Hybrid Dependence Analysis for Automatic Parallelization

Silvius Rus and Lawrence Rauchwerger
{rus,rwerger}@tamu.edu

Technical Report TR05-013
Parasol Lab
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112

November 21, 2005

Abstract

Automatic program parallelization has been an elusive goal for many years. It has recently become more important due to the widespread introduction of multi-cores in PCs. Automatic parallelization could not be achieved because classic compiler analysis was neither powerful enough and program behavior was found to be in many cases input dependent. Run-time thread level parallelization, introduced in 1995, was a welcome but somewhat different avenue for advancing parallelization coverage. In this paper we introduce a novel analysis, Hybrid Analysis (HA), which unifies static and dynamic memory reference techniques into a seamless compiler framework which extracts almost maximum available parallelism from scientific codes and generates minimum run-time overhead. In this paper we will present how we can extract maximum information from the quantities that could not be sufficiently analyzed through static compiler methods and generate sufficient conditions which, when evaluated dynamically can validate optimizations. A large number of experiments confirm the viability of our techniques, which have been implemented in the Polaris compiler.

1 Introduction

Compiler research has, for the most part, taken to two directions: Static analysis performed symbolically during compilation and Run-time analysis which validates transformations dynamically when variable values become available. Static analysis was always preferred because it does not cause execution overhead. Run-time analysis is a necessity because dynamic information is not available at compile time, is precise (and thus does not need to be conservative) but it generates overhead. Too much overhead reduces the profitability of optimizations; It can even lead to slowdown (e.g., speculative parallelization). These two approaches have so far progressed in parallel with little interaction.

In this paper we propose to unify the two approaches into the **Hybrid Analysis** (**HA**) framework. We modify both classic and dynamic analysis methods in order to integrate them seamlessly. This technology represents a departure from the classic analysis paradigm: Instead of only answering the question of whether an optimization is legal it also generates the dynamic conditions under which it could be legal. These conditions are frequently inexpensive to evaluate at run-time and thus can further increase the efficiency of run-time optimization to the point where they are almost always profitable. In this paper we present how we have designed new or modified classic analysis methods in order to transfer maximum information to the run-time evaluation phase and thus produce very good results. Without loss of generality, we have applied **HA** to the automatic detection of parallelism, probably the most important optimization, and obtained excellent results on a large number of scientific codes. The importance of making automatic parallelization viable has recently been brought to the forefront of research by the massive introduction of low cost parallel processors, all in need of software that can exploit them transparently.

1.1 Previous Approaches to Optimization

Most previous optimization methods can be divided into major categories: Static and dynamic. Static analysis of references to scalable data structures (e.g., arrays) for the purpose of thread level (iteration) parallelization have taken several directions: Proving sufficient conditions for independence about the behavior of array indexes (GCD, Banerjee, Itest [37]), solving (mostly) linear integer equations with constraints [9, 28], special cases of array data flow (privatization) and outright pattern matching. Another important technique which has been applied with limited success is pointer analysis and shape analysis. All such static techniques have advanced the state of the art without actually solving the parallelization problem. The reason behind their lack of full success is the limited applicability to real, complex programs which go well beyond linear, reasonably behaved memory reference patterns. Array references with non-linear subscripts, complex control flow guarding such references have proved to be complex for the most powerful compilers. Furthermore, when arrays are referenced through indirection (similar to pointers) then often the information necessary for analysis is not statically available because it depends on run-time values. In the more recent past dynamic techniques [33, 30] have analyzed various types of array references in loops during execution and accurately detected when such loops can be executed in parallel. Various flavors of such run-time testing and execution including speculative parallelization have been developed. The precision of these techniques is indeed absolute because all information needed for analysis is available during program execution; However their applicability has been limited by their inherent overhead. Analyzing access patterns only at run-time often results in work proportional to the dynamic reference count. While the most advanced run-time methods use the best parallel algorithms and have been optimized for scalable parallel execution, their overall overhead reduces the profitability of loop parallelization, sometimes to less than acceptable levels, even slowdowns (e.g., failed speculations). Vectorizing compilers like KAP had introduced simple run-time methods to decide when it is profitable to vectorize. e.g., a test on the length of the vector. (For a discussion of previous relevant work see Section 7).

We can therefore recognize that both static and dynamic methods have their pros and cons: no overhead but limited applicability or potentially high overhead but full applicability. These two approaches have been developed relatively independently with little cross-over interaction. The lack of significant information transfer between static and dynamic analysis has led to relatively poor results and thus to the practical failure of automatic parallelization. Furthermore, other potential optimizations of parallel programs have also suffered for the same reasons.

In [31] we have attempted to solve the automatic parallelization problem by bridging compile-time and run-time analysis. We had introduced an intermediate representation that could express all memory references in a program. The evaluation of the dependence sets was done at compile time with classical techniques and continued (with the same technique) at run-time yielding relatively high overhead.

1.2 Contribution

In this paper we have reworked most of our classical analysis algorithms to both perform static analysis as well as generate sufficient (sometimes also necessary) predicates for dynamic validation of optimization. In essence we have shifted our analysis paradigm in a novel, and based on results, very powerful way. We believe that this paper makes the several significant contributions:

- Hybrid Analysis represents a paradigm shift for optimizing (parallelizing) compilation. Results of the analysis tell not only if a transformation is safe but also what needs to be verified to make it safe. HA rewrites the classical compiler analysis techniques to produce more exploitable results even when definitive answers are not available.
- HA, through its novel approach bridges static and dynamic analysis in a seamless and possibly optimal manner which allows optimization when legal with minimum run-time overhead. It thus extends the profitability of dynamic methods – a necessity for full program parallelization.
- Introduces an extensible predicate extraction algorithm which results in a predicate "language" that can be directly mapped to executable.
- Fully implements the technique in a research compiler (Polaris) and shows through a large number of experiments that automatic parallelization is possible.

2 An Overview of Hybrid Analysis

The proposed **Hybrid Analysis** statically verifies memory reference properties, and when this is not possible, generates the conditions for which these properties can become true during program execution. These conditions (for parallelization) can then be evaluated dynamically with potentially little overhead and predicate the execution of the optimized (parallelized) blocks of code (loops). We have implemented **HA** in the context of automatic parallelizations of loops for Fortran programs and obtained excellent results. In Fig. 1 we sketch our general algorithm for analyzing memory references and extracting of conditions (if any) for parallel execution of loops.

```
Algorithm Automatic Parallelization
1. Call Memory Classification Analysis
      Aggregate References
         (across blocks, loops, subprograms)
      Classify references at each context
         (readonly, write first, readwrite)
2. ForEach loop
      DS = Dependence Set
         (All memory locations with loop carried
           dependences as aggregated reference set)
      DS = DS − memory related dependences
         (privatization, reduction, ...)
      Case (DS = ∅) Of
         True:   Generate Parallel Loop
         False:  Generate Sequential Loop
         Maybe:  (not sure at compile time)
            Extract condition P ≡ (DS = ∅)
            Generate Parallel Loop guarded by P
```

Figure 1: A compiler algorithm for automatic parallelization using Hybrid Memory Reference Analysis and Hybrid Data Dependence Analysis.

First individual array references are aggregated to their corresponding loop level and expressed with the help of a novel intermediate representation, the Uniform Sets of References (USR). This IR is capable of representing all memory references as sets at any level of the program. The USR is closed under set operations (see Section 3.2).

The collection of the aggregated memory references is followed by a memory classification. Using set operations references will be classified into Write First(WF), Read-Write(RW) and Read only (RO) sets (see Section 3.1). In the second phase important loops will be analyzed for the purpose of parallelization. This static analysis will check if the loop dependence sets (obtained from the WF, RW and RO sets) are empty. Some non-empty dependence sets can be removed by recognizing reductions, push-backs and other parallelizable patterns collected in a pattern library. The result of this analysis can yield three answers: Provable empty set implying the loop is parallel, provable non-empty set implying the loop is sequential and undecided. This latter case arises from either our inability to symbolically prove that the dependence set is empty or due to statically unavailable values. We will then extract from the undecided dependence sets a number of sufficient conditions which can guarantee correct parallel execution and place them in the order of their increasing dynamic cost (Section 4). These conditions range in complexity from scalar comparisons to the outcome of speculative parallelization (Section 5). They represent the sensitivity set of a program parallelization to its input values (and many times to dynamic values we could not evaluate symbolically).

Let us illustrate this algorithm with a very simple example. Consider the loop in Fig. 2. In order to generate code for its parallel execution, the compiler must disprove the existence of loop carried data dependences. It must prove that the set of *read* references [41:40+n] and the set of *write* references [1:n] are disjoint. A static parallelization decision cannot be taken regardless of the analysis method, because the validity of the decision depends on the input value n. In this case the compiler will need to defer the analysis of array A to a reference-based runtime test (LRPD) with a $O(n)$ run-time overhead

```
1 Read *, n                     D = [1 : n] ∩ [41 : 40 + n]
2 Do j = 1, n                   n= 5: D = ∅
3    A(j) = A(j+40)             n=45: D = [41:45] ≠ ∅
4 EndDo                         (D = ∅) ≡ (n ≤ 40)
```

Figure 2: Example of input-sensitive dependence. D is the set of memory locations with loop-carried dependences.

(albeit scalable). A more careful analysis (e.g., manual parallelization) will yield $n \leq 40$ as the only condition a compiler cannot prove statically because the value of n is input dependent. An inexpensive run-time test could verify only this condition before executing the parallel version of this loop instead of checking all references. However classic compiler dependence analysis does not generate conditions for which loops are independent. They only decide if a loop can be parallelized or not and, sometimes, return also exact dependences. The need for this type of analysis has been recognized in the past by compilers like KAP from KAI and other vectorizers. They used such run-time tests, but only for cases as simple as the one in Fig. 2.

The following section presents, for clarity, a summary of our previous work [31] on a new intermediate representation of all memory references and lends itself to hybrid analysis. We have changed its name from RT_LMAD (run-time lmad) to USR because it is used mainly as an intermediate representation subject to our predicate extraction analysis.

3 Memory Reference Analysis

Loop-level parallelization requires the analysis of all memory references that could cause cross iteration data dependences. It is more efficient at a higher level of granularity, where the fixed cost of thread synchronization becomes insignificant compared to the large amount of work between synchronization points. In order to detect large amounts of good quality parallelism we must be able to represent memory references (1) in analytical form (independent of the data set size), (2) across large program blocks, and (3) without relying on overly conservative approximations.

3.1 Aggregation and Classification

The analysis of memory references over large, interprocedural program contexts requires the use of *summary sets*, symbolic descriptions of sets of memory references (or locations) over arbitrarily large program blocks. Extensive work on the use of summary sets in program analysis and especially parallelization has been reported in [34, 5, 9, 7, 21, 11, 6, 28, 17, 15, 26].

Data dependence analysis requires precise information on the relative order of *write* memory references with respect to all other references. We define three types of summary sets to represent this information. The RO summary set records all memory locations only read (not written) within a section of code, the WF summary set records all memory locations that are written first and then possibly read and written, and the RW summary set records all other memory locations referenced from within a context. For a given variable and a given code block, the RO, WF, and RW sets form a partition, or *classification* of all references to the given variable, and store sufficient ordering information to solve data dependence problems.

The RO, WF, and RW summary sets can be computed in a bottom-up traversal of the whole program. Initially, they are collected from individual statements. When two successive statements may reference the same memory locations, the corresponding summary sets are updated to reflect their order. For instance, a RO with footprintS_1 followed by a WF with footprint S_2 results in: RO with footprint $S_1 - S_2$, RW with footprint $S_1 \cap S_2$ and WF with footprint $S_2 - S_1$.

It is thus important that a representation for summary sets be closed with respect to elementary set operations. Otherwise, even straight line code programs could not be handled by the analysis. In order to be control-flow sensitive, the representation must be closed with respect to predication. When a variable is passed by reference to subprograms, its generic summary set within the callee must be translated to the specific call site. The summary set representation must be closed with respect to symbolic cross-procedure translation. In order to represent sets of memory locations referenced within loops, the representation must be closed with respect to expansion across iteration spaces. A scalable analysis requires that the result of such an expansion to be analytical, i.e., independent of the size of the iteration space.

$$
\begin{aligned}
T = \{&\cap, \cup, -, (,), \#, \otimes^{\cup}, \otimes^{\cap}, \bowtie,\\
&LMADs, Gate, Recurrence, Call\ Site\}\\
N = \{&USR\},\ \ S = USR\\
P = \{&USR \rightarrow LMADs|(USR)\\
&USR \rightarrow USR \cap USR\\
&USR \rightarrow USR \cup USR\\
&USR \rightarrow USR - USR\\
&USR \rightarrow USR\#Gate\\
&USR \rightarrow \otimes^{\cup}_{Recurrence} USR\\
&USR \rightarrow \otimes^{\cap}_{Recurrence} USR\\
&USR \rightarrow USR \bowtie Call\ Site\}
\end{aligned}
$$

Figure 3: USR formal definition. $\cap$, $\cup$, $-$ are elementary set operations: intersection, union, difference. $USR\#Gate$ represents reference set USR predicated by condition $Gate$. $\otimes^{\cup}_{i=1,n} USR(i)$ represents the union of reference sets $USR(i)$ across the iteration space $i = 1 : n$. $USR(formals) \bowtie Call\ Site$ represents the image of the generic reference set $USR(formals)$ instantiated at a particular call site.

3.2 Memory Reference Summary Set Representation

The *Uniform Sets of References*) (**USR**) has been first introduced in [31] as RT_LMAD because in addition to being a intermediate representation it was also used to be directly evaluated at run-time for parallelization validation. It is a general, symbolic and analytical representation of memory reference sets in a program. It can represent the aggregation of scalar and array memory references at any hierarchical level (on the loop and subprogram call graph) in a program. It can represent the control flow (predicates), inter-procedural issues (call sites, array reshaping, type overlaps) and recurrences.

The building block of the USR is the Linear Memory Access Descriptor (LMAD) [15, 26], a symbolic representation of memory reference sets accessed through linear index functions. It may have multiple dimensions, and all its components may be symbolic expressions. Throughout this paper we will use the simpler interval notation for unit-stride single dimensional LMADs. For the loop in Fig. 2, the *read* pattern on array A is [41:40+N].

```
Algorithm Build Dependence Set
  Input:  $RW_j, WF_j, RW_j$ as USRs, Loop $DO\ j = 1, n$
  Output: Dependence Set $DS$ as USR
1. $DS = (WF_1^n \cap RO_1^n) \bigcup (WF_1^n \cap RW_1^n) \bigcup (RO_1^n \cap RW_1^n)$
   ($flow\ and\ anti\ dependences$)
2. $DS = DS \cup \otimes_{j=1,n}^{\cup}(WF_j \cap WF_1^{j-1})$
   ($output\ dependences$)
3. $DS = DS \cup \otimes_{j=1,n}^{\cup}(RW_j \cap RW_1^{j-1})$
   ($flow\ dependences$)
```

Figure 4: Construction of the set of memory locations with loop-carried dependences. S_m^n means $\otimes_{j=m,n}^{\cup} S_j$.

A USR can be viewed as an expression, or a program that takes as input values from the code under analysis and produces a set of memory references. The USR is stored as an abstract syntax tree with respect to the language presented in Fig. 3 and where operands are lists of LMADs. We could adopt a different primary representation by replacing the LMAD with a new data structure that preserves the same semantics (e.g., linear constraint sets).

The power of the USR consists in its inherent ability to tolerate static analysis failure by being closed with respect to all the operations required by the aggregation and classification process. Linear-based methods [8, 27, 14, 15, 11] often cannot represent memory references across large program contexts because they stop at the first point when the indexing scheme or the control predicates are not linear functions, or when operations such as set difference cannot be performed or would result in combinatorial explosion. Although the USR is faced with the same challenges, instead of stopping at the first point of failure, it saves the cause of failure in analytical form and allows aggregation to continue.

We have implemented a memory reference aggregation and classification algorithm in which we represent the RO, WF, and RW *summary sets* as USRs. This way, the information passed to data dependence analysis is precise, i.e., not conservatively approximated. When all references are linear, the result of the aggregation is a USR consisting of a single node, an LMAD. Otherwise, the USR contains partially aggregated descriptors and exposes the exact points of failure of the aggregation and classification process.

4 Hybrid Data Dependence Analysis (HDA)

We formulate loop data dependence problems by (1) expressing the set of all memory locations that carry cross-iteration dependences. i.e., the *Dependence Set* as a USR and (2) proving that this set is empty (*Dependence Set* $= \emptyset$). This condition is necessary because we consider in this paper only DOALL parallelization (no synchronizations).

Fig. 4 presents the construction of the *Dependence Set* as a USR based on the RO, WF, and RW summary sets across the loop body. Lines 1 and 3 find all the locations that are written and read in different iterations, respectively. Line 2 finds locations that are written in at least two different iterations. The dependence set collected at line 2 can be excluded through privatization. The dependence set collected at line 3 can be excluded in case the operation on the data under analysis is recognized as a reduction (and parallelized as such). In the example in Fig. 2, the dependence set is $[1 : n] \cap [41 : 40 + n]$.

When the dependence set can be proved at compile-time empty we generate a parallel loop and when it is proved not empty we generate a sequential loop. When it cannot be decided at compile-time whether the dependence set is empty or not, we generate a parallel version of the loop and guard it with an *independence condition*.

Hybrid Dependence Analysis represents the process of (1) extracting independence conditions from dependence equations at compile-time and (2) evaluating them efficiently at run-time. In [31], we expressed dependence sets using USRs, but did not present a method to extract independence conditions. We just evaluated the USR operations contained in *Dependence Set*, a rather inefficient method. We do not need to compute exact dependences but only a *True/False* assertion whether the loop is parallel or not. The actual dependences may be useful for other optimization but are currently beyond the scope of this work.

$$\begin{aligned}
&T = \{\wedge, \vee, \neg, (,), \otimes^{\wedge}, \otimes^{\vee}, \bowtie, LogicalExpression, Recurrence,\\
&\qquad Call\ Site, Library\ routine, Reference\ based\ test\}\\
&N = \{PDAG\},\ S = PDAG\\
&P = \{PDAG \rightarrow LogicalExpression | (PDAG)\\
&\qquad PDAG \rightarrow PDAG \wedge PDAG\\
&\qquad PDAG \rightarrow PDAG \vee PDAG\\
&\qquad PDAG \rightarrow \neg PDAG\\
&\qquad PDAG \rightarrow \otimes^{\wedge}_{Recurrence} PDAG\\
&\qquad PDAG \rightarrow \otimes^{\vee}_{Recurrence} PDAG\\
&\qquad PDAG \rightarrow PDAG \bowtie Call\ Site\\
&\qquad PDAG \rightarrow Library\ routine\\
&\qquad PDAG \rightarrow Reference\ based\ test
\end{aligned}$$

Figure 5: PDAG formal definition. $\wedge$, $\vee$, $\neg$ are the elemenary logical operators *and*, *or*, *not*. $\otimes^{\wedge}_{i=1,n} PDAG(i)$ holds true if and only if each of $PDAG(i)$ holds true, $i = 1, n$. $PDAG(formals) \bowtie Call\ Site$ represents the instantiation of a generic $PDAG$ at a particular call site. A specialized *library routine* may be employed to produce the value of the predicate. If a test based on simple comparisons and logical operations cannot be found, we fall back to a $reference\ based\ test$.

The grammar in Fig. 5 defines our language to express independence conditions as Predicate Directed Acyclic Graphs (PDAGs). PDAGs are very expressive but rely mostly on simple, logical operations and have a direct mapping to executable code.

In addition to classic $\wedge$, $\vee$, and $\neg$ operators, PDAGs can also express conjunction ($\otimes^{\wedge}$) and disjunction ($\otimes^{\vee}$) of predicates over iteration spaces. Library routines such as monotonicity checks may be called to address particular patterns, and reference based tests represent the fallback when cheaper conditions cannot be extracted. The HDA problem can now be expressed as the translation:

$$(Dependence\ Set = \emptyset) \mapsto P, \tag{1}$$

Dependence Set is a USR, P is a PDAG s.t. $P \equiv (Dependence\ Set = \emptyset)$.

4.1 Extracting Predicates from Dependence Equations

Our technique of extracting sufficient conditions from dependence equations (expressed using USRs) is a divide and conquer method that descends recursively the USR tree structure. It attempts to get answers from simpler dependence sets using known set boolean algebra relations. Fig. 6 presents algorithm *Solve* that extracts conditions based on the USR type (union, intersection etc.) of the dependence set.

```
Algorithm Solve
 Input:        $D = \emptyset$ (as USR)
 Output:       $P$ (as PDAG)
Case D of:
 $LMADs$:      P = $false$
 $A \cup B$:   P = $Solve(A = \emptyset) \wedge Solve(B = \emptyset)$
 $A \cap B$:   P = $Solve(A = \emptyset) \vee Solve(B = \emptyset) \vee SolveDisjoint(A, B)$
 $A - B$:      P = $Solve(A = \emptyset) \vee SolveIncluded(A \subseteq B)$
 $q\#A$:       P = $\overline{q} \vee Solve(A = \emptyset)$
 $\otimes^{\cup}_{i=1,n}(A_i)$:   P = $\otimes^{\wedge}_{i=1,n} Solve(A_i)$
 $A \bowtie Call\ Site$:  P = $Solve(A = \emptyset) \bowtie Call\ Site$
```

Figure 6: Extraction of an equivalent PDAG from an equation USR = $\emptyset$. The extraction process choses applies simple rules as directed by USR syntax. The rules are based on elementary set algebra identities, e.g., $(A \cup B = \emptyset) \equiv (A = \emptyset \wedge B = \emptyset)$.

Each USR production rule is associated with semantics particular to the set operation it represents. For instance, in order to prove a union empty, it is necessary and sufficient to prove both its terms empty. Finally, the recursion will either stop at the leaves, i.e., dependence set is not empty and thus translate into *false*, or, call a subalgorithm (presented later). Our implementation is biased toward extracting conditions sufficient for independence (optimistic). A similar approach can be used to extract conditions sufficient for dependence (pessimistic). In both cases, the partial (optimistic or pessimistic) conditions are complemented by a fallback reference-based test, which will return an exact answer for each instantiation.

Low-cost, precise conditions can be extracted from all USR operations except for intersection and set difference. An intersection could be empty even if none of its terms are (e.g. a set of odd numbers vs. a set of even ones). Algorithms *SolveDisjoint* (Fig. 7) and *SolveIncluded* (Fig. 8) show which cases can and which cannot be handled in a generic way. They rely on dividing more complex equations (such as $A \cap (B \cup C)$) into simpler equations (such as $A \cap B = \emptyset$ and $A \cap C = \emptyset$), based on elementary set identities.

In the example in Fig. 10, array W could be proved privatizable by showing that the *read* at line 5 is covered by the *write* at line 16. However, the shape of the USR that describes the *write* pattern is outside any of the cases in the *Solve* algorithms presented above. Over the following section we will show that even when two USRs cannot be compared directly, a meaningful PDAG can often be extracted from comparisons between predicated approximations of the USRs.

4.2 Extracting PDAGs from USR Approximations

Several memory reference analysis techniques have proposed the use of approximations of reference sets in the presence of subscript arrays or arrays of conditionals [6, 15, 31]. These techniques generally approximate a memory reference set P that does not fit a particular model with a pair $(\lfloor P \rfloor, \lceil P \rceil)$ such that (1) $\lfloor P \rfloor \subseteq P \subseteq \lceil P \rceil$ and (2) $\lfloor P \rfloor$ and $\lceil P \rceil$ fit their model. We apply this to our framework by approximating complex USRs with LMADs.

Returning to the example in Fig. 10, when trying to prove array W privatizable we cannot compare the USRs of the *read* and *write* descriptors directly. Instead, we compute $\lceil read \rceil$ and $\lfloor write \rfloor$ as LMADs

```
                        Algorithm SolveDisjoint
                          Input:        $A, D$ (as USRs)
                          Output:       $P$ (as PDAG)

Pattern                 / P = $SolvePatternMatch(A \cap D = \emptyset)$
Matching                \ If (Solved) Then Return

                        / Case D of:
                        |   $B \cap C$:             P = $Solve(A \cap B = \emptyset) \wedge Solve(A \cap C = \emptyset)$
                        |   $q\#B$:                 P = $\overline{q} \vee Solve(A \cap B = \emptyset)$
Syntax-directed         |   $\otimes^{\cup}_{i=1,n}(B_i)$:  P = $\otimes^{\wedge}_{i=1,n} Solve(A \cap B_i)$
Rules                   | Case A of:
                        |   $B \cap C$:             P = $Solve(B \cap D = \emptyset) \wedge Solve(C \cap D = \emptyset)$
                        |   $q\#B$:                 P = $\overline{q} \vee Solve(B \cap D = \emptyset)$
                        \   $\otimes^{\cup}_{i=1,n}(B_i)$:  P = $\otimes^{\wedge}_{i=1,n} Solve(B_i \cap D)$

                        / If (Not Solved) Then
Approximative           |   $(p_A, \lceil A \rceil)$ = a conditional LMAD overestimate of A
Estimates               |   $(p_D, \lceil D \rceil)$ = a conditional LMAD overestimate of D
                        \   P = P $\vee (p_A \wedge p_D \wedge SolveDisjointLMADs(\lceil A \rceil, \lceil D \rceil))$

Reference-based         / If (Not Solved) Then
Tests                   \   P = P $\vee ReferenceBasedTest(A \cap D = \emptyset)$
```

Figure 7: Transformation of a disjointness test into a PDAG. When a solver (pattern matching, syntax-based rules, approximative estimates and reference-based-tests) produces a PDAG that is equivalent to the original problem, we consider it *solved.* When the PDAG is not equivalent to the original problem, we consider it partially solved and continue to the next solver. Reference-based tests are guaranteed to be accurate in all cases, thus the algorithm always produces PDAGs equivalent to the original problem.

and record the assumptions made during the approximation process. The problem reduces to proving $\lceil read \rceil \subseteq \lfloor write \rfloor$. Since $read \subseteq \lceil read \rceil$ and $\lfloor write \rfloor \subseteq write$, this condition is sufficient to prove that $read \subseteq write$. In our example, $\lfloor read \rfloor = [1 : l(j)]$, and $\lceil write \rceil = [1 : l(j)]$, when $\otimes^{\wedge}_{i=1,l(j)} y(i,j).GT.0$. The approximation process is invoked by algorithms *SolveDisjoint* and *SolveIncluded.*

4.3 Predicate Extraction from Finite Valued USRs

USRs are symbolic sets that depend on the value of program variables. When a variable may take a known, limited number of values (possibly symbolic) we can partially evaluate the USR for all these possible values. Then the USR can be represented as a union of all its specialized versions, each guarded by its assumption. Consider USR $d = \{f(MOD(i,2))\}$. Then

$$d = (MOD(i,2).EQ.0)\#\{f(0)\} \cup (MOD(i,2).EQ.1)\#\{f(1)\}$$

based on the fact that intrinsic $MOD(*,2)$ may take only one of two values, 0 or 1. Assuming that $f(j) = j/2$, the USR reduces to $\{0\}$ at compile-time. Similarly, a set difference $\{j\} - \{k\}$ can be expressed as $(j.NE.k)\#\{j\}$, which does not solve the problem at compile-time, but *leaves less to be done at run-time.*

```
                      Algorithm SolveIncluded
                        Input:        $A \subseteq D$ as (USRs)
                        Output:       $P$ (as PDAG)

Pattern               / P = $SolvePatternMatch(A \subseteq D)$
Matching              \ If (Solved) Then Return

                      / Case D of:
                      |   $B \cup C$:      P = $Solve(A - B = \emptyset) \vee Solve(A - C = \emptyset)$
                      |   $B \cap C$:      P = $Solve(A - B = \emptyset) \wedge Solve(A - C = \emptyset)$
                      |   $B - C$:         P = $Solve(A - B = \emptyset) \wedge Solve(A \cap C = \emptyset)$
Syntax-directed       |   $q\#B$:          P = $q \wedge Solve(A - B = \emptyset)$
Rules                 | Case A of:
                      |   $B \cup C$:      P = $Solve(B - D = \emptyset) \wedge Solve(C - D = \emptyset)$
                      |   $B \cap C$:      P = $Solve(B - D = \emptyset) \vee Solve(C - D = \emptyset)$
                      |   $B - C$:         P = $Solve(B - D = \emptyset)$
                      \   $q\#B$:          P = $\overline{q} \vee Solve(B - D = \emptyset)$

                      / If (Not Solved) Then
Approximative         |   $(p_A, \lceil A \rceil)$ = a conditional LMAD overestimate of A
Estimates             |   $(p_D, \lfloor D \rfloor)$ = a conditional LMAD underestimate of D
                      \   P = P $\vee(p_A \wedge p_D \wedge SolveIncludedLMADs(\lceil A \rceil, \lfloor D \rfloor))$

Reference-based       / If (Not Solved) Then
Tests                 \   P = P $\vee ReferenceBasedTest(A - D = \emptyset)$
```

Figure 8: Transformation of an inclusion test into a PDAG.

When the number of values taken by an input sensitivity variable is not known at compile-time, we can still enumerate a small set of important cases followed by a fallback solution. We evaluate the USR instances (similar to abstract interpretation) across the first few iterations of loops, which may lead to much simpler conditions for the case when the iteration counts are small.

An extreme case is when complex subscripts are created within the program, resulting in nontrivial USRs with empty input sensitivity sets. In such a case we generate executable code from PDAGs and run it at compile-time.

In summary we are blurring the line between what is done statically and dynamically.

4.4 Extracting PDAGs from LMAD Equations

Algorithms in Fig. 7 and Fig. 8 usually stop by calling subroutines *SolveDisjointLMADs* and *SolveIncludedLMADs*. Although in general predicate extraction from dependence sets consisting of set intersections and differences is a hard problem even for linear memory reference descriptors like the LMAD [15, 27], most practical cases are tractable. We have modified the multi-dimensional LMAD intersection and subtraction algorithms presented in [15] to return sufficient conditions under which their result is empty. For instance, in order to prove two 1-dimensional LMADs disjoint, it is necessary and sufficient to perform interval bound checks and a GCD test.

4.5 Extracting PDAGs from Known Reference Patterns

4.5.1 Monotonicity and Sorting

Consider the dependence problem on array A in the example in Fig 10. A direct application of the *SolveDisjoint* algorithm would result in a test of $n * (n-1)/2$ bound checks, one for each pair ([p(j)+1:p(j)+l(j)], [p(k)+1:p(k)+l(k)]), where $j = \overline{1:n}$ and $k = \overline{1, j-1}$. However, a less expensive solution exists for this case: We can verify, dynamically, in $O(n \log(n))$ time, that the sequence $\lceil D_i \rceil = [lower_i : upper_i]$ is non-overlapping by sorting the pairs $lower_i : upper_i$ (based on $lower_i$) and verifying that $upper_i < lower_{i+1}$. A quicker ($O(n)$) and sufficient but not necessary version of the test verifies whether the intervals already form a monotonic sequence.

It is important to note that n may be much smaller than the actual number of dynamic memory references since it represents the number of partially aggregated intervals, rather than individual references. We extended the applicability of this test to multi-dimensional LMADs by defining order in multi-dimensional integer spaces.

4.5.2 Reference Pattern Library: Extensible Compiler

Recognizing and taking advantage of particular patterns such as sortable intervals will always provide better solutions to some classes of problems, since the programmer's level of abstraction is often above the programming language semantics. Pattern recognition presents two main challenges. First, a pattern must be general enough so that two semantically equivalent patterns will be recognized as such even when they are textually different. Second, the pattern recognition must be quick in order to be applicable (pattern recognition in programs is often associated to subgraph isomorphism).

We have created an offline XML database of memory reference patterns as USRs. These patterns can be analyzed by programmers and can be associated specialized library routines for a particular analysis, such as sorting-based checks for dependence analysis. A dependence test can then be broken up into parts for which solutions are known, and an overall solution can be composed using the general algorithm *Solve*.

By storing patterns as USRs, the possibility of finding a match in the library is greatly enhanced. Aggregation and normalization bring textually different memory reference patterns to a comparable form. We have identified several similar patterns across different subroutines within the same program and even between completely different programs. *Semantic* reference pattern matching reduces (partially) to *syntactic* pattern matching on the USR grammar, which can be implemented efficiently.

4.6 Fallback: Reference-based Dependence Tests

Extraction of equivalent simple conditions from dependence equations is not always possible, for instance in the example in Fig. 9. We have two generally applicable solutions that can solve arbitrarily complex dependence equations. In case the aggregation process was partially successful, we can embed the USRs in the generated code and evaluate them at run-time [31]. The run-time dependence test will consist of checking whether the result is empty. When (partial) aggregation and predicate extraction is not possible we fall back to the LRPD test [30], which has a complexity proportional to the dynamic memory reference count, but scales well with the number of processors.

```
Read *, (p(j), j=1,1000),
        (q(j), j=1,1000)
Do j = 1, 1000
    A(p(j)) = A(q(j))
EndDo
```

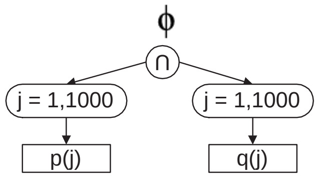


Figure 9: A Hybrid Analysis extreme: in general, no test can solve this problem faster than the reference-by-reference LRPD test.

```
 1 Read *, n, x,
    (p(j), l(j), (y(i,j),
     i=1,l(j)), j=1,n)
 2 Do j = 1, n
 3    Call geteu(W, j)
 4    Do i = 1, l(j)
 5       A(p(j)+i) = W(i)+
            + 1/A(p(j)+i)
 6    EndDo
 7 EndDo
...
...
```

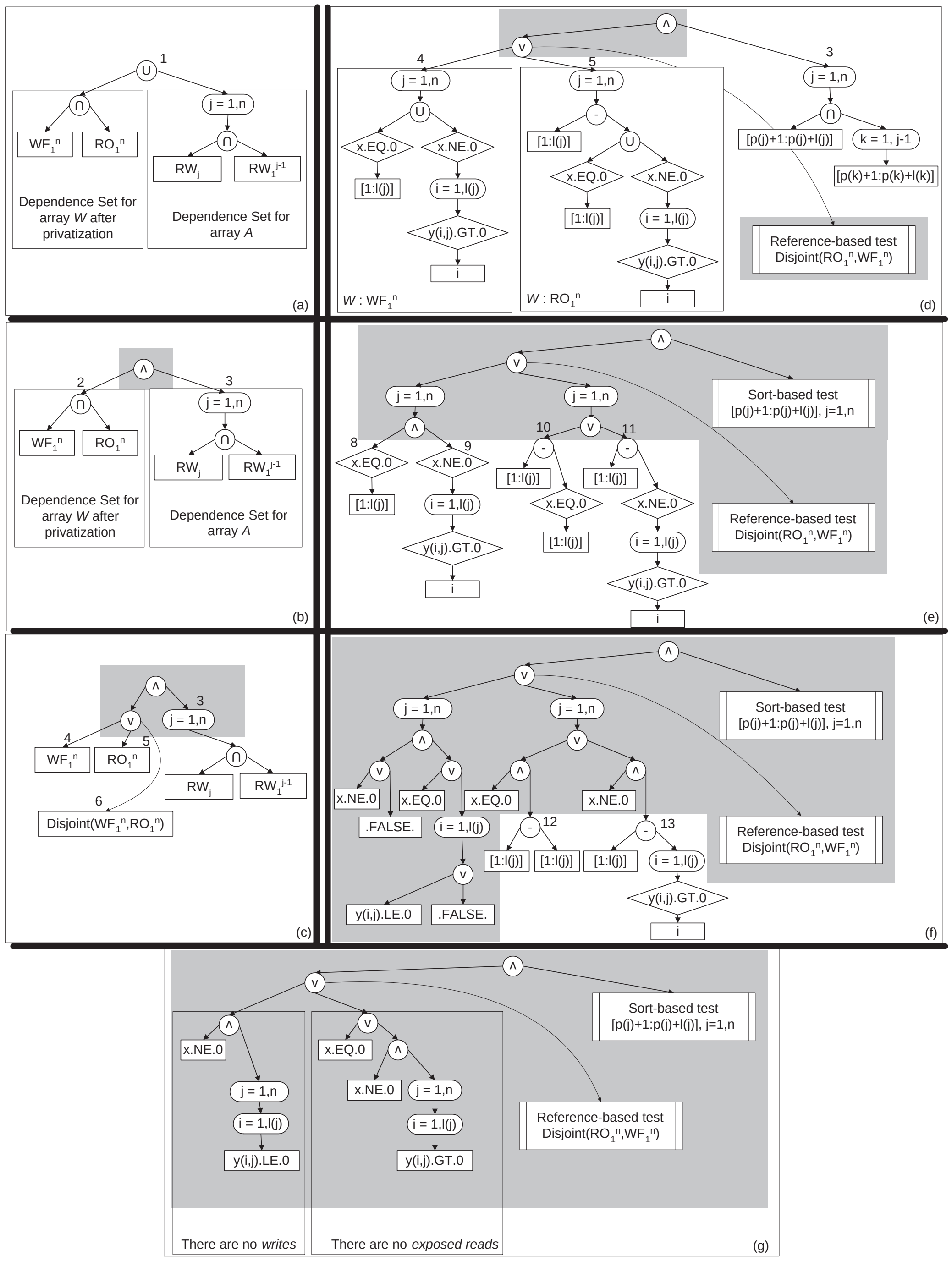
```
 8 Subroutine geteu(W, j)
 9 If (x.EQ.0)
10   Do i = 1, l(j)
11     W(i) = ...
12   EndDo
13 Else
14   Do i = 1, l(j)
15      If (y(i,j).GT.0)
16        W(i) = ...
17      EndIf
18   EndDo
19 EndIf
```

Figure 10: Example extracted from DYFESM, loop SOLVH_do20. The loop at line 2 can be executed in parallel if and only if there are no cross-iteration dependences on arrays W and R.

4.7 A Hybrid Dependence Analysis Example

Fig. 11 presents the HDA process for the loop at line 2 in Fig. 10. Block (a) shows the dependence set resulting from the invocation of algorithm *Build Dependence Set* for variables W and A (empty descriptors such as RW for W are not shown). (b) Problem 1 is divided into subproblems 2 and 3 by applying algorithm *Solve*. (c) Problem 3 is divided into subproblems 4 and 5 by applying algorithm *SolveDisjoint*. Because problems 4 and 5 are sufficient but not equivalent to problem 3, algorithm *SolveDisjoint* will add the fallback solution, a reference-based test on array W. (d) Problems 3, 4 and 5 are detailed by showing the exact shape of their USRs. (e) Algorithm *Solve* transforms problem 4 into the conjunction of problems 8 and 9 over the iteration space of the loop. Algorithms *Solve* and *SolveIncluded* transform problem 5 into a disjunction of problems 10 and 11. Problem 3 is recognized as a pattern by algorithm *SolveDisjoint* and is assigned a library routine solution. (f) Problems 8 and 9 are transformed in simple equivalent conditions by applying algorithm *Solve* recursively. Problem 10 is transformed in condition ($x.EQ.0$) and problem 12. Problem 12 will then reduce to *true* at compile-time after it undergoes the approximation phase in algorithm *SolveIncluded* and is applied algorithm *SolveIncludedLMADs*. Similarly, problem 11 reduces to problem 13, which is also solved by the approximation phase in *SolveIncluded* followed by *SolveIncludedLMADs*. (g) The final result is shown after symbolic simplification and hoisting of loop invariants.

It is important to note that a fully automated analysis technique of USRs and PDAGs produced run-time tests that have a clear meaning to a programmer. The independence conditions on W shown



Dependence Set for array W after privatization
Dependence Set for array A
W : WF1n
W : RO1n
Reference-based test Disjoint(RO1n,WF1n)
Sort-based test [p(j)+1:p(j)+l(j)], j=1,n
Disjoint(WF1n,RO1n)
There are no writes
There are no exposed reads
(a)
(b)
(c)
(d)
(e)
(f)
(g)

Figure 11: PDAG extraction from the parallelization problem for the loop at line 2 in Fig. 10 (partial PDAGs are shaded). The numeric labels represent dependence equations $DS = \emptyset$, where DS is the corresponding node. For instance, equation 2 in block (b) has as DS the dependence set for variable W.

in the boxes in block (g) can be identified as (1) the absolute lack of write references and (2) the absolute lack of exposed reads. The second one is the actual behavior of the application and a common pattern observed in several applications. This makes us believe that HA operates at the right level of abstraction and thus manages to extract high semantics such as data dependence from low-level program representation, similar to the way a programmer would do.

5 Implementation and Complexity

We have implemented Hybrid Dependence Analysis for automatic parallelization in the Polaris [3] research compiler.

To streamline our work we have first restructured our F77 benchmarks to structured form though a chain of relatively simple filters. For example, we have transformed *DATA* statements into assignments, unstructured loops into *DO* and *WHILE* loops, etc.

Additional passes recognize induction variables and propagate constants inter-procedurally. We use our techniques presented in [32] to remove dependences caused by conditional linear recurrences, and pushback patterns. We then use our hybrid dataflow framework to implement hybrid (static and dynamic), selective (array subregion) and conditional run-time reduction parallelization and privatization, including copy-in and last value computation. USR operations are simplified mainly by means of symbolic range comparisons and set algebra identities. Ranges are extracted from predicates, loop bounds, array dimension declarations, standard nonaliasing assumptions, and computed using abstract interpretation methods [4] and Value Evolution Graphs [32]. Other simplification techniques include set identities and are based on comparisons of conservative estimates expressed as LMADs. *Predication* is optimized based on boolean algebra identities. The USR data structures are reference counted, which limits the number of USR nodes per program analysis to at most a constant factor of the number of static data references [31].

The extraction of PDAGs from USR dependence equations consists of several processes: (1) the translation based on USR syntax presented in the *Solve* algorithms, (2) tests based on USR approximation, (3) limited abstract interpretation of control flow and finite values, (4) pattern matching, and (5) generation of reference-based tests.

Complexity. The complexity of the syntax-directed translation could be exponential in the worst case, due to productions such as: $A \cap (B \cup C) = \emptyset \mapsto (A \cap B = \emptyset) \wedge (A \cap C = \emptyset)$. However, this tendency is avoided through aggressive memoization of solutions to common subproblems. The extraction of approximative tests and the pattern matching algorithms have complexities at most linear with the size of the given USR. The effort of abstract interpretation is currently limited to enumerations of at most 10 particular input sensitivity sets.

Table 1(a) presents compilation statistics. The number of USR and PDAG nodes is relatively small. On the average, a USR node occupies about 3 KB, while a PDAG node occupies about 24 bytes. There is no precise correlation with the number of lines of code because each application undergoes several transformations such as forward substitution that may increase the static number of memory references. In some cases the compilation times are long because of failed attempts to simplify USRs, which may result in up to quadratic complexity [31].

Code	**Lines**	**Time**	**USRs**	**PDAGs**
ADM	5,791	455	35,249	10,456
ARC2D	3,099	102	13,178	22
BDNA	4,919	36	11,181	156
DYFESM	3,903	38	6,841	756
FLO52	2,508	120	8,371	0
MDG	1,237	15	8,085	744
OCEAN	2,738	122	14,664	208
SPEC77	4,582	303	75,032	4,733
TRACK	2,523	245	27,790	2,931
TRFD	656	120	1,684	139
APPLU	3,980	56	13,212	34
APSI	7,488	399	36,593	10,800
MGRID	489	108	2,089	0
SWIM	435	7	1,785	0
WUPWISE	2,184	45	4,710	60
HYDRO2D	4,461	33	5,911	11
MATRIX300	439	3	1,458	0
MDLJDP2	4,172	18	6,928	444
NASA7	1,204	48	8,545	547
ORA	373	7	2,562	0
SWM256	487	8	1,520	0
TOMCATV	194	5	1,056	32

Table 1: Compile-time analysis statistics (seconds). Column 4 and 5 show the total number of USR and PDAG nodes created (operator or leaves).

Code	**Op Ratio**
ADM	$1.8 * 10^5$
ARC2D	$1.2 * 10^7$
BDNA	–
DYFESM	$1.5 * 10^4$
FLO52	–
MDG	$6.7 * 10^0$
OCEAN	$2.1 * 10^4$
SPEC77	$1.0 * 10^0$
TRACK	$1.0 * 10^0$
TRFD	$5.6 * 10^4$
APPLU	–
APSI	$1.6 * 10^7$
MGRID	–
SWIM	–
WUPWISE	–
HYDRO2D	–
MATRIX300	–
MDLJDP2	–
NASA7	$3.0 * 10^6$
ORA	–
SWM256	–
TOMCATV	–

Table 2: Run-time test overhead reduction through HDA: ratio between the number of memory references and the number of PDAG operations.

```
Independent = .F.
If (x.EQ.0)             Then Independent = .T.
ElseIf (y(:,:).GT.0)    Then Independent = .T.
ElseIf (y(:,:).LE.0)    Then Independent = .T.
Else Independent = ReferenceTest(W)
EndIf
Independent = Independent .AND. SortTest(p,l,n)
```

Figure 12: Executable code generated from the PDAG in Fig. 11(g).

5.1 Generation of Executable Code from PDAGs

PDAGs contain four types of run-time operations: (1) evaluation of elementary conditional expressions, (2) sorted checks, (3) partially aggregated USR run-time evaluation and (4) reference-by-reference LRPD. We extract, for each dependence equation, a **cascade of tests** (Fig. 12). ordered by their estimated complexity. They range from $O(1)$ tests as the one in Fig. 2 to $O(n)$ dynamic reference instrumentation as is the case in Fig. 9.

We apply loop invariant hoisting to USRs and PDAGs by performing aggressive invariance analysis on their input sensitivity sets. *We can identify invariant reference patterns in the presence of subscripted subscripts even when subregions of the subscript array are not invariant.* This accuracy is achieved by representing the input sensitivity sets of a USR as USR themselves, and thus identifying the exact subregion of the subscript array that affects the shape or size of the memory pattern. We transform the invariance problem into proving that the input sensitivity set is disjoint from the set of variant subscripts, which is essentially a dependence problem on the subscript array.

For instance, for the final test in Fig. 11(g), we hoist the $O(1)$ check, $x.EQ.0$ and insert it first in the cascade of independence tests for array W (Fig. 12). In the actual application (DYFESM/SOLVH_do20), the $O(1)$ tests succeeds so the other more expensive tests are avoided.

In general, sorted-based tests are faster then reference-based tests. Evaluating USRs at run-time consists of fewer (but more complex) operations, then reference-by-reference tests. In some cases they may either degenerate into inefficient enumerations or take conservative decisions that can lead to false negatives. The LRPD has overhead proportional to the dynamic reference count, but is optimal for cases where aggregation and equation inversion are not possible (Fig. 9), and is always applicable, precise, and has a more predictable complexity.

All the tests can be run in either inspector/executor mode, or during speculative parallel executions of the code. In both cases, we reuse the test results by means of inspector hoisting, PDAG and USR common subexpression recognition, and run-time test result memoization. The choice between inspector/executor and speculative execution requires a complex cost model. Presently, we choose speculation over inspector/executor only if (1) a parallel inspector cannot be extracted or (2) if we cannot extract a light inspector (a slice made of only scalar definitions). The actual code generation consists of a syntax-based translation from the PDAG grammar to Fortran.

6 Experimental Results

The experimental evaluation presented in the following sections will show that Hybrid Dependence Analysis (a) extracts a very high degree of parallelism and, often, all the available parallelism from a large number of applications, (b) it is applicable to a large number of applications (c) allows the generation of minimal run-time tests (d) contributes significantly to the overall parallelization of programs, i.e., they are instrumental in obtaining the presented results.

We have focused on the detection of parallelism rather than on optimizing parallel code execution (e.g. locality enhancement, load balancing). We believe that the major challenge in front us is to detect loops parallel, a step which preconditiones any further optimizations.

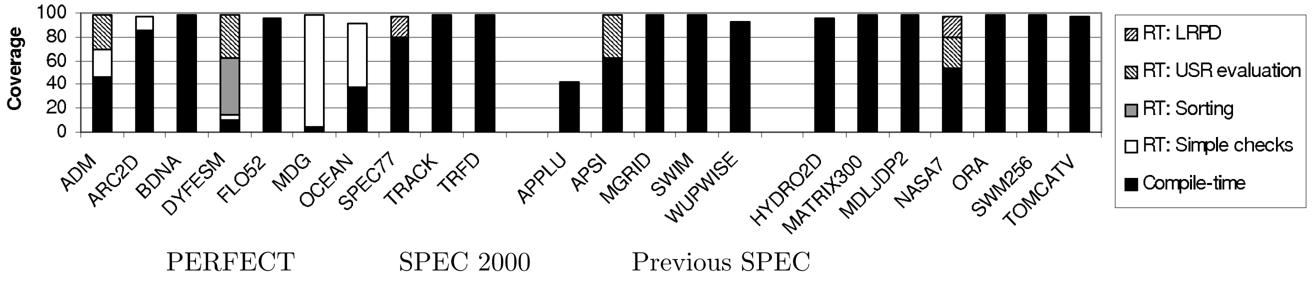


Figure 13: Automatic parallelization coverage as percentage of the sequential execution time.

6.1 Experiment Setup

We applied HA and then parallelized a set of 22 benchmark programs from the PERFECT and SPEC benchmark suites. The parallel code generation is done automatically using OpenMP directives without any further optimizations. The selection of the loops for which parallel code and (possibly) dynamic tests were generated based on profiling their sequential execution time. The automatic selection of parallelization candidates based on some more sophisticated performance model is beyond the scope of this paper.

The experiments were run on a parallel processor SGI Altix 3700 machine in non-dedicated mode. The reference times for all runs are those of the original benchmark codes running sequentially with the largest input set available.to us. All codes were compiled with the Intel Fortran Compiler using compilation option *-openmp*. While the SPEC codes were compiled with the *-O2* optimization flag, some of the older PERFECT codes were compiled with *-O0* (both the sequential and the parallel versions). The reason for this choice was to increase in the most uniform manner the execution times of these codes.These benchmarks and their (initially) reduced input sets have, on todays machines, a very short sequential execution time. Their parallelization, while correct, brings the time for some loops down to the execution time of barriers, making it impossible to measure the effect of parallelization. We strongly believe that their structural characteristic is still quite relevant and that expanding their execution time by reducing sequential optimizations is a reasonable experiment for measuring parallelism. In fact they are harder to parallelize than newer benchmarks with larger input sets. For Perfect codes MDG and TRACK we have larger input sets and thus they have been compiled with *-O2*.

Fig. 13 shows the coverage of parallelization achieved by **HA**. For 21 out of 22 applications the coverage is over 90% and many are at the 99% level. The exception, *APPLU*, contains a large section with loop-carried flow dependences. The excellent coverage does not sufficiently do justice to the power of **HA** because it does not quantify the fact that we can detect course grain parallelism (outer loop level) as well as fine grain level (inner loops).The exception was TOMCATV, where the outer loop was found sequential and thus only inner loops were parallelized. In the near future we plan to run our experiments on a machine that supports well nested parallelism in order to better present the quality of the parallelization we obtain.

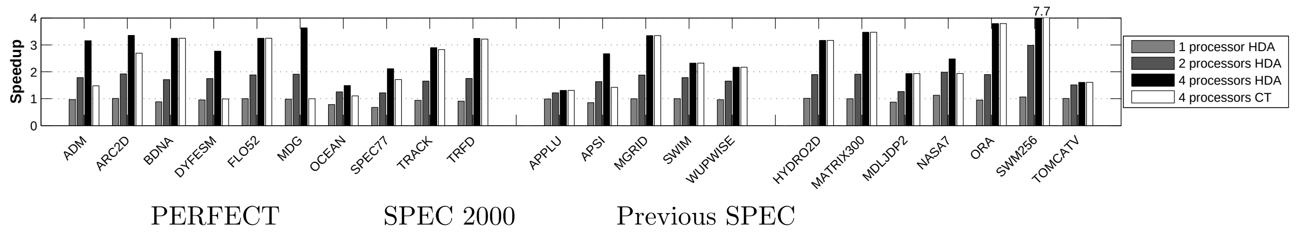


Figure 14: Speedups relative to the sequential version of automatically parallelized benchmark applications on 1, 2, and 4 processors. CT = speedups obtained using only compile-time methods.

6.2 Run-time Test Applicability and Evaluation

Overall, HDA generated 42 tests based on evaluation of elementary conditional expressions, 30 sorted-based tests and 81 based on USR run-time evaluations. The parallelization of only 4 loops required the application of the reference-by-reference LRPD test. Fig. 13 shows the coverage (and thus importance) of the PDAG technique (evaluation of simple comparisons, sorting-based checks, USR evaluation and reference-by reference LRPD) in parallelizing the codes. Table 2 presents the reduction in dynamic operations achieved by HDA relative to reference-by-reference (LRPD) tests as being at least four orders of magnitude in 7 applications. The overhead of run-time tests for all the applications that could not be parallelized statically proves to be negligible (less than 0.1%) in most cases. In *ADM*, the overhead of 4.67% is due to the run-time evaluation of complex USRs. However, because this run-time test can be reused (outer loop invariant) its overhead decreases to 0.1% in the *APSI* version (much larger input set). The total run-time overhead in *MDG* is 2.8% and is partially due to checkpointing for speculative parallelization.

6.3 Speedup Measurements

Fig. 14 presents full application speedups, measured by dividing the sequential execution time of the whole application to its parallel execution time including the runtime overhead. Automatic parallelization based on HDA resulted in speedups of at least 3 on 4 processors for 11 out of 22 applications and of at least 2 on 4 processors on 18 out of 22 applications. The static part of HDA is powerful in itself and manages to parallelize more loops that previous static analysis methods in Polaris. Its strength lies primarily in its ability to analyze large interprocedural contexts such as GLOOP_do1000 (over 1,000 lines of code), which could not be previously parallelized by Polaris. More importantly, the speedup improvement through the dynamic component of HDA is significant in 8 out the 22 applications.

OCEAN and *NASA7* (partially) suffer from lack of memory locality in their time consuming FFT loop nests. *APPLU* has outer loop flow dependences and cannot be parallelized using the *DOALL* model. Several loops in *TOMCATV* could not be parallelized at the outermost level resulting in low granularity and limited speedup despite large parallelization coverage.

We believe that the consistent solid performance results across a large number of standard benchmark applications proves our claims on the effectiveness of HDA. Comprehensive analysis reports, performance tables and graphs can be found at `http://parasol.tamu.edu/compilers/ha`.

7 Related Work

Data Dependence Analysis. Most of the previous data dependence work was based on the representation of memory reference sets using linear constraints. Dependence questions are reduced to proving that a system of linear constraints had no integer-valued solution [8, 36, 2, 10, 22, 18, 27, 26]. In all these systems, the number of variables must be known at compile-time and the symbolic expressions must be linear (some particular extensions can handle certain classes of nonlinearity). They cannot generally be used to analyze (1) memory references through index arrays and (2) memory references controlled by arrays of conditionals.

Pattern recognition and index property analysis were proposed as solutions for nonlinear reference patterns [20]. Its applicability is limited to the cases studied. Storing patterns as a USR abstraction (Section 4.5.2) may allow more cases to be matched.

Symbolic value range [4] and monotonicity analysis [13, 12, 20, 38, 35, 32] also targeted some nonlinear reference patterns. They are generally not well integrated with other techniques and thus lack generality. For instance, the *Range Test* compares the value ranges of two reference sets, but does not deal with strided patterns. We use value ranges and monotonicity information [32] in elementary USR operations, such as intersection or set difference: comparison of offsets, but also strides and spans, predicate implication and redundancy proofs.

Run-time data dependence tests were proposed to solve dependence problems that did not have compile-time solutions [33, 16, 19, 30, 29]. Their overhead may some times void the optimization benefits they bring.

Hybrid Dependence Analysis and Parallelization. One of the first forms of hybrid analysis was conditional vectorization [37]. It is an effective technique, but limited in scope to vectorized loops. [1] presents a powerful interprocedural partial redundancy elimination analysis and its application to the detection of array data flow relations on particular control flow paths, which in turn leads to aggressively optimized placement of communication primitives. This is similar to the aggregation and classification phase of our previous approach [31]. HDA performs more aggressive static analysis and produces lighter run-time conditions by partially solving statically general dependence equations involving complex control-flow, indirect addressing and recurrences.

[24, 25, 23] relies on *predicate extraction* to synthesize simple conditions from data dependence and data flow equations on arrays. Their applicability is limited to checks on scalars such as loop bounds, so they cannot extract predicates for general reference patterns through indirection arrays or arrays of conditionals. In such cases they choose to take conservative decisions. Similarly, [15] presents the extraction of safety guards for optimistic LMAD operations. This approach has essentially the same power and applicability as predicate extraction.

[39] focuses on reducing the overhead of reference-by-reference run-time tests by grouping together reference sets that have the same dependence patterns. Only one representative test is performed, resulting in lower overhead. However, only accesses that have identical control and very similar indexing (e.g., differ by constant offset) are recognized as similar. The USRs are more expressive and can be used to compare more complex patterns.

8 Conclusions and Future Work

The **Hybrid Analysis** (**HA**) framework represents a paradigm shift in the way static analysis is done. It does not only validate transformations but also generates sufficient, simple conditions which can validate optimizations at runtime. It has required the modification of many standard compiler analysis methods as well as a novel predicate extraction algorithm. To give a few examples, memory classification techniques have been transformed in *hybrid* classification algorithms, data dependence analysis has become *hybrid* dependendence analysis. We have also introduced a more systematic way to use reference patterns to discover parallelism. We have implemented our new technology in the Polaris compiler and shown the most important aspect: It works. We could automatically detect almost all the parallelism in 23 codes and then, without further optimization get very good speedups. This result also shows that automatic parallelization, long in coming, is actually possible.

It is true that our results have been validated only on F77 programs. But Hybrid Analysis can be applied to programs in any language. It is not language dependent because it represents a paradigm of analysis and not a specific technique. We therefore hope that HA can be used in other compilation systems.

References

[1] G. Agrawal, J. H. Saltz, and R. Das. Interprocedural partial redundancy elimination and its application to distributed memory compilation. In *Conf. on Programming Language Design and Implementation*, 1995.

[2] U. Banerjee. *Dependence Analysis for Supercomputing.* Norwell, Mass.: Kluwer Academic Publishers, 1988.

[3] W. Blume et al. Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, 29(12):78–82, Dec. 1996.

[4] W. Blume and R. Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. *Proc. of Supercomputing, Washington D.C.*, Nov. 1994.

[5] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Trans. Program. Lang. Syst.*, 12(3):341–395, 1990.

[6] B. Creusillet and F. Irigoin. Exact vs. approximate array region analyses. In *Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, 1996.

[7] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical data flow framework for array reference analysis and its use in optimizations. In *Conf. on Programming Languages Design and Implementation*, Albuquerque, N.M., June 1993.

[8] P. Feautrier. Parametric integer programming. *Operations Research*, 22(3):243–268, 1988.

[9] P. Feautrier. Dataflow analysis of array and scalar references. *Int. Journal of Parallel Programming*, 20(1):23–54, 1991.

[10] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *Conf. on Programming Languages Design and Implementation*, Toronto, Ont., June 1991.

[11] J. Gu, Z. Li, and G. Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Conf. on Supercomputing*, San Diego, CA, 1995.

[12] M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. *Int. Journal of Parallel Programming*, 28(6):537–562, 2000.

[13] M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Trans. on Programming Languages and Systems*, 18(4):477–518, 1996.

[14] M. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, Dec. 1996.

[15] J. Hoeflinger. *Interprocedural Parallelization Using Memory Classification Analysis.* PhD thesis, Univ. of Illinois, Urbana-Champaign, Aug., 1998.

[16] D. K. Chen, J. Torrellas, and P.-C. Yew. An Efficient Algorithm for the Run-time Parallelization of DOACROSS Loops. *In Conf. on Supercomputing*, Washington D.C., Nov. 14-18, 1994

[17] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *Symp. on Principles of Programming Lang.*, 1998.

[18] X. Kong, D. Klappholz, and K. Psarris. The I test: An improved dependence test for automatic parallelization and vectorization. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):342–349, July 1991.

[19] S. Leung and J. Zahorjan. Improving the performance of runtime parallelization. In *Symp. on Principles and practice of parallel programming*, San Diego, CA, May 1993.

[20] Y. Lin and D. Padua. Analysis of irregular single-indexed array accesses and its application in compiler optimizations. In *Int. Conf. on Compiler Construction*, 2000.

[21] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array data-flow analysis and its use in array privatization. In *Symp. on Principles of Programming Languages*, Charleston, SC, Jan. 1993.

[22] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *Conf. on Programming Languages Design and Implementation*, Toronto, Ont., June 1991.

[23] S. Moon and M. W. Hall. Evaluation of predicated array data-flow analysis for automatic parallelization. In *Symp. on Principles and Practice of Parallel Programming*, New York, NY, 1999.

[24] S. Moon, M. W. Hall, and B. R. Murphy. Predicated array data-flow analysis for run-time parallelization. In *Int. Conf. on Supercomputing*, Melbourne, Australia, July 1998.

[25] S. Moon, B. So, M. W. Hall, and B. R. Murphy. A case for combining compile-time and run-time parallelization. In *4th Workshop on Languages, Compilers, and Run-Time Systems*, London, UK, 1998. Springer-Verlag.

[26] Y. Paek, J. Hoeflinger, and D. Padua. Efficient and precise array access analysis. *ACM Trans. of Programming Languages and Systems*, 24(1):65–109, 2002.

[27] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91*, Albuquerque, N.M., Nov. 1991.

[28] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In *1993 Workshop on Languages and Compilers for Parallel Computing*, in LNCS 786, Portland, OR, 1993.

[29] C. G. Quiñones et al. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Conf. on Programming Languages Design and Implementation*, New York, NY, 2005.

[30] L. Rauchwerger and D. A. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *Conf. on Programming Languages Design and Implementation, La Jolla, CA*, June 1995.

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

[32] S. Rus, D. Zhang, and L. Rauchwerger. The Value Evolution Graph and its use in memory reference analysis. In *Conf. on Parallel Arch. and Compilation Techniques*, Antibes, France, 2004.

[33] J. H. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Trans. on Computers*, 40(5):603–612, May 1991.

[34] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of Call statements. In *Symp. on Compiler Construction*, Palo Alto, CA, 1986.

[35] R. van Engelen, et al. A unified framework for nonlinear dependence testing and symbolic analysis. In *Int. Conf. on Supercomputing*, Saint Malo, France, 2004.

[36] M. Wolfe and C.-W. Tseng. The Power test for data dependence. *IEEE Trans. on Parallel and Distr. Syst.*, 3(5):591–601, Sept. 1992.

[37] M. J. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1995.

[38] P. Wu, A. Cohen, and D. Padua. Induction variable analysis without idiom recognition: Beyond monotonicity. In *Workshop on Languages and Compilers for Parallel Computing*, Cumberland Falls, KY, 2001.

[39] H. Yu and L. Rauchwerger. Run-time parallelization overhead reduction techniques. In *Int. Conf. on Compiler Construction*, Berlin, Germany, March 2000.