Sensitivity Analysis for Migrating Programs to Multi-Cores

Silvius Rus, Marinus Pennings, Lawrence Rauchwerger
`silvius.rus@gmail.com, {pennings,rwerger}@cs.tamu.edu`

Technical Report TR06-015
Parasol Lab
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112

December 4, 2006

Abstract

Recently, a technique known as Hybrid Analysis has demonstrated the possibility of automatic coarse grain program parallelization through an integrated combination of static and dynamic analysis techniques. The recent introduction of multi-cores for the mass market has on the one hand exacerbated the need for such a technology and, on the other hand, changed the cost/benefit ratio of parallelization. Multi-Cores have low communication costs but the degree of parallelism is relatively small (¡100 processors for the next 5 years). Therefore hybrid parallelization techniques (static/dynamic analysis) need to keep their dynamic overhead very low in order to benefit multi-core systems.

The Sensitivity Analysis (SA), presented in this paper is a novel technique that can reduce the dynamic overhead of previous Hybrid Analysis technology. Instead of dynamically evaluating the aggregated memory reference sets representing the potential data dependences, SA can frequently extract light weight, sufficient conditions for which these dependence sets are empty. The cost of the run-time evaluation of these conditions, named predicates, is low enough to make parallelization profitable on multi-cores.

We have implemented the SA technology in the Polaris compiler and evaluated its performance on 22 benchmark SPEC and PERFECT codes. We have obtained speedups superior to the Intel Ifort compiler in most cases because we could extract most of the available coarse grained parallelism.

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. A key enabling technology in this domain is parallelizing compilers because they offer the advantage of automation and thus high productivity.

Compiler research has, for the most part, taken 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 progressed in parallel with little interaction for quite some time.

More recently, the Hybrid Analysis (HA) technique [27] has demonstrated a compiler framework that could seamlessly bridge the static and dynamic analysis technology. Briefly stated, Hybrid Analysis recursively aggregates memory references into memory reference descriptor sets. These sets summarize the data flow behavior of the program and are grouped in *write-first* (WF), *read-write* (RW) and *read-only* (RO) reference classes. The aggregation proceeds bottom-up and *data dependence memory reference sets* (**DS**) are computed from the WF, RW, and RO sets using set operations. When the data dependence sets cannot be conclusively evaluated statically the compiler either continues to aggregate them upwards into the program or generates executable code for their dynamic evaluation. Their outcome, selects between the parallel and sequential versions of the code. A worst case scenario arises when memory is referenced using indirection (aka subscripted subscripts). In this case Hybrid Analysis may use the LRPD Test, a speculative parallelization technique described in detail in [26]. In essence the LRPD test optimistically executes a loop in parallel while simultaneously collecting a (partial) trace of its memory references which could not be conclusively analyzed by the compiler. After the parallel execution, the trace is analyzed and if the optimistic parallel execution is found to be invalid (*potential* data dependences are uncovered) the loop is re-executed sequentially. The run-time overhead associated with this technique, while scalable, can be quite costly at times because it is proportional to the the dynamic memory reference count and can potentially lead to slowdowns (when the speculation fails).

To reduce the overhead of dynamic evaluation of the dependence sets produced by Hybrid Analysis we introduce in this paper a novel technique, Sensitivity Analysis (**SA**). Sensitivity Analysis starts where the Hybrid Analysis presented in [27] left off, i.e, after the data dependence sets have been generated (through bottom-up aggregation). These set expressions are, in their final form, the result of an inconclusive static analysis and represent potential dependences. Instead of directly evaluating them at run-time, **SA** finds, statically, the input conditions for which the dependence set is empty. This is achieved by a recursive, top-down, distribution of the "empty set" ($DS == \phi$) equation over the terms of the dependence set by applying known set transformations. At each level, logic inference, known constants and value ranges are used to simplify the resulting expressions. The final result is often a list of simple, input dependent conditions which represent suficient (and sometimes necessary) conditions for the root data dependence set to be empty, i.e., the condition for which parallelization is legal. In other words, the set of obtained *predicates* represents the input sensitivity of the parallelization transformation of the program block under consideration.

Our results indicate that employing **SA** frequently results in lowering the overhead of dynamic parallelization to the point where it becomes profitable on multi-core architectures. In particular, 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 significantly superior to the one obtained by the Intel ifort compiler. On average, we obtained speedups of 1.5 on a dual core and 4 on a quad socket dual core.

1.1 Contribution

In this paper we have built upon previous compiler techniques and developed a novel method that extracts sufficient and light weight conditions (predicates) that can dynamically validate optimized (parallelized) versions of the code. This paper makes the following contributions:

- Develops Sensitivity Analysis, a 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. These conditions are then evaluated dynamically, in order of their complexity.
- Extends the applicability of hybrid analysis to multi-cores by reducing the weight of the run-time overhead to (mostly) insignificant levels.
- A full implementation of this technique in a research compiler (Polaris) and experimental results showing that coarse grained automatic parallelization is possible and essential for multi-cores.

We believe 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 step has been implemented in our parallelizing compiler (based on the original Polaris compiler) as a postpass to the Hybrid Dependence Analysis. Together these two techniques constitute the automatic parallelization capability of the current version of the 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. The evaluation of this dependence set can be done statically (preferably) or dynamically. This technique has been presented in [27] and was shown to obtain fairly good results on an SMP.
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 ouput of this step is a disjunction (OR) of sufficient *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 Generation

Loop-level parallelization requires the analysis of all memory references that could cause loop carried data dependences. To analyze memory references over large, interprocedural program contexts, Hybrid Analysis uses *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. Three types of summary sets are defined to represent this information. 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, recursively 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 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 the 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 $W \cap R$. 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

```
Do i = 1, n
 s = 0
 Do j = 1, m                  W = [1 : n]
   s = s + A(i+p(j))          R = ∪_{j=1}^{m} [p(j) + 1 : p(j) + n]
 EndDo                        Independent ⇔ W ∩ R = ∅
 A(i) = s
EndDo
```

Figure 1: Parallelization with Hybrid Analysis.

constant factor can be fairly large.

2.2 Sensitivity Analysis Reduces Dynamic Overhead

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 can be true (optimistic approach) or for which it cannot be true (pessimistic approach). **SA** will now derive at compile time, symbolic conditions under which the intersection is empty. From the independence equation we get
$[1:n] \cap \cup_{j=1}^{m}[p(j)+1:p(j)+n] = \emptyset$.
After distributing the intersection operator, we have
$\cup_{j=1}^{m}([1:n] \cap [p(j)+1:p(j)+n]) = \emptyset$.
Since 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
$\bigwedge_{j=1}^{m}(n < p(j)+1)$.
This test still has to be evaluated at run time, but it consists of just $\Theta(m)$ very light weight operations.

```
Do i = 1, n
 s = 0
 Do j = 1, m
   s = s + A(i+p(j))          W = save#[1 : n]
 EndDo                        R = ∪_{j=1}^{m} [p(j) + 1 : p(j) + n]
 If (save)                    Independent ⇔ W ∩ R = ∅
  A(i) = s
 EndIf
EndDo
```

Figure 2: Notatation: $x\#Region$ means region referenced predicated by the X.

Let us further complicate our example and consider the code 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 run-time this condition becomes true, then no further dynamic tests are needed thus making parallelization dependent on a simple scalar comparison.

Let us recall the important 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 references 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 which convert this problem ($DS == \phi$) into an equivalent, but (hopefully) simplified, logical expression that can be evaluated efficiently at run time. These transformations are applied in a recursive descent on the tree representation of the dependence set and are based on set algebra semantics.

In the more general case, the relevant sets of references *read* and *write* cannot 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. The implementation of **SA** will thus need an expressive, scalable intermediate representation of the relevant parts of the program. We will now present such a representation and how it can be used by **SA** on arbitrary codes.

3 Sensitivity Analysis

3.1 Memory Reference and Dependence Set Representation

In order to aggregate memory references in a scalable manner and generate the dependence sets we will use the same representation as was presented by the authors of [27], the RT_LMAD. However, we will name it *Uniform Memory Representation* (**USR**) because in the context of Sensitivity Analysis the name RT_LMAD is no longer appropriate. As we have already indicated, we no longer evaluate dependence sets at run-time. In the interest of completeness and clarity we will briefly 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 later transform the dependence equations into a new representation which enables the Sensitivity Analysis.

3.2 Sensitivity Representation: 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 is in fact a an 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.

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

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

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, $i = 1, n$. $SG(formals) \bowtie Call\ Site$ represents the instantiation of a generic SG at a particular call site.

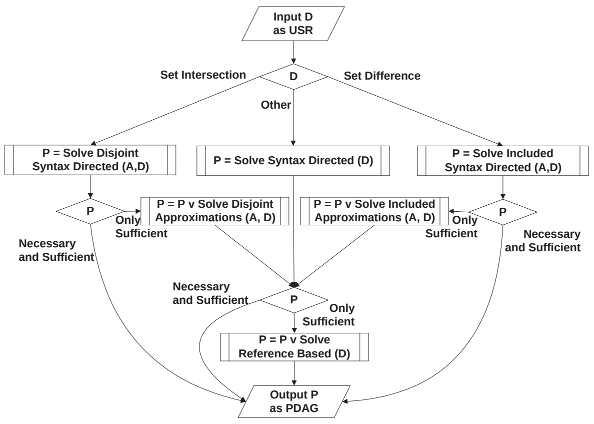
we have formulated the algorithm *Solve* shown in Fig. 4. This algorithm extracts a set of conditions, represented as a SG, which, when evaluated dynamically, returns *true* if and only if the dependence set is empty.

Algorithm *Solve* extracts the SG from the dependence set by recursively descending its USR tree (recollect 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, $A \cup B = \emptyset \Leftrightarrow A = \emptyset \wedge B = \emptyset$.

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 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. 5 presents such a case where a simple independence condition cannot be extracted.

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 USR configurations that cannot be broken up, 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.


Input D as USR
D
Set Intersection
Other
Set Difference
P = Solve Disjoint Syntax Directed (A,D)
P = Solve Syntax Directed (D)
P = Solve Included Syntax Directed (A,D)
P
Only Sufficient
P = P v Solve Disjoint Approximations (A, D)
P = P v Solve Included Approximations (A, D)
Only Sufficient
P
Necessary and Sufficient
Necessary and Sufficient
Necessary and Sufficient
P
Only Sufficient
P = P v Solve Reference Based (D)
Output P as PDAG

(a)

```
Algorithm Solve Syntax Directed
  Input:        $D$ as USR
  Output:       $P$ as SG s.t. $P \Rightarrow A = \emptyset$
Case D of:
  $LMADs$:        $P = HasEmptyDimension(LMADs)$
  $A \cup B$:         $P = Solve(A = \emptyset) \wedge Solve(B = \emptyset)$
  $q \# A$:          $P = \overline{q} \vee Solve(A = \emptyset)$
  $\cup_{i=1}^{n}(A_i)$:     $P = \wedge_{i=1}^{n} Solve(A_i)$
  $A \bowtie Call\ Site$: $P = Solve(A = \emptyset) \bowtie Call\ Site$
```

```
Algorithm Solve Disjoint Approximations
  Input:          $A, D$ as USRs
  Output:         $P$ as SG s.t. $P \Rightarrow (A \cap D = \emptyset)$
$(cond_A, \lceil A \rceil)$ = a conditional LMAD overest. of A
$(cond_D, \lceil D \rceil)$ = a conditional LMAD overest. of D
$P = cond_A \wedge cond_D \wedge SolveDisjointLMADs(\lceil A \rceil, \lceil D \rceil)$
```

(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.

```
Read *, (p(j), j=1,100),
        (q(j), j=1,100)
Do j = 1, 100
    A(p(j)) = A(q(j))
EndDo
```

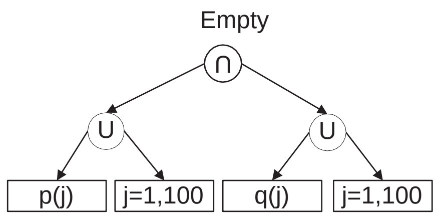


Figure 5: An extreme parallelization: in general, no test can solve this problem faster than the reference-by-reference LRPD test. The subtrees rooted at the $\cup$ nodes represent $\cup_{j=1}^{100} p(j)$ and $\cup_{j=1}^{100} q(j)$, respectively.

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

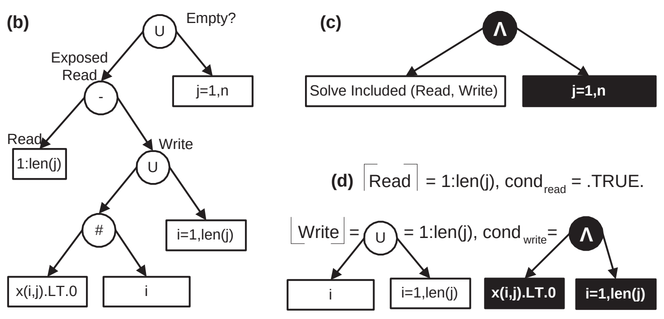


Figure 6: Extraction of an independence predicate using approximation. The subtrees rooted at the ∪ and ∧ nodes represent $\cup_{i=1}^{len(j)} i$ and $\wedge_{i=1}^{len(j)} x(i,j) < 0$.

3.3.2 Extracting SGs from USR Approximations

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 $\wedge_{i=1}^{len(j)} x(i,j) < 0$. 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. From this LMAD list we can extract the SG, a much simpler problem than extracting it from the original USR.

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 Testing Monotonicity and Disjoint Intervals

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

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

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 ($O(n)$) 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 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-by-reference test costs $O(n^4)$, the aggregated monotonicity test costs only $O(n)$.

```
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 8: Automatic Parallelization Algorithm

4 Implementation and Complexity

4.1 Implementation of an Automatic Parallelizer

We have implemented Sensitivity Analysis within an automatic parallelizer based on Hybrid Dependence Analysis 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 and resolved statically at every loop level. Some dependences are removed through using known techniques such as privatization (renaming), reduction recognition and pushback recognition [28]

Every remaining dependence problem is then formulated as 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 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 SG.

4.2 Complexity of Compile Time Analysis

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

Table 1 presents compilation statistics. The number of USR and SG nodes is relatively small. On the 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].

4.3 Complexity of Run-Time SG Evaluation

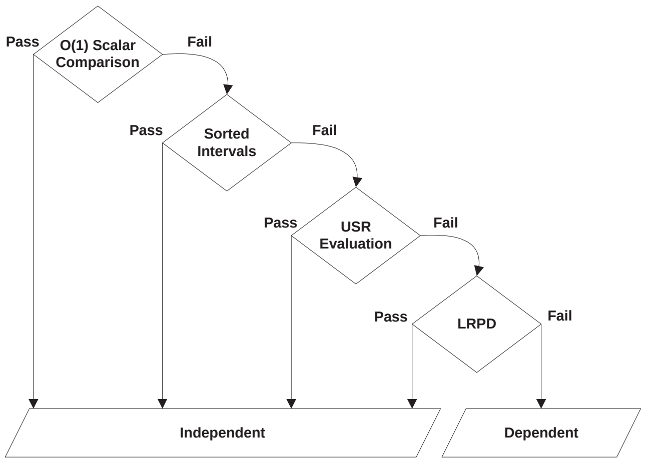


Figure 9: Cascade of sufficient run time tests in order of complexity.

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). ordered by 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 aggregation and equation inversion are not possible (Fig. 5), 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, 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 made of only scalar definitions). 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

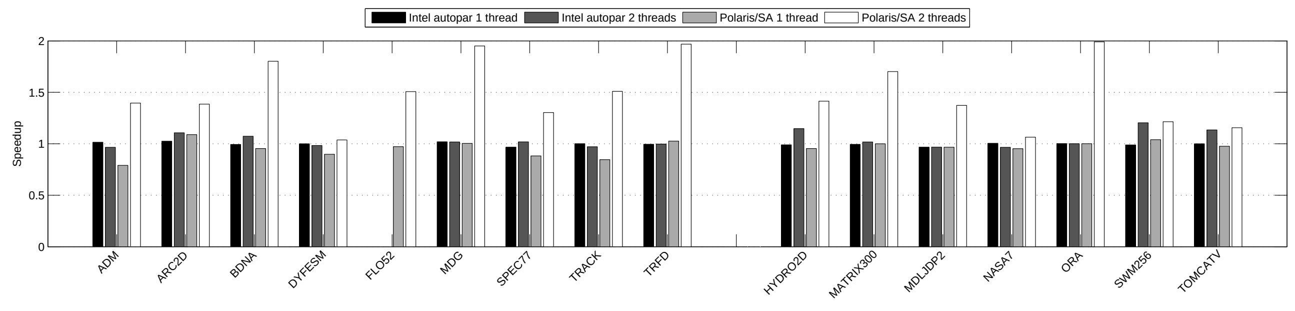
Code	**Lines**	**Time**	**USRs**	**SGs**
ADM	5,791	455	35,249	10,456
ARC2D	3,099	102	13,178	22
BDNA	4,919	36	11,181	156
DYFESM	3,903	38	6,841	756
FLO52	2,508	120	8,371	sr
MDG	1,237	15	8,085	744
SPEC77	4,582	303	75,032	4,733
TRACK	2,523	245	27,790	2,931
TRFD	656	120	1,684	139
APPLU	3,980	56	13,212	34
APSI	7,488	399	36,593	10,800
MGRID	489	108	2,089	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
ORA	373	7	2,562	sr
SWM256	487	8	1,520	sr
TOMCATV	194	5	1,056	32
BWAVES	918	716	10617	sr

(a)

Op Ratio
$1.8 * 10^5$
$1.2 * 10^7$
–
$1.5 * 10^4$
–
$6.7 * 10^0$
$1.0 * 10^0$
$1.0 * 10^0$
$5.6 * 10^4$
–
$1.6 * 10^7$
–
–
–
–
–
–
$3.0 * 10^6$
–
–
–
–

(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.



PERFECT SPEC 89-92

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.

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 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

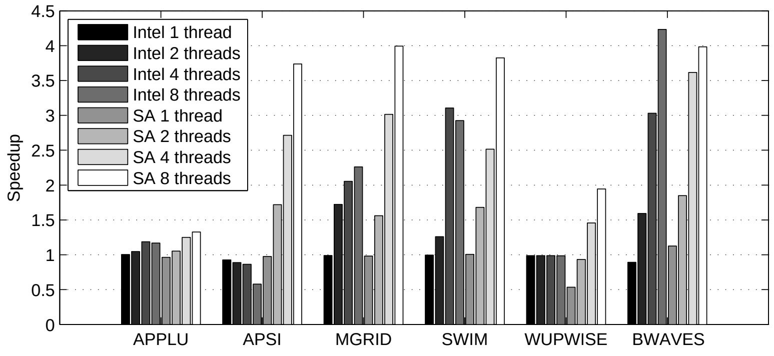
The experimental evaluation presented in the following sections will show that Sensitivity Analysis, when coupled with the existing Hybrid Analysis in Polaris, (a) extracts a very high degree of parallelism and, often, all the available parallelism, from a large number of applications, (b) is applicable 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 in front us is to detect loops parallel, 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



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 SPEC 2006 benchmark suites respectively.

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 152 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 Speedup Obtained with 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 runtime 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 fewer 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 (Intel has assured us that the sonn to be released Ifort v. 10.0 will resolve these issues). 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. We will investigate further the reason.

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 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 processors (4 dual 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 the 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

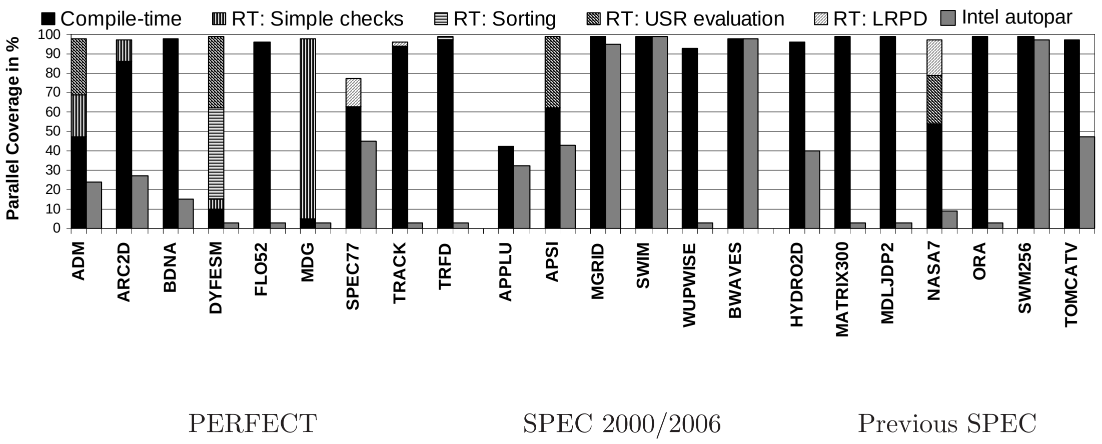


Figure 12: Automatic parallelization coverage as percentage of the sequential execution time of Polaris with SA and Intel Ifort 9.1 compiler. Each bar shows the contribution of each employed technique to parallelization. Ifort auto parallelizer does not specify employed techniques.

its cost model to disallow parallelization of small loops (they are rather simple).

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. As we mentioned, the excellent coverage does not sufficiently do justice to the power of SA because it does not quantify the fact that we can detect course grain parallelism (outer loop level) as well as fine grain level (inner loops).

5.4 Run Time Tests Applicability and Effectiveness

Overall, we generated 42 tests based on the evaluation of elementary conditional expressions, 30 sorting-based tests (for monotonicity verification), 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

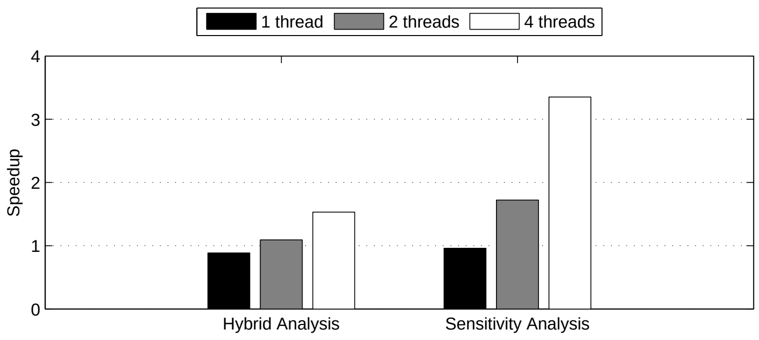


Figure 13: Comparison of speedup on benchmark application MDG using three types of run time tests: evaluation of LMADs at run time [27], reference instrumentation LRPD [26] and simple conditions extracted by SA.

speedups due to their low overhead.

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

6 Related Work

Most previous optimization methods can be divided into major categories: Static and dynamic. As we shall see, several researchers have also tried to combine the two approaches.

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]. Their 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.

Dynamic Dependence Analysis and Parallelization. One of the first forms of dynamic analysis was conditional vectorization [32]. Vectorizing compilers like KAP had introduced simple run time methods to decide when it is profitable and/or correct to vectorize. The technique [32] has remained limited to solving simple cases.

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. In [1] the authors introduce a powerful interprocedural partial redundancy elimination analysis and its application to the detection of array data flow relations on particular control flow paths, which in turn leads to aggressively optimized placement of communication primitives; this is similar conceptually to hoisting USR computation to the data flow locations where their input variables become available [27]. By comparison, **SA** pushes symbolic analysis further and extracts SGs as cascades of sufficient 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.

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. The SGs 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.

In [18, 23, 27] all authors had recognized the need to bridge compile-time and run time analysis. 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. [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.

In [27] the authors propose the evaluation of USRs (named RT_LMAD) at run time followed by comparison to the empty set. They are however not able to extract only the input sensitive entities which need to be tested at run-time, thus causing more overhead than generally necessary. While at the time their results were promising the introduction of multi-cores has reduced some of the profitability of this approach because they generate too much overhead. **SA** tries improves on their technique.

On a similar note we should mention the JIT compiler technology which can fully specialize code at run-time. We are not aware of its specific use in auto-parallelization but we recognize it may have this potential. However the issue of run-time overhead is even more severe than in the previously mentioned approaches which use multi-version, precompiled, partially specialized code blocks.

7 Conclusions

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 – even for multi-cores.

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. Irigoin. Exact vs. approximate array region analyses. In *Workshop on Languages and Compilers for Parallel Computing*, LNCS, 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 Parallel Prog.* , 20(1):23–54, 1991.

[8] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *ACM SIGPLAN PLDI'91*, 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 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. 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 TPDS*, 2(3):342–349, July 1991.

[15] S. Leung and J. Zahorjan. Improving the performance of runtime parallelization. In *ACM SIGPLAN PPOPP*, 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, 2000.

[17] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *ACM SIGPLAN PLDI*, Toronto, Ont., June 1991.

[18] S. Moon and M. W. Hall. Evaluation of predicated array data-flow analysis for automatic parallelization. In *ACM SIGPLAN PPOPP* 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*, 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*, LNCS, London, UK, 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 *Supercomputing '91*, 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 LCPC*, in LNCS 768, Portland, Ore., Aug. 1993.

[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*, Kluwer, 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 PLDI*, 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 PLDI*, La Jolla, CA, 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 *Proc. of PACT*, 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*, 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, USA, 1995.

[33] P. Wu, A. Cohen, and D. Padua. Induction variable analysis without idiom recognition: Beyond monotonicity. In *2001 Workshop on LCPC*, Cumberland Falls, KY, 2001, LNCS 2624.

[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, March 2000.