

Compiler Technology for Migrating Sequential Code to Multi-threaded Architectures

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

Technical Report TR06-006
Parasol Lab
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112

April 28, 2006

Abstract

Executing sequential code in parallel on a multithreaded machine has been an elusive goal for many years. It has recently become quite important due to the widespread introduction of multi-cores in PCs. Automatic multi-threading could not be achieved so far because classic compiler analysis was not powerful enough and program behavior was found to be in many cases input dependent. Run time, speculative 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 technique, 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. 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 fully 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 and is also more precise (and thus does not need to be conservative) but it generates overhead. These two approaches have so far progressed in parallel with little interaction, leaving optimizations like parallelization at a level reached almost 10 years ago. Recently however, the decision of the major vendors to move to multi-core based architectures, has brought program parallelization of existing codes to the forefront. In fact there seems to be a degree of urgency from the part of the major vendors to enable their users to exploit the coarser level parallelism offered by these new micros with their existing software base. To this end we are proposing in this paper to advance the state of the art in automatic parallelization to a practical level by introducing a novel and powerful compiler framework, **Hybrid Analysis (HA)**. The general Hybrid Analysis approach modifies both classic and dynamic analysis methods in order to integrate them seamlessly. It represents a departure from the classic analysis paradigm insofar as it answers not only if an optimization is legal but 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 well established domain of data dependence analysis, which is essential to the automatic detection of parallelism. We have implemented our **Hybrid Dependence Analysis (HDA)** in the Polaris [3] compiler and achieved an impressive degree of good quality parallelization on a large number of scientific codes. We believe that this advance could significantly improve the utilization of multi-core processors in particular, and expand the coverage and profitability of compiler optimizations in general.

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 [32]), solving (mostly) linear integer equations with constraints [7, 23], 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 because of their 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 cannot be computed statically. In the more recent past, dynamic techniques [29, 26] 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, even when optimized for scalability. 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. In [18, 23, 27] the authors had recognized the need to bridge compile-time and run time analysis. However, their solutions, even when implemented, did not go far enough for practical impact. For a more detailed discussion of previous relevant work see Sec. 6.

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. We further recognize, as have others, that the best solution is to combine the power of both static and dynamic analysis in a seamless framework in order to improve the profitability and thus the coverage of automatic parallelization and thus the impact of multi-core processors.

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 (frequently 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 Dependence Analysis (HDA), implements, in a comprehensive manner a seamless bridge between static and dynamic analysis. Instead of only verifying that an optimization is valid, it can also generate, often optimally, the statically unknown conditions under which a transformation could be legal.
- Introduces a novel technique to transform statically unresolved data dependence relations represented as memory reference sets, into a predicate set expressing (necessary) and sufficient conditions for parallelization. These conditions can then be evaluated dynamically.
- A full implementation of the HDA framework in a research compiler (Polaris) and experimental results showing that automatic parallelization is possible.

We further believe that HA is a general approach many aggressive optimizations beyond HDA for which we cannot obtain definitive answers during static compilation.

2 An Overview of Hybrid Dependence Analysis

Hybrid Dependence Analysis (HDA) consists of a compile time phase which extracts symbolic data independence conditions and of a run time phase in which the parallel version of the code is predicated by the actual values of the associated independence conditions.

The compile-time part of HDA formulates an independence problem in terms of sets of references: the set of memory locations *read* in one iteration must not overlap with the set of memory locations *written* in another iteration. Let us illustrate this discussion with a very simple example. Consider the loop in

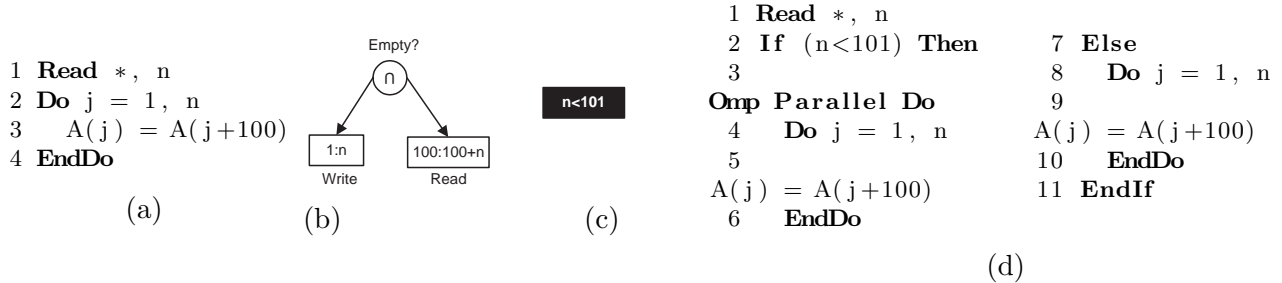


Figure 1: Example of an input-sensitive dependence and corresponding predicated parallelized code. (a) Original code. (b) Are there any memory locations that carry cross-iteration dependences? (c) Sufficient independence condition. (d) The parallelized code contains a predicated parallel loop.

Fig. 1(a). In order to generate code for its parallel execution, the compiler must disprove the existence of loop carried data dependences. It must prove (b) that the set of *read* references [101:100+n] and the set of *write* references [1:n] are disjoint, i.e., that their intersection is empty. 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 a simple condition, $n < 101$ is sufficient to prove the *read* and *write* sets disjoint. The extracted condition (c), shown on black background, is inserted in the generated code and used to predicate the parallel version of the loop (d). When $n=50$, the loop is executed in parallel. When $n=150$, the loop is executed sequentially. The extraction of conditions from intersections of linear intervals such as the one in Fig. 1(b) was used by vectorizers such as KAP and later documented in [24, 19, 20, 18, 12].

Most reference patterns in loops are more complex than those in Fig. 1(a). The relevant sets of references *read* and *write* cannot, in general, be represented as linear intervals. Moreover, parallelization is profitable at large levels of granularity, which correspond to large loops, spanning a large amount of code. Quite often, such loops contain nonlinear patterns, such as indirect memory references or nonlinear control flow. Let us follow the slightly more complex example in Fig. 2(a). The independence question is still represented as whether a set intersection is empty (b), but the set of *read* references is not a simple interval because of the unknown control flow value of $x < 0$. However, the problem of proving the *read* and *write* sets disjoint can be divided into two subproblems (c). In order to solve the first subproblem, let us notice that when $x < 0$ is false, the corresponding predicated set becomes empty. When $x < 0$ is true, the subproblem reduces to proving the sets disjoint, which is similar to the simpler problem in Fig. 1. The subproblem on the right in Fig. 2(c) is similar to the simpler problem in Fig. 1. The final result is shown in Fig. 2(f) as a simple logical expression which can be evaluated quickly at run time. Let us point out the important steps taken to solve these problems.

- We represent the set of memory locations that carry cross-iteration dependences as a tree in which the leaves are linear intervals and the internal nodes are operators, such as set union, set intersection and predication (Fig. 2(b)). We formulate the independence problem as testing whether this dependence set is empty.
- We apply a sequence of transformations which convert this problem into an equivalent logical expression that can be evaluated efficiently at run time (Fig. 2(c-e)). These transformations are applied in a recursive descent on the tree representation of the dependence set and are based on set algebra semantics.

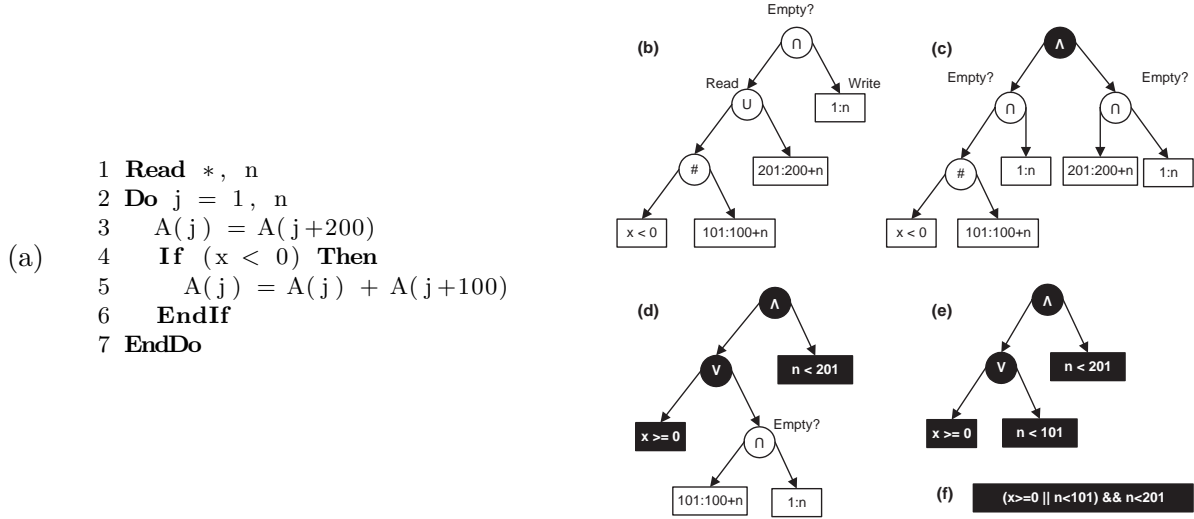


Figure 2: Extraction of an independence predicate from an independence equation. The black nodes represent simple conditions and logical operations that are easier to evaluate at run time than it is to solve the original independence problem. (a) Original code. (b) Independence equation as intersection of *read* and *write* reference sets. \cap and \cup stand for set intersection and union respectively and $\#$ means predication. (c) The original problem was divided into two subproblems. \wedge and \vee stand for logical *and* and *or* respectively. (d) Intermediate result. (e) The final result is an accurate independence predicate which is inserted in the generated code (f) and that will be evaluated efficiently at run time.

The following section presents in detail how this process is generalized to extracting sufficient conditions from general data independence problems in arbitrary structured programs.

3 Hybrid Dependence Analysis (HDA)

Hybrid Dependence Analysis represents the process of solving dependence equations efficiently by using a mix of compile time and run time techniques. We represent dependence equations as $DS = \emptyset$, where DS is the set of all memory locations that carry dependences. The compile time part of HDA starts by trying to prove statically that the dependence set is empty. If it succeeds, the corresponding loop will be run in parallel without using any run time tests. If it proves statically that the dependence set is not empty, the loop will be run sequentially. When a decision cannot be made at compile time, HDA extracts statically simple independence conditions that are (1) sufficient to prove the loop parallel and (2) easy to evaluate at run time. Its run time analysis consists of evaluating the independence condition and selecting the parallel code version when it holds true.

In this section we will describe how we automate the process of extracting simple independence conditions from general dependence equations. We will first present formally the data structures involved in this transformation and then follow with a detailed presentation of the algorithms. We use an existing USR representation [27] for sets of references, which has a tree structure as shown in Fig. 2(b). We will introduce a representation named PDAG to represent independence conditions. They can be seen as symbolic expressions that will produce at run time the boolean value of a dependence test.

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

Figure 3: USR formal definition. \cap , \cup , $-$ are elementary set operations: intersection, union, difference. $Gate\#USR$ 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.

3.1 Symbolic Representations

3.1.1 Sets of Memory References: the USR

The *Uniform Set of References* (**USR**) previously introduced in [27] 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 simplest form of a USR is the Linear Memory Access Descriptor (LMAD) [12, 21], 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. 1, the *read* pattern on array A is an LMAD, $[101:100+n]$.

The USR is stored as an abstract syntax tree with respect to the language presented in Fig. 3. The reference sets in Fig. 1(b) and Fig. 2(b) are USRs. When memory references are expressed as linear functions, USRs consist of a single leaf, i.e., a list of LMADs. In Fig. 1, the *read* pattern is represented as an LMAD, $[101:100+n]$. When the analysis process encounters a nonlinear reference pattern or when it performs an operation (such as set difference) whose result cannot be represented as a list of LMADs, we add internal nodes that record accurately the operations that could not be performed. In Fig. 1(b) we added an intersection operator node that shows that the intersection could not be performed statically.

The USR is also used to formulate the **Dependence Set (DS)** of a program block. The DS is obtained through various set operations (mainly intersection and difference) of the aggregated memory reference descriptors (USRs). (For example, Write First, Read Only, Read Write sets). The internal nodes of the Dependence Set tree pinpoint the exact points of static analysis failure and constitute an excellent starting point for extracting run time conditions from a USR equation.

3.1.2 Independence Conditions: the PDAG

The Predicate Directed Acyclic Graph (PDAG) is an analytical, symbolic representation of a boolean expression. PDAGs are extracted automatically from dependence equations that cannot be solved

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

Figure 4: PDAG formal definition. \wedge, \vee, \neg are the elementary logical operators *and, or, not*. $\otimes_{i=1,n}^\wedge PDAG(i)$ holds true if and only if each of $PDAG(i)$ holds true, $i = 1, n$. $PDAG(\text{formals}) \bowtie \text{Call Site}$ 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*.

statically $DS = \emptyset$, where DS is represented as a USR. *PDAGs are the boundary between the compile time and run time analysis*. They are the final result of static analysis: Conditions used to predicate the validity of dynamic optimizations. They are inserted in the generated code and evaluated at run time. Their dynamic values are used to choose between sequential and parallel code versions. In its simplest form, the PDAG is a logical expression such as $x < 0$ in Fig. 1(c). At the other extreme, it can be an arbitrary program slice that produces a boolean value. PDAGs are represented as trees having logical expressions as leaves and operators as internal nodes. The PDAG tree structure generally mirrors the tree structure of the dependence set as a USR, which in turn generally mirrors the block structure of the program. This makes PDAGs relatively easy to associate with sections in the original program, which makes it easier for compiler writers to program and understand the analysis process.

PDAGs are expressive enough to represent any possible dependence question, and simple enough to be quickly evaluated dynamically. The grammar in Fig. 4 defines PDAGs formally. They 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 employed to express particular problems more efficiently, and reference based tests represent the fallback when cheaper conditions cannot be extracted.

3.2 Symbolic Analysis Algorithms

3.2.1 Syntax Directed Predicate Extraction

After resolving all statically analyzable dependence questions we are left with a Dependence Set (DS), represented as an USR, for which we could not give a definitive answer. For the resolution of this problem we have formulated the algorithm *Solve* shown in Fig. 5. This algorithm extracts a set of conditions, represented as a PDAG, which, when evaluated dynamically, returns *true* if and only if the dependence set is empty.

Algorithm *Solve* extracts the PDAG from the dependence set by recursively descending its USR

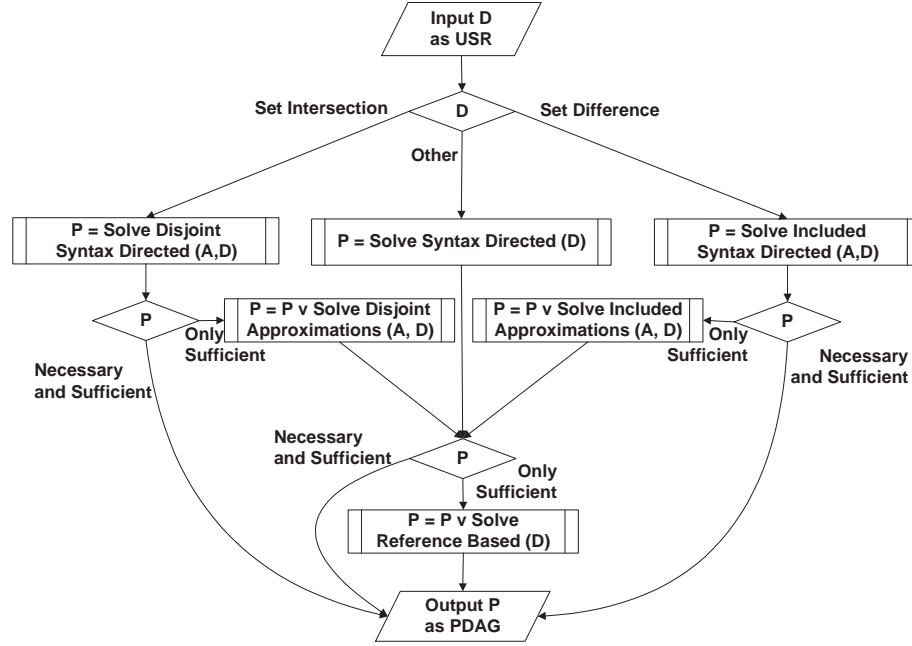


Figure 5: **Algorithm Solve**: Extraction of a sufficient run time test as a PDAG from a dependence equation $D = \emptyset$. Details on the implementation of the subalgorithms are presented in the Appendix. We accumulate PDAGs in increasing order of complexity when the partial solutions are sufficient but not necessary, using the logical or operator \vee .

tree and decomposing the nodes using elementary set algebra identity transformations. For instance, in order to prove a union of two terms empty, it is necessary and sufficient to prove both terms empty. In other words, $A \cup B = \emptyset \Leftrightarrow A = \emptyset \wedge B = \emptyset$.

Our current implementation is optimistic, i.e., it extracts sufficient independence conditions. A similar approach can be used to extract pessimistic dependence conditions. Inexpensive pessimistic conditions could be used at run time to flag the sequential loops quickly and thus avoid the overhead of more expensive dependence tests. The algorithm maintains throughout the recursive descent process information on whether the current solution is equivalent to the original independence problem. When the solution obtained by the recursive descent approach is sufficient but not necessary, more specialized and expensive reference based tests [26, 27] can be generated, thus avoiding a conservative decision (i.e., not parallel). The dynamic evaluation of these tests will then ensure an exact answer but will cost a higher run-time overhead, proportional to the dynamic reference count of the Dependence Set we started from. Fig. 6 presents such a case where a simple independence condition cannot be extracted.

Unfortunately, the recursive descent approach does not work for set intersections and differences as well as for unions. 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 *Solve Disjoint Syntax Directed* and *Solve Included Syntax Directed* continue the recursive descent according the syntax of the terms of intersections and differences. They rely on dividing more complex equations such as $A \cap (B \cup C) = \emptyset$ into simpler equations such as $A \cap B = \emptyset$ and $A \cap C = \emptyset$, based on elementary set identities. However, there are USR configurations that cannot be broken up, such as $A \cap B \cap C = \emptyset$.

```

1 Read *, (p(j), j=1,100),
              (q(j), j=1,100)
2 Do j = 1, 100
3   A(p(j)) = A(q(j))
4 EndDo

```

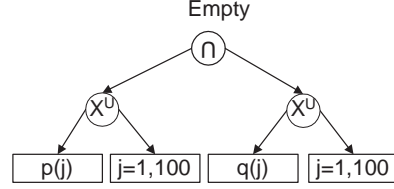


Figure 6: A Hybrid Analysis extreme: in general, no test can solve this problem faster than the reference-by-reference LRPD test.

```

1 Read *, x(i, j),
              j=1,n, i=1, len(j)
2 Do j = 1, n
3   Do i = 1, len(j)
4     If (x(i, j) < 0)
5       W(i) = ...
6     EndIf
7   EndDo
8   Do i = 1, len(j)
9     ... = W(i)
10  EndDo
11 EndDo

```

(a)

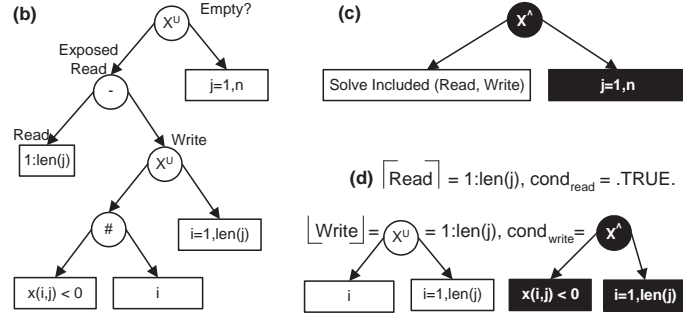


Figure 7: Extraction of an independence predicate using approximation.

When the recursive descent described in algorithm *Solve* reaches such a point, it resorts to approximation to extract conditions that, in most cases, are sufficient but not necessary to prove independence.

3.2.2 Extracting PDAGs from USR Approximations

In the example in Fig. 7, array W could be proved privatizable by showing that the *read* at line 9 is covered by the *write* at line 5. However, the shape of the USR that describes the *write* pattern is outside any of the cases in the *Solve* algorithms presented above. We will show that even when two USRs cannot be compared directly, a meaningful PDAG can often be extracted based on comparisons between predicated approximations of the USRs.

Several memory reference analysis techniques have proposed the use of approximations of reference sets in the presence of subscript arrays or arrays of conditionals [5, 12, 27]. 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 $\lfloor P \rfloor \subseteq P \subseteq \lceil P \rceil$ and $\lfloor P \rfloor$ and $\lceil P \rceil$ fit their model. We apply this to our framework by approximating complex USRs with predicated LMADs.

Returning to the example in Fig. 7, 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 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, $\lceil read \rceil = [1 : len(j)]$, and $\lfloor write \rfloor = [1 : len(j)]$, when $\bigotimes_{i=1, len(j)}^{\wedge} x(i, j) < 0$. The approximation process is invoked by algorithms *Solve Disjoint Approximations* and *Solve Included Approximations* shown in Fig. 5.

The approximation algorithm (not shown) is based on a recursive descent on the USR structure. When looking for underestimates, the algorithm makes choices that maximize the size of the result, and when looking for overestimates, it minimizes. In the example in Fig. 7, the underestimate of *write* is maximized optimistically and ends up covering the overestimate of the *read*. In general, this aggressive approach increases the chances of extracting nontrivial conditions.

Extracting PDAGs from LMAD Equations

When the recursive descent on USRs reaches leaves, we have to extract conditions from equations involving linear intervals. Although in general these are hard problems even for linear memory reference descriptors like the LMAD [12, 22], most practical cases are tractable. We have modified the multi-dimensional LMAD intersection and subtraction algorithms presented in [12] to return sufficient conditions under which their result is empty. For instance, the problem of proving two 1-dimensional LMADs disjoint, is equivalent to a bounds check and a GCD test.

3.3 Testing Monotonicity and Disjoint Intervals

```

1 Do j = 1, n
2   Do i = 1, len(j)
3     A(ptr(j)+i) = ...
4   EndDo
5 EndDo

```

Figure 8: Example of a case where a sorting based test is more efficient than applying the *Solve* algorithm.

Consider the dependence problem on array *A* in the example in Fig 8. A direct application of the *Solve* algorithm would result in a test of $n*(n-1)/2$ bound checks, one for each pair $([ptr(j)+1:ptr(j)+len(j)], [ptr(k)+1:ptr(k)+len(k)])$, 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 $[D_i] = [lower_i : upper_i]$ 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.

Sorting-based tests are generated whenever the per-iteration reference set can be bounded by a symbolic interval.

4 Implementation and Complexity

4.1 Implementation of an Automatic Parallelizer

We have implemented an automatic parallelizer based on Hybrid Dependence Analysis in the Polaris [3] research compiler Fig. 9 shows the compile time analysis organization. First individual array references

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** ($DS = \emptyset$) **Of**
 - True:** Generate **Parallel** Loop
 - False:** Generate Sequential Loop
 - Maybe:** (not sure at compile time)
Extract condition $P \Leftrightarrow (DS = \emptyset)$
Generate **Parallel** Loop guarded by P

Figure 9: Automatic parallelization using Hybrid Data Dependence Analysis.

are aggregated to loop levels as USRs. The collection of the aggregated memory references is followed by memory classification. Using set operations, references are classified into Write First(WF), Read Write(RW) and Read Only (RO) sets represented as USRs. This classification preserves sufficient *write* ordering information to formulate accurate data dependence and privatization problems [12, 27]. The aggregation as USRs is crucial to the analysis of large loops possibly spanning multiple subroutines in a scalable manner. In the second phase important loops are analyzed for the purpose of parallelization. This static analysis checks if the loop dependence problems (obtained from the WF, RW and RO sets) can be solved at compile time. Some dependences are removed by recognizing reductions, push-backs and other parallelizable patterns collected in a pattern library. Every remaining dependence problem is formulated as an HDA problem $DS = \emptyset$, where DS is the USR that describes the set of all dependent memory locations. The result of this analysis can yield three answers: Provable empty set implying the loop is parallel (PDAG with a constant value of *true*), provable non-empty set implying the loop is sequential (PDAG with a constant value of *false*), and undecided (PDAG with a dynamic value). In this latter case we generate a parallel loop version guarded by a dynamic predicate whose value is determined by the run time evaluation of the extracted PDAG.

4.2 Complexity of Compile Time Analysis

The complexity of the memory reference aggregation process is at most quadratic with the size of the program [27]. 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.

Table 1 presents compilation statistics. The number of USR and PDAG nodes is relatively small.

Code	Lines	Time	USRs	PDAGs	Op Ratio
ADM	5,791	455	35,249	10,456	$1.8 * 10^5$
ARC2D	3,099	102	13,178	22	$1.2 * 10^7$
BDNA	4,919	36	11,181	156	—
DYFESM	3,903	38	6,841	756	$1.5 * 10^4$
FLO52	2,508	120	8,371	0	—
MDG	1,237	15	8,085	744	$6.7 * 10^0$
OCEAN	2,738	122	14,664	208	$2.1 * 10^4$
SPEC77	4,582	303	75,032	4,733	$1.0 * 10^0$
TRACK	2,523	245	27,790	2,931	$1.0 * 10^0$
TRFD	656	120	1,684	139	$5.6 * 10^4$
APPLU	3,980	56	13,212	34	—
APSI	7,488	399	36,593	10,800	$1.6 * 10^7$
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	$3.0 * 10^6$
ORA	373	7	2,562	0	—
SWM256	487	8	1,520	0	—
TOMCATV	194	5	1,056	32	—

(a)

(b)

Table 1: (a) Compile-time analysis statistics (seconds). Column 4 and 5 show the total number of USR and PDAG nodes created (operator or leaves). (b) Run time test dynamic overhead reduction through HDA: ratio between the number of actual memory references and the number of PDAG operations performed at run time.

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 applications differ greatly in 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 [27].

4.3 Complexity of Run Time PDAG Evaluation

PDAGs contain four types of run time operations: (1) evaluation of elementary conditional expressions, (2) sorted checks, (3) actual evaluation of USRs and comparison to the empty set and (4) reference-by-reference LRPD. We extract, for each dependence equation, a **cascade of tests** (Fig. 10). ordered by their estimated complexity. They range from $O(1)$ tests as the one in Fig. 1 to $O(n)$ dynamic reference instrumentation as is the case in Fig. 6. Evaluating USRs at run time generally consists of fewer (but more complex) operations than the reference-by-reference LRPD [26]. 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

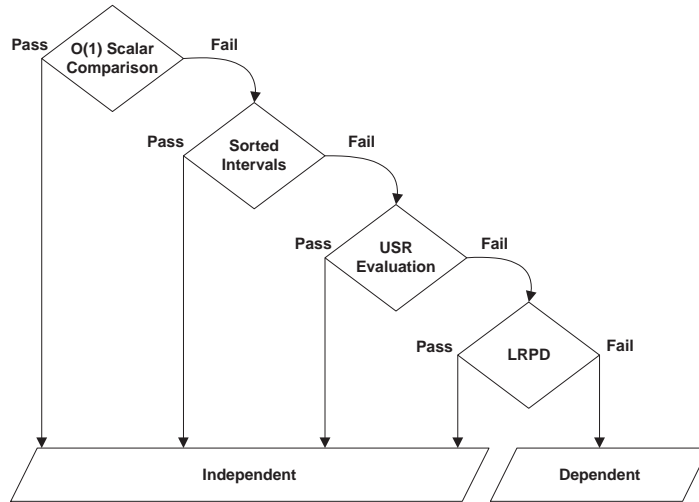


Figure 10: Cascade of sufficient run time tests in increasing order of complexity.

aggregation and equation inversion are not possible (Fig. 6), 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 test code generation consists of a syntax-based translation from the PDAG grammar to Fortran.

We apply loop invariant hoisting to USRs and PDAGs by performing aggressive invariance analysis on their sets of input variables. Invariance problems on USRs resulted from subscripted subscripts are formulated as dependence problems on the subscript arrays, which *are solved by the same HDA algorithm*. This is achieved by representing the exact referenced memory regions of the subscript array as USR themselves, and thus identifying the exact subregion of the subscript array that affects the shape or size of the memory pattern on the host array. An interesting problem arises when a more expensive test such as LRPD can be hoisted out of a loop, but a simpler $O(1)$ version is loop variant. At this time we (simplistically) hoist tests as far away as possible and build cascades from tests at the same nesting level respectively.

5 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 and (d) contributes significantly to the overall parallelization of programs, i.e., they are instrumental in obtaining the presented results.

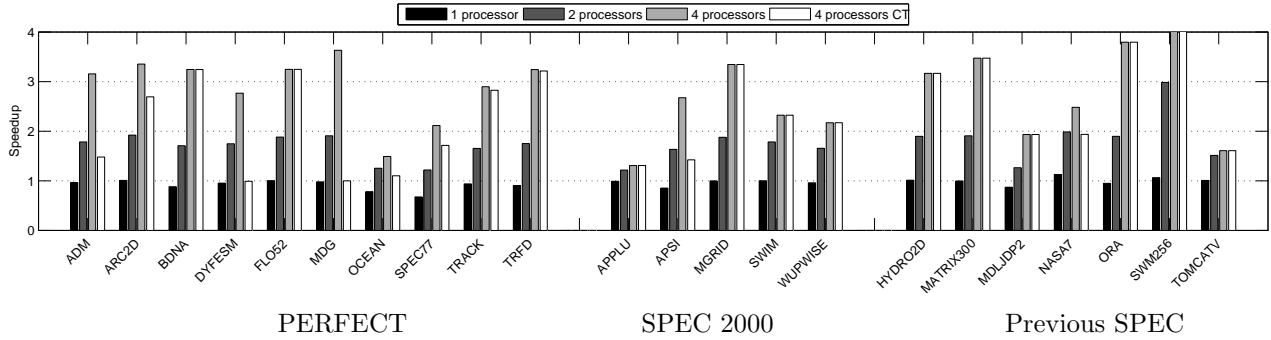


Figure 11: Speedups relative to the sequential version of automatically parallelized benchmark applications on 1, 2, and 4 processors. CT = speedups obtained using only compile-time methods.

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 preconditions any further optimizations.

5.1 Experimental Setup

We ran the automatic parallelizer based on HDA on a set of 22 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 was 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. All codes were compiled with the Intel Fortran Compiler version 9.0 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 today’s machines, a very short sequential execution time. Their parallelization, while correct, brings the time of some loops down to the execution time of barriers, making it impossible to measure the effect of parallelization. We strongly believe that the structural characteristics are still quite relevant and that expanding the execution times by disabling 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`.

5.2 Speedup Measurements

Fig. 11 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

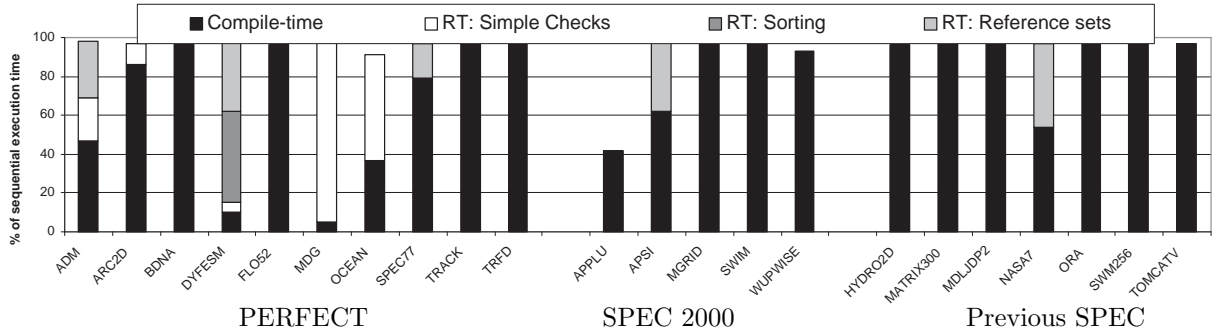


Figure 12: Automatic parallelization coverage as percentage of the sequential execution time.

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 than 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 of 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.

5.3 Run Time Tests 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. 12 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 1(b) 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.

Fig. 12 shows the coverage of parallelization achieved by HDA. 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 HDA because it does not quantify the fact that we can detect coarse 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.

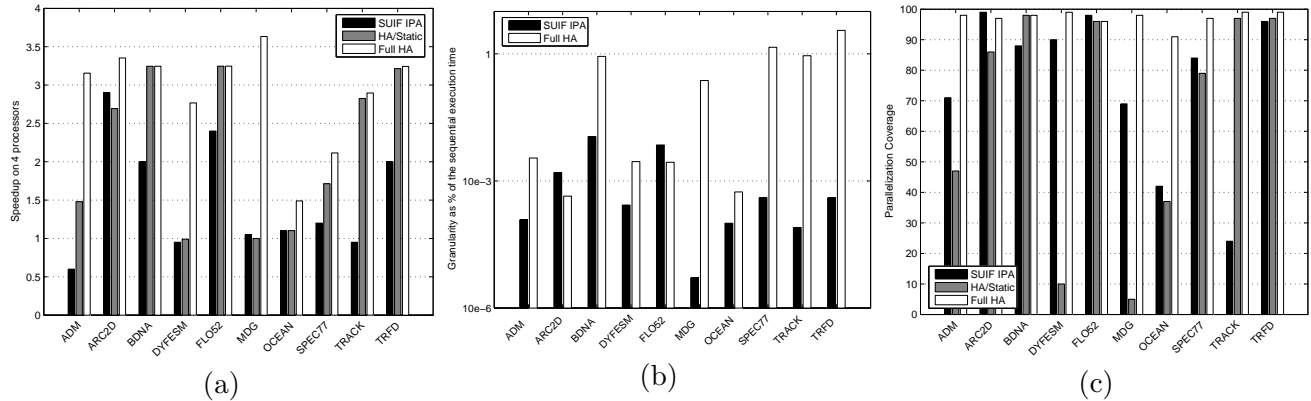


Figure 13: Comparison of automatic parallelization using HDA against SUIF Interprocedural Automatic Parallelizer [11]: (a) Speedup on four processors, (b) Granularity as average duration of a parallelized section expressed as percentage of the execution time, and (c) Coverage as percentage of the execution time.

5.4 Comparison to the Current State of the Art

We cannot compare our results with any of the previous hybrid parallelization techniques [24, 19, 20, 18, 27] because they did not provide extensive results across whole benchmark suites. The techniques described by [1] are applied to other classes of programs and thus cannot be compared directly. [11] presents the most recent (2005) comprehensive results in automatic parallelization, although based only on static analysis. Their techniques were implemented in the SUIF compiler and include interprocedural data dependence analysis and privatization. Fig. 13 presents a detailed comparison with our results in the automatic parallelization of PERFECT benchmarks. We compare SUIF/IPA with the static part of HDA (HA/Static) and with the full, static and dynamic, HDA (Full HA). It is not perfectly accurate because the measurements were taken on different machines. Furthermore, SUIF uses locality enhancement transformations which we did not do.

Program ARC2D was parallelized with approximately same coverage and at the same granularity by both compilers and resulted in similar speedups, which shows that the speedup comparison is relevant. SUIF/IPA shows better parallelization coverage on ADM than the HA/Static. However, HA/Static favors parallelization at a higher level of granularity which results in positive, though modest, speedup. Given an appropriate system we could generate nested parallelism and exploit both fine and coarse grain parallelism. Full HA has better coverage and higher granularity resulting in speedup of more than 3 on 4 processors. The execution time in ADM (and more so in DYFESM and MDG) is dominated by large loops that iterate over the whole data set and which can only be parallelized at run time. SUIF/IPA manages to parallelize only inner loops and gets good coverage but cannot achieve speedups. Full HA parallelizes them at the highest level of granularity available and achieves good speedups on all three of them.

We believe that the consistent solid performance results across a large number of standard benchmark applications proves our claims on the effectiveness of HDA.

6 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 were reduced to proving that a system of linear constraints had no integer-valued solution [6, 31, 2, 8, 17, 14, 22, 21]. In all these systems, the symbolic expressions must be linear, although some particular extensions can handle certain classes of nonlinear references. They cannot generally be used to analyze (1) memory references through index arrays, (2) memory references controlled by arrays of conditionals and (3) memory references indexed or controlled by data values computed within the code section under analysis.

Pattern recognition and index property analysis were proposed as solutions for nonlinear reference patterns [16]. Its applicability is limited to the cases studied. Symbolic value range [4] and monotonicity analysis [10, 9, 16, 33, 30, 28] also targeted some classes of nonlinear reference patterns. They are generally not integrated well with other techniques and thus lack generality. For instance, the *Range Test* [4] compares the value ranges of two reference sets, but does not deal with strided patterns directly. We use value ranges and monotonicity information [4, 28] in a more general way, not only to compare offsets, but also strides and spans, and to prove predicate implication, redundancy or contradiction.

Run time data dependence tests were proposed to solve dependence problems that did not have compile-time solutions [29, 13, 15, 26, 25]. Their overhead may sometimes void the optimization benefits they bring. Our approach reduces the overhead by performing much of the analysis symbolically, at compile-time.

Hybrid Dependence Analysis and Parallelization. One of the first forms of hybrid analysis was conditional vectorization [32]. It is an effective technique, but limited in scope to small 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, which is similar conceptually to hoisting USR computation to the data flow locations where their input variables become available [27]. HDA pushes symbolic analysis further and extracts PDAGs as cascades of conditions that are later hoisted in a similar fashion, which leads to even lighter run time tests. We cannot make a quantitative comparison with [1] because we targeted different classes of programs.

[19, 20, 18] synthesize simple conditions from data dependence and data flow equations on arrays. Their applicability is limited to checks on scalars such as loop bounds or scalar control flow values 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. A similar approach of comparable symbolic power is presented by [12]. Safety guards are inserted to predicate optimistic results of statically undecidable LMAD operations. [24] showed how sufficient predicates can be extracted by simplifying Presburger formulas with uninterpreted function symbols. Although our implementations are very different, they are fundamentally very similar. Unfortunately they did not apply it to real applications so we cannot compare the quality of the generated run time tests, which is what makes the difference in dynamic optimization methods. [27] uses USRs to express dependence tests but does not provide a way to extract simple run time tests. They propose the evaluation of USRs at run time followed by comparison to the empty set. However, in general a simple *Yes/No* answer is sufficient. The evaluation of USRs is generally not needed and it often results in unnecessary run time overhead. For instance, the best speedup presented by [27] on MDG on 4 processors is 2.1, while our best is 3.7, albeit on different machines.

[34] 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 PDAGs can express much more complex relations between reference patterns and eliminate more classes of redundant checks. A decisive role is played by the USRs unification of apparently different patterns which would otherwise appear to be unrelated. Their best speedup on MDG on 4 processors is 1.7, while ours is 3.7 (on different machines).

7 Conclusions

The **Hybrid Analysis (HA)** framework substantially increases the coverage of compiler optimizations, especially parallelization because it not only validates transformations but also generates sufficient, simple *dynamic* validation conditions. The extraction of these low cost, dynamically verifiable conditions is achieved by using a novel *predicate extraction* algorithm. 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 *SIGPLAN Conf. on Prog. Language Design and Implementation*, pp. 258–269, 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, 1996.
- [4] W. Blume and R. Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. *Proc. of Supercomputing, Washington D.C.*, pp. 528–537, November 1994.
- [5] B. Creusillet and F. Irigoien. Exact vs. approximate array region analyses. In *Workshop on Languages and Compilers for Parallel Computing*, LNCS, Springer-Verlag, 1996, pp. 86–100.
- [6] P. Feautrier. Parametric integer programming. *Operations Research*, 22(3):243–268, 1988.
- [7] P. Feautrier. Dataflow analysis of array and scalar references. *Int. Journal of Parallel Prog.*, 20(1):23–54, 1991.
- [8] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *ACM SIGPLAN '91 Conf. on Prog. Language Design and Implementation*, pp. 15–29, Toronto, Ont., June 1991.

- [9] M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. *Int. Journal of Parallel Prog.*, 28(6):537–562, 2000.
- [10] M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Trans. on Prog. Languages and Systems*, 18(4):477–518, 1996.
- [11] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Interprocedural parallelization analysis in suif. *ACM Trans. Program. Lang. Syst.*, 27(4):662–731, 2005.
- [12] J. Hoeflinger. *Interprocedural Parallelization Using Memory Classification Analysis*. PhD thesis, Univ. of Illinois, Urbana-Champaign, Aug., 1998.
- [13] D. kai Chen, J. Torrellas, and P.-C. Yew. An Efficient Algorithm for the Run-time Parallelization of DOACROSS Loops. in *Proc. for Supercomputing '94, Washington D.C., Nov. 14-18, 1994, Oct.*, 1994.
- [14] 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.
- [15] S. Leung and J. Zahorjan. Improving the performance of runtime parallelization. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog.*, pp. 83–91, May 1993.
- [16] Y. Lin and D. Padua. Analysis of irregular single-indexed array accesses and its application in compiler optimizations. In *Int. Conf. on Compiler Construction*, LNCS, Springer Verlag, pp. 202–218, 2000.
- [17] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *ACM SIGPLAN Conf. on Prog. Language Design and Implementation*, pp. 1–14, Toronto, Ont., June 1991.
- [18] S. Moon and M. W. Hall. Evaluation of predicated array data-flow analysis for automatic parallelization. In *Proc. of ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog.*, pp. 84–95, New York, NY, USA, 1999.
- [19] S. Moon, M. W. Hall, and B. R. Murphy. Predicated array data-flow analysis for run-time parallelization. in *Proc. of ACM Int. Conf. on Supercomputing*, pp. 204–211, Melbourne, Australia, July 1998.
- [20] S. Moon, B. So, M. W. Hall, and B. R. Murphy. A case for combining compile-time and run-time parallelization. In *LCR '98: Selected Papers from the 4th Int. Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pp. 91–106, London, UK, 1998. LNCS, Springer-Verlag.
- [21] Y. Paek, J. Hoeflinger, and D. Padua. Efficient and precise array access analysis. *ACM Trans. of Prog. Languages and Systems*, 24(1):65–109, 2002.
- [22] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91*, pp. 4–13, Albuquerque, N.M., Nov. 1991.

- [23] 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 768, pp. 546–566, Portland, Ore., Aug. 1993. Berlin: Springer Verlag.
- [24] W. Pugh and D. Wonnacott. Nonlinear array dependence analysis. In *Proc. of the 3-rd Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pp. 1–14, Kluwer Academic Publishers, Boston 1995.
- [25] C. G. Quiñones, *et. al.* Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proc. of the 2005 ACM SIGPLAN Conf. on Prog. Language Design and Implementation*, pp. 269–279, New York, NY, USA, 2005.
- [26] L. Rauchwerger and D. A. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *Proc. of the SIGPLAN Conf. on Prog. Language Design and Implementation, La Jolla, CA*, pp. 218–232, June 1995.
- [27] S. Rus, J. Hoeflinger, and L. Rauchwerger. Hybrid analysis: static & dynamic memory reference analysis. *Int. Journal of Parallel Prog.*, 31(3):251–283, 2003.
- [28] S. Rus, D. Zhang, and L. Rauchwerger. The value evolution graph and its use in memory reference analysis. In *13th Conf. on Parallel Architecture and Compilation Techniques*, pp. 243–254. IEEE Computer Society, 2004.
- [29] 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.
- [30] R. van Engelen, J. Birch, Y. Shou, B. Walsh, and K. Gallivan. A unified framework for nonlinear dependence testing and symbolic analysis. In *ACM Int. Conf. on Supercomputing*, pp. 106–115, 2004.
- [31] M. Wolfe and C.-W. Tseng. The Power test for data dependence. *IEEE Trans. on Parallel and Distributed Systems*, 3(5):591–601, Sept. 1992.
- [32] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing, Inc., Boston, MA, USA, 1995.
- [33] P. Wu, A. Cohen, and D. Padua. Induction variable analysis without idiom recognition: Beyond monotonicity. In *2001 Workshop on Languages and Compilers for Parallel Computing*, Cumberland Falls, KY, 2001, LNCS 2624, Springer Verlag, pp. 427–441.
- [34] H. Yu and L. Rauchwerger. Run-time parallelization overhead reduction techniques. In *Proc. of the 9th Int. Conf. on Compiler Construction, Berlin, Germany*. LNCS 1781, Springer-Verlag, March 2000.

Appendix

Text of the algorithms referenced in Fig. 5.

Algorithm Solve Syntax Directed

Input: D as USR
Output: P as PDAG s.t. $P \Rightarrow A = \emptyset$
Case D of:
 $LMADs:$ $P = HasEmptyDimension(LMADs)$
 $A \cup B:$ $P = Solve(A = \emptyset) \wedge Solve(B = \emptyset)$
 $q\#A:$ $P = \bar{q} \vee Solve(A = \emptyset)$
 $\otimes_{i=1,n}^{\cup}(A_i):$ $P = \otimes_{i=1,n}^{\wedge} Solve(A_i)$
 $A \bowtie Call\ Site:$ $P = Solve(A = \emptyset) \bowtie Call\ Site$

Algorithm Solve Disjoint Syntax Directed

Input: A, D as USRs
Output: P as PDAG s.t. $P \Rightarrow (A \cap D = \emptyset)$
Case D of:
 $B \cup C:$ $P = Solve(A \cap B = \emptyset) \wedge Solve(A \cap C = \emptyset)$
 $q\#B:$ $P = (\bar{q}, \mathbf{false}) \vee Solve(A \cap B = \emptyset)$
 $\otimes_{i=1,n}^{\cup}(B_i):$ $P = \otimes_{i=1,n}^{\wedge} Solve(A \cap B_i)$
Case A of:
 $B \cup C:$ $P = Solve(B \cap D = \emptyset) \wedge Solve(C \cap D = \emptyset)$
 $q\#B:$ $P = \bar{q} \vee Solve(B \cap D = \emptyset)$
 $\otimes_{i=1,n}^{\cup}(B_i):$ $P = \otimes_{i=1,n}^{\wedge} Solve(B_i \cap D)$

Algorithm Solve Included Syntax Directed

Input: A, D as USRs
Output: P as PDAG s.t. $P \Rightarrow (A - D = \emptyset)$
Case D of:
 $B \cup C:$ $P = Solve(A - B = \emptyset) \vee Solve(A - C = \emptyset)$
 $B \cap C:$ $P = Solve(A - B = \emptyset) \wedge Solve(A - C = \emptyset)$
 $B - C:$ $P = Solve(A - B = \emptyset) \wedge Solve(A \cap C = \emptyset)$
 $q\#B:$ $P = (q, \mathbf{true}) \wedge Solve(A - B = \emptyset)$
Case A of:
 $B \cup C:$ $P = Solve(B - D = \emptyset) \wedge Solve(C - D = \emptyset)$
 $B \cap C:$ $P = Solve(B - D = \emptyset) \vee Solve(C - D = \emptyset)$
 $B - C:$ $P = Solve(B - D = \emptyset)$
 $q\#B:$ $P = (\bar{q}, \mathbf{false}) \vee Solve(B - D = \emptyset)$

Algorithm Solve Disjoint Approximations

Input: A, D as USRs
Output: P as PDAG s.t. $P \Rightarrow (A \cap D = \emptyset)$
 $(cond_A, \lceil A \rceil)$ = a conditional LMAD overestimate of A
 $(cond_D, \lceil D \rceil)$ = a conditional LMAD overestimate of D
 $P = cond_A \wedge cond_D \wedge SolveDisjointLMADs(\lceil A \rceil, \lceil D \rceil)$

Algorithm Solve Included Approximations

Input: A, D as USRs
Output: P as PDAG s.t. $P \Rightarrow (A - D = \emptyset)$
 $(cond_A, \lceil A \rceil)$ = a conditional LMAD overestimate of A
 $(cond_D, \lfloor D \rfloor)$ = a conditional LMAD underestimate of D
 $P = P \vee (cond_A \wedge cond_D \wedge SolveIncludedLMADs(\lceil A \rceil, \lfloor D \rfloor))$
