Run-time Assisted Interprocedural Analysis of Memory Access Patterns

Silvius Rus
rus@tamu.edu
Texas A&M University

Lawrence Rauchwerger
rwerger@cs.tamu.edu
Texas A&M University

Jay Hoeflinger
jay.p.hoeflinger@intel.com
Intel-KAI Corporation

Texas A&M University Technical Report TR01-001, January 2001

ABSTRACT

In this paper we present a unified framework based on the existing IPA and speculative run-time parallelization technology analysis in the Polaris compiler. The framework is able to analyze efficiently all statically un-analyzable cases in a uniform manner. We use the same technique to extract run-time assertions from any loop and depending on the case at hand, generate run-time tests that range from a low cost scalar comparison to a full, reference by reference LRPD test. We accomplish this by both extending compile time IP analysis techniques and by incorporating speculative run-time techniques when necessary. In essence our solution is to bridge 'free' compile time techniques with exhaustive run-time techniques through a continuum of simple to complex solutions. We have implemented our framework in the Polaris compiler by introducing an innovative intermediate representation (RT_LMAD) and associated run-time operations. It includes intersection, union, last-value assignment, aggregated inspectors and LRPD tests for both dense and sparse reference patterns.

1. INTRODUCTION

The analysis of memory reference patterns is one of the most important phases performed by a compiler in order to transform the original code into an optimized version. Memory access analysis is crucial for parallelization and locality enhancement. In the case of scientific codes, automatic parallelization is obtained by employing various forms of array data dependence analysis techniques. It is widely accepted that good parallel code requires the detection and exploitation of parallelism at all hierarchical levels of the code (loop structure) with an emphasis on outer, large granularity loops. This requirement implies that compilers must be able to perform their data dependence analysis interprocedurally. At times this has been performed through procedure inlining when applications were small. However, large codes require a scalable (proportional with the size of the code) interprocedural analysis method which can produce results in a reasonable time on a moderately powerful machine. Such techniques have been developed and incorporated in research compilers such as Parascope [2], SUIF [4] and Polaris [5] among others. In [6] we have developed a framework for interprocedural analysis of memory access patterns which can perform a global data dependence analysis and thus extract good quality parallelism. In essence, it is based on an aggregation of individual references into concise access descriptors called LMADs (linear memory access descriptors). This compacted information can be propagated and aggregated to a global scope in a fairly scalable manner.

The results of classic compiler analysis are by nature conservative for several reasons. Algorithms are not sufficiently powerful to deal with all cases encountered in practice which is frequently due to their inability to perform accurate symbolic analysis. Another fundamental reason for conservative compiler decisions is the static unavailability of crucial information. Certain values become available only after execution has started or are compute dependent. This problem has become more important in the last years due to the dynamic nature of modern simulations and the ascendancy of Java. Many optimizations, most notably parallelization, cannot be performed at compile time because the access pattern is either too complex for current data dependence algorithms or simply statically unavailable. The challenge posed by the dynamic nature of these modern (and some older) applications has been addressed in the last years through run-time techniques. In essence all run-time methods detect sections of code that can be run safely in parallel by recording and analyzing a (compressed) trace of all relevant memory accesses during program execution. We have developed such techniques that can extract at run-time either full parallelism (DOALL parallelism) [8, 3] or partial parallelism [7].

There is, however, a fairly clear separation between the compile time and run-time techniques. Static compiler technology uses analytical methods to draw conclusions about entire sections of memory references and iteration spaces. Run-time techniques use, for the most part, an exhaustive analysis method of all points referenced and thus can be very expensive. There is, to this date, very little static partial information that flows from the compiler to the run-time optimization system. Attempts have been made to integrate the two approaches [11, 10] but with limited applicability.

In this paper we present an attempt to create such a unified framework that can seamlessly integrate compile time and run-time analysis. This system can extract maximum parallelism with minimum overhead by passing all partial (but insufficient) static information to the run time system and thus can eliminate most redundant analysis.

1.1 The Need for Run-time Assisted IPA

In this section we present several examples from real codes for which classic, static compilation produces only conservative output. In all these cases the compiler could produce optimized code by using run-time checks to verify whether the reference pattern of the loops allows parallelization. In

```
subroutine FOO(A,N,L)
integer A(100)
Do j=1,L
  A(j) = A(j+N)
Enddo
Return
```

```
Program Main
integer A(1:100,1:100)
READ *,A(:,:),N,L
Do i=1,N
  Call FOO(A(1,i),N,L)
Enddo
```

Figure 1:

```
subroutine FOO(C,TMP,N,M,lim)
integer C(*),TMP(*)
Do i=1,N
  If (C(I).lt.lim) then
    TMP(1:M) = ...
  Endif
  ... = TMP(1:M)
Enddo
Return
```

```
Program Main
integer C(1:N),TMP(1:M)
READ *,C(:),lim
Do k=1,L
  Call FOO(C,TMP,N,M,lim)
Enddo
```

Figure 2:

each case we sketch the 'ideal', minimum run-time overhead check that could be obtained if the compiler could extract the necessary and sufficient conditions which it cannot verify at compile time. We further briefly present previous work in this domain and then summarize the main contributions of our work presented in this paper.

Let us consider the pseudo-code example in Fig. 1. Because the value of N is unknown at compile time we cannot decide if the loop in the subroutine FOO can be executed as a DOALL or not. Moreover, the character of the outer loop (in the main program) is also undecidable. However for subroutine FOO a simple dependence analysis can generate the appropriate run-time assertion that will check whether or not the maximum write offset is smaller than the minimum read access. Furthermore, we need to propagate this assertion to the outer loop through the procedure boundary and hoist it to the beginning of the program, right after the crucial value N has been read in. In this case the compiler generated run-time assertion takes the form of a *simple scalar comparison* which can decide the character of the loops.

Fig. 2 shows a more complex situation for both inner and outer loops. We first need to analyze the loop nest in subroutine FOO and then report the findings at the program level. Because nothing can be said about the values of C(i) in FOO we cannot conclude whether array TMP is privatizable, i.e., whether all uses of TMP are covered by a write for every iteration. It all depends on the values of C which are read in the main program from a file. To disambiguate such a situation we need to (a) be able to collect the loop level information and propagate it at the main program level and (b) generate a minimal run-time assertion that would check the contents of C and decide if the outer loop is parallel. More specifically we need to take advantage of the compile time analysis that concludes under what conditions the array TMP is or is not privatizable. As we will show later the generated run-time assertion consists of a relatively simple parallel inspector loop of the array C. Because C is loop invariant the inspector loop can be hoisted from the main loop which allows us to reuse the decision of the assertion (the technique is known as 'schedule reuse'[9]).

Fig. 3 complicates the previous problem further by computing the values of C at every iteration, i.e., C is loop variant. This situation does not allow us generate a simple,

```
subroutine FOO(C,TMP,N,M,lim)
integer C(*),TMP(*)
Do i=1,N
  If (C(I).lt. lim) then
    TMP(1:M) = ...
  Endif
  ... = TMP(1:M)
Enddo
Return
```

```
Program Main
integer C(1:N),TMP(1:M)
READ * ,lim
TMP(:) = 0
Do k=1,L
  C(k) = TMP(k)
  Call FOO(C,TMP,N,M,lim)
Enddo
```

Figure 3:

```
subroutine FOO(C,TMP,N)
integer C(*),TMP(*)
Do i=1,N
  TMP(C(i)) = ...
Enddo
Return
```

```
Program Main
integer C(1:100),TMP(1:M)
READ *,C(:),N
TMP(:) = 0
Do k=1,L
  Call FOO(C,TMP,N)
  ... = TMP(k)
Enddo
```

Figure 4:

reusable inspector before the main loop. A solution is to insert an assertion about the values of C inside the loop. Deciding when and where to use such an assertion is a key technique for a compiler.

Fig. 4 shows the case when the addresses are read-in from a file. Classic static compilation cannot produce any results. However the compiler could extract the condition under which the array TMP is either independent or privatizable at the outer loop level. The result of the run-time analysis can be reused because C is loop invariant.

The case presented in Fig. 5 is similar to the previous one except that the subscripts of array TMP are computed inside the inner most loop instead of being read in. This produces a dependence cycle between address and data computation and obliges us to insert a run-time check for every reference to TMP. We have shown in previous work that speculative execution is a viable solution.

Finally, Fig. 6 illustrates an example of a loop where privatization is conditioned by the values taken by a recurrence. The least expensive run-time assertion would be for the compiler to test the first value taken by NA in the inner loop (If NA=1 then the array CC is privatizable).

1.2 Current State of Art

The generation of run-time assertions for a possibly more aggressive optimization has been applied at least since the introduction of vectorizing compilers. A simple check on the length of the vector was used to make the dynamic decision whether to use the vectorized or the scalar version of the code.

In later years various techniques for run-time paralleliza-

```
subroutine FOO(C,TMP,N)
integer C(*),TMP(*)
Do i=1,N
  TMP(C(i)) = ...
  C(i) = f(TMP(I))
Enddo
Return
```

```
Program Main
integer C(1:100),TMP(1:M)
READ *,C(:),N
TMP(:) = 0
Do k=1,L
  Call FOO(C,TMP,N)
  ... = TMP(k)
Enddo
```

Figure 5:

```
                          subroutine ADM_RFFTF(CC,CH)
                          integer CC(*), CH(*)
subroutine RAD(CC,CH)     NA = 1
integer CC(*), CH(*)      Do i=1,N
Do i=1,N                    NA = NA-1
  CC (:) = ...              If (NA.EQ.0) then
  ....    = CH(:)             Call RAD(CC,CH)
Enddo                       else
Return                        Call RAD(CH,CC)
                            endif
                          Enddo
```

Figure 6:

tion and partial redundancy elimination have been proposed. Saltz [9] and, later Rauchwerger [8, 7] have proposed either inspector/executor or speculative methods that can trace and analyze the references of a loop and decide before and respectively after execution if the loop is parallel. Polaris has been the only compiler that has benefited from this technology [11].

Interprocedural analysis has been introduced to research compilers by Hall in her PhD thesis and then also implemented in SUIF [4]. In Polaris [1] IPA has been done by Paek, Hoeflinger and Padua and reported in [6, 5].

The use of run-time assertions within the IPA context is a recent development and has been reported by Mary Hall in [10] and Hoeflinger in [5]. Both of these compilers generate relatively simple low cost assertions and are capable of solving problems similar to those shown in Figs. 1. The coverage of the techniques used is mostly restricted to the cases when a predicate can be extracted outside the analyzed loop and a low cost run-time test can be generated. In [10] the technique is based on a predicated data flow analysis with emphasis given to combining predicates that affect parallelization conditions. The result are low cost run-time assertions with multiversion loops. When predicates are loop variant, sequential and are scoped only within a lower level procedure conservative assumptions are made. This has the advantage of simplifying compile time analysis but does not significantly reduce run-time overhead (especially given the profitability of parallelization). In [5] the infrastructure is based on predicated memory access descriptors (LMAD). These LMADs have the advantage of representing memory references in an aggregated form, making static analysis easier.

In [11] the authors start from the other 'end', i.e., from exhaustive run-time parallelization techniques (LRPD test) and reduce their complexity through predicated reference aggregation and logical implication. Their emphasis are cases similar to those shown in Figs. 3– 6 which generally require speculative execution. Their results are good but do not extend beyond the procedural level.

In most of the previous work reported so far, the authors have to perform some manual transformation (e.g., loop peeling for ADM) before being able to apply their techniques.

1.3 Our Contribution

In this paper we present a unified framework based on the existing IPA and speculative run-time parallelization technology analysis in the Polaris compiler. The main contribution of our framework is its ability to analyze efficiently all statically un-analyzable cases in a uniform manner. We use the same technique to extract run-time assertions from any loop and depending on the case at hand, generate run-time tests that range from a low cost scalar comparison to a full, reference by reference LRPD test. Moreover we can order the run-time tests in increasing order of complexity (overhead) for any loop and thus risk the minimum necessary overhead. We accomplish this by both extending compile time IP analysis techniques and by incorporating speculative run-time techniques when necessary. In essence our solution is to bridge 'free' compile time techniques with exhaustive run-time techniques through a continuum of simple to complex solutions. Our uniform approach allows us to always obtain maximum performance for the minimum necessary cost (overhead). We implement our framework in the Polaris compiler by introducing an innovative intermediate representation (RT_LMAD), a run-time library that can operate on it. It includes intersection, union, Last-value assignment, aggregated inspectors and aggregated LRPD tests for both dense and sparse reference patterns. Our analysis is accurately flow sensitive on any control flow graph.

In the following sections we first give a short overview of needed background material and then present our framework. After some implementation notes we present some experimental results under the form of case studies.

2. FRAMEWORK FOR MEMORY ACCESS ANALYSIS AT RUN-TIME

2.1 Review: Static Analysis Framework

```
C A declared with m dimensions
DO I1 = 0, U1 {
   DO I2 = 0, U2 {
      ...
         DO Id = 0, Ud {
            ... A(s1(I), s2(I), ..., sm(I)) ...
         }
      ...
   }
}
```

Figure 7: General form of a reference to array A in a d-nested loop. The notation $\vec{I}$ represents the vector of loop indices: $(I_1, I_2, \dots, I_d)$.

The *memory space* of a program is the set of memory locations which make up all the memory usable by a program. When an m-dimensional array is allocated in memory, it is linearized and usually laid out in either row-major or column-major order, depending on the language being used. To map the array space to the memory space of the program, the subscripting function must be mapped to a single integer that is the offset from the beginning of the array for the access. We define this *subscript mapping* F_a for an array reference with a *subscripting function* s, as in Figure 7 by

$$F_a(s_1, s_2, \cdots, s_m) \quad = \quad \sum_{k=1}^{m} s_k \cdot \lambda_k.$$

When the language allocates an array in column-major order, $\lambda_1 = 1$ and $\lambda_k = \lambda_{k-1} \cdot n_{k-1}$ for $k \neq 1$. If the language allocates the array in row-major order, $\lambda_m = 1$ and $\lambda_k = \lambda_{k+1} \cdot n_{k+1}$ for $k \neq m$.

If the linearized form of the subscript expression s can be written in a sum-of-products form with respect to the individual loop indices,

$$F_a(\mathbf{s}(\vec{I})) \quad = \quad f_0 + f_1(I_1) + f_2(I_2) + \cdots + f_m(I_m). \tag{1}$$

then, we can isolate the effect of each loop index on the subscripting offset sequence. In this case, there is no restriction on the form of the functions f_k. They can be subscripted-subscripts, or any non-affine function. We define the isolated effect of any loop in a loop nest on a memory reference pattern to be a *dimension* of the access. A dimension k can be characterized by its *stride*, δ_k, and the number of iterations in the loop, $U_k + 1$. An additional expression, the *span*, is carried with each dimension since it is used in some operations.

A Linear Memory Access Descriptor (LMAD) is a representation of the subscripting offset sequence. *It can be built for any array reference whose subscript expressions can be put in the form of Equation 1, so all algorithms in this paper will assume this condition has been met for all LMADs.* It contains all the information necessary to generate the subscripting offset sequence. Each loop index in the program is *normalized* internally for purposes of the representation, and called the *dimension index.*

The LMAD contains:

- a starting value, called the *base offset*, represented as τ, and
- for each dimension k:
  - a dimension index I_k, taking on all integer values between 0 and U_k,
  - a stride expression, δ_k
  - a span expression, σ_k.

The general form for an LMAD is written as

$$\mathcal{A}\,^{\delta_1,\delta_2,\cdots,\delta_d}_{\sigma_1,\sigma_2,\cdots,\sigma_d} + \tau.$$

Memory Classification Analysis

Data dependence analysis can be formulated in terms of a scheme of classifying memory locations, called Memory Classification Analysis (MCA) [5], based on the order and type of the accesses within a section of code. The method of classifying memory locations is a general one, based on abstract interpretation[?, ?] of a program, and may be used for purposes other than dependence analysis.

We can classify a set of memory locations according to their access type by adding a symbolic representation of them to the appropriate *summary set.* A **summary set** is a symbolic description of a set of memory locations.

We use LMADs to represent memory accesses within a summary set. To represent memory accesses for use in MCA, we use three summary sets: ReadOnly (RO), WriteFirst (WF), and ReadWrite (RW).

The RO summary set records all the memory locations that are only read in a section of code. The WF summary set records all memory locations that are written before being read in a code section. The RW summary set consists of all other memory locations referenced in a code section. RW can include locations for which we are just not sure about the order.

Classification of Memory References

Each memory location referred to in a code section to be analyzed must be entered into one of these summary sets, in a process called *classification.* A program is assumed to be a series of nested contexts[1].

Classification combines summary sets with an *intersection* operation, in a way that maintains proper classification for each memory location. For more information, see [5].

2.2 Defining The Run-time Framework

The framework for interprocedural analysis of access memory references described in the previous section can handle well most statically defined cases. However it has some limitations: If the descriptors are statically undefined as is the case in Figs. 1–6 the compiler must make conservative decisions, i.e., generate sequential code. Yet another limitation is its ability to successfully aggregate LMADs as they are propagated upwards, to the global level. This inability is due to either incompatible LMADs (that cannot be aggregated due to stride combination) or due to an insufficiently powerful symbolic analysis that is needed when comparing LMADs. The result is an ever increasing number of LMADs that is carried upwards, in the program hierarchy which makes compilation in reasonable time practically impossible. In other words, the framework is not always scalable.

To overcome these limitations we extend our static IPA framework to a more general and more powerful Run-time Assisted Interprocedural Framework (RTA-IPA) that can deal uniformly with all the difficulties mentioned above. More specifically we introduce the run-time extension to the LMADs, the RT_LMAD. As we will detail in the next section it is an aggregated representation of a reference pattern which, at run-time has its own allocated data structure. We define an extended set of operations which can be performed on these RT_LMADs using a run-time library. We extend our compile time analysis and decision algorithms to classify RT_LMADs and generate appropriate code. For example, if the parallelization decisions can be made using only a few RT_LMAD intersections which are fully defined before loop execution then we will generate a very simple inspector code. If the compile time analysis decides that the RT_LMADs are dependent on the data computed by the loop then the operations using the LMADs will be inserted in a loop prepared for speculative execution. We should mention here that sometimes the LMAD representation 'decays' into an equivalent point to point enumeration of the reference space. Such cases are equivalent to applying the LRPD test, as described in [8, 11].

2.3 RT_LMAD: Structure and Operations

The basic data structures are run-time valued descriptors of memory access patterns. They are extensions of LMADs designed to store information for run-time evaluation: RT_LMAD.

An RT_LMAD is represented as a formal expression having as operands lists of LMADs. The operators of these expressions can be: *union* ($\cup$), *difference* ($\backslash$), *intersection* ($\cap$). They are generated when we perform some operations with sets at compile-time but the result cannot be computed due to an unknown value. Let us follow the example in Fig. 1 Here, the LMAD that aggregates the reads from array `A` across all iterations is: `[1+N:10+N]`, and the write LMAD is `[1:10]`. Their intersection cannot be performed

[1] If the programming language does not force this through its structure, then the program will be transformed into that form through a *normalization* process.

```
DO I = 1, 10
   IF (C(I)) THEN CALL W(TMP, 1, N, 1)
   CALL R(TMP, 1, N, 1)
ENDDO
```

Figure 8: Subroutine W writes array TMP and subroutine R reads it.

at compile-time, but we can represent it as an RT_LMAD: `[1+N:10+N]` $\cap$ `[1:10]`.

There is another operator, that allows representation of the aggregation of access descriptors across all iterations of a loop. *Expand* has two operands: one of them is an induction variable together with a range of values, and the other one is a description as a function of the induction variable. Let us follow another example. Let us assume we want to express the set of reads from memory location `TMP` that are not covered by writes to the same memory location within the same iteration. For every iteration we have a descriptor $ExposedRead_i = ([1:N-1]+1) \setminus (\{C(I)\}[1:N-1]+1)$. Across all iterations:

$ExposedRead = Expand((I,1,10,1), ExposedRead_i)$, or

$ExposedRead = Expand((I,1,10,1), ([1:N-1]+1) \setminus (\{C(I)\}[1:N-1]+1))$

We follow the same algorithm as the one for obtaining descriptor sets RO, WF, RW. However, in addition to the static version of the framework, there are three run-time evaluateable sets for every symbol: RT_RO, RT_WF, RT_RW. They are all represent ed as RT_LMADs. The only addition to the algorithm is that whenever a symbolic operation cannot be performed, instead of approximating we generate an RT_LMAD and we place it in the corresponding descriptor (RT_RO, RT_WF, or RT_RW).

The decisions taken statically rely on data from static descriptors RO, WF, RW. If there is enough information in them, decisions are taken and analysis stops here. If not, code is generated to take them at run-time, after all necessary values will have been computed.

When Do We Need RT_LMADs?

We generate an RT_LMAD for any operation that could not be done statically. Here are a few cases for which we will definitely need an RT_LMAD.

(a) A simple operation ($\cup, \cap, \setminus$) has as an operand some input value. If our symbolic simplifier fails to perform the operation, or *the result is prohibitively large*, an RT_LMAD is generated, having as operator the operation we failed to perform. Assume we need to intersect `[1:N]` with `[10:20]`, we have two choices. First, generate a list of LMADs: all possible outcomes: ϕ, `[10:N]`, `[10:20]`. We prefix them with predicates to distinguish the case they cover: `{N.LT.10}`ϕ, `{N.GE.10.AND.N.LE.20}[10:N]`, `{N.GT.20} [10:20]`. Here there is only an unknown (`N`) and the dimension of the descriptors was 1. When there are many unknowns, finding the guarding predicates becomes untractable. Also, it leads to exponential growth of the representation. With our framework, generate RT_LMAD $[1:N] \cap [10:20]$.

(b) Assume we try to aggregate accesses based on a subscript array across an iteration space, e.g., Fig. 4. In this case, an RT_LMAD will be generated. Its operator will be *Expand*. This case cannot be handled by the static framework, and leads to an approximation. The run-time evaluation can be performed in parallel using the LRPD test in either inspector/executor or speculative mode.

(c) Assume we try to aggregate access based on a recurrence across its value set. If the recurrence has a closed form solution that we can express symbolically, we can generate LMADs for the aggregation. However, if a closed form solution is not known, we are forced to represent it as an RT_LMAD with an *Expand* operator. The run-time evaluation is usually an inspector, that may be executed in parallel for some types of recurrences.

Last Value Computation Using RT_LMADs

The problem arises when a loop containing output (write after write) dependencies is executed in parallel. To guarantee correctness, after the execution of the loop every memory location must contain the same value it would have in a sequential execution. Since we have assumed the only possible dependency can be write-after-write, if more than one iteration writes to the same memory location, it is only the last o ne (in sequential order) that actually must be output. All other writes to the same memory location can be ignored. In some cases, even the code that computes them can be deadcoded. We can formalize the problem a little bit. Given a loop that writes locations W_i in iteration `I` we determine the set of memory locations that it writes which will be visible after the execution of the entire loop (sequentially). We call them *OUT* sets, because they are the only ones that need be output.

With RT_LMADs, we can express the output sets for iteration `I` as $OUT_i = W_i \setminus AW_i$, where AW_i is the set of writes corresponding to all iterations after `I`. Mathematically, $AW_i = W_{i+1} \cup W_{i+2} \cup \ldots \cup W_n$. With RT_LMADS's, $AW_i = Expand((J, I+1, N, 1), W_i)$. From the formula, we can see that sets OUT_i are distinct.

We can apply this formula at compile-time or run-time, depending on how hard it is to aggregate sets W_i. If we can do it at compile-time, we can statically dead-code the writes that will not be output and possibly the code computing the values written to them.

At run-time we distinguish two cases: When an inspector is available and speculative execution. If we can generate an inspector to compute RT_LMADs OUT_i, then we can use this RT_LMAD as a mask for the writes in iteration `I`. Since it is computed *a priori*, it will be available before the execution of the real code, so whenever we want to write something in the real code, we can check whether the memory location where we want to write is in OUT_i. The inspector has two attractive features: we can solve the dependencies without additional space (no privatization). Also, we could perform run-time deadcode to avoid computation of values that do not get output. If there is a cycle between address and data computation then we have to execute the loop speculatively in parallel. In this case we have to evaluate the OUT_i on-the-fly and store them in private memory. After loop execution we copy out using OUT_i as the mask for iteration `I`.

2.4 Code Generation

In order to continue analysis at run-time we must define data structures and algorithms in the output language (FORTRAN for Polaris) similar to the ones used in the static framework.

First, we need a run-time representation for lists of LMADs,

```
   DO I = 1, 10
      IF (C(I)) THEN
C        Initialize it to LMAD [1:N]+0
         CALL initialize(RT_LMAD1, 1, N, 0)
      ENDIF

C     Initialize it to LMAD [1:N]+0
      CALL initialize(RT_LMAD2, 1, N, 0)

C     Compute the difference R \ W)
C     Put the result in RT_LMAD2
      CALL difference(RT_LMAD2, RT_LMAD1)
C     Aggregate this iteration to the rest.
      CALL union(RT_LMAD_ALL, RT_LMAD2)
   ENDDO
```

Figure 9: Inspector to aggregate exposed reads across all iterations of a loop.

since they are our operands in all RT_LMADs. We choose a simple representation as two dimensional integer arrays. The first dimension enumerates the LMADs in a list and the second contains the list of pairs (*stride, span* as integers) within every LMAD.

Then, we need to generate code to evaluate the RT_LMADs. First, we need to generate code to initialize the FORTRAN arrays with the operands. Assignment statements are inserted to initialize them with corresponding offsets, strides, spans. Then we need to generate code for the operations. $\cup, \cap, \setminus$ are implemented as calls to a run-time library.

Operation *Expand* needs special attention. Since the expansion by the loop index could not be performed symbolically, there must be a loop variant in the expression of the descriptor being aggregated across all iterations. We need to generate an inspector to compute it. In the example in Figure 8, the expression of Exposed Reads is: $ExposedRead = Expand((I, 1, 10, 1), [1 : N] + 0 \setminus \{C[I]\}[1 : N] + 0)$. The evaluation of this descriptor is the inspector in Figure 9.

Fig. 10 shows the pseudocode for the code generation routine. It takes as input an `RT_LMAD` and a FORTRAN array symbol `A_LMAD`. It generates FORTRAN code for the computation of `RT_LMAD`, so that the result will be stored in the array `A_LMAD`.

Note that when the operator is *Expand*, the first operand is the iteration expression (such as `(I,1,10,1)`) and the second one is the per-iteration RT_LMAD to be aggregated across this iteration space.

We have omitted from our algorithm description the stopping condition, corresponding to the case when RT_LMAD is a leaf node, a list of LMADs. There, we insert code to initialize the FORTRAN arrays with the list of LMADs. When loop variants are used to initialize FORTRAN arrays their definitions must be included in the inspector too. In the worst case (e.g., list traversals) the inspector can be as comlex as the loop itself. In this case, or, when a proper inspector cannot be found (due to cycles between address and data computation) speculative execution is a better solution. The algorithm to generate code for speculative execution is similar to the one for inspectors, with two differences. First, every call to the run-time library is inserted in the real code, after the definitions of the values contained in the operands. The evaluation of *Expand* is done without loop insertion. The aggregation code (the call to the run-time routine for $\cup$) is inserted at the end of the loop.

```
Algorithm CodeGen(RT_LMAD, A_LMAD)
IF (Operator(RT_LMAD) in ∪,∩,\) THEN
   GenCode(Operand1(RT_LMAD), TMP1_A_LMAD)
   GenCode(Operand2(RT_LMAD), TMP2_A_LMAD)
   InsCall(Operator(RT_LMAD),
      TMP1_A_LMAD, TMP2_A_LMAD, A_LMAD)
ELSE
   //Operator is Expand.
   GenCode(Operand2(RT_LMAD), TMP1_A_LMAD)
   InsCall(∪, TMP1_A_LMAD, TMP2_A_LMAD, TMP2_A_LMAD)
   InsLoop(Operand1(RT_LMAD), code generated above)
ENDIF
```

Figure 10: Inspector loop generation with RT_LMADs. TMP1(2)_A_LMAD are arrays to store LMADs, Operand(1,2) retrieve RT_LMAD's operands, InsCall inserts a call to the corresponding RT routine, InsLoop inserts a loop with given iteration-expression which encloses the generated code.

At this point we also generate code to decide whether the loop is parallel. If it is, or speculative execution is needed, then our code will decide when and how much to privatize. The generated code makes use of a run-time library that can handle RT_LMADs and other run-time operations needed for the Recursive LRPD test [3].

2.5 Dynamic Aggregation and Decisions

This run-time time phase continues the aggregation started at compile time and then decides which pre-compiled version of the loop to execute.

The aggregation continues if it was not completed during static compilation. That is, if there are any RT_LMADs to compute. This part will evaluate an RT_LMAD to a final list of LMADs containing actual values (integers). We have implemented operations on LMADs ($\cup, \cap, \setminus$) as calls to a run-time library. The code of the run-time library is a subset of the code used for symbolic operations in the compiler.

The results of RT_LMAD computation (lists of LMADs with actual values) are passed to the decision routines. They are implemented as calls to the run-time library and are semantically equivalent to the ones used to take decisions in the static framework. The only difference is they can only work on integer values (no symbolic computation). The code of the run-time library is again a subset of the code used in the compiler.

We did not use the same code for the compiler and run-time library for two reasons. First, the compiler is written in `C++`, while the code is `FORTRAN`. Second, symbolic manipulation is more general but less efficient, inducing undesirable overhead.

If the parallelization test fails, the initial version of the loop is run sequentially. In case of success, the loop is run in parallel.

2.6 Complexity of Generated Code

The atomic operations performed at run-time aggregation are $\cup, \cap, \setminus$ on lists of LMADs. If the access pattern is linear, then their results are lists of LMADs of the same size as the input lists. For linear accesses the size of the lists does not increase as we perform the aggregation. In the best case, the result of an aggregation will be a single LMAD.

However, for non-linear access patterns such as those using subscript arrays, the aggregation will lead to a large number of LMADs. The worst case is when the number of LMADs equals the number of references, and every LMAD represents a single reference. Then the LMAD representation is not efficient anymore - it takes up more space than it saves. A DOALL test is the better alternative since it works directly with memory references (very little additional space for shadow arrays). The DOALL test is also faster since its operations are significantly faster than LMAD operations.

We are usually somewhere in between. The result of aggregation is a list of LMADs and every one of them represents a set of references corresponding to a linear section of the access pattern. There is a trade-off between the amount of aggregation and the cost of operations on aggregated descriptors. A very interesting case is when the size of the list of LMADs does not increase with the program data set. Then the increase in the data set is reflected only in an increase of the linear sections. That makes our analysis scalable, since our linear section representation (LMAD) will just increase a *span* to represent a larger linear access region. In most cases, the test will be like a DOALL based on LMADs, not on references.

Table 1 presents three classes of complexity. It considers a simple intersection test. It presents the complexity as a function of how the memory reference sets need to be represented: LMADs, LLMADs (lists of LMADs), or lists of references. For the remainder of this paper we assume that d, the dimensiuon of the LMAD is smaller than 3 and thus the complexity of LMAD intersection is $O(1)$.

Type of Test	Complexity
$LMAD1 \cap LMAD2$	$O(2^d)$
$LLMAD1 \cap LLMAD2$	$O(s1 + s2)$
$LRef1 \cap LRef2$	$O(n)$

Table 1: Complexity of Generated Code. LLMAD1(2) are lists of LMADs of size $s1(2)$, n is data set size, d is the dimension of the LMAD

Case study DYFESM presents a situation where two tests of increasing complexity are needed to solve the problem in the general case. The first one is $O(1)$ and the second one is $O(s)$, where s is much smaller than the data set of the application.

Reducing the Complexity of Run-time Analysis

Let us follow an example. The loop in Figure 8 is parallelizable if C(I) is always true. To find out whether TMP is privatizable in the DO I loop, we need to generate an inspector for the memory references. A naive implementation of an inspector would check every reference. There are $2 * 10 * N = 20 * N$ total references.

Using RT_LMADs we take advantage of static aggregation. Since the access pattern in loops DO J can be expressed as an LMAD, we will only have to compare those descriptors at run-time. That can be done by 10 LMAD operations, reducing the computation by a factor of N. LMAD operations have complexity $O(1)$ (with a slightly higher constant factor though).

This is only a direction of reducing inspector complexity due to *redundancy below our analysis level.* Let us assume that the same piece of code is executed many times, with-

```
P = 1
DO I = 1, 10
   IF (C(I)) THEN CALL W(TMP1, 1, N, 1)
   IF (C(I)) THEN CALL W(TMP2, 1, N, 1)
   CALL R(TMP1, 1, N, 1)
   CALL R(TMP2, 1, N, 1)
ENDDO
```

Figure 11:

out redefining the condition array C. We can then hoist the whole inspector up to the point where C gets redefined and compute it only once for all the times the piece of code gets executed. This way, we reduce inspector complexity due to *redundancy above our analysis level.* Now suppose the code looked like in Figure 11: The access pattern on TMP1 is identical to the one on TMP2. We can prove that at compile time. Therefore we can use the results of the inspector for TMP1 and skip analysis for TMP2. This way, we reduce inspector complexity due to *redundancy at our analysis level.* A crude analysis of access pattern similarities on the PERFECT benchmark suite shows about 30% repetition of patterns. This observation also saved us compilation time and space, which becomes very important for JIT compilers.

2.7 Implementation Details

High-level Optimization of Generated Code. In order to reduce overhead in case of failure, we want to be able to quit an inspector or speculative execution as soon as the test fails. We check whether privatization fails at every iteration step when aggregating exposed reads. That way, we avoid executing useless code. More than that, in many cases if privatization fails for an iteration, it fails for all of them. So failure will be detected in the first one.

We must also generate efficient code for RT_LMAD evaluation. The RT_LMAD is viewed as a symbolic expression of lists of LMADs. We can change the order of operations so that *Expand* operations are executed as late as possible. Also, by identifying common subexpressions in our tree representation of RT_LMADs, we lower memory requirements at compile-time (which is actually a big problem) and employ memorization at run-time (thus reduce overhead).

Privatization of Nested Loops. Our static framework offers support for automatic privatization. It tries to compute the set of exposed reads across all iterations of a loop, and see whether they might overlap with a write. If so, the overlapping parts cannot be privatized.

The computation was done as symbolic expansion of the generic per-iteration descriptor of exposed reads across the iteration space. In the following case, it will produce the per-iteration descriptor ϕ, which is expanded across the iteration space to ϕ. That is, the writes cover the subsequent read so TMP is privatizable. This approach solves the case in Figure 12(a).

The process is continued for the next outer loop. The information we have computed here (exposed read) will probably influence decisions at that level. This whole loop becomes a part of an iteration of another loop. However, we will see that this test fails in the case is Figure 12(b) Here we will first try to analyze the exposed read for the DO J loop. We can see that TMP may be read in the second iteration without being written before in the same J iteration. We

```
DO I = 1, 10
   CALL W(TMP, 1,10,1)
   CALL R(TMP, 1,10,1)
ENDDO
```

(a)

```
DO I = 1, 10
   DO J = 1, 2
      IF (J.EQ.1) THEN
         CALL W(TMP, 1,10,1)
      ELSE
         CALL R(TMP, 1,10,1)
      ENDIF
   ENDDO
ENDDO
```

(b)

Figure 12:

conclude that the set of exposed reads for this loop is the union of all exposed reads in its iteration. That will correspond to the access on TMP in the second J iteration. When we start analyzing the DO I loop, we have the information that in every iteration there is an exposed read. Just by looking at the code, we can tell that is wrong.

We have made a wrong assumption when we analyzed the outer loop. The assumption was that the inner loop might be executed in parallel. We did not take advantage of the fact that the inner loop is actually sequential, so its iteration space will be traversed in sequential order. The analysis was not wrong, just overly conservative.

In our new framework we have corrected the analysis. It is easy to express the change using RT_LMADs. That also enables us to perform it at run-time. Consider a generic loop where all writes in iteration I (W_i) happen before all reads in iteration I (R_i). Then if we assume this loop's iterations are executed out-of-order, the expression of Exposed Reads is $(R_1 \setminus W_1) \cup (R_2 \setminus W_2)$. If the loop is executed sequentially, the expression of Exposed Reads is $(R_1 \setminus AW_1) \cup (R_2 \setminus AW_2) \cup \ldots \cup (R_n \setminus AW_n)$, where $AW_i = W_1 \cup W_2 \cup \ldots \cup W_i$. The second one is more accurate while still conservative. In the *Case Studies* section we present the analysis of ADM, a PERFECT benchmark that can only be parallelized automatically using the second technique at run-time.

Program Representation and Code Restructuring. Our internal representation is based on SSA. That allows us to identify efficiently definition sites for values contained in the LMADs.

Our approach is control-flow sensitive. We use the Control Dependence Graph (CDG) to reorganize programs as structured code. The operations involved by restructuring are: remove multiple routine entries, remove premature loop exits (which eliminates non-trivial cycles in the CDG), and clone nodes with multiple parents in the CDG. As a result, the CDG will become a tree, and the code will be completely structured. The only control statements will be IF, DO, WHILE. There are few cases (4 routines in all PERFECT and SPEC codes) where restructuring leads to a large increase in the size of the code (more than 3 times as compared to the original). Even though the CDG is not a tree, it is a DAG with self loops, so we can apply our analysis directly on the CDG (bottom-up top-sort traversal).

The implementation of the Run-time Framework consists of 15,000 lines of code organized as a pass in Polaris.

3. CASE STUDIES

We will present three case studies which are representative of problems encountered in programs from the PERFECT suite.

3.1 ADM:RUN DO 20

This loop (DO 20), and similar loops DO 30, DO 40, DO 50, DO 60, DO 70 could not be parallelized completely automatically before. The special situation arising in these loops has been extracted in Fig. 6. We need to prove that CC is written before it is read. If that happens, than this write will cover subsequent reads in an iteration of the immediately outer loop. That guarantees safe privatization for the outer loops.

rad is a routine that reads the second argument, does some computation and writes the first one from 1 to some M.

There are two problems here: first, the access pattern depends on a non-linear recurrence. Our static representation cannot handle it properly. Fortunately, this is not a problem for the run-time framework. Second, the aggregation scheme for privatization must use the fact that this loop is executed in sequential order. Otherwise we will not be able to exploit the fact that CC gets written in the first iteration. With our old aggregation scheme we would have missed this essential piece of information.

In our representation, the expression of reads for iteration I is R_i = Expand((I,1,N,1), .NOT.NA.EQ.0[1:M]). The expression for writes in iteration I is W_i = Expand((I,1,N,1), NA.EQ.0[1:M]). Let us recall the expression of exposed reads across the whole loop, considering it will be executed sequentially: $(R_1 \setminus AW_1) \cup (R_2 \setminus AW_2) \cup \ldots \cup (R_n \setminus AW_n)$, where $AW_i = W_1 \cup W_2 \cup \ldots \cup W_i$. We will show the evolution of these sets as I traverses the iteration space.

I	NA	R	W	$R_i \setminus W_i$	AW	$R_i \setminus AW_i$
1	0	ϕ	[1:M]	ϕ	[1:M]	ϕ
2	1	[1:M]	ϕ	[1:M]	[1:M]	ϕ
3	0	ϕ	[1:M]	ϕ	[1:M]	ϕ
4	1	[1:M]	ϕ	[1:M]	[1:M]	ϕ

It is clear now that an aggregation of $R_i \setminus W_i$ will not give us the right result (it will give us [1:M]). The aggregation of $R_i \setminus AW_i$ is the right answer (ϕ).

This is our inspector for computing Expand($R_i \setminus AW_i$):

```
na = 1
DO I = 1, N, 1
   NA = NA-1
   IF (NA.EQ.0) THEN
        CALL union(CC_AW,[1:M],CC_AW)
   ELSE
        CALL union (CC_ER,difference([1:M],CC_AW),CC_ER)
   ENDIF
ENDDO
```

CC_AW is the cumulative descriptor of writes, and CC_ER is the cumulative descriptor of exposed reads.

Observations:

- Another approach described in [10] can analyze all the access patterns in the outer loop DO 20 except for this descriptor (CC). This technique can only generate O(1) tests. Any data that is loop variant is approximated to a conservative value in order to simplify descriptors. However, in this case this approach ignores local information decisive to optimizing a large part of the code. In order to parallelize it automatically, [10] found it necessary to peel the loop so that the first iteration gets moved in front of it. That helps in this particular case (the recurrence on NA has this particular form - flips between two values), but will not solve the

```
DO I = 1, Q
  DO K = 1, R
    ...
    CALL geteu(ID, XE, X)
    CALL solvhe( ... , XE)
  ENDDO
ENDDO
```

(a)

```
IF (SYMM.EQ.0) THEN
  DO I=1, N
    XE(I) = ...
  ENDDO
ELSE
  DO I=1, N
    IF (ICOND(I).LT.0) THEN
      XE(I) = ...
    ENDIF
  ENDDO
ENDIF
```

(b)

Figure 13: (a) Loop SOLVH_DO20, (b) Subroutine GETEU

general case. Also, it is not very clear when loop peeling should be chosen.

• Theoretically, we could prove at compile-time that the write happens before the read. For instance, Maple can find out the general formula for this particular recurrence. However, symbolic computations are hard in both execution time and programming complexity using current techniques. As future work, we plan to integrate a state-of-the-art symbolic instrument in Polaris. On the compiler side, that means developing new methods to use the information provided by a symbolic calculator.

3.2 DYFESM:SOLVH

This routine contains a loop (`DO 20`)that takes 12% of the whole program execution when run sequentially.

We will only focus here on a problem encountered when trying to parallelize the outer loop. That is, deciding whether `XE` is privatizable. In `geteu`, it is only written. We want to check whether what is written covers what is read later, in `solvhe`. The read pattern in `solvhe` is easy to obtain: the whole array `[1:N]`. The problem is to show that the write in `geteu` covers the whole array.

The descriptor for the writes in the `THEN` branch is {`SYMM.EQ.0`}`[1:N]`, and the descriptor for the writes in the `ELSE` branch is {`.NOT.SYMM.EQ.0`} `Expand((I,1,N,1),` {`ICOND(I).LT.0`}`[I])`.

In the phase of static aggregation, the analysis of `geteu` will generate information for the call sites. In the analysis of the `DO K` loop in `solvh`, the pattern on `XE` is $R_k \setminus W_k$. The expression of R_k is `[1:N]`, and the expression of write is $W1_k \cup W2_k$, where the first term corresponds to the writes in the `THEN` branch in `geteu`, and the second one to the `ELSE` branch.

For every iteration in the `DO K` loop in `solvh` there will be a computation of $R_k \setminus (W1_k \cup W2_k)$. This will actually be generated as $R_k \setminus W1_k \setminus W2_k$ because we try to perform $O(1)$ tests first, hoping to get a positive answer and skip the evaluation of *Expand* (which is always an inspector). This is the generated code:

```
CALL initialize R(1,N,1)
IF (SYMM.EQ.0) THEN
   CALL initialize W1(region, 1,n)
ENDIF
CALL difference(R, W1, ER)
IF (check_empty(ER)) THEN GOTO 10
IF (.NOT.SYMM.EQ.0) THEN
   DO I=1, N
      IF (ICOND(I).LT.0) THEN CALL union(W2, [I], W2)
      ENDIF
   ENDDO
ENDIF
CALL difference(R, W1, ER)
IF (check_empty(ER)) THEN GOTO 10
C  XE is NOT privatizable
   ...
10 CONTINUE
C  XE is privatizable
```

Even though this inspector seems expensive, it depends only on input data (no recurrences). Since the program reads data only once, the inspector can be hoisted up to the point where data is read from the input file. Thus, it needs to be executed only once per program execution.

An $O(1)$ method (such as [10, 5]) will work in this case only if `SYMM.EQ.0`. They will give overly conservative results in the other case: they will declare the loop sequential even though it can be parallel.

3.3 MDG:INTERF

This routine contains a loop (`DO 1000`) that theoretically can be proven parallel at compile-time. However, an automatic method to parallelize it must be able to make inferences involving arithmetic operations and comparisons.

The piece of code we want to focus on presents a part of an iteration of this loop: a possible write access to a part of an array followed by a possible read. We want to prove that the region in discussion is privatizable, i.e., it is written before it is read regardless of the conditions guarding the memory access. We have renamed some variables and reduced the code substantially to make it easier to understand. However, we did not change the complexity of the problem (for parallelization).

```
OUT = 0
DO K = 1, 10, 1
   IF (R(K).GT.MAX_R) THEN OUT = OUT + 1
ENDDO
DO K = 1, 10, 1
   IF (R(K).LT.MAX_R) THEN VALUE(K) = ...
ENDDO
IF (OUT.EQ.0) THEN
   DO K = 1, 10, 1
       ... = VALUE(K)
   ENDDO
ENDIF
```

In the first loop the objects with radius greater than a given threshold are counted. In the second loop, the ones that are below the threshold get written. In the third loop, they are all read. However, this last loop gets executed only if no objects have radius greater than the threshold. That means, if all of them have radius below the threshold.

In order to prove it statically, the compiler needs to infer that `(OUT.EQ.0)` ⇒ `(R(K).LT.MAX_R,` ∀ `K=1:10)`. Although it is theoretically possible, our static analysis does not have that power.

We choose to generate a run-time test to compute the set of writes `Expand((K,1,10,1), (R(K).LT.MAX_R) [K])`, then the set of reads OUT.EQ.0 [1:10].

In this loop, computing the values of `radius` produces much data as a side effect. Any inspector would have to either recompute or store that data for every iteration of the `DO 1000` loop. A better approach is to execute the loop in parallel speculatively. The code to evaluate these LMADs and possibly bail out in case a test fails, is inserted in the loop body, not in front of it as in inspectors. The loop will look like this:

```
   OUT = 0
   DO K = 1, 10, 1
      IF (R(K).GT.MAX_R) THEN OUT = OUT + 1
   ENDDO
   DO K = 1, 10, 1
      IF (R(K).LT.MAX_R) THEN
         VALUE(K) = ...
         CALL union(W, [K], W)
      ENDIF
   ENDDO
   IF (OUT.EQ.0) THEN
      DO K = 1, 10, 1
         ... = VALUE(K)
      ENDDO
      CALL initialize(R, 1, 10, 1)
   ENDIF
   CALL difference(R, W, ER)
   IF (.NOT.check_empty(ER)) THEN
      RT_LMAD_ERROR = 1
      GOTO 10
    ...
10 CONTINUE
C  Restore modified data and re-execute sequentially.
```

Observation. This particular loop was proven parallel at run-time by other tests ([8]) We have presented it to illustrate how we integrate speculative execution in our framework.

4. CONCLUSIONS AND FUTURE WORK

We have presented a unified framework based on the existing IPA and speculative run-time parallelization technology analysis in the Polaris compiler. This framework is able to analyze efficiently all statically un-analyzable cases in a uniform manner. We use the same technique to extract run-time assertions from any loop and depending on the case at hand, generate run-time tests that range from a low cost scalar comparison to a full, reference by reference LRPD test. We accomplish this by both extending compile time IP analysis techniques and by incorporating speculative run-time techniques when necessary. In essence our solution is to bridge 'free' compile time techniques with exhaustive run-time techniques through a continuum of simple to complex solutions. We have implemented our framework in the Polaris compiler by introducing an innovative intermediate representation (RT_LMAD) and associated run-time operations. It includes intersection, union, last-value assignment, aggregated inspectors and LRPD tests for both dense and sparse reference patterns. We are currently working on the full integration of all our compiler 'passes' in this framework.

5. REFERENCES

[1] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[2] K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. Mc Kinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope Parallel Programming Environment. *Proc. IEEE*, 81(2):244–263, February 1993.

[3] F. Dang and L. Rauchwerger. Speculative parallelization of partially parallel loops. In *Proc. of the 5th Int. Workshop, Languages, Compilers and Run- time Systems for Scalable Computing, Lecture Notes in Computer Science*, May 2000.

[4] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[5] J. Hoeflinger. *Interprocedural Parallelization Using Memory Classification Analysis.* PhD thesis, University of Illinois, Urbana-Champaign, August, 1998.

[6] Yunheung Paek, Jay Hoeflinger, and David Padua. Simplification of Array Access Patterns for Compiler Optimizat ions. In *Proceedings of the SIGPLAN 1998 Conference on Programming Language Design and Implementation, Montreal, Canada*, June 1998.

[7] L. Rauchwerger, N. Amato, and D. Padua. Run–time methods for parallelizing partially parallel loops. In *Proceedings of the 9th ACM International Conference on Supercomputing, Barcelona, Spain*, pages 137–146, July 1995.

[8] Lawrence Rauchwerger and David A. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *Proceedings of the SIGPLAN 1995 Conference on Programming Language Design and Implementation, La Jolla, CA*, pages 218–232, June 1995.

[9] J. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.*, 40(5), May 1991.

[10] Brian R. Murphy Sungdo Moon, Mary W. Hall. Predicated array data-flow analysis for run-time parallelization. In *Proceedings of the 12th ACM International Conference on Supercomputing*, July 1988.

[11] Hao Yu and L. Rauchwerger. Run-time parallelization overhead reduction techniques. In *Proc. of the 9th International Conference on Compiler Construction (CC2000), Berlin, Germany.* Lecture Notes in Computer Science, Springer-Verlag, March 2000.