Sensitivity Analysis for Automatic Parallelization on Multi-Cores *

Silvius Rus
Texas A&M University
Dept. of Computer Science
Parasol Laboratory
3112 TAMUS
College Station, TX 77843
silvius.rus@gmail.com

Maikel Pennings
Texas A&M University
Dept. of Computer Science
Parasol Laboratory
3112 TAMUS
College Station, TX 77843
pennings@cs.tamu.edu

Lawrence Rauchwerger
Texas A&M University
Dept. of Computer Science
Parasol Laboratory
3112 TAMUS
College Station, TX 77843
rwerger@cs.tamu.edu

ABSTRACT

Sensitivity Analysis (SA) is a novel compiler technique that complements, and integrates with, static automatic parallelization analysis for the cases when relevant program behavior is input sensitive. In this paper we show how SA can extract all the input dependent, statically unavailable, conditions for which loops can be dynamically parallelized. SA generates a sequence of sufficient conditions which, when evaluated dynamically in order of their complexity, can each validate the dynamic parallel execution of the corresponding loop. For example, SA can first attempt to validate parallelization by checking simple conditions related to loop bounds. If such simple conditions cannot be met, then validating dynamic parallelization may require evaluating conditions related to the entire memory reference trace of a loop, thus decreasing the benefits of parallel execution.

We have implemented Sensitivity Analysis in the Polaris compiler and evaluated its performance using 22 industry standard benchmark codes running on two multicore systems. In most cases we have obtained speedups superior to the Intel Ifort compiler because with SA we could complement static analysis with minimum cost dynamic analysis and extract most of the available coarse grained parallelism.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors,Compilers

General Terms

Performance, Experimentation

Keywords

Parallelism, Multicore, Sensitivity Analysis

*This research supported in part by NSF Grants EIA-0103742, ACR-0081510, ACR-0113971, CCR-0113974, EIA-9810937, ACI-0326350, and by the DOE Office of Science.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS '07 June 18-20, Seattle, WA, USA
Copyright 2007 ACM 978-1-59593-768-1/07/0006 ...$5.00.

1. INTRODUCTION

The recent introduction of multi-core based architectures to the mass market has brought program parallelization of the existing code base 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. Parallelizing compilers are a key enabling technology in this domain because they offer the advantage of automation and thus high productivity.

Parallelizing compilers focus, at least as a necessary step, on proving which loops can be executed in parallel (as a `doall`) Data dependence analysis techniques as simple as the GCD test [32] and as sophisticated as the Omega test [22] have been employed to statically prove the independence of memory references within a loop. After some limited success it had become clear that sparse, dynamic programs could not be automatically parallelized using these static techniques alone because their memory reference pattern is input dependent. The proposed solution was dynamic (run-time) analysis with the advantage of high accuracy (most symbolic data is instantiated) but with the drawback of run-time overhead. The dynamic approach has taken two directions: (a) a continuation of the static compilation analysis at run-time, and (b) a memory reference trace based analysis approach. In the first approach, symbolic expressions that could not be evaluated statically are postponed for run-time evaluation which then decides the (in)dependence of a loop. For example, if the static analysis cannot conclusively perform a standard data dependence test, e.g., a GCD test, because some of its parameters can be evaluated, we can always perform it at run-time when all information becomes available. In the second approach, more general and better suited for codes using indirection, the memory references are recorded and analyzed at run-time either before a loop is executed (inspector–executor mode [29]) or after an optimistic (speculative) parallel execution [26]. The complexity of this method is proportional to the number of dynamic references and thus is potentially expensive.

Overall, static and run-time approaches to automatic parallelization have progressed independently without significant integration. Partial, but insufficient, static analysis

was not used effectively to simplify run-time analysis. An improvement over this state of the technology was presented in [27]. Instead of performing a reference-based test, the technique, named Hybrid Analysis, uses an aggregated reference representation and performs dynamic analysis using set and interval operations very similar to those performed statically by a compiler. This often results in a significant reduction of run-time overhead.

A step further in automatic parallelization has been the re-formulation of the loop independence analysis into sufficient conditions (predicates) for which a loop can be parallelized. These conditions represent the *sensitivity* of parallelization to some input (dynamic) conditions. For example, in [24] the authors showed some limited examples of how sufficient predicates could be extracted by simplifying Presburger formulas with uninterpreted function symbols. These predicates are then returned to the programmer for evaluation (for interactive compilation). Further research [19, 20, 18, 12, 27] showed how to extract simple scalar conditions from relatively simple array data dependence predicates for a limited number of cases.

In this paper we present *Sensitivity Analysis* (**SA**), a technique that can extract a disjunction of simple to complex predicates that guard the dynamic parallelization (execution) of a loop. This "cascade" of conditions can start with a simple scalar comparison of input dependent variables and can end with a complex reference trace analysis based test. The *Sensitivity Analysis* presented here, unlike all previous related work, addresses all forms of flow sensitive memory references including nonlinear patterns such as indirection arrays in loops. **SA** seamlessly integrates all the results obtained by static analysis with those obtained through dynamic analysis thus ensuring minimum run-time overhead. Because this framework integrates dynamic analysis, where all needed information is available, it also assures a high degree of parallelization, i.e., most available parallelism. Its low overhead ensures its wide applicability.

We further show how we have implemented **SA** in the Polaris [3] compiler and extracted a high degree of coarse level parallelization on 22 scientific codes from the SPEC and PERFECT benchmark set. Our results are significantly superior to the one obtained by the Intel Ifort commercial compiler. On average, we obtained speedups of 1.5 on a dual core and 4 on a quad socket dual core.

1.1 Contribution

This paper makes the following contributions:

- Sensitivity Analysis, a general technique that transforms statically unresolved data dependence relations represented as memory reference sets, into a light weight predicate set expressing necessary and sufficient conditions for parallelization.
- A full implementation of this technique in a research compiler (Polaris) and experimental results showing that coarse grained automatic parallelization is profitable on multi-cores.

We believe that Sensitivity Analysis is a general approach to static/dynamic optimizations and is a key technology for the exploitation of multi-cores. It allows the use of dynamic information without paying a heavy price in dynamic overhead, and can significantly improve the profitability of coarse level parallelization on multi-core processors in particular, and has the potential to expand the coverage and profitability of many other compiler optimizations as well.

2. THE SENSITIVITY ANALYSIS CONTEXT

The Sensitivity Analysis has been implemented in a parallelizing compiler based on the original Polaris compiler.

In the interest of completeness and clarity of presentation of the **SA** technique we will now present a sketch of the overall organization of our automatic parallelization technique around two major steps.

1. Construction of Dependence Sets through a bottom up traversal of the program (from inner to outer loop). The output of this step is a complex set expression which, if proved empty, can lead to *forall* style parallelization of the analyzed loops. This technique has been described in [27].
2. Sensitivity Analysis. It is a recursive distribution of the independence condition $DS == \phi$ over the terms of the dependence set and its transformation into an equivalent logical equation set for which the dependence set must be empty. The output of this step is a disjunction (OR) of *predicates* (conditions) for which the dependence set is empty and thus allows loop parallelization. This technique reduces the overhead of the possible run-time phase of parallelization and will be presented in detail in Section 3.

2.1 Dependence Set Construction

Loop-level parallelization requires the analysis of all memory references that could cause loop carried data dependences. The analysis of memory references over large, interprocedural program contexts requires the use of *summary sets*, i.e., symbolic descriptions of sets of memory references (or locations) over arbitrarily large program blocks.

Data dependence analysis requires information on the relative order of *write* memory references with respect to all other references. This information can be represented efficiently using three types of summary sets [12]: Read Only (RO), Write First (WF) and Read Write (RW). They represent the specific data flow information needed for dependence analysis. 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. The RO, WF, and RW summary sets are computed in a bottom-up traversal of the program. They are individually aggregated (summarized) to the next (loop) level and then, recursively, to the program level. When two successive statements may reference the same memory locations, the corresponding summary sets are updated to reflect their order. For instance, a RO region S_1 followed by a WF in region S_2 results in: RO for region $S_1 - S_2$, RW for region $S_1 \cap S_2$ and WF for region $S_2 - S_1$.

The Dependence Sets (DS) are computed for each loop nest from the memory summary sets (RW, WF, RO) using set algebra operations based on the known dependence relations. It is important to note that the memory reference order across iterations of a loop (dependence direction) is taken into consideration when building the DS set for each

```
1 Do i = 1, n
2   s = 0
3   Do j = 1, m
4     s = s + A(i + p(j))
5   EndDo
6   A(i) = s
7 EndDo
```

(a)

$W = [1:n]$
$R = \cup_{j=1}^{m}[p(j)+1 : p(j)+n]$
$Independent \Leftrightarrow W \cap R = \emptyset$

(b)

```
Do j = 1, m
  R = R ∪ R_i
EndDo
DS = R ∩ W
indep = (DS == ∅)
```

(c)

$indep = \bigwedge_{j=1}^{m}(n < p(j)+1)$

(d)

Figure 1: Run time parallelization. (a) Code sample. (b) Independence compile-time conditions. (c) Evaluation of sets at run time. (d) Sensitivity Analysis test.

loop nest. Further order information for analysis of the enclosing nesting levels is not preserved. However this information is sufficient to determine whether loops are parallel or not and provide a scalable analysis algorithm up to program level.

The resulting DS set represents the possible loop-carried dependences which we would like to disprove. If enough information is available statically and when symbolic analysis is powerful enough the DS set will be conclusively proved either empty or non-empty, i.e., the loop will be proved parallel or not.

Let us consider the code example in Fig. 1. The outer loop (line 1) is independent (and thus parallelizable) if, across all iterations, the *read* references (line 4) do not overlap with the *write* references (line 6) across iterations. We can thus prove independence by showing that the set of all read memory locations (R) is disjoint from the set of all written memory locations (W) across the entire loop.

This problem cannot be solved statically because it depends on the input values n, m and the subscript array p. In [27], the proposed solution is to evaluate at run time the set $W \cap R$ (c). However, this approach incurs significant overhead. Each of the m unions involves coalescing, interleaving and contiguous aggregation checks [12]. When the actual values of $p(j)$ are not a linear function of j, these checks result in $\Theta(m^2)$ comparisons, because each new item in the union is compared against all the previous ones to see if they form a linear combination (contiguous aggregation check). Moreover, the constant factor can be fairly large.

2.2 Sensitivity Analysis

Let us now reconsider the example in Fig. 1. and show how our approach can prove the intersection of the W and R sets empty without ever computing it. The general idea is to recursively distribute the $DS == \phi$ question over its terms and extract the conditions for which it is true (optimistic approach) or for which it is false (pessimistic approach). **SA** derives at compile time symbolic conditions under which the intersection is empty. From the independence equation:
$[1:n] \cap \cup_{j=1}^{m}[p(j)+1 : p(j)+n] = \emptyset$.
After distributing the intersection operator, we have
$\cup_{j=1}^{m}([1:n] \cap [p(j)+1 : p(j)+n]) = \emptyset$.
A union is empty iff each of its members is empty, i.e.,
$\bigwedge_{j=1}^{m}([1:n] \cap [p(j)+1 : p(j)+n] = \emptyset)$.
Proving the intersection of two intervals empty reduces to a bound check and a GCD test. The final result of the static analysis will be
$\bigwedge_{j=1}^{m}(n < p(j)+1)$. (For simplicity, this example assumes that p(j) are positive.)
This test still has to be evaluated at run time, but it consists of just $\Theta(m)$ very light weight operations.

```
1 Do i = 1, n
2   s = 0
3   Do j = 1, m
4     s = s + A(i + p(j))
5   EndDo
6   If (save)
7     A(i) = s
8   EndIf
9 EndDo
```

(a)

$W = save\#[1:n]$
$R = \cup_{j=1}^{m}[p(j)+1 : p(j)+n]$
$Independent \Leftrightarrow W \cap R = \emptyset$

(b)

```
Do j = 1, m
  R = R ∪ R_i
EndDo
DS = R ∩ W
indep = (DS == ∅)
```

(c)

```
If !save
  indep = true
Else
  indep = ∧_{j=1}^{m} (n < p(j) + 1)
```

(d)

Figure 2: Run time parallelization. a) Code sample. (b) Compile-time independence conditions. (c) Evaluation of sets at run time. (d) Run time test extracted by Sensitivity Analysis. Notation: $x\#R$ represents a set R of memory references predicated by condition x.

Let us further complicate our example and consider the code in Fig. 2. It differs from the one in Fig. 1 in that the *write* operations are guarded by a loop invariant predicate, *save*. The approach in [27] would be to evaluate $W \cap R$, and check whether it is empty, which would result in high overhead. The complex union of all $[p(j)+1 : p(j)+n]$ sets would still be performed even though *save* may be false and thus W may be empty, so the intersection $W \cap R$ will be empty regardless of the form of R.

In contrast, after a similar derivation to that shown for the example in Fig. 1, **SA** obtains the final independence condition $\overline{save} \vee \bigwedge_{j=1}^{m}(n < p(j)+1)$, which will add (over the previous example) the sufficient condition $\overline{save}$. If at runtime this condition becomes true, then no further dynamic tests are needed thus making parallelization dependent on a simple scalar comparison.

Let us recall the steps taken to solve these problems.

1. We represent the potential dependences of a loop, i.e., the dependence set (DS) as a predicated memory reference set. We are trying to answer the question whether DS is empty (i.e., the loop is independent).

2. We apply a sequence of transformations that convert problem $DS == \phi$ into an equivalent, but (hopefully) simplified, logical expression that can be evaluated efficiently at run time. The transformations are applied in a recursive descent on the tree representation of the dependence set and are based on set algebra semantics.

3. SENSITIVITY ANALYSIS

3.1 Dependence Set Representation

In order to aggregate memory references in a scalable manner and generate the dependence sets we use the USR *Uniform Set of References* representation, initially introduced as RT_LMAD in [27]. In the interest of completeness and clarity we will summarize the properties of the USRs.

*USR*s can represent the aggregation of scalar and array memory references at any hierarchical (interprocedural) level (on the loop and subprogram call graph) as well as control flow (predicates). 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 example, for the loop in Fig. 1, the *Write* pattern on array A is a USR which takes the simple form of an LMAD, [1:n], while the *Read* pattern is the USR with the form $\cup_{j=1}^{m}[p(j)+1 : p(j)+n]$. The reference sets in Fig. 2 are more complex. For the *Write*, the USR is save # [1:n] (where the symbol # indicates a predicated execution). For the *Read*, the USR is $\cup_{j=1}^{m}[p(j)+1 : p(j)+n]$. When memory references are expressed as linear functions, USRs consist of a single leaf, i.e., a list of LMADs. Internally, the compiler will store the USRs in a tree-data structure. When the analysis process encounters a non-linear reference pattern or when it performs an operation (such as set difference) whose result cannot be represented as a list of LMADs, internal nodes are added to record accurately the operations that could not be performed. In Figs. 1 and 2 the dependence set DS is represented by the USR ($W \cap R$). The use of the $\cap$ operator shows that the intersection could not be performed statically (otherwise it would have been evaluated). The USR representation has two important characteristics: (a) it is closed with respect to all the operations needed for dependence analysis, unlike linear representations, which often have to resort to approximation, and (b) it represents memory references in an aggregated form, thus making them scale to program level.

These properties allow us to transform the dependence equations into a new representation which enables the Sensitivity Analysis.

3.2 The Sensitivity Graph (SG)

In order to inexpensively evaluate at run-time if a loop is parallel or not we need to generate a persistent data structure that can transfer the result of static analysis into executable code. To this end we define and use the *Sensitivity Graph* (**SG**), an analytical, symbolic representation of a boolean expression. SGs are extracted automatically from dependence equations $DS = \emptyset$ that cannot be solved statically where DS is represented as a USR. *SGs 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, evaluated at run time and used to choose between sequential and parallel code versions.

The SG is an arbitrary logical expression such as $(\overline{save}) \vee \bigwedge_{j=1}^{m}(n < p(j)+1)$ from Fig. 2. It is composed (recursively) of simpler SG's: A simple one $\overline{save}$ and a complex one $\bigwedge_{j=1}^{m}(n < p(j)+1)$, which may be any arbitrary program slice that produces a boolean value. SGs are represented by trees having logical expressions as leaves and operators as internal nodes. The SG 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 SGs relatively easy to associate with sections in the original program, thus making it easier for compiler writers to program and understand the analysis process.

$$
\begin{aligned}
\Sigma = \{&\wedge, \vee, \neg, (,), \wedge_{i=1}^{N}, \vee_{i=1}^{N}, \bowtie, LogicalExpression, Recurrence,\\
&Call\ Site, Library\ routine, Reference\ based\ test\}\\
N = \{&SG\},\ S = SG\\
P = \{&SG \rightarrow LogicalExpression|(SG)\\
&SG \rightarrow SG \wedge SG\\
&SG \rightarrow SG \vee SG\\
&SG \rightarrow \neg SG\\
&SG \rightarrow \wedge_{Recurrence} SG\\
&SG \rightarrow \vee_{Recurrence} SG\\
&SG \rightarrow SG \bowtie Call\ Site\\
&SG \rightarrow Library\ routine\\
&SG \rightarrow Reference\ based\ test\}
\end{aligned}
$$

Figure 3: SG formal definition. $\wedge$, $\vee$, $\neg$ are the elementary logical operators *and*, *or*, *not*. $\wedge_{i=1}^{n} SG(i)$ holds true if and only if each of $SG(i)$ holds true, $i = \overline{1,n}$. $SG(formals) \bowtie Call\ Site$ represents the instantiation of a generic SG at a given call site.

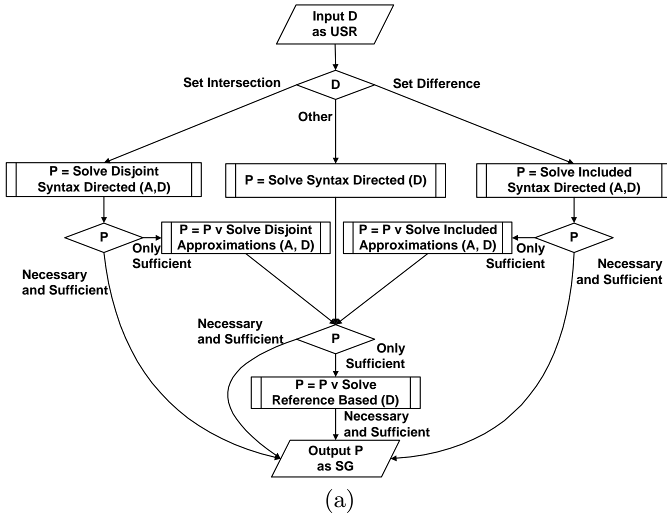
SGs are expressive enough to represent any possible dependence question, and simple enough to be quickly evaluated dynamically. The grammar in Fig. 3 defines SGs formally. They rely mostly on simple, logical operations and have a direct mapping to executable code. In addition to classic $\wedge$, $\vee$, and $\neg$ operators, SGs can also express conjunction ($\wedge_{i=1}^{N}$) and disjunction ($\vee_{i=1}^{N}$) 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.

3.3 Symbolic Analysis Algorithms

3.3.1 Syntax Directed Predicate Extraction

After resolving all statically analyzable dependence questions we are left with a Dependence Set (DS), represented as a 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. 4. This algorithm extracts a set of conditions, represented as a SG, which, when evaluated dynamically, returns *true* if and only if the dependence set is empty. In some cases, such as those in Figs. 1, 2, the algorithm can extract simple necessary and sufficient conditions. In other cases, the algorithm extracts a sequence of sufficient predicates in increasing order of complexity. The last predicate is always necessary and sufficient, which means that our run-time test is *always necessary and suffient*, thus *always accurate*. This is illustrated in Fig. 4 by the fact that the final node (the extracted SG) can only be reached on an edge labeled *necessary and sufficient*.

Algorithm *Solve* extracts the SG from the dependence set by recursively descending its USR tree (recall that we represent USRs as trees) 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, $A \cup B = \emptyset \Leftrightarrow A = \emptyset \wedge B = \emptyset$.



(a)

```
Algorithm Solve Syntax Directed
  Input:           D as USR
  Output:          P as SG s.t. P ⇒ A = ∅
Case D of:
  LMADs:           P = HasEmptyDimension(LMADs)
  A ∪ B:           P = Solve(A = ∅) ∧ Solve(B = ∅)
  q#A:             P = q̄ ∨ Solve(A = ∅)
  ∪_{i=1}^{n}(A_i): P = ∧_{i=1}^{n} Solve(A_i)
  A ⋈ Call Site:   P = Solve(A = ∅) ⋈ Call Site
```

```
Algorithm Solve Disjoint Approximations
  Input:           A, D as USRs
  Output:          P as SG s.t. P ⇒ (A ∩ D = ∅)
(cond_A, ⌈A⌉) = a cond. LMAD overestimate of A
(cond_D, ⌈D⌉) = a cond. LMAD overestimate of D
P = cond_A ∧ cond_D ∧ SolveDisjointLMADs(⌈A⌉, ⌈D⌉)
```

(b)

Figure 4: (a) **Algorithm Solve**: Extraction of a sufficient run time test as a SG from a dependence equation $D = \emptyset$. We accumulate SGs in increasing order of complexity when the partial solutions are sufficient but not necessary, using the logical or operator $\vee$. (b) Some representative subalgorithms; the other algorithms are defined similarly.

```
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
```

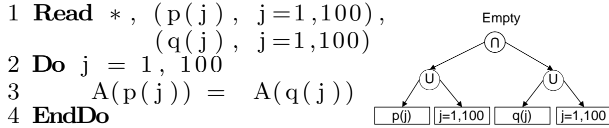


Figure 5: Worst case scenario: in general, no test can solve this problem faster than the reference-by-reference LRPD test. The subtrees rooted at the $\cup$ nodes represent $\cup_{j=1}^{100} p(j)$ and $\cup_{j=1}^{100} q(j)$, respectively.

Our current implementation optimistically assumes independence and extracts sufficient *independence* conditions to prove that. Similarly, we could pessimistically assume dependence and extract sufficient *dependence* conditions to prove that. Inexpensive pessimistic conditions could be used at run time to flag the sequential loops quickly and thus avoid 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 (not parallel). The dynamic evaluation of these tests will then ensure an exact answer but will cost a higher runtime overhead, proportional to the dynamic reference count of the Dependence Set we started from. Fig. 5 presents such a case where a simple independence condition cannot be extracted, in which case the algorithm falls back to reference-by-reference LRPD.

The recursive descent approach is more complex for set intersections and differences than 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 to 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 cases when the equation cannot be further divided, such as $A \cap B \cap C = \emptyset$.

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.

3.3.2 Extracting SGs from USR Approximations

```
     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) Then
     5       W(i) = ...
(a)  6     EndIf
     7   EndDo
     8   Do i = 1, len(j)
     9     ... = W(i)
    10   EndDo
    11 EndDo
```

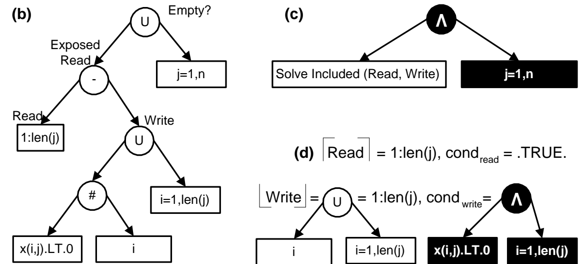


Figure 6: Extraction of an independence predicate using approximation. The subtrees rooted at the $\cup$ and $\wedge$ nodes represent $\cup_{i=1}^{len(j)} i$ and $\wedge_{i=1}^{len(j)} x(i,j) < 0$.

In the example in Fig. 6, array W could be proved privatizable (e.g., a local variable that could be renamed) 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 SG 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. 6, 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 $\wedge_{i=1}^{len(j)} x(i,j) < 0$ (by choosing the value of the conditionals so that $\lfloor write \rfloor$ gets maximized). The approximation process is invoked by algorithms *Solve Disjoint Approximations* and *Solve Included Approximations* as shown in Fig. 4.

We extract tight conditional approximations of USRs as pairs (SG, list of LMADs) by making optimistic assumptions about the gates within the USR. For instance, when computing the underestimate of a gated USR, we optimistically assume that the gate is taken (in order to tighten the underestimate, i.e., to be as large as possible). We then continue the analysis based on this assumption, which takes the role of a sensitivity graph of all subsequent transformations, and which will be validated later, possibly at run time. These optimistic approximations are computed through a recursive descent on the given USR during which gates are extracted from the USR and inserted into the sensitivity graph. The remainder of the USR becomes a list of LMADs.

Although approximation with underestimates and overestimates has been presented before by [5, 12], our technique is more aggressive because it can afford to make crucial optimistic assumptions. In the example in Fig. 6, their techniques must assume conservatively that the gate could be false, and thus the underestimate becomes the empty set, and the privatization problem would not be solved. However, in our case using **SA**, the underestimate of *write* is maximized optimistically and ends up covering the overestimate of the *read*.

Extracting SGs 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.

3.4 Monotonicity and Disjoint Intervals

Consider the dependence problem on array A in the example in Fig 7. 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 $\lceil D_i \rceil = [lower_i : upper_i]$ 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 $O(n)$ sufficient but not necessary version of the test verifies whether the intervals already form a monotonic sequence.

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

Figure 7: Example where a sorting based test is more efficient than applying the *Solve* algorithm.

```
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 8: Automatic Parallelization Algorithm

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 for monotonicity are generated whenever the per-iteration reference set can be bounded by a symbolic interval. We have obtained an efficient 2-dimensional monotonicity test for loop *TRFD/INTGRL_do140* from the *PERFECT* benchmark suite. While a reference test costs $O(n^4)$, the aggregated monotonicity test costs only $O(n)$.

4. IMPLEMENTATION AND COMPLEXITY

4.1 Implementation of an Automatic Parallelizer

We have implemented Sensitivity Analysis within an automatic parallelizer in the Polaris [3] research compiler. Fig. 8 shows the compile time analysis organization. First, individual array references are aggregated to loop level as USRs. Then, as presented in [12, 27], dependence sets are computed

Code	Size	Time	USRs	SGs	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	sr	–
MDG	1,237	15	8,085	744	$6.7 * 10^0$
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	sr	–
SWIM	435	7	1,785	sr	–
WUPWISE	2,184	45	4,710	60	–
HYDRO2D	4,461	33	5,911	11	–
MATRIX300	439	3	1,458	sr	–
MDLJDP2	4,172	18	6,928	444	–
NASA7	1,204	48	8,545	547	$3.0 * 10^6$
ORA	373	7	2,562	sr	–
SWM256	487	8	1,520	sr	–
TOMCATV	194	5	1,056	32	–
BWAVES	918	716	10,617	sr	–

(a) (b)

Table 1: (a) Compile-time analysis statistics (seconds). Column 4 and 5 show the total number of USR and SG nodes created (operator or leaves). Entries denoted by "sr" indicate the parallelization was statically decided, i.e., no run-time test was required. (b) Run time test dynamic overhead reduction through **SA**: ratio between the number of actual memory references and the number of SG operations performed at run time. Entries denoted by "–" indicate no run-time test was performed, either because it was resolved during compilation or it was removed during a post-processing optimization phase.

and resolved statically at each loop level. Some dependences are removed using known techniques such as privatization (renaming), reduction and pushback recognition [28].

Every remaining dependence problem is then formulated as a Sensitivity Analysis problem with input $DS == \emptyset$, where DS is the USR that describes the set of all dependent memory locations. **SA**, in its recursive descent on the DS can call, on demand, other techniques, e.g., simple logic inference, and yields three possible answers: (a) provable empty set implying the loop is parallel (SG with a constant value of *true*), (b) provable non-empty set implying the loop is sequential (SG with a constant value of *false*), and (c) undecided (SG with a dynamic value). In this last case we generate a parallel loop version guarded by a dynamic predicate whose value is determined by the run time evaluation of the extracted SG.

4.2 Complexity of Compile Time Analysis

The complexity of the memory reference aggregation process has been shown to be at most quadratic in 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 has complexity at most linear in the size of the given USR.

Table 1 presents compilation statistics. The number of USR and SG nodes is relatively small. On average, a USR node occupies about 3 KB, while a SG 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].

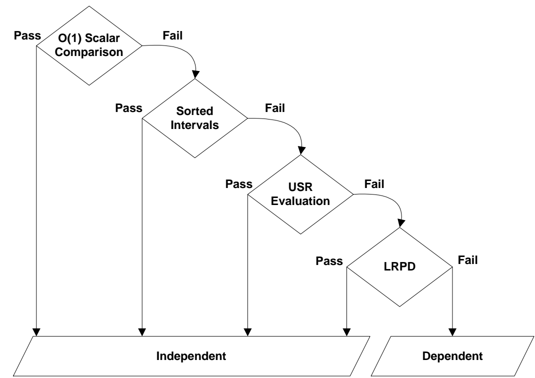


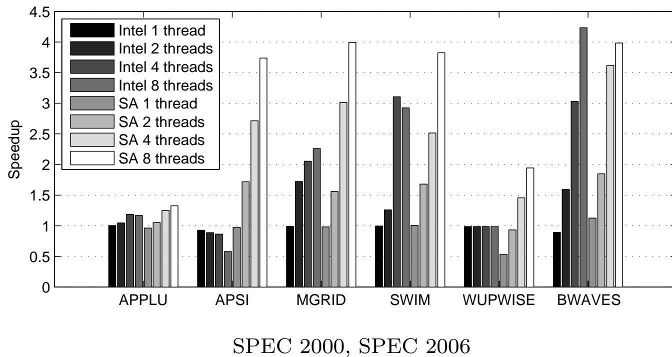
Figure 9: Cascade of sufficient run time tests in order of increasing complexity.

4.3 Complexity of Run-Time SG Evaluation

SGs 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 analysis, e.g., the LRPD test [26]. We extract, for each dependence equation, a **cascade of tests**, (Fig. 9), i.e., a disjunction of tests sorted in increasing order of their estimated complexity. They range from $O(1)$ tests as the one in Fig. 1, $save == true$, to $O(n)$ dynamic reference instrumentation as is the case in Fig. 5. For some extreme cases, when indirection is used, the **SA** does not yield an inexpensive test so we generate code for reference based (enumeration of all references) parallelization (LRPD test [26]). The LRPD has overhead proportional to the dynamic reference count, but is optimal for cases where equation inversion are not possible (Fig. 5), and is always applicable, precise, and has a 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, SG 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 in which all defined variables are scalar). The actual test code generation consists of a syntax-based translation from the SG grammar to Fortran.

We apply loop invariant hoisting to USRs and SGs 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 our already presented algorithms. This is achieved by representing the exact referenced memory regions of the subscript array as USRs themselves, and thus identifying the exact subregion of the subscript array that affects the shape or size of the



SPEC 2000, SPEC 2006

Figure 11: Speedups relative to the sequential version of automatically parallelized benchmark applications on 1, 2, and 4 processors on a SUN quad socket, AMD dual core system. CT = speedups obtained using only compile-time methods. The applications are from the SPEC2000 and SPEC2006 benchmark suites.

memory pattern on the host array. An interesting problem arises when a more expensive sufficient test can be hoisted out of a loop, but a less expensive one is loop variant. We simply hoist tests as far as possible.

5. EXPERIMENTAL RESULTS

The experimental evaluation presented in the following sections will show that Sensitivity Analysis(a) extracts a very high degree of parallelism and, often, all the available parallelism, from a large number of applications, (b) 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.

We compare our results against the Intel Ifort parallelizing compiler v9.1 and show that in many cases we can uncover more coarse grain parallelism.

In this paper 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 is to detect parallel loops, a step which preconditiones any further optimizations.

5.1 Experimental Setup

We ran our automatic parallelizer on a set of 22 programs from the PERFECT and various 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 has used a very simple cost model. When static analysis found a loop nest parallel we have selected the outer one for parallel execution. For those loops needing run-time analysis, we have profile information, i.e., percentage of the sequential execution time. The automatic selection of parallelization candidates based on some more sophisticated performance model is beyond the scope of this paper and has been already employed by commercial compilers with some degree of sophistication.

The experiments were run on two parallel systems: A dual core (Core Duo) Intel based Lenovo Thinkpad laptop and a Sun quad socket, dual core AMD Opteron system.

The reference times for all runs are those of the original benchmark codes running sequentially on the machines and compiled with the Intel Ifort 9.1.036 compiler with options `-O3 -ipo -xP` for Intel chip based system and `-O3 -ipo` for the AMD based system. All codes compiled by us have then used the Intel Fortran Compiler version 9.1.036 backend with compilation option `-O3 -ipo -openmp`. Our reference parallelization performance is that of the Intel compiler with options `-O3 -ipo -parallel`, with `-xP` only for the Intel based dual core system.

5.2 Speedups: Polaris–SA and Intel Ifort 9.1

Figs. 10 and 11 present full application speedups, measured by dividing the sequential execution time of the whole application by its parallel execution time including the run-time overhead, if any.

Fig. 10 shows the speedups obtained for some PERFECT and older SPEC 89 and SPEC 92 programs on a modern Intel dual core based Lenovo laptop. We have not scaled up these measurements to more processors because the data sets are small and thus results cannot scale. However, we have obtained an average speedup of 1.5 which given the small data sets is quite reasonable. When comparing with the performance of the Intel compiler we can remark that (a) our results are always better for multiple cores and (b) our results on one dual core are sometimes lagging. The explanation for our better speedup is that we manage to uncover more coarse grain loops than Intel. The slowdowns on a single core are due to several factors. First, Intel will use loop unroll-and-jam freely when compiling with the `-parallel` switch (when it uses its own auto parallelizer) However, when we pass our code with OpenMP directives through it (-openmp switch) the Ifort does less loop unrolling. This fact can be observed from the Ifort optimization report. We believe that the OpenMP directives may also inhibit other, unreported optimizations. The Intel compiler group has confirmed our finding that loop unrolling is not yet totally compatible with OpenMP parallelization directives. The second reason that our single core version (parallelized code running on 1 core) is sometimes slower than the sequential execution is the use of run-time tests. They represent a small overhead (as we shall see later). The only time when this is actually “visible” is in the case of ADM which uses an inspector loop which cannot be amortized sufficiently through reuse. NASA7 (partially) suffers from lack of memory locality in their time consuming FFT loop nests. Several loops in TOMCATV could not be parallelized at the outermost level resulting in low granularity and limited speedup despite large parallelization coverage. For FLO52 we could not obtain a valid result for the Ifort parallelized version in time for this publication.

Fig. 11 shows the speedups obtained for SPEC 2000 and 2006 programs which have a larger data set size and thus a longer running time. We have measured their execution times on up to four cores in a dual core based AMD system (made by SUN). The results show clearly that we can find more coarse grain parallelism than Ifort. In particular, WUPWISE shows good scalability but starts with a slowdown for the reasons previously mentioned. Ifort will unroll twice as many loops when its own auto-parallelizer is used than when using OpenMP directives generated by our analysis. We believe that a better integration of OpenMP with Ifort could yield much better absolute speedups. The graph for APSI indicates our parallelization of outer loops. Ifort is in fact getting a slowdown because they parallelize inner loops (in RADF and RADB routines) while we find

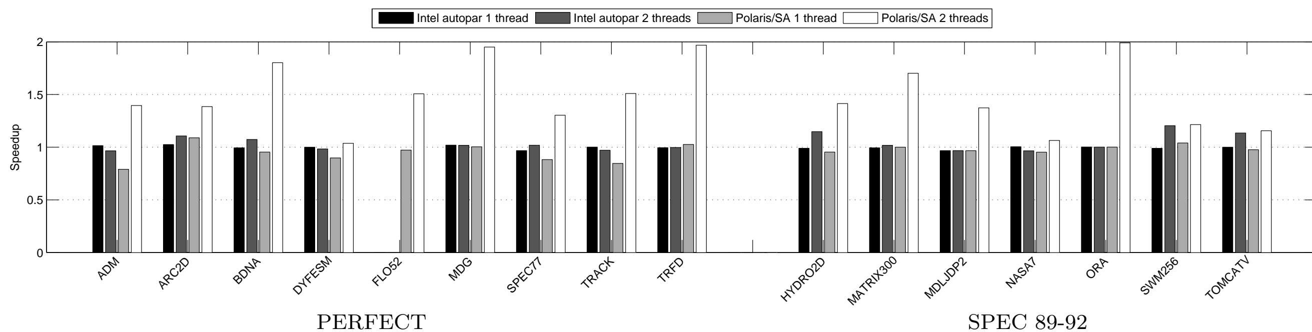


Figure 10: Speedups relative to the sequential version of automatically parallelized benchmark applications on 1, 2 processors of a Core Duo Intel system. CT = speedups obtained using only compile-time methods. The applications are from the PERFECT and previous SPEC 89-92 benchmark suites respectively.

the outer loops parallel. It seems that Ifort's cost model is not accurate in this particular instance. APPLU is not well covered by Ifort. Our coverage is better but does not result in much better speedup. It is important to note that Ifort may have a better cost model for selecting which loops to parallelize and that we do not employ any locality enhancement transformations. We have also started applying our technique to the recent SPEC 2006 programs. So far we report only speedups for BWAVES, slightly less than Ifort's. We will continue to expand our experiments to the rest of the F77 SPEC 2006 codes.

5.3 Parallelism Coverage

We define as *coverage* the ratio between the sequential time of the parallelized loops and the total sequential execution time of a program. This measure is important because it gives a certain measure of potential scalability of the parallelization. If we can cover all the code with coarse grain parallelism, then chances are we can scale the performance (speedup) to many processors. If we cannot, then performance will be subjected to Amdahl's law (bottleneck). In Fig. 12 we present the coverage obtained by Ifort and our compiler. It is important to note that for Polaris we report all loops that can be parallelized while for Intel's Ifort we report only what its cost model decided to parallelize. Thus these results do not show the full capability of Intel's compiler. However, these coverage results can be somewhat correlated with the obtained speedup. Low coverage results in poor speedup on 8 cores. However, the reverse is not always true. Good coverage needs to be of good quality, i.e., coarse grain parallelism. While we will attempt in the future to define a good measure for coverage granularity, in this paper we will point to our obtained speedups. A good example is TOMCATV where we have excellent coverage when compared to Ifort but do not obtain any better results. In fact, Ifort is most likely using its cost model to disallow parallelization of small loops.

It is important to note that we have obtained for most codes a coverage of over 90% and for many we 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 SA because it does not quantify the fact that we detect coarse grain parallelism (outer loops) as well as fine grain (inner loops).

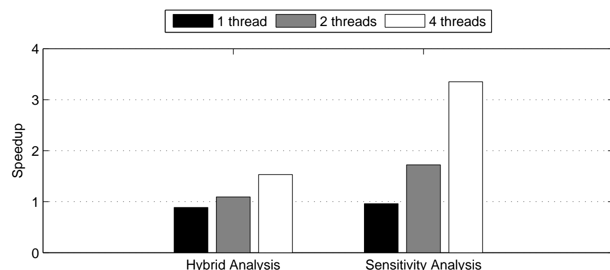


Figure 13: Comparison of speedup on benchmark application MDG using three types of run time tests: evaluation of LMADs at run time, reference instrumentation LRPD, and simple conditions extracted by SA.

5.4 Evaluation of Run Time Tests

Overall, we generated 42 tests based on the evaluation of elementary conditional expressions, 30 monotonicity tests, and 81 tests 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 SG 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 us relative to reference-by-reference (LRPD) tests as being at least four orders of magnitude in 6 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).

In Fig. 13 we give an example on how the various parallelization techniques compare against SA. We compiled with different versions of the compiler: using a run-time evaluation of the USRs ([27]), and SA. Clearly, the light weight run-time tests generated by the SA technique offer the best absolute speedups due to their low overhead.

In conclusion, we believe that the consistent solid performance results across a large number of standard benchmark applications prove our claims on the effectiveness of SA.

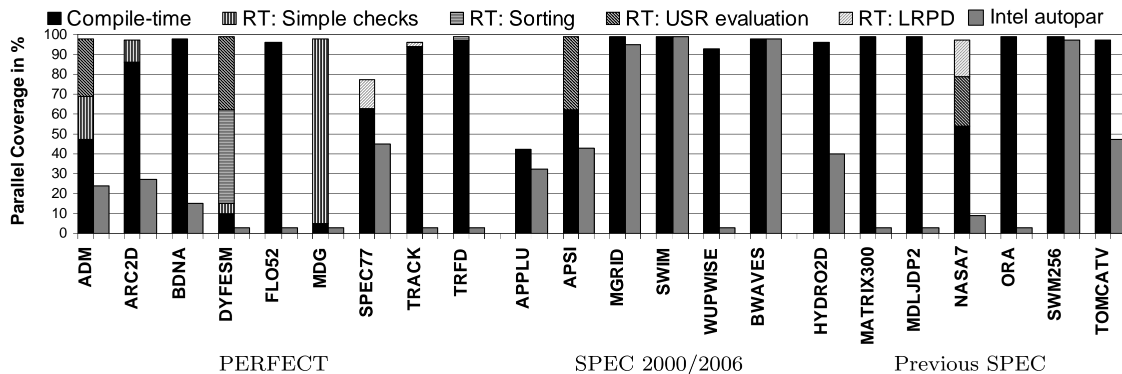


Figure 12: Automatic parallelization coverage as percentage of the sequential execution time of Polaris with SA and Intel Ifort 9.1 compiler. Each bar shows the contribution of each employed technique to parallelization. Ifort auto parallelizer does not specify employed techniques.

6. RELATED WORK

Traditionally, automatic parallelization relied on representing memory reference sets as linear systems [6, 31, 2, 8, 17, 14, 22, 21]. These systems did not work with nonlinear reference patterns, such as those caused by indirection arrays or control predicates containing floating point values. *In essence, classic compile-time methods based on linear algebra are not always applicable.*

Pattern recognition and index property analysis were proposed as solutions for nonlinear reference patterns [16]. Their applicability is limited to a small number of cases. 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 cannot deal with strided patterns. *Pattern-matching based extensions to handle nonlinear cases have limited applicability as well.* In contrast, Sensitivity Analysis uses 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. When a full proof is not possible, we expose just the remaining conditions as SGs.

General run time data dependence tests were proposed to solve dependence problems that did not have compile-time solutions [1, 29, 13, 15, 26, 27, 25]. [1, 27] used interprocedural aggregation and redundancy elimination analysis to simplify the formulas describing sets of memory references corresponding to a data flow or dependence relation. However, their run time tests still had to manage sets of references. In [34] the authors start from an exhaustive run-time test (the LRPD test) and focus on its overhead reduction. They obtain improvements 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. *Dynamic analysis methods based on the construction and analysis of memory reference sets at run time are expensive.*

One of the first forms of predicate extraction was conditional vectorization [32]. Vectorizing compilers like KAP introduced simple run-time methods to decide when it was profitable and/or correct to vectorize. The technique [32] has remained limited to solving simple cases. In [19, 20, 18] simple conditions are synthesized 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. However, none of these methods can handle loop nests in which memory accesses are through indirection arrays or controlled by arrays of predicates. *All these methods are limited to scalar nonlinear terms.* [24] showed how predicates can be extracted by simplifying Presburger formulas with uninterpreted function symbols. Their purpose was not to extract run-time optimization predicates, but to extract optimization predicates that could be validated at compile-time by an expert programmer, thus we cannot compare quantitatively to their method.

In essence, SA has lower overhead than run-time memory reference analysis methods, and can be applied in more cases than previous predicate extraction methods.

7. CONCLUSION

The **Sensitivity Analysis** (**SA**) 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, the Sensitivity Analysis 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 22 codes and then, without further optimization get very good speedups. This result also shows that automatic parallelization, long in coming, is actually possible and profitable.

8. REFERENCES

[1] G. Agrawal, J. H. Saltz, and R. Das. Interprocedural partial redundancy elimination and its application to distributed memory compilation. In *Proc. of Prog.*

Lang. Design and Impl., pp. 258–269, La Jolla, CA, June 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. In *Proc. of Supercomputing '94*, Washington DC, pp. 528–537, Nov. 1994.

[5] B. Creusillet and F. Irigoin. Exact vs. approximate array region analyses. In *Proc. of Workshop on Lang. and Comp. for Par. Computing*, LNCS, vol. 1239, pp. 86–100, Springer 1996.

[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 Par. Prog.*, 20(1):23–54, 1991.

[8] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *Proc. of Prog. Lang. Design and Impl.*, pp. 15–29, Toronto, ON, June 1991.

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

[10] M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM TOPLAS*, 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 TOPLAS.*, 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. of Supercomputing '94*, Washington DC, Nov. 1994.

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

[15] S. Leung and J. Zahorjan. Improving the performance of runtime parallelization. In *Proc. of Principles and Practice of Par. Prog.*, pp. 83–91, ACM Press, May 1993.

[16] Y. Lin and D. Padua. Analysis of irregular single-indexed array accesses and its application in compiler optimizations. In *Proc. of Int. Conf. on Compiler Constr.*, pp. 202–218, LNCS vol. 1781, 2000.

[17] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *Proc. of Prog. Lang. Design and Impl.*, pp. 1–14, Toronto, ON, June 1991.

[18] S. Moon and M. W. Hall. Evaluation of predicated array data-flow analysis for automatic parallelization. In *Proc. of Principles and Practice of Par. Prog.*, pp. 84–99, ACM Press, 1999.

[19] S. Moon, M. W. Hall, and B. R. Murphy. Predicated array data-flow analysis for run-time parallelization. in *Proc. of Int. Conf. on Supercomputing*, Melbourne, Australia, pp. 212–219, 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 *Proc. of Workshop on Lang., Comp, and Run-time Support for Scalable Systems*, pp. 91–106, London, 1998.

[21] Y. Paek, J. Hoeflinger, and D. Padua. Efficient and precise array access analysis. *ACM TOPLAS*, 24(1):65–109, 2002.

[22] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proc. of Supercomputing '91*, Albuquerque, NM, pp. 4–13, Nov. 1991.

[23] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In *Proc. of Workshop on Lang. and Comp. for Par. Computing*, in LNCS 768, pp. 546–566, Portland, OR, Aug. 1993.

[24] W. Pugh and D. Wonnacott. Nonlinear array dependence analysis. In *Proc. of Workshop on Lang., Comp. and Run-Time Support for Scalable Systems*, Kluwer, Boston 1995.

[25] C. G. Quiñones, *et. al.* Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proc. of Prog. Lang. Design and Impl.*, pp. 269–279, New York, NY, 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 Prog. Lang. Design and Impl.*, pp. 218–232, La Jolla, CA, June 1995.

[27] S. Rus, J. Hoeflinger, and L. Rauchwerger. Hybrid analysis: static & dynamic memory reference analysis. *Int. Journal of Par. 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 *Proc. of Par. Arch. and Comp. Techniques*, pp. 243–254, Antibes, France, Oct. 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 *Proc. of Int. Conf. on Supercomp.*, pp. 106–115, ACM Press, 2004.

[31] M. Wolfe and C.-W. Tseng. The Power test for data dependence. *IEEE TPDS*, 3(5):591–601, Sept. 1992.

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

[33] P. Wu, A. Cohen, and D. Padua. Induction variable analysis without idiom recognition: Beyond monotonicity. In *Proc. of Workshop on Lang. and Comp. for Par. Computing*, pp. 427–441, Cumberland Falls, KY, 2001, LNCS 2624.

[34] H. Yu and L. Rauchwerger. Run-time parallelization overhead reduction techniques. In *Proc. of Int. Conf. on Compiler Construction, Berlin, Germany*, pp. 232–248, Springer LNCS 1781, March 2000.