the number of eliminated symbols. The typical case is that we eliminate only the outermost loop index via Fourier-Motzkin. Hence, we expect $O(1)$ overhead, where the inner-loop indexes are eliminated by LMAD-level aggregation. Furthermore, while the factorization algorithm has worst-case exponential complexity, the typical USR input exhibits a sparse structure in the operations that cause the exponential behavior. This means that, in practice, the compile time is dominated by the quadratic time of building USRs [28].

We model a predicate's runtime complexity after the loop-nest depth exhibited by its implementation, where we bound a potential explosion in predicate size via a convenient constant factor. We generate the entire cascade of predicates, noting that we have not yet encountered a first-successful predicate of complexity greater than $O(N)$, where N is the number of iterations of the targeted loop. The last test is always an exact one, i.e., implemented either as USR evaluation or by means of thread-level speculation.

There several possible avenues for future investigation such as: More aggressive rules to the translation, e.g., nonlinear accesses, currently disambiguated by checking their monotonicity, could be more accurately modeled if existential quantifiers would be part of the predicate language. Furthermore, we could enhance the precision of the LMAD-level comparison, which currently results in only sufficient conditions even in the case of unidimensional LMADs.

4. Other Uses of the Infrastructure

The factorization algorithm extracts predicates that satisfy an arbitrary equation in the USR domain, and as such, its applicability goes beyond proving loop independence. For example, one can disambiguate at runtime whether a symbol requires only static, rather than dynamic last value computation. Static last value (SLV), can be represented via USR equation $\cup_{i=1}^{N}\texttt{(WF}_i\texttt{)} \subseteq \texttt{WF}_{i\leftarrow N}$, which says that the whole write-first set of the loop of index $i \in [1, N]$ is included in the write-first set of the last iteration. Array `array_psi` of loop EMIT_DO5 of SPEC89's `nasa7` benchmark is such a case, where the SLV successful predicate is: $\wedge_{i=1}^{N-1}\texttt{(arrays\_nwall(}i\texttt{)} \leq \texttt{arrays\_nwall(}N\texttt{))}$.

Similarly, we apply runtime tests to optimize and extend the applicability of reduction parallelization. Consider the simple loop:

```
      DO i = 1, N, 1
S1 :      A(i) = ...
S2 :      A(B(i)) = A(B(i)) + ...
      ENDDO
```

First, disregarding statement S_1, we observe that S_2 matches the reduction pattern and the loop can always be parallelized by computing the changes to `A` locally for each processor, and by merging (adding) them in parallel at the end of the loop. However, if B is injective, this treatment is unnecessary because each iteration reads/writes a distinct element of `A`, and thus every processor can work safely, directly with the shared array `A`. The predicate[5] obtained from solving equation $\cup_{i=1}^{N}\texttt{(RW}_i \cap \cup_{k=1}^{i-1}\texttt{(RW}_k\texttt{))=}\emptyset$ is a sufficient condition for the access to be independent, and thus will guard a conditional application of *reduction* at *runtime* (RRED).

Second, considering now S_1, one can observe that while S_1 and S_2 do not form a reduction group [17] (i.e., S_1 is not a reduction), one can still parallelize the loop by treating S_2 as a reduction *if* `A`'s index sets for statements S_1 and S_2 do not overlap (e.g., predicate $\wedge_{i=1}^{N} N < B(i)$). We name this case *extended reduction* (EXT-

[5] With our framework we obtain predicate $\wedge_{i=1}^{N-1} B(i) < B(i+1)$ extracted by the monotonicity rule.

RRED), where we allow A to be written outside reduction statements as long as these writes do not precede, on any path, any reduction statement. One can observe that EXT-RRED instances will have non-empty write-first and read-write sets, corresponding to statements such as S_1 and S_2, respectively, and a necessarily empty read-only set. Enabling parallelism in this case requires proving (i) flow independence, i.e., $\cup_{i=1}^{N}(\mathtt{WF}_i) \cap \cup_{i=1}^{N}(\mathtt{RW}_i)=\emptyset$, and either (ii) output independence, i.e., $\cup_{i=1}^{N}(\mathtt{WF}_i \cap \cup_{k=1}^{i-1}(\mathtt{WF}_k))=\emptyset$ or (iii) computing the last value of the write-first set (e.g., the last value corresponds to the last iteration if $\cup_{i=1}^{N}(\mathtt{WF}_i) \subseteq \mathtt{WF}_{i\leftarrow N}$). Loops MXMULT_DO10 and FORMR_DO20 that cover almost 55% of dyfesm's sequential runtime, exhibit the EXT-RRED pattern.

Finally, in some cases such as gromacs and calculix benchmarks, the reduction is statically recognized but the bounds of the reduction array cannot be estimated at compile time because, for example, the array is passed as a parameter of assumed size to a Fortran subroutine called from C.

Our solution, shown in Figure 7(a) on the simplest example we encountered, is to compute at runtime the upper and lower bounds of the array indexes touched in the loop. This is achieved via overestimating $\cup_{i=1}^{N}(\mathtt{RW}_i)$ by removing terms B from nodes such as $A - B$, $A \cap B$, etc., such that the resulting USR exhibits only $\cup$, call site and recurrence nodes. In contrast to USR's exact evaluation, our lightweight USR-*bounds estimation*, named BOUNDS-COMP, allows parallel evaluation, where the lower and upper bounds are MIN/MAX-reduced across iterations. On gromacs we are 1.66x faster than IBM's xlf_r compiler, which, in the absence of array-bounds information, appears to parallelize the loop by executing its reduction statements atomically causing frequent cross-processor conflicts.

While the example of Figure 7(a) is trivial, a challenging application for BOUNDS-COMP are arrays such as AUB from calculix's mafillsm_do7, where $\mathtt{RW}_i$ is complex and prohibitively expensive to compute via exact USR evaluation. Still, BOUNDS-COMP's overhead is less than 9% of the parallel runtime and scales well.

5. Code Generation: Putting Everything Together

The factorization analysis is implemented in a variant of the Polaris research compiler [5] for Fortran77 and is applied on a control-flow-structured program under SSA representation. First, accesses are summarized via read-only, write-first and read-write USRs, and flow and anti independence USRs are computed for each symbol in the targeted loop. If IND-USR is decidable to be empty for all symbols or non-empty for at least one symbol then the loop is statically recognized independent or dependent, respectively. Exceptions are the cases when the symbol access is in a reduction pattern or when output dependencies can be fixed via the application of privatization and static last value. Next, a sequence of flow and output independence predicates is extracted for each unresolved symbol, where a true predicate still classifies independence statically.

Predicate code generation first extracts the loop slice that corresponds to the CDG-transitive closure of all statements that are necessary to compute the symbols appearing in the target predicate, where the non read-only symbols are privatized and copied in, if necessary. Next, one can observe that the definitions of the symbols appearing in a leaf node are necessarily on the same CFG path, hence the code for the leaf-node is placed immediately after the definition of the last symbol; we denote it the most dominated definition (MDD). Composition nodes, such as $\vee/\wedge$ are placed at the immediate common post-dominator of its child nodes, etc.

Predicates of non-constant complexity are evaluated in parallel, where we use and/or reduction to merge boolean results across iterations. Redundancy is further reduced by hoisting the calls to these predicates interprocedurally at the highest loop-dominator point where all predicate's input values are available (i.e., the MDD of its input values). It follows that all symbols' predicates are called on one CFG path, hence predicates can be cascaded (i.e., the first successful predicate disables the evaluation of the rest).

If all predicates fail, then we apply an exact, albeit potentially expensive, runtime test. If we can amortize the cost of the exact test against many execution of the loop (i.e., via hoisting), then we use direct evaluation of IND-USR, otherwise we use TLS [25].

Finally, we optimize parallelism by implementing conditional privatization, reduction or static-last value, where loop parallelization uses OpenMP directives. For example, code generation for an array X that requires the EXT-RRED of Section 4 uses a private copy of X, X1. Prior to (parallel) loop execution the reduction part of X1, i.e., $\cup_{i=1}^{N}(\mathtt{RW}_i)$ is zeroed out. Next, each iteration computes its $\mathtt{WF}_i$ set, and writes back to X the locations in $\mathtt{WF}_i$ at iteration's end. After loop termination, the locations involved in reduction statements, i.e., $\cup_{i=1}^{N}(\mathtt{RW}_i)$, are reduced across the X1's copies and X is accordingly updated. If the access is proven independent at runtime, then none of the above are necessary and the code uses shared-array X instead of its private copies X1.

6. Experimental Results

This section evaluates our auto-parallelizing approach on 26 benchmarks from the PERFECT-CLUB, SPEC1992, 2000 and 2006 suites. Tables 1, 2 and 3 characterize each benchmark as a whole, named in column one, and several of its representative loops, where columns three and four show the loop name and its contribution to sequential coverage (LSC) as percentage of the sequential runtime, respectively. Column five shows how loops have been classified: whether the loop has been proven sequential/parallel statically[6](STATIC-PAR/SEQ), or it uses predicates to prove flow/output independence (F/OI), and the complexity of the runtime test (O(1)/O(N)), where N refers to the number of iterations of the outermost loop.

Finally, column two characterizes the benchmark as a whole: (i) the sequential coverage (SC) and the corresponding number of measured loops (NL_{sc}), but also the total number of analyzed loops (NL_{tot}) when different than NL_{sc}, (ii) the sequential coverage of the loops that require runtime independence tests ($\mathrm{SC}_{\mathtt{rt}}$), and the overhead of these tests ($\mathrm{RT}_{\mathtt{ov}}$) represented as percentage of the parallel runtime, and (iii) the parallelization techniques used: privatization (PRIV), static/dynamic last value (SLV/DLV), static/runtime/extended reduction (SRED/RRED/EXT-RRED/BOUNDS-COMP), as presented in Section 4.

Additionally, (i) UMEG refers to the transformation of Section 3.4 that preserves the USR particular shape of an union of mutually exclusive gates, (ii) MON signals the use of monotonicity tests, and $\mathrm{CIV}_{\mathtt{agg}}$ and CIV-COMP refer to the summarization refinement in the presence of *conditional induction variables* (CIV) and to the parallel pre-computation of their iteration-wise values, as summarized in Section 3.3, and (iii) HOIST-USR means that independence was proven via runtime USR evaluation, where the USR has been successfully hoisted outside at least one loop, i.e., the overhead was amortized across the many loop executions.

Our compiler generates OpenMP-annotated Fortran source code. PERFECT-CLUB and SPEC92 benchmarks were compiled (-O2 -ipo) and compared with INTEL's ifort compiler version 11.1 (-O2 -ipo -parallel) on a commodity INTEL quad-core Q9550@2.83GHz machine with 8Gb memory. The larger SPEC2000/2006 benchmarks were compiled (-O4) and compared with IBM's xlf compiler version 13 (-O4 -qsmp=auto) on a 8 dual-core POWER5+@1.9GHz, 32Gb-memory machine. Benchmarks were run three times and the average was taken; This was

[6] In some cases the static decision refers to extracting a true predicate.

PERFECT CLUB Suite					
BENCH	BENCH PROPERTIES	SELECTED LOOPS	LSC %	GR ms	PAR/SEQ/ RT TEST
FLO52	SC=95%,NL$_{sc}$=30 SC$_{rt}$=.3%, RT$_{ov}$=0% PRIV,SRED,SLV RRED,NL$_{tot}$=199	PSMOO_do40 DFLUX_do30 EFLUX_do10 DFLUX_do40	19.5% 9.6% 8.2% 0.3%	.04 .08 .02 .01	STATIC-PAR STATIC-PAR STATIC-PAR OI O(1)
BDNA	SC=94%,NL$_{sc}$=6 SC$_{rt}$=0%, RT$_{ov}$=0% PRIV,S/RRED,CIV$_{agg}$ NL$_{tot}$=272	ACTFOR_do500 ACTFOR_do240 RESTAR_do15 ACTFOR_do320	59.5% 31.5% 4.8% 1.8%	69 36 28 .1	STATIC-PAR CIV$_{agg}$ STATIC-PAR STATIC-PAR
ARC2D	SC=97%,NL$_{sc}$=34 SC$_{rt}$=20%, RT$_{ov}$=.2% PRIV,SLV,MON NL$_{tot}$=207	STEPFX_do210 STEPFX_do230 XPENT2_do11,etc FILERX_do15	16.3% 11.9% 10.7% 9.0%	.8 .6 .002 1.3	STATIC-PAR STATIC-PAR FI O(1) FI O(1)
DYFESM	SC=97%,NL$_{sc}$=10 SC$_{rt}$=96%, RT$_{ov}$=.3% PRIV,EXT-RRED HOIST-USR,MON NL$_{tot}$=195	MXMULT_do10 SOLXDD_do10/4 SOLVH_do20 FORMR_do20	43.9% 27.3% 14.2% 10.5%	.006 .007 .03 .02	FI HOIST-USR OI O(N) OI O(N) F/OI O(1)/O(N) FI HOIST-USR OI O(N)
MDG	SC=99%,NL$_{sc}$=12 SC$_{rt}$=0%, RT$_{ov}$=0% PRIV,RRED,NL$_{tot}$=59	INTERF_do1000 POTENG_do2000 CORREC_do1000	92% 7.2% 0.1%	24 19 .04	STATIC-PAR STATIC-PAR STATIC-PAR
TRFD	SC=99%,NL$_{sc}$=4 SC$_{rt}$=34.8%, RT$_{ov}$=0% PRIV,SLV,MON NL$_{tot}$=44	OLDA_do100 OLDA_do300 INTGRL_do140 INTGRL_do20	63.7% 30.9% 3.9% 0.1%	18 9 2 .006	STATIC-PAR FI O(1) OI O(N) STATIC-PAR
TRACK	SC=97%,NL$_{sc}$=3 SC$_{rt}$=97%, RT$_{ov}$=47% PRIV,CIV$_{agg}$,NL$_{tot}$=88	EXTEND_do400 FPTRAK_do300 NLFILT_do300$3	49.2% 47.7% 1.2%	117 121 3.6	CIV-COMP CIV-COMP TLS
SPEC77	SC=76%,NL$_{sc}$=4 SC$_{rt}$=11%, RT$_{ov}$=0% PRIV,SRED,SLV	GLOOP_do1000 GWATER_do190 SICDKD_do1000	57.1% 16.5% 2.6%	31 9.5 1.3	STATIC-PAR TLS FI O(1)
OCEAN	SC=65%,NL$_{sc}$=14 SC$_{rt}$=45%, RT$_{ov}$=.1% PRIV,SLV,MON NL$_{tot}$=134	FTRVMT_do109 CSR_do20 SCSC_do30/40 RCS_do20	45.4% 5.2% 3.8% 1.8%	.01 .04 .03 .04	FI O(1) STATIC-PAR STATIC-PAR STATIC-PAR
QCD	SC=99%,NL$_{sc}$=6 SC$_{rt}$=1%, RT$_{ov}$=0% NL$_{tot}$=113	UPDATE_do1 UPDATE_do2 INIT_do2	31.9% 31.6% 1%	22 22 1.5	STATIC-SEQ STATIC-SEQ .. OI O(1)

Table 1. Properties of the PERFECT CLUB suite.
The layout of this table is explained in the beginning of Section 6

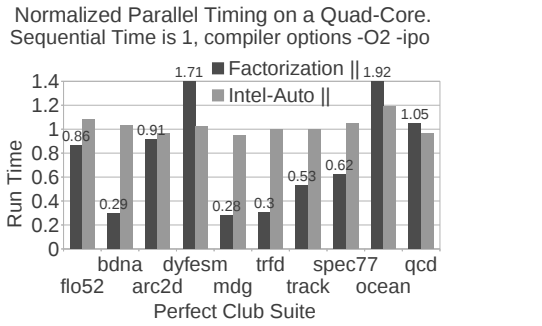


Figure 10. Timing Results for PERFECT-CLUB Suite.

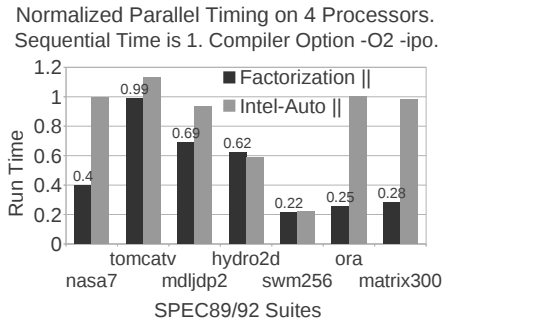


Figure 11. Timing Results for SPEC89/92 Suite.

deemed sufficient because their runtime typically exhibited negligible standard deviation.

6.1 Results Summary

Examining Figures 10, 11, 12 and 13, one can draw several high-level observations: First, the speedups achieved via our factorization approach are superior to the ones of INTEL's `ifort` and IBM's `xlf_r` compilers in all but four cases: `dyfesm`, `ocean`, `hydro2d`, and `qcd`. With `qcd` the results are close, and neither approach extracts significant parallelism. In the other cases, our approach successfully parallelizes a number of small-granularity loops which results in slowdown compared to sequential execution, while `ifort/xlf_r` fails to prove those loops parallel, hence executes them sequentially.

Second, the gains mainly reflect the commercial-compiler inability to parallelize the important loops of the corresponding benchmarks: (i) either because it lacks interprocedural dependence analysis, e.g., the benchmarks that were statically parallelized by SUIF a decade ago, or (ii) because it lacks extensive use of runtime-validation of parallelization (conditional parallelization, inspector/executor, speculative parallelization).

Third, the overhead of our techniques that enable parallelism at runtime is negligible in most cases, i.e., less than 1% of the parallel timing; the notable exceptions of `track`, `gromacs`, `calculix`, which still exhibit scalable speedup, will be discussed separately.

Fourth, we have classified parallelism exhaustively on a number of benchmarks, indicated via a large $\texttt{NL}_{\texttt{tot}}$, and the results support the feasibility of exploiting nested parallelism (e.g., `bdna` and `apsi` exhibit many inner loops that are solved via light predicates).

Finally, there are only two loops that require thread-level speculation (TLS): NLFILT_DO300 of `track` and GWATER_DO190 of `spec77`, and only one notable example where independence is proven via hoistable-USR evaluation: `apsi`'s RUN_DO20/40/etc.

6.2 PERFECT CLUB Suite Results

While being the oldest, PERFECT-CLUB suite is the most difficult to parallelize: `arc2d`, `dyfesm`, `trfd`, `ocean` and `bdna` all exhibit an abundance of flow and output independence predicates of $O(1)$ and $O(N)$ runtime complexity. Loops MXMULT_DO10 and FORMR_DO20 of `dyfesm` use the extended treatment for reduction of Section 4, while `trfd` and `dyfesm` are rich in the monotonicity predicates of Section 3.3.

Furthermore, `bdna`, but especially `track` use the CIV-aggregation refinement (CIV_{agg}) of Section 3.3. In `track`'s case, two while loops sum up to 97% of the sequential coverage. Parallelizing the two `while` loops requires (pre)computing the number of iterations of the loops, together with the per-iteration CIV values (CIV-COMP). The corresponding loop slice is almost as expensive as the loop, hence the runtime overhead is 47% of the total parallel timing. However, tested on extended datasets, parallelism scales well up to at least 16 processors (7.3x speedup).

Figure 10 shows the parallel timings under normal `-O2 -ipo` compilation. Unfortunately, PERFECT-CLUB uses (outdated) small datasets, which, in the cases of `flo52`, `arc2d`, `dyfesm`, and `ocean` results in loop granularities in the range of tens of microseconds, which are simply too small to amortize the thread-spawning overhead. However, extended datasets would likely enable scalable speedups on all four benchmarks; for example artificially increasing granularity by compiling under option `O0` results in speedups: 2.6x, 2.1x, 2.2x and 1.6x, respectively. The modest speedup of `ocean` is mainly due a sequential coverage of only 65%, because some parallel loops (`in,out`) with prohibitively small granularities (in the range of one microsecond) have not been considered. We have encountered similar problems in the `spec77` and `qcd` codes.

SPEC89 and SPEC92 Suites					
BENCH	BENCH PROPERTIES	SELECTED LOOPS	LSC %	GR ms	PAR/SEQ/ RT TEST
MATRIX300	SC=100%,NL$_{sc}$=9 SC$_{rt}$=26% RT$_{ov}$=0% PRIV,RRED	SGEMM_do160 SGEMM_do120 SGEMM_do20/40 SGEMM_do60/100	30.2% 30% 12.8% 12.8%	160 159 34 34	STATIC-PAR STATIC-PAR OI O(1) OI O(1)
SWM256	SC=99%,NL$_{sc}$=18 SC$_{rt}$=0% PRIV RT$_{ov}$=0% SRED	CALC2_do200 CALC3_do300 CALC1_do100	40.6% 29.7% 27.8%	.7 .5 .5	STATIC-PAR STATIC-PAR STATIC-PAR
ORA	SC=100%,NL$_{sc}$=4 SC$_{rt}$=0% PRIV, SLV RT$_{ov}$=0% SRED	MAIN_do9999 MAIN_do25 MAIN_do412	99.9% 0% 0%	999 0 0	STATIC-PAR STATIC-PAR STATIC-PAR
NASA7	SC=90%,NL$_{sc}$=9 SC$_{rt}$=43.6% RT$_{ov}$=.03% PRIV SLV, SRED, CIV$_{agg}$	GMTTST_do120 EMIT_do5 BTRTST_do120	21.1% 13.2% 9.4%	980 61 436	FI O(1) SLV O(N) CIV-COMP FI O(1)
TOMCATV	SC=100%,NL$_{sc}$=9 SC$_{rt}$=0% RT$_{ov}$=0% PRIV,SLV,SRED	MAIN_do60 MAIN_do100$2 MAIN_do120$2 MAIN_do80	37.8% 26.6% 10.9% 10.8%	7 .01 .01 2	STATIC-PAR STATIC-PAR STATIC-PAR STATIC-PAR
MDLJDP2	SC=87%,NL$_{sc}$=6 SC$_{rt}$=0% RT$_{ov}$=0% PRIV,S/RRED	FRCUSE_do20 POSTFR_do20 PREFOR_do60 POSTFR_do60	82.4% 1.6% 1.5% 1.1%	.9 .02 .02 .01	STATIC-PAR STATIC-PAR STATIC-PAR STATIC-PAR
HYDRO2D	SC=92%,NL$_{sc}$=58 SC$_{rt}$=0% RT$_{ov}$=0% PRIV	TISTEP_do400 FILTER_do300 T1_do10	17.6% 14.2% 7.5%	1.2 .1 .07	STATIC-PAR STATIC-PAR STATIC-PAR

Table 2. Properties of the SPEC89 and SPEC92 suites.
The layout of this table is explained in the beginning of Section 6

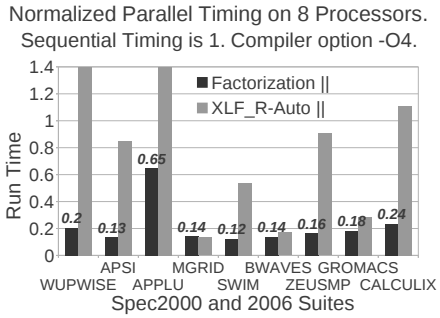


Figure 12. Timing Results for SPEC2000/2006 Suite.

6.3 SPEC89 and SPEC92 Suites Results

Figure 11 shows four-processor timings for several benchmarks in SPEC92 suite, from which `nasa7` uses independence predicates and CIV aggregation. While predicate overheads are negligible, `nasa7` obtains only 2.5x speedup because loops GMTTST_DO120 and EMIT_DO5 have 2 and 5 iterations, respectively. As with PERFECT-CLUB, we observe that our speedups are superior to the ones of `ifort`, and our lower speedups correspond to benchmarks that exhibit small loop granularity: `mdljdp`, `hydro2d` and `tomcatv`.

`Tomcatv` shows the need for further transformations in order to take advantage of parallelization. Loops MAIN_DO100$2 AND 120$2 exhibit too small granularities, to benefit from parallelization (in fact they suffer a slowdown). However the speedup increases to 2 on 8 procesors when the granularity is increased via loop interchange (though loosing locality). We expect loop tiling [27] (not implemented) to improve matters further.

6.4 SPEC2000 and SPEC2006 Suites Results

Figure 12 compares the parallel execution time of our approach against the one `xlf_r` on eight processors and shows that our speed-up is superior in most cases. Figure 13 shows our scalability speedups up to sixteen processors. Benchmark `gamess`, not measured, is notoriously difficult to parallelize [3] and we disambiguated only small-granularities loops that do not exhibit speedup, while `applu` exhibits two sequential loops that sum-up to 56% of sequential coverage and contain only sequential or small-granularity inner loops, which do not contribute to speedup.

SPEC2000 and SPEC2006 Suites					
BENCH	BENCH PROPERTIES	SELECTED LOOPS	LSC %	GR ms	PAR/SEQ/ RT TEST
WUPWISE	SC=93%,NL$_{sc}$=4 SC$_{rt}$=93% RT$_{ov}$=0%, PRIV RRED,SLV	MULDEO_do100 MULDEO_do200 MULDOE_do100 MULDOE_do200	20.6% 25.8% 20.7% 25.9%	206 258 207 259	F/OI O(1) F/OI O(1) F/OI O(1) F/OI O(1)
APSI	SC=99%,NL$_{sc}$=25 SC$_{rt}$=28% RT$_{ov}$=.2% HOIST-USR,PRIV,SRED SLV, NL$_{tot}$=252	RUN_do20/30/40 RUN_do50/60/70 WCONT_do40 DVDTZ_do40	17.6% 10.4% 11% 10.3%	176 122 330 314	FI HOIST-USR FI HOIST-USR STATIC-PAR STATIC-PAR
APPLU	SC=98%,NL$_{sc}$=9 SC$_{rt}$=0% RT$_{ov}$=0% PRIV,S/RRED,SLV	BLTS_do10 BUTS_do1 JACLD_do1 JACU_do1	28.4% 28.1% 14.1% 10%	119 117 59 314	STATIC-SEQ STATIC-SEQ STATIC-PAR STATIC-PAR
MGRID	SC=100%,NL$_{sc}$=12 SC$_{rt}$=0% RT$_{ov}$=0% PRIV	RESID_do600 PSINV_do600 INTERP_do800 RPRJ3_do100	51.5% 28.9% 4.9% 4.5%	42 7 2 2	STATIC-PAR STATIC-PAR STATIC-PAR STATIC-PAR
SWIM	SC=100%,NL$_{sc}$=18 SC$_{rt}$=0% RT$_{ov}$=0%, PRIV SRED, NL$_{tot}$=252	SHALOW_do3500 CALC2_do200 CALC1_do100 CALC3_do300	44.8% 20.5% 18% 15.4%	116 53 47 40	STATIC-PAR STATIC-PAR STATIC-PAR STATIC-PAR
BWAVES	SC=100%,NL$_{sc}$=20 SC$_{rt}$=0%,PRIV,SLV SRED,NL$_{tot}$=85	MAT*VEC_do1 FLUX_do2 SHELL_do5$2	75.1% 5.8% 4.2%	206 236 509	STATIC-PAR STATIC-PAR STATIC-PAR
ZEUSMP	SC=99%,NL$_{sc}$=51 SC$_{rt}$=10%,RT$_{ov}$=.01% PRIV,SLV UMEG	HSMOC_do360 MOMX3_do3000 TRANX2/3_do2100 TRANX1_do100	10.3% 5.1% 7.6% 2.4%	783 13 24 26	STATIC-PAR STATIC-PAR F/OI O(1) OI O(1)
GROMACS	SC=90%,NL$_{sc}$=4 SC$_{rt}$=90%,RT$_{ov}$=3.4% PRIV,RRED BOUNDS-COMP	INL1130_do1 INL1100_do1 INL1000_do1 INL0100_do1	84.8% 2.2% 1.9% 0.8%	33 5 4 1	BOUNDS-COMP BOUNDS-COMP BOUNDS-COMP BOUNDS-COMP
CALCULIX	SC=74%,NL$_{sc}$=1 SC$_{rt}$=74%,RT$_{ov}$=8.5% SRED,PRIV,UMEG BOUNDS-COMP	MAFILLSM_do7	73.7%	14s	BOUNDS-COMP F/OI O(N) F/OI O(1)
GAMESS	SC=32%,NL$_{sc}$=2 SC$_{rt}$=0% PRIV,RRED	DIRFCK_do300 GENR70_do170	18% 14.4%	.04 .03	STATIC-PAR STATIC-PAR

Table 3. Properties of the SPEC2000 and SPEC2006 suites.
The layout of this table is explained in the beginning of Section 6

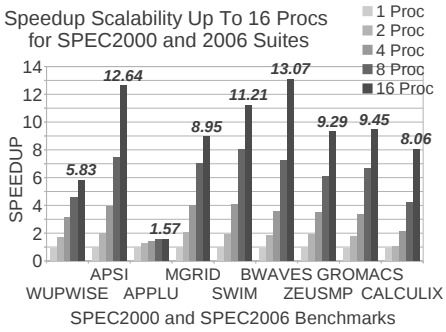


Figure 13. Scalability Results for Spec2006 Suite. Only the Fortran Parts of `gromacs` and `calculix` are measured.

We observe that speedups do not scale well between 8 to 16 processors; this is likely because the machine has eight dual-core processors, and executing on both cores decreases the per-core bandwidth. Benchmarks `mgrid`, `swim` and `bwaves` show good speedups, extracted statically. Both `calculix` and `gromacs` are written in a mixture of `C` and `Fortran`, from which we have analyzed and measured only the `Fortran` part, which shows a sequential coverage of 74% and 90% respectively. We remark that our speed-ups are superior to those of IBM's `xlf_r` compiler.

Half of the benchmarks use runtime parallelization techniques: `wupwise`, `zeusmp` and `calculix` use $O(1)$ and $O(N)$ flow and output independence tests, while `apsi` proves flow independence of loops such as RUN_DO20 via hoistable-USR evaluation. Both `gromacs` and `calculix` use reductions, where the target array is allocated in the `C` part and used in `Fortran` as an assumed-size-array parameter. Typical reduction implementation (e.g. OpenMP) requires to know the upper and lower bounds of the target array.

Our bounds-estimation technique, BOUNDS-COMP, described in Section 4, is responsible for the overheads $RT_{\text{ov}} = 3.4\%$ and 8.5%

of the parallel runtime of `gromacs` and `calculix`, respectively. BOUNDS-COMP's overhead (i) slightly increases for `gromacs` from normalized runtime .01 on one processor to .02 on 16 processors, due to the small granularity of the BOUNDS-COMP loop, but (ii) it scales perfectly with parallelism for `calculix`: from .16 on one processor to .01 on 16 processors. We note that our technique results in a parallel runtime about 1.66x faster than the one of `xlf_r` on `gromacs`, where `xlf_r`, in the absence of bounds information, appears to be wrapping reduction statements into atomic blocks (another `xlf` version exhibits slowdown).

7. Related Work

Solutions based on Presburger arithmetic [12, 22] analyze an entire loop nest at a time, albeit in the narrower affine domain where subscripts, loop bounds, `if` conditions are affine expressions of loop indexes. Both the memory dependencies and the flow of values between every pair of read-write accesses are accurately modeled via systems of affine inequations, which are solved by gaussian-like elimination. These solutions drive powerful code transformations to optimize and enable parallelism [21], but they are most effective when applied on relatively-small, intra-procedural loop nests exhibiting simple control flow. In comparison, our technique is better suited to parallelize larger loops, but is less effective in driving code transformations such as loop interchange, skewing, tiling, etc.

Pugh and Wonnacott are the first to show how to interpret irreducible Presburger formulas as simple predicates that can be verified either by the user or at runtime. The approach, named conditional dependency analysis [24], existentially quantifies the variables that correspond to the loop index in the Presburger formula of the flow dependence and computes the "gist" of the obtained formula, by removing false-alarm terms. Analysis is extended with uninterpreted-symbol functions [23] to model non-affine terms and a limited notion of control flow, where inductive simplification derives simpler, (only) sufficient conditions for independence.

To extend analysis to program level, several solutions have been proposed that (i) summarize accesses via an array abstraction, as dictated by (structural) data-flow analysis, and (ii) model loop independence via equations on these summaries. These approaches typically perform conservative approximations to keep the (successive) summary results within the array abstraction domain, e.g., they fail to disambiguate some more complicated cases of coupled subscripts, but they accommodate better more-complex control flow. For example, the simple loop [24]:

```
DO i = 1, N, 1
   IF(p[i] > 0) THEN A = ...
   ELSE              A = ... ENDIF
ENDDO
... = A
```

produces an inexact flow-dependence result for scalar `A` under Pugh and Wonnacott's approach: $\{[] \mid \nexists i\ s.th.\ 1 \leq i \leq n \wedge p(i) > 0\}$, but the `IF`-data-flow equation of a summary-based solution would identify that both mutually exclusive branches guard the same summary, hence the non-affine gate (`p[i]`) can be safely discarded.

Hoeflinger *et* al. use the LMAD [20] array abstraction to summarize accesses into read-only, write-first and read-write sets. The ART test [14] builds iteration-level summaries and aggregates them over the targeted loop. If this aggregation creates a new LMAD dimension that does not overlap with the existent dimensions, then a monotonicity-based argument establishes that independence holds.

Hall *et* al. organize summaries as systems of affine inequations and analyze the read, write and exposed-read abstract sets to establish loop independence [13]. In both approaches summaries are paired with predicates, typically extracted from control flow, that guard the summary existence. In addition, Moon and Hall also extract predicates that guard (otherwise unsafe) simplifications in the array abstraction domain, and use invariants synthetized from branch conditions to enhance the precision of the summary [18].

Adve and Mellor-Crummey present an interesting instance of an equational system, where summaries are represented via Presburger formula, and summary equations model computation partitioning and communication analysis for data-parallel programs [1]. The system defines a rich compositional algebra that starts from a set of user-define inputs, such as the alignment of an array with a template and the home of certain statements, and computes the set of locations that need to be sent/received to/from other processors. This formalism drives several important optimizations, such as message vectorization, in-place communication and in-place splitting. Whenever the result summary falls outside the affine domain, inspector/executor techniques are used to verify the desired invariant.

In comparison, rather than requiring constraints to be affine or resorting to conservative approximations, we build on Rus *et* al.'s language of USRs [28], recalled in Section 2, to construct exact read-only, read-write, and write-first summaries. Since runtime evaluation of USRs exhibits in many cases unacceptably-large overheads, we define a *language translation* scheme to an equally expressive language of *predicates*.

The obtained predicate program is less constrained than the ones of related solutions, and its construction typically involves less conservative approximations. For example, when predicate symbols are mutable we use a program slice to compute them. We improve both qualitatively and quantitatively on Rus *et* al. work [28, 29] in that: we define a complex system of inference rules, which are used more aggressively to derive predicates, and we show that predicates can disambiguate many loops that were either unreported or previously solved via the more expensive USR evaluation. Finally, we present the compiler infrastructure for improving predicates' accuracy (e.g., USR reshaping rules), for factorizing and cascading the predicate program into a set of increasingly-complex conditions for independence, and for implementing conditional reduction, etc.

A significant amount of work was aimed at disambiguating a class of irregular subscripts exhibiting quadratic or array indexing or induction variables without closed-form solutions. One direction was to enhance the mathematical support with more accurate symbolic ranges [8, 10], or more encompassing algebras, such as representing induction-variables via chains of recurrence [9]. For example, Blume and Eigenmann's Range Test [7] uses the extended range support and exploits the monotonicity of read-write accesses to disambiguate a class of quadratic or coupled indexing (e.g., loops `olda_do100/300` from `trfd` or `ocean`'s `ftrvmt_do109`).

Lin and Padua extend the library of recognizable access patterns to solve: (i) stack/queue-access patterns [16] (e.g., loops from `bdna`, `p3m` and `tree`, but not `track`), and (ii) index-array accesses [15]. The latter extends the applicability of the Range Test to cover array indexing, for the cases when the index-array value properties assumed by pattern recognition can be statically verified (e.g. loops from `trfd` and `dyfesm`, but not `solvh_do20`).

Pugh and Wonnacott extension of Presburger arithmetic [23] also solves a number of loops that would fall outside the traditional affine domain: for example some tricky cases of coupled subscripts, or the indirect array pattern of loop `intgrl_do540` from `trfd` or the non-trivial control flow of `mdg`'s `interf_do1000` and `poteng_do2000`, or the quasi-affine pattern of loop `filerx_do290` from `arc2d`. However, loops `olda_do100/300` from `trfd` are only recognized to form a monotonic indexing sequence before induction variable substitution, while loops `actfor_do240` and `extend_do400` from `bdna` and `track` are not disambiguated.

Finally, the ART test [14] of Hoeflinger *et* al. disambiguates a class of coupled subscripts and exponential indexing.

In comparison, we present a *unified framework* that parallelizes a large class of loops that have been previously analyzed with a

number of different techniques: for example we match the results reported by SUIF [13] on a number of statically analyzable benchmarks (e.g., `mdg`, `ora`, `swim`, `applu`, `mgrid`, `hydro2d` etc.), and also solve most of the non-affine benchmarks that require conditional analysis. The latter corresponds in part to a complex translation rule [19] that exploits the monotonicity of a summary of a particular shape (see Section 3.3). Notable exceptions are the stack-access pattern of benchmark `p3m`, and several loops exhibiting exponential indexing and tricky instances of coupled subscripts. We solve loop `run_do20` from `apsi` via expensive USR evaluation, but which still results in negligible overhead due to the use of memoization. In addition we parallelize a number of previously unreported benchmarks using our minimal-weight predicates (e.g., `calculix`, `zeusmp`, `wupwise`, `nasa`, `track`, `gromacs`).

The other main direction of approaching autoparallelization has been to analyze memory references at run-time, either via inspector/executor [26], or via TLS techniques [25], or via faster but less scalable techniques [30]. These techniques have overhead proportional to the number of the original-loop accesses, and hence we use then only as last resort, once all the lighter predicates failed.

8. Conclusions

In this paper we presented a fully automatic approach to loop parallelization that integrates the use of static and run-time analysis and thus overcomes many previously known difficulties. Starting from our rich array reference representation, the USR language, we expressed the independence condition as an equation, $S = \emptyset$, where S is a set expression representing array indexes. We introduced a language translation $\mathcal{F}$ from the USR set-expression language to a language of predicates ($\mathcal{F}(S) \Rightarrow S = \emptyset$). Loop parallelization is then validated using a novel logic inference algorithm that factorizes the obtained complex predicates ($\mathcal{F}(S)$) into a sequence of sufficient-independence conditions which are evaluated first statically and, when necessary, at run-time, in increasing order of their estimated complexities. The experimental evaluation on 26 benchmarks from PERFECT-CLUB and SPEC suites and show speedups as high as 4.5x and 8.4x and on average 2.4x and 5.4x on four and eight processors, respectively, which are superior to the ones of INTEL's `ifort` and IBM's `xlf_r` commercial compilers. Our translation technique from set to predicate language and our factorization algorithm are extensible and can be applied to a variety of conditional optimizations. It is powerful because it can make use of dynamic information very efficiently.

Acknowledgments

This work was supported in part by NSF awards CRI-0551685, CCF-0833199, CCF-0830753, IIS-0917266, NSF/DNDO award 2008-DN-077-ARI018-02, by the DOE NNSA PSAAP grant DE-FC52-08NA28616, IBM, Intel, Oracle/Sun and Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST). Since November 2011 Cosmin Oancea was supported by Danish Strategic Research Council, for the HIPERFIT research center under contract 10-092299.

References

[1] V. Adve and J. Mellor-Crummey. Using Integer Sets for Data-Parallel Program Analysis and Optimization. In *Procs. Int. Conf. Prog. Lang. Design and Implementation*, 1998.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002. ISBN 1-55860-286-0.

[3] B. Armstrong and R. Eigenmann. Application of Automatic Parallelization to Modern Challenges of Scientific Computing Industries. In *Int. Conf. Parallel Proc.* , pages 279–286, 2008.

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

[5] W. Blume *et* al. Parallel Programming with Polaris. *Computer*, 29(12), 1996.

[6] W. Blume and R. Eigenmann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. *IEEE Transactions on Parallel and Distributed Systems*, 3:643–656, 1992.

[7] 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.

[8] W. Blume and R. Eigenmann. Demand-Driven,Symbolic Range Propagation. In *Procs. Int. Lang. Comp. Par. Comp.* , 1995.

[9] R. A. V. Engelen. A unified framework for nonlinear dependence testing and symbolic analysis. In *Procs. Int. Conf.on Supercomp*, pages 106–115, 2004.

[10] T. Fahringer. Efficient Symbolic Analysis for Parallelizing Compilers and Performance Estimator. *Journal of Supercomp*,12:227–252,1997.

[11] P. Feautrier. Parametric Integer Programming. *Operations Research*, 22(3):243–268, 1988.

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

[13] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Interprocedural Parallelization Analysis in SUIF. *Trans. on Prog. Lang. and Sys.*, 27(4):662–731, 2005.

[14] J. Hoeflinger, Y. Paek, and K. Yi. Unified Interprocedural Parallelism Detection. *Int. Journal of Par. Prog*, 29(2):185–215, 2001.

[15] Y. Lin and D. Padua. Demand-Driven Interprocedural Array Property Analysis. In *Procs. Int. Lang. Comp. Par. Comp.*, 1999.

[16] Y. Lin and D. Padua. Analysis of Irregular Single-Indexed Arrays and its Applications in Compiler Optimizations. In *Procs. Int. Conf. on Compiler Construction*, pages 202–218, 2000.

[17] B. Lu and J. Mellor-Crummey. Compiler Optimization of Implicit Reductions for Distributed Memory Multiprocessors In *Int. Par. Proc. Symp.*, 1998

[18] S. Moon and M. W. Hall. Evaluation of Predicated Array Data-Flow Analysis for Automatic Parallelization. In *Proc. of Principles and Practice of Parallel Programming*, pages 84–95, 1999.

[19] C. E. Oancea and L. Rauchwerger. A Hybrid Approach to Proving Memory Reference Monotonicity. In *Procs. Int. Lang. Comp. Par. Comp.*, 2011.

[20] Y. Paek, J. Hoeflinger, and D. Padua. Efficient and Precise Array Access Analysis. *Trans. on Prog. Lang. and Sys.*, 24(1):65–109, 2002.

[21] L.N. Pouchet, et al. Loop Transformations: Convexity, Pruning and Optimization. In *Procs. of Princ. of Prog. Lang.*, 2011.

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

[23] W. Pugh and D. Wonnacott. Nonlinear Array Dependence Analysis. In *Proc. Lang. Comp. Run-Time Support Scal. Sys.*, 1995.

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

[25] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Trans. Par. Distrib. Sys*, 10(2):160–199, 1999.

[26] L. Rauchwerger, N. Amato, and D. Padua. A Scalable Method for Run Time Loop Parallelization. *Int. Journal of Par. Prog*,26:26–6,1995.

[27] L. Renganarayanan, et. al. Parameterized Tiled Loops for Free. In *Int. Conf. Prog. Lang. Design and Implementation.*, 2007.

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

[29] S. Rus, M. Pennings, and L. Rauchwerger. Sensitivity Analysis for Automatic Parallelization on Multi-Cores. In *Procs. Int. Conf. on Supercomp*, pages 263–273, 2007.

[30] X. Zhuang, et. al. Exploiting Parallelism with Dependence-Aware Scheduling. In *Int. Conf. Par. Arch. Compilation Tech.*, 2009.