Hybrid Analysis: Static & Dynamic Memory Reference Analysis

Silvius Rus
rus@tamu.edu

Lawrence Rauchwerger
rwerger@cs.tamu.edu

Jay Hoeflinger
jay.p.hoeflinger@intel.com

January 26, 2002

Technical Report 02-002
PARASOL LAB
Department of Computer Science
Texas A&M University

Abstract

We present a novel **Hybrid Analysis** technology which can efficiently and seamlessly integrate all static and all run-time analysis of memory references into a single framework that is capable of performing all data data dependence analysis and can generate necessary information for all associated memory related optimizations. We use **HA** to perform automatic parallelization by extracting run-time assertions from any loop and generating appropriate run-time tests that range from a low cost scalar comparison to a full, reference by reference run-time analysis. Moreover we can order the run-time tests in increasing order of complexity (overhead) 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. 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 called RT_LMAD and a run-time library that can operate on it. From the experimental results obtained to date we believe that we will be able to automatically parallelize all PERFECT codes, a significant accomplishment.

Keywords: Hybrid Analysis, run-time parallelization, memory reference analysis

1 Introduction

The analysis of memory reference patterns is one of the most important phases performed by an optimizing compiler Memory access analysis is crucial for parallelization and locality enhancement. In many 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 requires inter-procedural data dependence analysis. While procedure inlining removes the requirement for actual inter-procedural analysis techniques, the blow-up in code size makes this approach practical only for small codes. Some true inter-procedural analysis techniques have been developed and incorporated into research compilers such as Parascope [7], SUIF [10] and Polaris [12] among others. In [15] we have developed a framework for inter-procedural 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 First, due to their inability to perform accurate symbolic analysis, there are many practical cases compilers cannot handle. 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 recently due to the dynamic nature of modern simulations and the ascendancy of Java. Many optimizations, most notably parallelization, often cannot be performed at compile time because the access pattern is either too complex for current data dependence algorithms or is simply statically unavailable. The challenge posed by the dynamic nature of these modern (and some older) applications has been addressed in recent 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) [19, 6] or partial parallelism [18].

There is, however, a fairly clear separation between 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. To date, there is 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 [25, 22] but with limited applicability.

In this paper we present a new *Hybrid Analysis (HA)* system, which represents a unified

```
subroutine FOO(A,N)
integer A(100)
Do j=1,N
  A(j) = A(j+40)
Enddo
Return
```

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

Figure 1:

inter-procedural framework that can seamlessly integrate compile-time and run-time analysis. This integrated IPA system minimizes run-time overhead by performing as much analysis as possible statically, and then passing the partial results for the statically indeterminate cases to the run-time system.

1.1 State of the Art vs. IP Hybrid Analysis

In this section we present several examples extracted from real codes which, for the most part, could not be analyzed by the most advanced commercial and research compilers. We contrast the current conservative results with those possible using our **Hybrid Analysis (HA)** framework. The main idea is that when the compiler cannot complete its analysis statically, it should generate code that starts where the static analysis left off and finishes the analysis at run-time. This combination of static and dynamic analysis can generate aggressively optimized code with minimum run-time overhead. We henceforth refer to this approach as **Hybrid Analysis (HA)**.

In each of the following cases we sketch the 'ideal', minimum run-time overhead check and also sketch our own results. We further briefly present previous work in this domain and then summarize our main contributions.

In the pseudo-code example in Fig. 1. the value of N is unknown at compile time and thus neither the loop in subroutine FOO nor the outer loop in MAIN is statically parallelizable. However, a simple dependence test of the loop in FOO can generate a run-time assertion to check if the maximum write offset is smaller than the minimum read access. This assertion can then be propagated through the outer loop in MAIN up to the point where N is read in. The run-time check takes the form of *simple scalar comparisons* between N and 40 and N and 100.

Fig. 2 shows a more complex situation for both inner and outer loops. Because the values of C(i) in FOO are unknown, we cannot conclude whether array TMP is privatizable, i.e., whether all uses of TMP are covered by a write for every iteration. The privatization transformation is input dependent. In this situation most compilers known to us take the conservative approach of issuing sequential code. However, a compiler could collect loop level information, propagate it to the MAIN program level and then generate a run-time assertion to analyze the contents of the crucial array $C(I)$. The outcome of this analysis will decide whether TMP is privatizable and thus if the loop in MAIN is parallel. As we

```
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(:),L,lim
Do k=1,L
  Call FOO(C,TMP,N,M,lim)
Enddo
```

Figure 2:

```
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
Do k=1,L
  Call FOO(C,TMP,N)
  ... = TMP(k)
Enddo
```

Figure 3:

will show later, in this particular example, such a run-time assertion is equivalent to the generation of an inspector loop of the array C. Its results can then be reused (the technique is known as 'schedule reuse' [21]).

Fig. 3 shows a typical example when arrays (TMP in this case) are indexed indirectly (subscripted subscripts). Because the contents of index array C are input dependent, classic static compilation generates sequential code. However, instead of being conservative a compiler using our proposed **hybrid analysis (HA)** can predicate the parallelization of the main loop to the contents of array C and generate the appropriate run-time assertion (to compare the contents of C to the interval [1,L]). The result of the run-time analysis can be reused because C and L are loop invariant.

The loop in Fig. 4 also uses subscripted subscripts but, in contrast to the previous example, it computes them within the loop. This produces a dependence cycle between address and data computation and prevents the compiler from inserting a run-time test before the loop. However, our **HA** framework still isolates the fact that *only* the contents of C need to be analyzed at run-time. In our work [19] we show that speculative parallelization followed by run-time analysis is probably the only viable solution for such cases.

Finally, Fig. 5 illustrates an example of a loop where privatization is conditioned by the values taken by a recurrence that has a closed form solution (the 'flip variable' `NA`) in addition to the values taken by `L(:)`. The 'ideal' **HA** output for `NA` would be a run-time test for the initial value taken by `NA` in the inner loop. As we will later show our compiler is not yet powerful enough to compute the closed form solution of the recurrence (Mathematica can though) and we will generate a small inspector loop that will verify all values taken by NA.

```
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
Do k=1,L
  Call FOO(C,TMP,N)
  ... = TMP(k)
Enddo
```

Figure 4:

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. Other simple scalar run-time checks verifying parallelization have been used in [22, 12].

In later years various techniques for run-time parallelization and partial redundancy elimination have been proposed. Saltz [21] and, later Rauchwerger [19, 18] 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 [25].

Inter-procedural analysis is present in many research compilers [23, 8, 13, 9, 3, 10, 11, 20]. In Polaris [2] IPA has been reported in [15, 12].

The use of run-time assertions within the IPA context is a recent development and has been reported in SUIF [22] and Polaris [12]. Both SUIF and Polaris generate relatively simple low cost assertions and are capable of solving problems similar to those shown in Fig. 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 [22] the technique is based on a predicated data flow analysis with emphasis given to combining predicates that affect parallelization conditions. The result is low cost run-time assertions with multi-version loops. When predicates are loop variant, sequential, and scoped only within a lower level procedure, then conservative assumptions are made. This has the advantage of simplifying compile time analysis but does not significantly reduce run-time overhead. In [12] the infrastructure is based on predicated memory access descriptors, LMADs, which aggregate memory references and thus simplify static analysis.

At the other end of spectrum in [25] the authors start from exhaustive run-time parallelization techniques (the LRPD test [19], which assumes no partial compiler information) and reduce the complexity through predicated reference aggregation and logical implication. Their emphasis is on cases similar to those in Figs. 3–5 which generally require speculative execution. Their results are good but do not extend beyond the procedural level.

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

```
subroutine RUN(W, D)
integer W(*), D(N,*)
Do i=1, 1024
  Call RFFTF(W,D(1,i))
Enddo

subroutine RAD(CC,CH,L)
integer CC(*), CH(*), L
common /input/ G(1024)
Do i=1,L
  CC(i) = CH(i) + G(i)
Enddo
```

```
subroutine RFFTF(CC,CH)
integer CC(*), CH(*)
common /length/ L(10)
NA = 1
Do j=1, 10
  NA = 1-NA
  If (NA.EQ.0) then
    Call RAD(CC,CH,L(j))
  else
    Call RAD(CH,CC,L(j))
  endif
Enddo
```

Figure 5:

We should note that the value based analysis reported in [14] which successfully parallelizes some important loops is complementary to this work. It can disambiguate some special cases of indirect addressing at compile time where our analysis generates run-time tests.

1.2 Our Contribution

Our main contribution is the **Hybrid Analysis** technology which can efficiently and seamlessly integrate all static and all run-time analysis of memory references into a single framework. This framework is capable of performing all data data dependence analysis and can generate necessary information for all associated memory related optimizations. We have used **HA** to perform automatic parallelization by extracting run-time assertions from any loop and generating appropriate run-time tests that range from a low cost scalar comparison to a full, reference by reference run-time analysis (e.g., the LRPD test). Moreover we can order the run-time tests in increasing order of complexity (overhead) 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.

We implement our framework in the Polaris compiler by introducing an innovative intermediate representation called RT_LMAD and 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. Additionally, we augment the GSA representation in Polaris for inter-procedural use. Our analysis is flow sensitive on any control flow graph. From the experimental results obtained to date we believe that we will be able to automatically parallelize all PERFECT codes, a significant accomplishment.

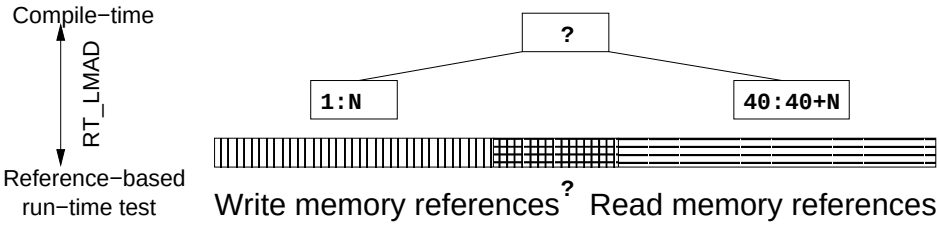

Figure 6: Compile-time to Run-time transition

2 Hybrid Memory Access Analysis

Static analysis of memory access patterns, essential to many optimizations and especially to parallelization, is efficient because it uses time and memory proportional to the size of the program. It aggregates all references in a symbolic representation, performs operations on them, and draws conclusions about their shape, size and other characteristics. In contrast, 'pure' *run-time analysis* of memory access patterns, is proportional to the dynamic number of individual memory references and is therefore a source of (scalable) inefficiency. Dynamic references have been mostly represented through enumeration of either all or, in some cases, all unique, memory references.

In *hybrid analysis*, performed both at compile and at run-time, the goal is to obtain definitive answers with a time and memory budget as close as possible to that of static analysis. For this, we need a flexible memory access descriptor that can efficiently handle any memory reference pattern from inefficient enumerated lists to almost fully aggregated symbolic expressions. Such a flexible representation and its associated memory and analysis complexity seamlessly bridge static and dynamic analysis. The border between what occurs at compile time and what occurs at run-time depends to a large extent on the power of current compiler algorithms and, with their continuous improvement, can be smoothly shifted towards better performance and less overhead.

Fig. 6 shows the two extremes that run-time analysis of memory reference sets can span for the code in Fig. 1. To prove the loop parallel, the analysis must show that the intersection of the read and write memory reference sets is empty (true for `N<40`). At one extreme (bottom), we have a run-time test that analyzes every memory access and checks if any read and write overlap. This is a bulletproof test, but it can be costly as it is proportional to the *dynamic* number of memory reference instances. At the other extreme (top), we have a compile-time decision. If the value of `N` is known at compile-time, this decision could be made then. If `N` is read from an input file then the intersection of the `READ` and `WRITE` memory reference sets cannot be computed at compile-time and the conservative compile-time decision is not to parallelize. Moving from the bottom to the top, we aggregate the memory references into symbolic sets, the number of symbolic sets decreases and the generated run-time test will be based on sets rather than individual references.

While the ideal situation would be aggregate all the way to the top and have a compile-time decision, in this particular case this is not possible because we lack the crucial informa-

tion about the value of N. We need to stop at the second level and postpone the rest of the evaluation to run-time. The complexity of the run-time test is $O(1)$ (only one intersection left for run-time evaluation). To enable this smooth transition between compile time and run time we introduce a new structure, the RT_LMAD. We will now present its structure and usage in the realistic context of complex control flow, recurrences and procedure calls.

2.1 Memory Access Representation:RT_LMAD

The RT_LMAD, or *Run-time Linear Memory Access Descriptor*, is a symbolic and compact representation of memory reference sets in a program. It can represent symbolically the aggregation of array memory references at any hierarchical level (on the loop and procedure call graph) in a program. It can represent the control flow (gates), inter-procedural issues (call sites) and recurrences (when array references, i.e., indices or gates, have to be expressed symbolically as a recurrence with no closed form solution). Its evaluation and subsequent optimization decisions can be:
(a) initiated and completed at compile time if all symbolic values can be analyzed, compared, (b) initiated and completed at run-time - with associated run-time overhead, or (c) initiated at compile time with partial but insufficient results and completed at run-time.

Classic, compile time array reference representations have been documented in the literature, e.g., [17, 10, 12]. For practical reasons but *without any loss of generality* we have chosen to implement the RT_LMAD as a generalization of the Linear Memory Access Descriptor (LMAD) [12].

The RT_LMAD is a symbolic expression with essentially the same operands as the LMAD and operators which cannot be performed at compile time.

The original LMADs described in [12] can represent aggregated memory reference sets but have some limitations given by control flow, recurrences, and subprogram calls and, in general, symbolic evaluation. Thus, the primary representation is not closed with respect to several operations required by an aggregation process. The use of LMADs at compile time gives insufficient information to make a safe and accurate optimization (in particular, parallelization) decision for several fundamental and/or practical reasons: (a) some needed symbolic values correspond to program input values which are not available statically, (b) relations which could be evaluated statically require theorem proving capabilities not yet available, or (c) relations that are prohibitively expensive in time or space (memory) to evaluate symbolically, e.g., Presburger formulae of time complexity $O(2^{2^{2^N}})$ and LMADS of exponential storage complexity.

For completeness, we first briefly describe LMADs, and then present in detail the more general RT_LMAD.

2.1.1 Linear Memory Access Descriptors (LMADs)

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. Consider a loop nest of depth D with indices $I_k, k = 1, D$, where $I_k = 0, U_k$ (any loop can be brought to this form). Consider a reference to memory given by $\mathrm{A}(s_1(\vec{I}), s_2(\vec{I}), \ldots, s_m(\vec{I}))$, where $\vec{I}$=$(I_1, I_2, \ldots, I_d)$. If the subscripting function can be written in a sum-of-products form with respect to the individual loop indices,

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

then, we can isolate the effect of each loop index on the subscripting offset sequence.

A Linear Memory Access Descriptor (LMAD) is a representation of the subscripting offset sequence [12]. *It can be built for any array reference whose subscript expressions can be put in the form of Equation 1.* 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*, and the number of iterations in the loop. The LMAD contains a starting value, called the *base offset* and a set of dimensions. For loop `DO j` in Fig. 1, the `Read` pattern on array `A` is represented by a 1-dimensional LMAD, $40 + [1 : N - 1]$. The offset is 40, the stride of the single dimension is 1 and the iteration count is N.

The LMAD representation defined in [12] cannot represent a variety of complex operations. LMADs are not closed with respect to: (a) set operations $\cup$, $\cap$, and $-$ (closure for $\cup$ is achieved in LMAD-only representation by using lists of LMADs instead of individual LMADs), (b) the aggregation of memory references per iteration to the loop level if the pattern depends on a recurrence with no closed form solution, (c) the representation of gated references if the gating expression is a recurrence with no closed form solution, or (d) the aggregation of the LMADS containing local variables to their outside scope (outer procedure).

In most cases LMAD-only based analysis overcomes these shortcomings by approximating the results of an operation with a conservative LMAD. For example, sections of arrays are approximated with an the entire array. Sometimes it is possible to generate a series of possible results of LMAD operations that are true under certain conditions which can be verified at run-time. There is however no clear way to generate these 'truth' conditions and/or or to limit their number. It may in fact lead to an exponential growth of the number of (gated) LMADs. (Instead of propagating one resulting LMAD we have to propagate an entire list of LMADs with their existence conditions).

2.1.2 RT_LMAD Definition

The RT_LMAD representation provides closure to the LMAD operators by introducing a symbolic representation of run-time executable operations. This enhanced representation can then be propagated through the program hierarchy without creating an exponential growth in access descriptors. Moreover, operations between RT_LMADs can lead to statically fully analyzable situations which in fact remove the need for run-time operations (e.g., they cancel out).

$T = \{\cap, \cup, -, (,), \#, \otimes, \bowtie, LMADs, Gate, Recurrence, CallSite\}$
$N = \{RT\_LMAD\},\ S = RT\_LMAD$
$P = \{RT\_LMAD \rightarrow LMADs | (RT\_LMAD)$
$RT\_LMAD \rightarrow RT\_LMAD \cap RT\_LMAD$
$RT\_LMAD \rightarrow RT\_LMAD \cup RT\_LMAD$
$RT\_LMAD \rightarrow RT\_LMAD - RT\_LMAD$
$RT\_LMAD \rightarrow RT\_LMAD \# Gate$
$RT\_LMAD \rightarrow RT\_LMAD \otimes Recurrence$
$RT\_LMAD \rightarrow RT\_LMAD \bowtie CallSite\}$

Figure 7: RT_LMAD formal definition.

More formally, we can define the RT_LMAD with the grammar in Fig. 7. It is based on the primary static LMAD representation generalized through the addition of the *gate*, *recurrence*, and *subprogram call site* terminals.

In addition to the binary operators on sets: intersection ($\cap$), union ($\cup$), difference ($-$), the terminals set contains operators for guarding a descriptor with a gate ($\#$), quantifying a per-iteration set across all iterations of a recurrence space ($\otimes$), and translating a descriptor across different routines ($\bowtie$). The set of operands is made of lists of $LMADs$, $Gates$ (as conditionals), $Recurrences$ (as annotated statements), and $CallSites$ (as annotated statements). Throughout the rest of the paper, we will use the set theory notation ($\bigcup_{j=1}^{N} X_j$) as an alternative of the RT_LMAD equivalent ($X_j \otimes (j = 1, N)$) to make it easier to read formulae containing recurrences.

Let us illustrate the use if RT_LMADs by following the example in Fig. 5. The WRITE access pattern for array `W` in routine `RUN` corresponds to array `CC` in `RFFTF`. `CC` is not modified directly in `RFFTF`, but is passed to routine `RAD`, first as `CC`, then as `CH`. Routine `RAD` writes only `CC`, with a pattern easy to represent as an LMAD, $[1 : L]$. This pattern can be translated at the first call site in routine `RFFTF` as $[1 : L(j)]$. The second call site to `RAD` cannot generate any WRITE access to `CC`, as it is passed inside as `CH`. In order to summarize the access pattern for an iteration of `DO j=1,10` in `RFFTF`, we need to guard this LMAD with a gate, corresponding to the condition `NA.EQ.0`. The problem arises when we want to summarize the access pattern over all iterations of `DO j=1,10`. Let us assume that our compiler does not recognize `NA` as an induction variable (as is the case in Polaris). Since `NA` is loop variant, and has no known (to Polaris) closed form, it is not possible to represent it through a LMAD. Using RT_LMADs, we will represent it as: $([1 : L(j)] \# (NA.EQ.0)) \otimes (j = 1, 10)$. We also notice that `NA` is local to `RFFTF`. Since we need the value of the WRITE pattern on `W` in `RUN`, we need to translate it into its context. We represent it symbolically as

$$(([1 : L(j)] \# (NA.EQ.0)) \otimes (j = 1, 10)) \bowtie CallRFFTF(...)$$

This notation corresponds to an expression having objects from the primary representation (LMADs) and language elements (gates, recurrences, subprogram call sites) as operands.

The operators correspond to the limitations of the primary LMAD representation. Fig. 8 presents the same expression viewed as a tree.

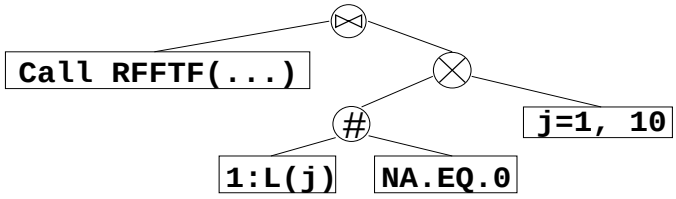

Figure 8: Write descriptor for array `W` for an iteration of loop `DO i=1, 1024` in routine `RUN`, Fig. 5.

We define **RT_LMAD Inputs** corresponding to the pair (memory reference set, code region) as the set of values, which upon entry to the code region, define the shape and size of a RT_LMAD. An RT_LMAD is loop invariant if and only if its *inputs* at the beginning of the loop body are loop invariant. This holds regardless of the complexity of the RT_LMAD (even if it contains variables that are not defined in the current routine). For the RT_LMAD in Fig. 8, the set of *inputs* is `L(j), j=1,10`. Since `L` is defined outside the loop, the whole descriptor is loop invariant in loop `DO i=1,1000`.

2.2 Program Analysis using RT_LMADs

Our HA framework is integrated into the Polaris compiler. A prepass brings the program into GSA form[1] and builds the program's control dependence graph (CDG) [4] and Call graph. We also compute control dependence regions (CDRegions) as defined in [5, 1]. Since our subsequent analysis assumes these two graphs are acyclic,[2] we ensure the CDG is acyclic by transforming any loops with premature exits into `WHILE` loops with exactly one exit. The call graph is assumed acyclic (Fortran does not allow recursive procedures).

The HA pass then performs a generic *program traversal* that traverses the CDG in reverse topological order, except that within a CDG region all nodes are ordered exactly as in the original program. Across routines, it uses a reverse topological sort of the Call graph. As the traversal visits a CDG node, it is processed as follows (see Fig. 9). First, in `GetPrimaryInfo`, we classify each memory reference statement associated with the CDG node. Next, memory reference(s) are aggregated with the appropriate RT_LMADs and then are added to the CDRegion containing the CDG node. Finally, the `PostAction` will analyze desired optimizations (such as parallelization) and will generate appropriate code.

We should note that the driver is generic in two ways because it does not specify what information to collect nor does it force us to respect a certain traversal order. We can collect different information or we can analyze and aggregate in a different order without having to

[1] We extend the existing Polaris GSA to represent inter-procedural use-def chains. We added a new class of ϕ functions at routine entries for every formal parameter and common variable used in the routine.

[2] Except for self-loops in the CDG.

```
ALGORITHM AnalyzeCDGNode(CDGNode)
CALL PreAction(CDGNode)
Info← GetPrimaryInfo(Statement(CDGNode))
CASE CDGNode OF
 Recurrence: Info← AggregateRec(Rec,CDRegion(T))
 Conditional:Info← AggregateCond(Cond,CDRegion(CDGNode,T)
             CDRegion(CDGNode,F))
 Call Site:  Info←AggregateTrans(CallSite,CDRegion(Callee))
CALL AggregateToRegion(CDRegion(CDGNode),CDGNode,Info)
CALL PostAction(CDGNode)
```

Figure 9: **HA** driver core: aggregation and actions. (T=TRUE, F=FALSE) at CDG node level

rewrite the entire compiler pass. We will now present in more detail the various algorithms invoked by the HA driver.

2.2.1 Reference Classification and Aggregation

We define a *summary set* as a symbolic description of a set of memory references (locations). We use RT_LMADs to represent memory accesses within a summary set. We define three types of summary sets: ReadOnly (RO), WriteFirst (WF) and ReadWrite (RW). The RO summary set records all read-only memory locations within a section of code, the WF summary set records all write-first memory locations and the RW summary set all other memory locations referenced in a code section.

Thus, we now detail the previously presented generic program traversal by explaining how we collect *summary sets* while walking up the CDG and Call Graph.

`AggregateCond` generates the summary set for an `IF` block. It guards the sets for the CD region on the `TRUE` branch with the condition in the `IF` statement, and the `FALSE` branch with the negated condition, then unites them. The operation is performed separately for WF, RO, and RW. [3] `AggregateTrans` generates the summary set for a subprogram (subroutine or function) call site. It takes as input the summary sets for the called subprogram and applies the translation operator corresponding to the given call site to WF, RO and RW respectively. Fig. 10 describes the other aggregation algorithms. `AggregateToRegion`(a) aggregates the summary set of a node to the summary set associated with the CD region. It assumes that the order in which the aggregation occurs is the same as in the input code.

When aggregating the summary set for a recurrence given the summary set per iteration (Fig. 10 (b)) we have the choice of either assuming no order between iterations or making an assumption about the type of parallelism used. For example, when aggregating an inner loop to the outer loop we can either assume that all iterations of the loop nest can be executed concurrently (one level parallelism of the coalesced loop nest, as in [12]) or, we

[3] There can be nodes in the CDG of a FORTRAN program with more than two branches, resulting from statements such as `COMPUTED GOTO`. We have replaced them during a preprocessing phase with semantically equivalent `IF` blocks.

ALGORITHM AggregateToRegion($CDRegion$, $CDGNode$, $Info$)
$(WF_1, RO_1, RW_1) \leftarrow CDRegion$
$(WF_2, RO_2, RW_2) \leftarrow Info$
$WF = WF_1 \bigcup (WF_2 - (RO_1 \cup RW_1))$
$RO = (RO_1 - (WF_2 \cup RW_2)) \bigcup (RO_2 - (WF_1 \cup RW_1))$
$RW = RW_1 \bigcup (RW_2 - WF_1) \bigcup (RO_1 \cap WF_2)$
$CDRegion \leftarrow (WF, RO, RW)$

(a)

ALGORITHM AggregateRec(Rec, $CDRegionTrue$)
$(WF_j, RO_j, RW_j) \leftarrow CDRegionTrue$
$WF = \bigcup_{j=1}^{N} (WF_j - \bigcup_{k=1}^{j-1} (RO_k \cup RW_k))$
$RO = \bigcup_{j=1}^{N} (RO_j - \bigcup_{k=1}^{j-1} (WF_k \cup RW_k))$
$RW = (\bigcup_{j=1}^{N} (RW_j - \bigcup_{k=1}^{j-1} WF_k)) \cup (\bigcup_{j=1}^{N} (WF_j \cap \bigcup_{k=1}^{j-1} RO_k))$
RETURN (WF, RO, RW)

(b)

Figure 10: Summary Sets Aggregation Algorithms

can assume that the outer loop sees the inner loop as having executed sequentially. Because most modern parallel machines support nested parallelism we view every loop as executing its iterations sequentially (on its own cluster). This assumption allows the aggregation algorithm to consider the parallelism of each loop level assuming the enforcement of all data dependences for the inner nest. This strategy will allow more aggressive parallelization.

Assuming no order in inner loops, then in the example from Fig. 5 the equation for quantifying RO across recurrence $j = 1, N$ would be $RO = \bigcup_{j=1}^{N} RO_j$. If we apply this equation to inner loop `DO j` in routine `RFFTF` then we will find the outer loop `DO i=1, 1024` in routine `RUN` sequential. When assuming a sequential order for the inner loop the equation for RO becomes $RO = \bigcup_{j=1}^{N} (RO_j - \bigcup_{k=1}^{j-1} (WF_k \cup RW_k))$, and the outer loop is found parallel.

The complex formulae for aggregating reference sets across recurrences, $(RO = \bigcup_{j=1}^{N} (RO_j - \bigcup_{k=1}^{j-1} (WF_k \cup RW_k)))$, seems to have quadratic complexity $(j = 1, N \wedge k = 1, j-1)$. We reduced its complexity in the code generation phase Section 2.2.3, by introducing the concept of *Lower Partial Union*. Any term $\bigcup_{k=1}^{j-1} X_k$ that shows in a `DO j` recurrence is a Lower Partial Union. The idea is to compute the lower partial unions at the same time we compute the other descriptors to aggregate on, which yields linear complexity (analogous to partial sum computation).

Fig. 11 presents the WF component of the summary set obtained by summarizing the memory access shown in Fig. 5 on array `W` in an iteration of loop `DO i=1,1024` routine `RUN`.

2.2.2 Parallelism Detection (HA: PostAction)

In this paper, we focus on global program analysis using RT_LMADs for extracting course grain parallelization. The decision when a loop can be executed as a DOALL is mainly con-

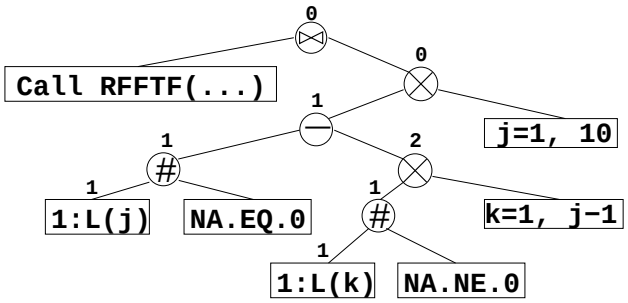

Figure 11: WriteFirst descriptor for array `W` in in an iteration of loop `DO i=1,1024`, routine `RUN` in Fig. 5.

```
ALGORITHM ClassifyRec(Rec, SymbolList)
FOR ALL Symbol IN SymbolList DO
(dep_fa, dep_o, is_red) ← (F, F, F)
    (s_dep_fa, s_dep_o, s_is_red) ←
       ClassifySymbol(Rec, Per Iteration Info (Symbol))
    dep_fa = MAX(dep_fa, s_dep_fa)
    dep_o = MAX(dep_o, s_dep_o)
    is_red = MAX(is_red, s_is_red)
RETURN (dep_fa, dep_o, is_red)

ALGORITHM ClassifySymbol(Rec, Per Iteration Info)
(WF_j, RO_j, RW_j)← Per Iteration Info
(dep_fa, dep_o, is_red) ← (F, F, F)
IF (CheckReduction(Symbol, Rec)=T) THEN RETURN (F, F, T)
dep_fa = DEP((⋃_{j=1}^{N} WF_j) ⋂ (⋃_{j=1}^{N} RO_j))
dep_fa = MAX(dep_fa, DEP((⋃_{j=1}^{N} WF_j) ⋂ (⋃_{j=1}^{N} RO_j)))
dep_fa = MAX(dep_fa, DEP((⋃_{j=1}^{N} WF_j) ⋂ (⋃_{j=1}^{N} RW_j)))
dep_fa = MAX(dep_fa, DEP((⋃_{j=1}^{N} RW_j ⋂ (⋃_{k=1}^{j-1} RW_k))))
dep_o = DEP(⋃_{j=1}^{N} WF_j ⋂ (⋃_{k=1}^{j-1} WF_k))
RETURN (dep_fa, dep_o, is_red)
```

Figure 12: Parallelization of Classification Algorithms

ditioned on proving that memory references across iterations do not cause data dependences. If only anti or output dependences are discovered then the privatization transformation can remove them. If flow dependences are present only in a reduction operation then known reduction parallelization algorithms can be used.

The loop parallelization detection algorithm takes as input the summary set for a loop and produces a triplet of values that represents the static decision about the loop: there are loop carried flow or anti dependences (dep_fa), output dependencies (dep_o), and this loop is a reduction (is_red). Because this triplet is expressed symbolically, we used a ternary logic with the values *NO, MAYBE* and *YES* in this (self-imposed) order.

Algorithm `ClassifyRec` (Fig. 12) finds the decision triplet by accumulating decisions taken per symbol referred in the loop. The usage of operator MAX reflects that a loop carried dependency for a single symbol is enough to classify the loop as serial. Function $DEP(X)$ returns *YES* if RT_LMAD X is not empty, *NO* if it is empty, and *MAYBE* if the decision cannot be taken at compile-time. `CheckReduction` is a pattern matching based reduction recognition algorithm.

For every symbol we also compute: individual (per element) privatizable memory areas per iteration `j` (WF_j); last value set to be output by iteration `j` ($WF_j - (\bigcup_{k=j+1}^{N} WF_k)$); signal possible false sharing if $\|WF_j - WF_{j+1}\| \leq CacheLineSize$.

A loop is declared statically parallelizable if dep_fa is NO, and statically serial if dep_fa is YES. If dep_fa is *MAYBE*, a run-time test is generated. The per-symbol value of dep_o specifies which symbols have an output dependence and the value of WF_j gives the precise location for a symbol (in an array) and per iteration `j` that would need to be privatized. A dep_o value of *MAYBE* requires a run-time test.

At the end of the analysis, a detailed report is presented for every symbol and for the whole loop. For every symbol, it presents all loop carried dependencies, their source (e.g., RO over WF), and their extent. Every dependency has a ternary value associated with it. A value of *YES* is followed by an RT_LMAD reduced to an LMAD list, and a value of *MAYBE* by an RT_LMAD that cannot be reduced to a list of LMADs.

2.2.3 Code Generation

This section describes how the run-time tests are generated with an algorithm semantically identical to the classification algorithm presented above. The only differences are that the decisions use boolean rather than ternary logic and the sets are represented as lists of LMADs rather than RT_LMADs.

We generate code to implement the `ClassifySymbol` and `ClassifyLoop` algorithms. The loop over symbols in *ClassifyLoop* is unrolled. Only code to compute `MAYBE` valued variables is needed, as the others are already known at compile-time. This phase is implemented as a simple code insert. The code generated to implement the function `DEP` is a call to a run-time library routine that checks whether a given list of LMADs is not empty.

The remainder of this section focuses on generating code to evaluate an RT_LMAD. The process starts by finding the program slice [24] that computes the symbolic values referred

to by the RT_LMAD. In an inspector-executor model of execution, the slice is cloned interprocedurally and the structures within the RT_LMAD are updated to point into the clone rather than to the original code. This unifies the rest of the code generation phase for both inspector-executor and speculative execution models.

RT_LMADs are expressions whose operands are lists of LMADs. We use a register-based expression evaluation of the RT_LMADs. The "registers" are actually FORTRAN arrays organized logically as lists of LMADs which we will call Run-time Registers (*RTRegs*). The code generation algorithm takes as input an RT_LMAD and an RTReg (to store the result) and traverses in postorder the RT_LMAD tree structure. At every node it first calls itself recursively to generate code to evaluate the children RT_LMADs and to store their values in temporary RTRegs. Then it generates code to merge the contents of these RTRegs into the final result. The merge depends on the RT_LMAD node type. Terminal nodes are lists of LMADs so they are inserted directly into the result. For binary operations ($\cup$, $\cap$, $-$), it inserts a call to the corresponding run-time library. For gates, it inserts an `IF` statement around a call to a routine to copy the RTReg for the child into the result. For translations, it inserts a call to a run-time library routine that adds the actual offset in the `CALL` statement to the descriptors of the called subprogram (e.g. the `10` in `CALL foo(A(10))`). For recurrences, it uses the result RTReg as an accumulator, creates a loop equivalent to the recurrence it represents, inserts code to accumulate the value of the child for every iteration, and inserts code to empty the accumulator before the loop. The definition of the result RTReg is placed after the immediate common postdominator of all the definition sites of the values referred to by the RT_LMAD. This way we make sure that the children are always available when needed for the computation of the parent.

Lower partial unions (such as $\bigcup_{k=1}^{j-1} X_k$, where $j = 1, N$) are evaluated within the defining recurrence (e.g., `DO j=1,N`). The only difference from computing normal recurrence aggregations is that the accumulation code is inserted at the very end of the loop, after all the uses of the accumulation register to ensure that the value used for the lower partial union is $\bigcup_{k=1}^{j-1} X_k$, and not $\bigcup_{k=1}^{j} X_k$. This leads to linear time complexity for terms such as the one in Fig. 11.

Fig. 13 shows the code generated to evaluate the RT_LMAD of Fig. 11. The complexity of this test is linear in the size of array `L`, and in fact, the entire. parallelism test has this same complexity since the values stored in `L` determine whether the loop is parallel.

Optimizations

We use several methods to reduce the run-time evaluation of RTLMADS. We identify equivalent RT_LMADs and feed the code generator with hints on opportunities for reusing values, and we select evaluation orders to minimize the decision time (e.g., when evaluating an intersection, evaluate the smallest term first and test whether it is empty). Many other optimizations are performed at the run-time library level, e.g., contiguous aggregation, coalescing and interleaving introduced in [15] as compile time optimizations..

The most effective transformation that reduces the cost of an RT_LMAD is hoisting

```
CALL MakeEmpty(RTReg_0, RTReg_2)
NA=1
DO j=1, 10
   NA=1-NA
   IF (NA.EQ.0) THEN CALL SetLMADs(RTReg_1, 1, L(j))
                ELSE CALL MakeEmpty(RTReg_1)
   CALL Subtract(RTReg_1, RTReg_2, RTReg_1)
   CALL Union(RTReg_0, RTReg_1, RTReg_0)
   IF (NA.NE.0) THEN CALL SetLMADs(RTReg_1, 1, L(j))
                ELSE CALL MakeEmpty(RTReg_1)
   CALL Union(RTReg_2, RTReg_1, RTReg_2)
ENDDO
```

Figure 13: Code generation for RT_LMAD displayed in Fig. 11. (The number above a node in Fig. 11 corresponds to the RTReg used to store its value at run-time.)

nodes that are invariant in a recurrence (e.g., $\bigcup_{j=1}^{N} A \rightarrow A$) from the inside of a loop (nest) to the level of the one considered for parallelization. The reduction is by a factor (the size of the recurrence space). There are RT_LMADs that contain parts that are recurrence invariants, but they are not invariant as a whole. We will call them partial invariants. We can still apply hoisting if they are of one of the forms $\bigcup_{j=1}^{N}(A_j \cup B) \rightarrow (\bigcup_{j=1}^{N} A_j) \cup B$ or $\bigcup_{j=1}^{N}(A_j \cap B) \rightarrow (\bigcup_{j=1}^{N} A_j) \cap B$ by pattern matching. We have shown here just operations involving union and intersection. Similar transformations are valid for set difference.

Inspector hoisting and schedule reuse were introduced by [21]. Using RT_LMADs it is very easy to express conditions for both these optimization schemes. Consider the *inputs* set associated with the RT_LMAD that represents the dependent memory area and with the code region it was aggregated over. The dependence test can be hoisted up to the immediate common postdominator of all the definitions of the *inputs*. Its value can be reused as long as the *inputs* are not modified.

The choice between inspector/executor and speculative execution is either dictated by the data dependence relations or by a performance model. When the *externals* set of an RT_LMAD contains a definition site within the loop, there may exist a cycle between the computation and address. For arrays this situation cannot always be proven at compile time (though a linked list traversal can be proven). Then we have the choice to either distribute the loop and isolate the statements that compute the value in our *external* set or to use the speculative parallelization strategy [19]. If we believe that the statements that potentially form a data dependence cycle are indeed sequential (e.g., linked list traversal) then speculative execution will fail and loop distribution is the better choice. The loop containing the cycle will be executed serially and its results will be used by the second, possibly parallel loop. When dependence cycles are not an issue, then the decision is based on the ratio between the execution time of an inspector loop and that of the entire loop. Small inspectors seem to perform well. A more detailed discussion about these choice can

be found in [16].

Regardless of the chosen strategy the run-time overhead for dependence testing is reduced by the level of aggregation that our **HA** framework achieves.

2.3 Complexity Analysis

We now show that the computational effort of **HA** is quite manageable both at compile-time as well as at run-time thus yielding a viable solution for full automatic parallelization.

2.3.1 Run-time Test Complexity

Memory usage. The additional memory required at run-time is for the Run-time Registers described in Section 2.2.3. The number of RTRegs is upper-bounded by the number of internal nodes of the largest RT_LMAD. In the worst case this number grows linearly with the number of memory reference statements in the code. However, in real codes this number did not exceed 50. If the access pattern is found linear at compile-time (as in direct indexing), then the size of RTReg is input data invariant (the size of an LMAD is proportional to the number of linear dimensions). If the access pattern is found linear at run-time even though it did not seem linear at compile-time (as in subscripted subscripts that take linear values at run-time), then the size of RTReg will still be constant, since the RTReg is implemented as a list of LMADs. The size of the RTReg increases only when a recurrence has a non-linear access pattern that cannot be aggregated above its defining loop.

The **time complexity** is worst case that of the LRPD test, i.e., proportional to either the number of distinct memory references or number of references for dense and sparse access patterns, respectively. However, in practice, the actual complexity is orders of maginitude smaller, depending on the degree of reference aggregation that the **HA** manages to extract. Many times we need only constant time to evaluate a small number of conditions. Even with linear RT_LMADs our framework can easily take advantage of value reuse (a.k.a. schedule reuse) through aggressive hoisting. Experimental results are our best proof.

2.3.2 Compilation Complexity

We will show that the memory and time used at compile-time is $O(\sum_{sym} StaticAccessCount(sym))$ if no RT_LMAD simplification is performed. Below, we give the complexity analysis for the memory access on a single symbol. We assume that the symbolic forward propagation, range dictionary, and interprocedural SSA passes have already run.

The overall memory budget is composed of the storage needed to keep the RT_LMAD nodes and the memory needed to store the primary representation objects. Based on the aggregation algorithms, the number of RT_LMAD nodes is upper bounded by $(15 * S + 9 * L + 9 * I + 3 * C)$, when there are S significant statements, L significant loops, I significant IF statements, C siginificant call sites. Every summary set update can create 15 more RT_LMAD nodes, every loop expansion 9 more, every conditional can create 9 nodes, and every routine 3 more (the number of nodes can be counted as the number of operators on the

Program	Loops	Cause of Difficulties	CT/RT	%S
ADM	RUN/do_20,do_30,do_40 RUN/do_50,do_60,do_100	Rec. with no closed form, Each loop spans 14 routines, Redimensioned arrays	RT	50
	DKZMH/do_30,do_60 WCONT/do_40 D?DTZ(?=C,T,U,V)/do_40	Spans multiple routines	CT CT	44 44
DYFESM	SOLXDD/do_4,do_10,do_30 SOLXDD/do_50,HOP/do_20	Subscripted subscripts	RT	17
	SOLVH/do_20	Input data determines privatization	RT	9
	BLCKMX/do_10,BLCKR0/_do10	Input data determines privatization	RT	73
	MXMULT/do_10,FORMR0/do_20	Subscripted subscripts/complex reduction	RT	73
MDG	INTERF/do_1000	Privatization needs theorem proving	RT	91
	POTENG/do_1000,do_2000	Routine calls	CT	8
TRACK	FPTRAK/do_300	Possible dependencies - cond. induct. var.	RT	46
	NLFILT/do_300	Partially parallel- input data sensitive	RT	2
	EXTEND/do_400	Possible dependencies - cond. induct. var.	RT	50

Table 1: Loops parallelized at compile-(CT) or run- (RT) time, sequential percentage from application (%S).

right hand side of the aggregation algorithms figures. Storage for the primary representation may increase exponentially (worst case) with the number of static memory references. We avoid this by limiting the number of LMADs that we store in an LMAD list to a constant (50, for now). In our experiments, the limit was never reached because most operations on LMADs either produce an LMAD (not increasing the size), or are not exact, in which case the result is represented as an RT_LMAD. The size of an LMAD is proportional to the number of dimensions of the access. which in practice is < 3. Thus, total memory usage is upper bounded by $(12 * S + 9 * L + 9 * I + 3 * C)$*sizeof(RT_LMAD)$+S * 50 * 10$*sizeof(1D-LMAD), i.e., it is linear in the number of program statements.

The cost-reducing transformations described in the section above increase time by an amount linear in the size of the optimized RT_LMADs. These optimizations use bottom-up traversals of RT_LMAD trees with constant-time pattern matching performed at every node. Because the size of RT_LMADs grows linearly, and there are a linear number of aggregations, the time complexity is upper bounded by $O(\sum_{sym} StaticAccessCount(sym)^2)$. The quadratic behavior is never reached in practice because of heuristic simplification that reduces the size of RT_LMAD as they are aggregated. Simplification is based on set and lattice theory and boolean logic identities and takes $O(1)$ regardless of the RT_LMAD size.

3 Experimental Results

We have integrated our **HA** analysis in the Polaris compiler and compiled four difficult PERFECT benchmarks: ADM, TRACK, DYFESM and MDG. The compiler has isolated the variables that could not be fully analyzed statically and generated minimal run-time tests for evaluating their RT_LMADs. For every symbol, array or scalar, we have three per-iteration descriptors WF, RO, and RW. Every processor p computes RT_LMADs WF_p, RO_p, RW_p by aggregating the descriptors within its domain. The RT_LMADs evaluation strategy has been biased towards hoisted and inspectors so that their outcome can be reused. For every run-time parallelized loop we have generated three code sections: an initialization, the loop itself, and a postprocessing phase. Only RW_p must be zero-ed out for reduction variables. If private storage is required (for reductions and to eliminate memory related dependencies), only RO_p sections must be copied in. The loop is run as a DOALL. In the post-processing phase, WF_p and RW_p are known and thus minimal accumulation (for reductions) and dynamic last value assignments (for privatized, live array sections) need to be performed. The last value computation may be helped by RT_LMAD based prefix sum operation across all WF_p (instead of the known exhaustive and veru expensive private storage traversals). All operations (intitialization, loop execution, post-processing are fully parallel).

The four selected benchmarks contain large, deeply nested loops with a wide variety of memory access patterns and a complex call graph which could not be analyzed using the existing technology in Polaris (Table 1). ADM has loop nests spanning multiple routines and arrays are reshaped. The access pattern in loops RUN/do_20,30,40,50,60,100 is based on a recurrence (NA) that has no closed form solution (in Polaris). There are also possible dependences when input variables (NX,NY) are odd integers > 1. Two arrays (SAVEX,SAVEY) are *logically* divided into parts that have different access patterns within the same loop. The run-time test generated by our **HA** pass takes the form of an inspector for the evaluation of the recurrence on NA. The inspector loop has been hoisted to the main program level and its iteration space is upper bounded by 15 thus requiring constant time regardless of the input size.

In DYFESM, most major loops share an offset/length pattern based on subscripted subscripts. Three loops SOLVH/do_20, BLCKMX/do_10 and BLCKR0/do_10 have input dependent access patterns. Some arrays exhibit different access patterns to their subregions within the same loop (MXMULT/do_10:MX, FORMR0/do_20:R0). The run-time test is an inspector of the index arrays, hoisted to the main program level. The complexity of the test is linear with only one of input set dimensions (number of substructures). Parallelism is also available at the second loop level (number of elements per substructure) and is extracted using a constant time run-time test. We note that in the specialized framework reported in [14] some of the loops have been parallelized at compile time.

In MDG, loop INTERF/do_1000, the complicating factor is the control flow. Even though the access pattern is input independent a deductive system would be required to make the necessary inference. Our **HA** limited the memory area with possible dependences to an interval of length 4 inside an array (RL. The test grows linearly with the iteration count but

is lightweight per iteration. Because an inspector would have to perform more than 1/3 of the work of the loop we used speculative run-time testing.

In `TRACK`, loop `NLFILT/do_300` is input dependent (array NUSED) and, for some instantiations, partially parallel. It was executed speculatively due to a potential cycle between data and address computation.

Loop `FPTRAK/do_300` presents possible output dependencies based on the values of a linear recurrence. The recurrence is based on all the computation inside the loop. We execute it speculatively since an inspector would be as complex as the loop itself. Similarly loop `EXTEND/do_400` presents a potential output dependence on several arrays (IHITS, XH1, ...) with essentially the same access pattern. The analysis reduces itself formally to a last-value computation of the arrays and of a scalars (LSTTRK, NXTTRK, JSAME) which indicate the offsets at which the arrays should be *copied out* (instead of last value assignment). We have executed the loop speculatively using a copy-in, merge-out technique [6]. We expect to soon improve our results.

Benchmark	**Lines**	**Comp. time**	**Par. Coverage**
ADM	4227	291	99.63
DYFESM	3435	45	99.72
MDG	938	47	98.87
TRACK	2157	143	98.00

Table 2: Compilation results

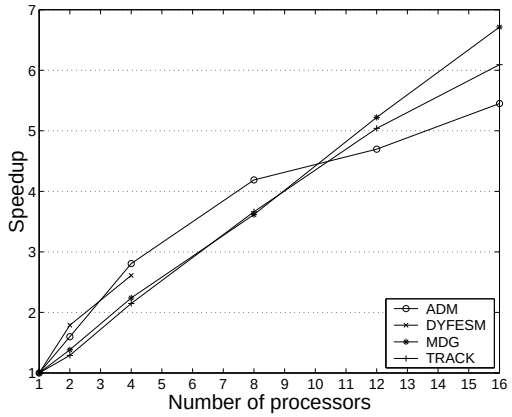

Figure 14: Speedups for `DYFESM`, `MDG`, `ADM`, `TRACK`

Table 2 presents the codes that were given as input to our optimizing pass and their compilation time (seconds). The last column presents the percentage of the sequential execution time that was parallelized. The unoptimized (g++ -g) Polaris was run on a Pentium III PC, 800Mhz, with 512 MB RAM. As we can see, the compilation time are practical.

Fig. 14 presents the speedups obtained by running the parallelized codes on a 16 processor `Convex V2200`. We used the native `HPUX f90` parallelizing directives as primitives. `DYFESM`

shows poor speedups because its data set is very small. `TRACK` is not fully parallel. `ADM/APSI` has used the large input set from SPEC2000 but the granularity of the parallelization is somewhat low. *All speedups are scalable and reflect a parallelization with practically full coverage.*

4 Conclusions and Future Work

In this paper we have presented a novel **Hybrid Analysis (HA)** technology which can efficiently and seamlessly integrate all static and all run-time analysis of memory references into a single framework. This framework is capable of performing all data data dependence analysis and can generate necessary information for all associated memory related optimizations. We have used **HA** to perform automatic parallelization by extracting run-time assertions from any loop and generating appropriate 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) 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.

We implement our framework in the Polaris compiler by introducing an innovative intermediate representation called RT_LMAD and 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. Additionally, we augment the GSA representation in Polaris for inter-procedural use. Our analysis is flow sensitive on any control flow graph. We intend to parallelize all PERFECT codes in the near future, which would be a significant accomplishment.

References

[1] T. Ball. What's in a region? -or- computing control dependence regions in linear time and space. Technical Report 1108, University of Wisconsin – Madison, Computer Sciences Department, 1992.

[2] 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.

[3] B. Creusillet and F. Irigoin. *Interprocedural array region analyses.* Springer-Verlag, August 1995.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and Kenneth Zadeck. An efficient method of computing static single assignment form. In *16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Austin, Tex., January 1989.

[5] Ron Cytron, Jeanne Ferrante, and Vivek Sarkar. Compact representations for control dependence. In *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 337–351, White Plains, N.Y., June 1990.

[6] 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.

[7] Keith Cooper et al. The parascope parallel programming environment. *Proceedings of IEEE*, pages 84–89, February 1993.

[8] M. Gupta, E. Schonberg, S. Midkiff, P. Sweeney, K.-Y. Wang, and M. Burke. Ptran ii – a compiler for high performance fortran. In *Proceedings of the 4th Workshop on Compilers for Parallel Computers*, December 1993.

[9] Mohammad R. Haghighat and Constantine D. Polychronopoulos. Symbolic analysis: A basis for parallelization, optimization and scheduling of programs. In *1993 Workshop on Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, pages 567–585, Portland, Ore., August 1993. Berlin: Springer Verlag.

[10] Mary Hall, Jennifer Anderson, Saman Amarasinghe, Brian Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica Lam. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[11] Mary Wolcott Hall. Managing interprocedural optimization. Technical Report TR91-157, Rice University – Computer Science Department, 28, 1998.

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

[13] J. Knoop. *Optimal interprocedural program optimization: A new framework and its application.* PhD thesis, Department of Computer Science, University of Kiel, 1993.

[14] Yuan Lin and David Padua. Compiler analysis of irregular memory accesses. In *ACM SIGPLAN '00 Conference on Programming Language Design and Implementation.*

[15] 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.

[16] D. Patel and L. Rauchwerger. Principles of speculative run–time parallelization. In *Proceedings 13th Annual Workshop on Programming Languages and Compilers for Parallel Computing*, pages 330–351, August 1998.

[17] William Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91*, pages 4–13, Albuquerque, N.M., November 1991.

[18] 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.

[19] Lawrence Rauchwerger and David A. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2), 1999.

[20] Radu Rugina and Martin C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 182–195, 2000.

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

[22] 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.

[23] Rémi Triolet, François Irigoin, and Paul Feautrier. Direct parallelization of Call statements. In *ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 175–185, Palo Alto, Calif., June 1986.

[24] Mark Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, July 1984.

[25] 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.