Implementation of Sensitivity Analysis for Automatic Parallelization

Silvius Rus, Maikel Pennings, and Lawrence Rauchwerger

Parasol Lab, Department of Computer Science, Texas A&M University
rus@google.com,{pennings,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 program behavior is input sensitive. 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. While SA's principles are fairly simple, implementing it in a real compiler and obtaining good experimental results on benchmark codes is a difficult task. In this paper we present some of the most important implementation issues that we had to overcome in order to achieve a fairly successful automatic parallelizer. We present techniques related to validating dependence removing transformations, e.g., privatization or pushback parallelization, and static and dynamic evaluation of complex conditions for loop parallelization. We concern ourselves with multi-version and parallel code generation as well as the use of speculative parallelization when other, less costly options fail. We present a summary table of the contributions of our techniques to the successful parallelization of 22 industry benchmark codes. We also report speedups and parallel coverage of these codes on two multicore based systems and compare them to results obtained by the Ifort compiler.

1 Introduction

1.1 Automatic Parallelization - Current State of the Art

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 must focus, at least as a necessary first step, on discovering which loops can be executed in parallel (ideally as a `doall`). Data dependence analysis techniques as simple as the GCD test [17] and as sophisticated as the Omega test [9] 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 [16]) or after an optimistic (speculative) parallel execution [12]. The complexity of this method is proportional to the number of dynamic references and thus is potentially expensive.

Overall, the 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 [13]. 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 [10] the authors showed some limited examples of how sufficient predicates could be extracted by simplifying Presburger formulas with uninterpreted function symbols. These predicates are returned to the programmer for evaluation (for interactive compilation). Further research [6, 7, 5, 2, 13] showed how to extract simple scalar conditions from relatively simple array data dependence predicates for a limited number of cases.

We have used a similar approach and recently presented Sensitivity Analysis (SA) [14] as a general framework to analyze memory references and used it to extract parallel loops from sequential programs. SA seamlessly bridges static and dynamic analysis of memory references. When the compiler cannot draw definitive conclusions about interesting properties of a memory reference pattern, SA can generate a set of sufficient conditions which, when evaluated, can (in)validate these interesting properties. Examples of such interesting properties are (in)dependent memory references, privatizable references, reductions, etc.

1.2 Automatic Parallelization with Sensitivity Analysis

In [13, 14] we have shown how our compiler using SA is able to extract most available loop level parallelism from various benchmark codes using a mix of advanced static analysis and aggressive optimizations that are validated dynamically with minimal overhead. This has resulted in fairly good speedups. In [13,

14] we have explained with some detail how the overall SA framework functions. However, obtaining good results requires us to apply and refine many general techniques that together contribute to good speedups.

For example, we mentioned that SA generates a set of sufficient conditions that can be evaluated dynamically and validate parallelization. However, the work (run-time overhead) involved in the dynamic evaluation of these predicates can vary greatly. Thus an ordering of their evaluations from simple to complex is crucial (somewhat similar to evaluating complex predicates) for obtaining good performance. In fact, based on performance models we can stop evaluating predicates if the effort outweighs the benefit of parallelization.

Further examples are simple algorithm substitution transformations. Exchanging a serial reduction with a parallel one can enable the parallelization of large loops. These transformations have to be proven correct though, and, in the case of complex or input sensitive memory reference patterns, this may not be possible statically. We use the same SA approach to generate dynamic conditions to validate parallelizing code transformations.

Contribution. In this paper we present some important aspects describing how the general framework of Hybrid Analysis (presented elsewhere [13, 14]) has been used and implemented in our parallelizing compiler (which is a derivative of the UIUC-Polaris compiler).

2 A Brief Introduction to Sensitivity Analysis

The Memory Reference Representation

There are three main concepts in our analysis. First, we introduce a powerful memory reference representation, the USR (uniform set representation). It was described in detail in [13] under the name RT_LMAD. In essence it can represent memory references of a program as an expression whose leafs are sets of LMADs (linear memory access descriptors) or enumerated sets of references which are composed (internal nodes of the expressions) through program operations (conditionals, loops, subroutine calls, etc.) A crucial advantage of this representation is that it is closed under composition - it can represent any memory reference pattern symbolically, at program level. When USRs cannot be evaluated to the exact sets of addresses they represent at compile time, they can be embedded in the generated code and be computed at run time, in the presence of actual input values. However, in most cases we do not need to compute the actual memory reference pattern, but rather prove a relation, which is generally easier.

Memory Reference Aggregation and Classification

The second concept in SA is memory reference aggregation, which ensures scalability of interprocedural analysis at the cost of losing dependence direction information. Memory references are aggregated bottom up on the Control Dependence Graph (CDG) within a subroutine, and on the call graph inter-procedurally. The program must have been restructured so the only loops in these graphs are the trivial self-loops in the CDG.

```
1   ... = A(6:15)        RO = [6 : 15], WF = ∅, RW = ∅
2   A(1:10) = ...        RO = ∅, WF = [1 : 10], RW = ∅
3   ...                  RO = [11 : 15], WF = [1 : 5], RW = [6 : 10]
```

Fig. 1. Memory reference classification example.

The process starts at leaf CDG nodes, which are simple statements. The set of memory locations *read* and *written* by the statement is computed from the statement type and symbolic expressions. This set is parameterized by symbolic variables referenced by the statement.

The sets corresponding to successive statements are then computed using set union, intersection and difference. All these operations are performed on USRs [13]. Special nodes in the CDG require more elaborate set operations, all of which are well defined and closed on USRs. Summarizing reference sets for *If-Then-Else* can result in predicated sets. Summarizing across *Do* loops requires symbolic expansion of sets across iteration spaces. Summarizing across the call graph requires symbolic translation of reference sets from a callee into the calling context.

These simple, node-local transformations on the CDG are applied repeatedly until the memory reference pattern has been completely summarized across the whole program.

Dependence Relations Based on Reference Summary Sets

While summarizing references, we also classify them into three disjoint sets [2]: 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. Computing the RO, WF and RW sets requires only the set operations discussed in the previous section. An example is given in Fig. 1.

Every time we reach a loop header in the aggregation process, we compute the cross iteration data dependence relations. If there are no dependences, then all the loop iterations can be executed in parallel. This is the most effective automatic parallelization method, as it scales with the number of iterations, thus it is likely to remain efficient as the underlying hardware evolves towards a larger number of processing units.

To express cross iteration dependence relations, we compute the set of memory locations that are referenced in two different iterations, and are written in at least one. At this point in the analysis, we have already computed RO_i, WF_i and RW_i, the per iteration reference sets.

One such dependence set is

$$DS = \cup_{i=1}^{n} RO_i \bigcap \cup_{i=1}^{n} WF_i$$

Similar dependence sets are expressed for combinations of RO, RW and WF sets [13]. If we can prove that $DS = \emptyset$, then no cross-iteration dependences may exist.

Sensitivity of Dependence Relations to Parameters

Finally, the third concept used in SA is the transformation of the USRs representing the aggregated memory references into a **Sensitivity Graph (SG)**, i.e., a boolean expression representing the parallelization conditions.

In many cases, proving the dependence set empty is trivial. It often results from a set intersection such as $[1:10] \cap [11:20]$, which evaluates to $\emptyset$ through symbolic calculus, at compile time. In other cases, proving the dependence set empty is not possible at compile time either because it depends on input data, e.g., $DS = [1:n] \cap [m:100]$ or because the relation is just too complicated for the compiler to evaluate.

We build SGs from dependence equations based on USRs by using a divide and conquer approach, which, at each step, breaks the dependence equation $DS = \emptyset$ into several simpler equations based on set identities [14]. For instance, equation $A \cup B = \emptyset$ is broken into $A = \emptyset$ and $B = \emptyset$. This algorithm is applied recursively until we reach equations involving only intervals, such as $[m:n] \cap [p:q]$. Such equations are translated into simple predicates based on bound comparison, e.g., $n < p$ *or* $q < m$.

We then extract a minimal (modulo the symbolic calculus capabilities of the compiler) run time check that guarantees that the loop is parallel. We then generate parallel code predicated by this condition. We use the SG [14] representation for these conditions. When they cannot be evaluated at compile time to a boolean value, they are embedded in the generated code and evaluated at run time, in the presence of actual values.

The aggregation and equation solving processes can deal with multidimensional strided reference patterns. In some cases, the divide and conquer process cannot extract a precise predicate from a dependence equation. In such cases, we approximate sets with optimistically predicated multidimensional strided intervals, and continue the analysis with affine sets, which are easier to compare. The optimistic assumptions are added to the dependence predicate, and are verified at run time through SG evaluation.

3 Engineering an Automatic Parallelizer

In the previous section we have provided an overview of the general approach to parallelization: We aggregate and, at the same time classify memory references (WF, RO, RW) at the program level into a set representation (USR) and then formulate the independence condition $DS = \emptyset$ (empty dependence set). Then, the compiler verifies the conditions for which this equation holds true by recursively descending on the equation $DS = \emptyset$ and, using boolean logic, generating a conjunction (OR) of simpler equations. Some of these equations can be proven true for all inputs, i.e., statically true, and others result in some constraints for the equation to be true. From these constraints (conditions) the compiler generates predicates (code) that are evaluated at run-time and can validate the

parallelization of a loop. The constraints are expressed as set of expressions which can be represented as a graph, the sensitivity graph (SG). This method was presented as SA (sensitivity analysis) [14].

Parallelism enhancing transformations. Our overall goal is to uncover as much parallelism (`doall` type only) as possible and exploit it when beneficial. To this we apply our SG based technique not only to prove that the original loops in a program are independent but also to validate code transformations that increase the intrinsic amount of parallelism. We will show how we can use our SA to perform powerful **dependence removing transformations**, e.g., reduction parallelization, pushback parallelization and array privatization. These are not new techniques, but the use of SA in their implementation makes them more powerful, i.e., more often successful.

Efficient Run-time Evaluation of Parallelization Conditions. After applying the dependence removing transformations the compiler needs to generate efficient parallel code. The outcome of the static Sensitivity Analysis may be the SG (sensitivity graph) which may be varying degree of complexity and which needs to be efficiently evaluated dynamically. It is important to perform the dynamic evaluation efficiently because this evaluation represents pure overhead. The novelty of our implementation lies in the way we generate efficient code for this dynamic validation.

We will present some of the more important aspects of this process, e.g., the generation of predicates that pre-validate parallel loop execution and the use of speculation and post execution validation. Sometimes we cannot extract a condition that can be evaluated before a loop is executed because it depends on the computed data. (There is a cycle between address and data computation). In this case, we have to resort to speculative execution [12]. This invokes other efficiency issues such as checkpointing (if used). Here too we use our program representation and SA to improve performance.

It is worth mentioning that our entire analysis framework is interprocedural. For the evaluation of USRs at run-time we have developed a library to which we generate calls. Similarly, when we employ LRPD we use a specialized library.

Let us now take a closer look at some powerful techniques.

3.1 Transformations to Remove Dependences

Conditional and Selective Array Privatization. Array privatization can be complex and expensive. In general, it means allocating a private array in each thread of execution. This replication can become quite costly if the array is big. In the most general case it is required to first copy-in from the shared array and then, after processing, copy-out the last value written. These two operations (copy-in and copy-out last value) can be very expensive because they do not scale. Thus we optimize them by performing selective copy-in and last value copy-out. In the case of relatively sparsely referenced arrays this can save significant time.

We can use USRs to express these in/out sets precisely in a general way and thus improve performance. Briefly, here is our approach:

By the time we reach a loop header, we have already classified all memory locations referenced within each iteration i into disjoint sets (USRs) RO_i, WF_i

and RW_i. Using only set operations, we put together the following descriptors ($\otimes^{\cup}_{i=1,n} A_i$ simply means $\cup^{n}_{i=1} A_i$, we just kept the notation used in [14]):

$$Memory\ to\ privatize : \otimes^{\cup}_{i=1,n} \left[WF_i \cap \left(\otimes^{\cup}_{k=1,i-1} WF_k \right) \right] \quad (1)$$

$$Subset\ to\ copy\ in = \left(\otimes^{\cup}_{i=1,n} RO_i \right) - \left[\left(\otimes^{\cup}_{i=1,n} WF_i \right) \cup \left(\otimes^{\cup}_{i=1,n} RW_i \right) \right] \quad (2)$$

$$Subset\ to\ copy\ out(per\ iteration) = WF_i - \left(\otimes^{\cup}_{k=i+1,n} WF_i \right) \quad (3)$$

The first descriptor contains the set of memory references that must be privatized because they are written to in at least two iterations. We chose to generate an OpenMP PRIVATE directive whenever this USR is not provably empty at compile time. This means we are possibly allocating too much private storage, since sometimes not all the elements in the array must be privatized. However, the alternative is to use an indirection table for just those locations that must be privatized, which introduces both complexity and overhead.

Although we privatize entire arrays, we perform selective and conditional copy in. Only those locations that are read before being written inside the loop are used in a memory copy operation from the shared object to the private copies. They are only copied if it turns out, at run time, that the values are needed inside the loop, based on actual control flow predicates. We wrote a simple *memcpy like* routine that uses a USR to control which locations get copied.

Conditional and Selective Array Reduction. Although implementations vary greatly, array reduction conceptually starts with an initialization of all the elements participating in the reduction with the null element of the reduction operator. The loop is executed in parallel. Upon exit from the parallel section, elements updated by more than one thread are merged using the reduction operation. We use USRs to describe the extent of the initialization and merge phases, and wrote simple library routines that use USRs to control the exact locations that are initialized and merged respectively.

$$To\ initialize = To\ reduce = \otimes^{\cup}_{i=1,n} \left[RW_i \cap \left(\otimes^{\cup}_{k=1,i-1} RW_k \right) \right] \quad (4)$$

Conditional Parallelization of Pushback Sequences. We have shown [15] how to recognize sequences of pushback operations that can be parallelized by using private storage, which simply need to be copied at the end of the loop to a specific location of the shared array. We use USRs WF_i to describe the extent of the writes to private storage, and a library function to perform the actual copies (the same used for copy in and copy out).

Not only do they get relocated efficiently, but this makes the transformation more general, since USRs can describe arbitrarily complex patterns. Previously, only pushback sequences made of contiguous locations could be parallelized.

3.2 Sensitivity Graph (SG) Evaluation

The outcome of the static Sensitivity Analysis may be either a definitive answer at compile time or the Sensitivity Graph (SG) which is a boolean expression which needs to be efficiently evaluated at run-time. It represents a conjunction

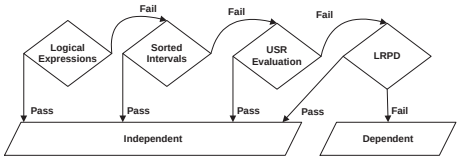

Fig. 2. Cascade of sufficient run time tests in increasing order of complexity.

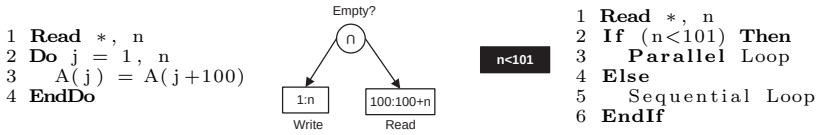

Fig. 3. Example of an input-sensitive memory reference pattern and corresponding code after parallelization.

(logic OR) of sufficient conditions which all can validate a loop to be parallel (including the associated dependence removing transformations). The SG can be of various complexities. They can be a:

a Boolean expression that can be evaluated in constant time.
b Boolean expression that can be evaluated in time proportional to some fraction of the size of the program data. For example, a triply nested loop with iteration spaces N,M,K can be parallelized by performing N (or N*M or K*N) work. This situation arises many times when aggregation works well in only some of the dimensions of the analyzed data structures.
c Boolean expression that can be evaluated in time proportional to data size.

In this latter case (c), some of the transformations of the equations ($DS = \emptyset$) involving the globally aggregated USRs into simpler ones has failed. This can happen when the recursive simplification of the $DS = \emptyset$ equation is not very successful or, in an extreme example, when the code uses indirection arrays. In effect, we need to generate code to dynamically evaluate the USR's (which can be seen as a program slice). The compiler will generate code for the evaluations of these conditions and sort them in order of their estimated complexity (similar

Fig. 4. A Hybrid Analysis extreme: in general, no test can solve this problem faster than the reference-by-reference LRPD test.

to the predicate of a branch condition). For illustration purposes, we have named the resulting code a *cascade of sufficient conditions* (Fig. 2).

There are four types of run time operations involved in the evaluation of the SG: (1) evaluation of elementary conditional expressions (constant time), (2) interval trees (some fraction of data size, simple operations), (3) actual evaluation of USRs (fraction of data size, complex operations) and comparison to the empty set and (4) reference-by-reference LRPD [12]. The estimated complexity of these tests ranges from $O(1)$ tests as the one in Fig. 3 to $O(n)$ dynamic reference instrumentation as is the case in Fig. 4. The evaluation of USRs at run time generally consists of fewer, but more complex operations than the reference-by-reference LRPD. In some cases they may either degenerate into inefficient enumerations or take conservative decisions that can lead to false negatives. The LRPD test has overhead proportional to the dynamic reference count, but is optimal for cases where aggregation and equation inversion are not possible (Fig. 4). It is always applicable, precise, and has a more predictable complexity. Perhaps the most important aspect of the "heavy" methods (USR evaluation or LRPD test) is that they have to be performed in parallel so that the overall obtained speedup scales with the number of processors.

There are two ways to validate parallel execution: Before the loop execution (similar to an inspector) or after its execution. In the latter case we have to use speculative execution [12].

In most cases, we can adopt either method and (hopefully) select the most efficient one. The correct choice involves a more complex cost model which is beyond the scope of this discussion. Presently, we choose speculation over pre-verification only if (1) a parallel inspector cannot be extracted (see next section) 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.

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. We apply loop invariant hoisting to USRs and SGs by performing aggressive invariance analysis on their sets of input variables. Invariance problems on USRs resulting from subscripted subscripts are formulated as dependence problems on the subscript arrays, which *are solved by the same SA algorithm applied to the subscript array*. 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 loop nesting level.

3.3 Speculative Execution

Sometimes we cannot extract a condition that can be evaluated before a loop is executed because it depends on the computed data. (There is a cycle between address and data computation). In this case, we have to resort to speculative execution [12]. This invokes other efficiency issues such as checkpointing (if used).

	Coverage		Speedup	
Suite	Polaris/HA	Intel	Polaris/HA	Intel
PERFECT	95%	14%	1.51	1.02
SPEC92/95	98%	29%	1.44	1.10
SPEC2000/2006	88%	62%	2.87	1.66

Table 1. Automatic parallelization coverage and speedups. PERFECT and SPEC92/95 speedups were measured on a 2-way Intel Core Duo. SPEC2000/2006 speedups were measured on an 8-way Sun server with 4 AMD dual core processors.

We identified previously the conditional pushback sequence pattern, which is perhaps the simplest such example. Other cases are more complex and do not follow a preset pattern. It should be noted that even when the dependence relation can be precomputed before the loop it may be worth executing the loop in parallel speculatively in order to reduce the overhead. A more detailed discussion about these choices can be found in [8].

If speculative parallelization is necessary, we take advantage of our novel representation and SA techniques to reduce overheads. We can compute the exact extent (as a USR) of memory that must be either saved at a checkpoint before the speculative loop, or committed from private speculative storage after the loop. The actual memory operations are implemented as calls to our memory copy routine used for copy in, copy out and pushback parallelization.

3.4 The Value Evolution Graph and Pushback Sequences

The *Value Evolution Graph (VEG)* [15] can represent the data flow in recurrences used as array indices which have no closed form solutions. The graphs are pruned based on control dependence predicates and produce tighter value ranges than abstract interpretation methods. These value ranges and their relations (overlapping, mutual exclusive) are used throughout our analysis, when building USRs and when extracting SGs.

Additionally, VEGs can be used to detect monotonic reference patterns in the code text. Unlike previous pattern recognition methods, we can analyze partially aggregated and classified memory descriptors (USRs). This single generic approach both extends and unifies in a single framework most cases which were previously solved using various, different, pattern matching techniques. It allows for the parallelization of important classes of memory reference patterns, e.g., sequences of pushback operations with complex footprints.

4 Experimental Results

Our experiments show that our techniques extract almost all the available parallelism at the highest granularity possible, which results in significant speedups on 22 codes from the PERFECT and various SPEC benchmark suites.

Table. 1 presents 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.

Technique	PERFECT	SPEC
CT	58.00	87.25
RT: Speculative	11.90	1.42
RT: Non-Speculative	26.60	4.08
Total	95.50	92.75

(a)

Technique	PERFECT	SPEC
SG: Simple Expressions	20.10	2.08
SG: Interval Trees	9.20	0.00
SG: LRPD	3.00	1.42
SG: USR Evaluation	13.80	4.08
Hybrid Priv, Red	12.60	0.50
VEG	12.80	0.91
Pushback Recognition	9.60	0.92

(b)

Table 2. Parallelization coverage breakdown (a) between compile time and run time (b) as contribution of each compiler technique. The coverages in (b) overlap because parallelizing some loops required several techniques. Coverage is measured as the percentage of the original sequential execution time that was parallelized.

Test	Type	Accuracy	Success	% S
Parallel/Sequential	Simple Expression	Sufficient	Fail	0.005
	USR Evaluation	Necessary&Sufficient	Pass	0.025
Indep. Update/Reduct.	Simple Expression	Sufficient	Fail	0.005
	USR Evaluation	Necessary&Sufficient	Fail	0.030
Indep. Write/Priv.	Interval Trees	Necessary&Sufficient	Pass	0.005

Table 3. Run time tests actually executed to decide whether the dependence structure on array *MX* prohibits or allows parallelization of loop *DYFESM/MXMULT_do10*. %S represents the time spent in the test as a percentage of the execution time of the loop.

Two main factors are behind these good speedups: high granularity and high coverage. The VEG, the USR and SG are all interprocedural and flow sensitive (though they use approximations), which makes our analysis apply to large program slices, resulting in higher granularity. Our hybrid approach pushed coverage over 90%. It also increased granularity significantly, since many outer loops could be proved parallel only at run time. A detailed discussion of the speedup numbers can be found in [14].

Table 2 presents the effect of each technique towards our goal of achieving highest parallelization coverage possible. It is important to note that our hybrid framework solves the parallelization problems uniformly at both compile-time and run-time, using SGs. The techniques presented in this paper contribute substantially to the coverage and granularity of parallelization. Comprehensive reports for a large set of loops are available at:

http://parasol.tamu.edu/compilers/ha

An interesting case is loop *MXMULT_do10*, which accounts for 73% of the sequential execution time on *DYFESM*. This loop contains an array *MX* which shows multiple patterns on different subsections. The first part of the array is only written to, while the last part is a reduction. The write section is fully independent, but this is not known until run time. The reduction section is only proven a proper reduction (not an independent update) at run time. Table 3 presents our run time tests, their dynamic outcomes and their relative overhead for this loop.

Tables 4 and 5 show the occurrence of each static and dynamic dependence test, privatization, reduction, pushback, and speculative parallelization in various benchmark programs.

Code	Loop	%	DD Test	Priv	Red	PB	IP	EX	Intel
ADM	RUN_do20,...,100	44	RT:SE,UE	CT,A	-	-	√	IE	-
	D*DTZ_do30	31	CT	CT,A	CT	-	√	-	-
	DKZMH_do20,50	11	CT	CT,A	-	-	√	-	-
	WCONT_do40	5	CT	CT,A	CT	-	√	-	-
ARC2D	STEPF*_do*	29	CT	CT	-	-	-	-	√
	*PENT*_do*	14	CT	CT	-	-	-	-	√
	FILERX_do15	14	RT:SE,UE	CT,A	-	-	-	IE	-
	RHS*_do*	10	CT	CT	-	-	-	-	√
	TK*_do1	8	CT	CT	-	-	-	-	-
BDNA	ACTFOR_do240,500	89	CT	CT,A	CT	-	-	-	-
DYFESM	MXMULT_do10	73	RT:IT,UE	RT:IT,A	RT:IT,UE	-	√	IE	-
	SOLVH_do20	9	RT:SE	RT:IT,A	-	-	√	IE	-
	FORMR0_do20	7	RT:IT,UE	RT:IT,A	RT:IT,UE	-	√	IE	-
	SOLXDD_do4,10,30,50	9	RT:IT	RT:IT,A	RT:IT	-	√	IE	-
FLO52	*FLUX*_do*	55	CT	CT	-	-	-	-	√
	PSMOO_do40,80	21	CT	CT	-	-	-	-	-
	EULER_do*	15	CT	CT	CT	-	-	-	√
MDG	INTERF_do1000	93	RT:SE	CT,A	CT	-	√	SP	-
	POTENG_do2000	6	CT	CT,A	CT	-	√	-	-
OCEAN	FTRVMT_do109	41	RT:SE	CT	-	-	-	IE	-
	IN_do10	15	CT	-	-	-	-	-	-
	OUT_do10	15	CT	-	-	-	-	-	-
	CSR,RCS_do20	7	CT	CT	-	-	-	-	-
	ACAC,SCSC_do30,40	6	CT	CT,A	-	-	-	-	-
SPEC77	GLOOP_do1000	48	CT	CT,A	CT	-	√	-	-
	GWATER_do1000	24	RT:LRPD	CT,A	CT	-	√	SP	-
	SICDKD_do1000	4	CT	CT,A	-	-	√	-	-
TRACK	EXTEND_do400	50	CT	CT,A	-	√	√	-	-
	FPTRAK_do300	46	CT	CT,A	-	√	√	-	-
	NLFILT_do300	2	RT:LRPD	CT,A	-	-	√	SP	-
TRFD	OLDA_do100	67	CT	CT,A	-	-	-	-	-
	OLDA_do300	28	CT	CT,A	-	-	-	-	-
	INTGRL_do140	3	RT:IT	RT:IT,A	-	-	-	IE	-

Table 4. Loop parallelization in PERFECT codes. % = percentage of total application execution time. DD Test = type of data dependence test required (CT = compile time, RT = run time, SE = simple logical expressions, IT = interval trees, UE = USR evaluation, LRPD = LRPD run time test) Priv = type of privatization required (A = array privatization). Red = type of reduction required. PB = pushback required. IP = loop contains subprogram calls. EX = execution type (IE = nonspeculative, SP = speculative execution). Intel = parallelized automatically by the Intel Compiler (version 9.1, *-parallel -par_threshold100*).

5 Conclusions

In this paper we have presented some of the more important issues involved in the implementation of the novel Sensitivity Analysis framework in our Polaris derived automatic paralleling compiler. We have shown that our powerful USR representation and our sensitivity analysis technique is useful not only in detecting independent loops but also in applying parallelism enhancing transformations (e.g, reduction and pushback parallelization, privatization). We have further shown that SA generates a flexible cascade of sufficient conditions applied in order of their estimated execution time complexity. Thus we allow a flexible cost-benefit analysis between the benefits of parallelization and the effort to obtain it. We further present the impact of our methods on 22 benchmark codes and report speedups that compare quite well with existing commercial compil-

Code	Loop	%	DD Test	Priv	Red	PB	IP	EX	Intel
APPLU	JACL*_do#1	34	CT	CT	-	-	-	-	-
	RHS_do#1,2,3,4	20	CT	CT	-	-	-	-	√
APSI	RUN_do*	25	RT:SE,UE	CT,A	-	-	√	IE	-
	D*DTZ_do40	40	CT	CT,A	CT	-	√	-	-
	DKZMH_do30,60	12	CT	CT,A	-	-	√	-	-
	WCONT_do40	6	CT	CT,A	CT	-	√	-	-
	HYD_do20	5	CT	CT	CT	-	-	-	-
MGRID	RESID_do600	52	CT	CT	-	-	-	-	√
	PSINV_do600	27	CT	CT	-	-	-	-	√
	RPRJ3_do100	7	CT	CT	-	-	-	-	√
	INTERP_do400,800	8	CT	CT	-	-	-	-	√
	COMM3_do100,200,300	5	CT	CT	-	-	-	-	-
SWIM	SHALLOW_do3500	48	CT	CT	CT	-	-	-	-
	CALC1_do100	14	CT	CT	-	-	-	-	√
	CALC2_do200	17	CT	CT	-	-	-	-	√
	CALC3_do300	19	CT	CT	-	-	-	-	√
WUPWISE	MULDEO_do100,200	47	CT	CT,A	-	-	√	-	-
	MULDOE_do100,200	46	CT	CT,A	-	-	√	-	-
HYDRO2D	FILTER_do*	42	CT	CT	-	-	-	-	√
	FCT_do*	18	CT	CT	-	-	-	-	√
	ARTDIF_do*	14	CT	CT	-	-	-	-	√
	TRANS*_do*	12	CT	CT	-	-	-	-	√
	TISTEP_do*	6	CT	CT	-	-	-	-	√
	S1,S2_do100	4	CT	CT	-	-	-	-	-
MATRIX300	LBMK14_do20	13	CT	CT	-	-	-	-	-
	SGEMM_do*	86	CT	-	-	-	√	-	-
MDLJDP2	FRCUSE_do20	76	CT	CT	CT	-	√	-	-
	FRCBLD_do20	11	CT	CT	CT	√	√	-	-
	POSTFR_do*	8	CT	CT	CT	-	-	-	-
	PREFOR_do*	5	CT	CT	-	-	-	-	-
NASA7	VPETST_do110	26	CT	CT	-	-	√	-	-
	GMTTST_do120	24	RT:UE	CT	-	-	√	IE	-
	CFFT2D*_do130,150	17	RT:LRPD	CT	-	-	-	SP	-
	BTRTST_do120	10	CT	CT	-	-	√	-	-
	CHOTST_do120	9	CT	CT	-	-	√	-	-
	EMIT_do5	6	CT	RT:IT,A	-	-	-	IE	-
ORA	MAIN_do9999	99	CT	CT	CT	-	√	-	-
SWM256	CALC1_do100	31	CT	CT	-	-	-	-	√
	CALC2_do200	38	CT	CT	-	-	-	-	√
	CALC3_do300	30	CT	CT	-	-	-	-	√
TOMCATV	MAIN_do100/2,120/2,60,...	96	CT	CT	CT	-	-	-	√

Table 5. Loop parallelization in SPEC codes. (Legend in Table 4.)

ers. These good results are due to our ability to uncover and efficiently exploit large granularity parallelism.

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 Programming Language Design and Implementation*, pages 258–269, 1995.
2. J. Hoeflinger. *Interprocedural Parallelization Using Memory Classification Analysis.* PhD thesis, University of Illinois, Urbana-Champaign, August, 1998.
3. D. kai Chen, J. Torrellas, and P.-C. Yew. An Efficient Algorithm for the Run-time Parallelization of DOACROSS Loops. *To appear in Proc. for Supercomputing '94, Washington D.C., November 14-18, 1994*, October 1994.
4. S. Leung and J. Zahorjan. Improving the performance of runtime parallelization. In *4th PPOPP*, pages 83–91, May 1993.

5. S. Moon and M. W. Hall. Evaluation of predicated array data-flow analysis for automatic parallelization. In *PPoPP '99: Proc. of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 84–95, New York, NY, USA, 1999. ACM Press.
6. S. Moon, M. W. Hall, and B. R. Murphy. Predicated array data-flow analysis for run-time parallelization. July 1988.
7. S. Moon, B. So, M. W. Hall, and B. R. Murphy. A case for combining compile-time and run-time parallelization. In *LCR '98: Selected Papers from the 4th Int. Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 91–106, London, UK, 1998. Springer-Verlag.
8. D. Patel and L. Rauchwerger. Principles of speculative run–time parallelization. In *Proc. 11th Annual Workshop on Programming Languages and Compilers for Parallel Computing*, pages 330–351, August 1998.
9. W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91*, pages 4–13, Albuquerque, N.M., Nov. 1991.
10. 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, 1995.
11. C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation*, New York, NY, USA, 2005.
12. 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 1995 Conf. on Programming Language Design and Implementation, La Jolla, CA*, pages 218–232, June 1995.
13. S. Rus, J. Hoeflinger, and L. Rauchwerger. Hybrid analysis: static & dynamic memory reference analysis. *Int. J. of Parallel Programming*, 31(3):251–283, 2003.
14. S. Rus, M. Pennings, and L. Rauchwerger. Sensitivity analysis for automatic parallelization on multi-cores. In Proc. of the ACM Int. Conf. on Supercomputing, Seattle, WA, 2007.
15. S. Rus, D. Zhang, and L. Rauchwerger. The value evolution graph and its use in memory reference analysis. In *Proc. of the 13-th Int. Conf. on Parallel Architectures and Compilation Techniques, Antibes Juan-les-Pins, France*, Oct. 2004.
16. J. H. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.
17. M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
18. H. Yu and L. Rauchwerger. Run-time parallelization overhead reduction techniques. In *Proc. of the 9th Int. Conf. on Compiler Construction, Berlin, Germany*. LNCS, Springer-Verlag, March 2000.