

Hybrid Analysis: Static & Dynamic Memory Reference Analysis

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

ABSTRACT

We present a novel **Hybrid Analysis** technology which can efficiently and seamlessly integrate all static and run-time analysis of memory references into a single framework that is capable of performing all data dependence analysis and can generate necessary information for most 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 **RTLMD** and a run-time library that can operate on it. Based on the experimental results obtained to date we hope to to automatically parallelize most and possibly all **PERFECT** codes, a significant accomplishment.

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 [6], SUIF [9] and Polaris [11] 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 both 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, 5] 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 [24, 14] but with limited applicability.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2001 ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

```

subroutine F00(A,N)      Program Main
integer A(100)         integer A(1:100*100)
Do j=1,N               READ *,A(:),N
  A(j) = A(j+40)      Do i=1,N
Enddo                  Call F00(A((i-1)*100+1),N)
Return                 Enddo

```

Figure 1:

work 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 reformulate it in function of the static unknowns and generate the code that will finish off 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 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 pro-

```

subroutine F00(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 F00(C,TMP,N,M,lim)
Enddo

```

Figure 2:

```

subroutine F00(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 F00(C,TMP,N)
  ... = TMP(k)
Enddo

```

Figure 3:

posed 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 routine FOO 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 no closed form solution (the 'flip variable' NA). 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 thus we will generate a small inspector loop that will verify all values taken by NA .

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

```

subroutine F00(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 F00(C,TMP,N)
  ... = TMP(k)
Enddo

```

Figure 4:

```

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

subroutine RAD(CC,CH)
integer CC(*), CH(*)
common /input/ G(10)
Do i=1,10
  CC(i) = 2*CH(i) + G(i)
Enddo

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

```

Figure 5:

of the code. Other simple scalar run-time checks verifying parallelization have been used in [14, 11].

In later years various techniques for run-time parallelization and partial redundancy elimination have been proposed. [21] and, later [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 [24].

Inter-procedural analysis is present in many research compilers [22, 7, 12, 8, 2, 9, 10, 20]. In Polaris [1] IPA has been reported in [15, 11].

The use of run-time assertions within the IPA context is a recent development and has been reported in SUIF [14] and Polaris [11]. 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 [14] 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 [11] 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 [24] 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.

We should note that the value based analysis reported in [13] 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 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 analysis with more expensive 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.

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),

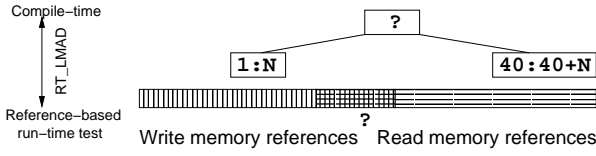


Figure 6: Compile-time to Run-time transition

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 information about the value of N . We need to stop at the second level and postpone the rest of the evaluation to run-time. The time 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 data structure, the RT_LMAD. We will now present its organization 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, 9, 11]. For practical reasons¹ but *without any loss of generality* we have chosen to use the triplet based LMAD as the elementary building block for RT_LMADs.

In [11] we defined a Linear Memory Access Descriptor (LMAD) as a representation of the subscripting offset sequence. Consider a loop nest of depth D with indices I_k , $k = 1, D$, where $I_k = 0, U_k$. Consider a reference to memory given by $A(s_1(\vec{I}), s_2(\vec{I}), \dots, s_m(\vec{I}))$, where $\vec{I} = (I_1, I_2, \dots, I_d)$. If the subscripting function can be written in a sum-of-products form with respect to the individual loop indices,

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

then, we can isolate the effect of each loop index on the

¹Our Polaris compiler already uses LMAD representation.

subscripting offset sequence.

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 D0 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 RT_LMAD is a symbolic expression in which the operands are LMADs. The operators identify operations that are not closed with respect to the set of all LMADs: (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 an array are approximated with the entire array. Sometimes we could 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 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).

2.1.1 RT_LMAD Definition

The RT_LMAD is a representation of the memory access pattern that takes into account the computation required to evaluate it. An RT_LMAD can be viewed as a piece of code that takes as input values from the program under analysis and produces the set of memory references. We store the RT_LMAD as a parse tree with respect to the language presented in Fig. 7. RT_LMADs are expressions in this language. These expressions have as operands lists of LMADs.² The operators reflect the operations that must be performed in order to evaluate the RT_LMAD to a set of integers. $RT_LMAD_2 = RT_LMAD_1 \otimes (DO\ i = 1, 10)$ means that in order to evaluate RT_LMAD_2 , we must evaluate RT_LMAD_1 for all iterations of $DO\ i = 1, 10$, and then perform set union on the results. Throughout the remainder of this 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 represent formulae containing recurrences.

Let us illustrate the use of 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

²We can easily adapt our system to a different primary representation by just replacing the LMAD data structure with a new data structure that preserves its semantics (such as systems of inequations).

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

Figure 7: RT_LMAD formal definition.

a pattern easy to represent as an LMAD, [1 : 10]. This pattern can be translated at the first call site in routine RFFTF as [1 : 10]. 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 an LMAD. Using RT_LMADs, we will represent it as: ([1 : 10]#(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

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

Fig. 8 presents the same expression viewed as a parse tree.

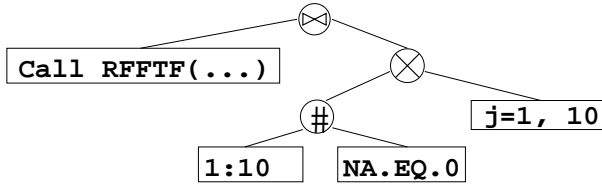


Figure 8: Write descriptor for array W for an iteration of loop DO i=1, 1024 in routine RUN, Fig. 5.

We define the **evaluation** of an RT_LMAD as the process of finding the set of indices as known integer values that it represents. Some RT_LMADs can be evaluated at compile-time (such as the ones that are made of an LMAD containing only known integer values). The ones that cannot be evaluated at compile-time can be translated into FORTRAN code using an attribute grammar associated with the language shown in Fig. 7.

2.2 Program Analysis using RT_LMADs

Our HA framework is integrated into the Polaris compiler. A prepass brings the program into GSA form³ and builds the program's control dependence graph (CDG) [3] and Call graph. We also compute control dependence regions (CDRegions) as defined in [4]. Since our subsequent

³We 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.

analysis assumes these two graphs are acyclic,⁴ 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).

```

Sort Call Graph topologically
FOREACH subprogram
  Sort CDG topologically
  FOREACH CDG node
    Process(CDG node)

```

Figure 9: High level view of the HA driver.

The HA driver (Fig. 9) performs a single pass over the whole program. Across routines, it traverses the Call Graph in reverse topological order. Within a routine, it traverses the CDG in reverse topological order. Within a CDG region all nodes are ordered exactly as in the original program. As the driver visits a CDG node, it is processed as follows (see Fig. 10). First, in `GetPrimaryInfo`, we classify each memory reference within the statement associated with the CDG node. At CDG internal nodes (loop headers and conditional branches), the driver invokes routines that perform the aggregation across all iterations (for loops) and across all branches (for conditionals). At call sites, the driver invokes routines that translate the RT_LMADs summarizing the called routine into the actual calling context. Last, memory reference are aggregated to the RT_LMADs associated with the CDRegion containing the CDG node. Parallelism detection and code generation are performed at loop header level (after processing the loop body, but before processing the header).

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), Write-First (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. An extensive description of the classification of references in (RO, WF, RW) and its use in parallelism detection is given in [11].

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.

⁴Except for self-loops in the CDG.

```

ALGORITHM AnalyzeCDGNode(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)

```

Figure 10: HA driver core: aggregation. (T=TRUE, F=FALSE) at CDG node level

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.⁵ **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. 11 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 the aggregation occurs in program order.

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

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

Figure 11: Summary Sets Aggregation Algorithms

When aggregating the summary set for a recurrence given the summary set per iteration (Fig. 11 (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 [11]) or, we 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 WF across recurrence $j = 1, N$ would be $WF = \bigcup_{j=1}^N WF_j - \bigcup_{j=1}^N (RO_j \cup RW_j)$. If we apply this equation to inner loop D0 j in routine RFFTF then we will find the outer loop D0 $i=1, 1024$ in routine RUN sequential. When assuming a sequential order for the inner loop the equation for WF becomes $WF = \bigcup_{j=1}^N (WF_j - \bigcup_{k=1}^{j-1} (RO_k \cup RW_k))$, and the outer loop is found parallel.

This formula seems to have quadratic complexity ($j = 1, N \wedge k = 1, j - 1$). We reduced its complexity in the code

⁵There 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.

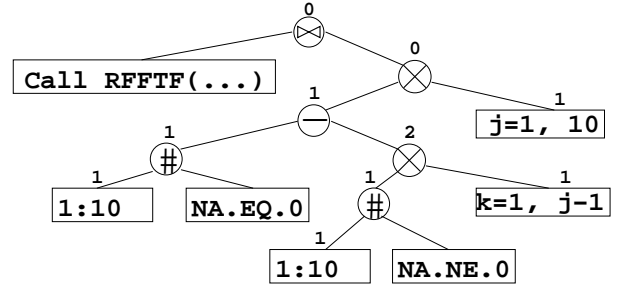


Figure 12: WriteFirst descriptor for array W in an iteration of loop D0 $i=1,1024$, routine RUN in Fig. 5.

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 D0 j recurrence is a Lower Partial Union. The idea is to compute the lower partial unions at the same time we aggregate the other descriptors, which yields linear complexity (analogous to partial sum computation).

Fig. 12 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 D0 $i=1,1024$ routine RUN.

2.2.2 Parallelism Detection

In this paper, we focus on global program analysis using RT_LMADs for extracting coarse grain parallelization. The decision when a loop can be executed as a DOALL is mainly conditioned 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 sets of memory access over the whole loop body (RO, WF, RW) for all the symbols referred in the loop and produces a triplet of values that represent the compile-time parallel decision for this loop: dep_fa (there are loop carried flow or anti dependences), dep_o (there are output dependencies), and is_red (there are reductions). Because this triplet is expressed symbolically, we used a ternary logic with the values *NO*, *MAYBE* and *YES*. Thus, $(dep_fa, dep_o, is_red) = (MAYBE, YES, NO)$ means there are variables that may have flow or anti dependences (it could not be decided at compile-time), there are variables that have output dependencies and there are no reductions.

Algorithm **ClassifyRec** (Fig. 13) finds the decision triplet by accumulating decisions taken per symbol referred in the loop. The usage of operator *MAX* (where $NO < MAYBE < YES$) 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. It checks, for every RW descriptor associated with this symbol, whether all its originating statements follow a reduction pattern: $A(ind)=A(ind)+expression$.

For every symbol we also compute: individual 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

```

ALGORITHM ClassifyRec(Rec, SymbolList)
(dep_fa, dep_o, is_red) ← (F, F, F)
FOR ALL Symbol IN SymbolList DO
  (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)
(WFj, ROj, RWj) ← Per Iteration Info
(dep_fa, dep_o, is_red) ← (F, F, F)
IF (CheckReduction(Symbol, Rec)=T) THEN is_red=T
dep_fa = DEP(( $\bigcup_{j=1}^N WF_j$ ) ∩ ( $\bigcup_{j=1}^N RO_j$ ))
dep_fa = MAX(dep_fa, DEP(( $\bigcup_{j=1}^N WF_j$ ) ∩ ( $\bigcup_{j=1}^N RO_j$ )))
dep_fa = MAX(dep_fa, DEP(( $\bigcup_{j=1}^N WF_j$ ) ∩ ( $\bigcup_{j=1}^N RW_j$ )))
dep_fa = MAX(dep_fa, DEP(( $\bigcup_{j=1}^N RW_j$ ) ∩ ( $\bigcup_{k=1}^{j-1} RW_k$ )))
dep_o = DEP( $\bigcup_{j=1}^N WF_j$  ∩ ( $\bigcup_{k=1}^{j-1} WF_k$ ))
RETURN (dep_fa, dep_o, is_red)

```

Figure 13: Parallel Classification Algorithms

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 can be privatized. A *dep_o* value of *MAYBE* requires a run-time test.

2.2.3 Code Generation

We generate code that implements a run-time algorithm that is semantically identical to the parallel classification algorithm described above. The only differences are that the decisions use boolean rather than ternary logic and the sets are represented as lists of LMADs (represented using FORTRAN arrays in the generated code) 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 code generation for RT_LMAD evaluation is a translation to FORTRAN of the language presented in Fig. 7. The translation is based on an attribute grammar. The only attribute we use is the run-time value of the RT_LMAD as a list of LMADs made of known integer values. Gates translate to IF statements, recurrences to DO loops. Set operations and `CallSite` nodes translate into calls to a run-time support library that operates on lists of LMADs with known integer values. The generated statements are inserted in the code at the first point where all the values they use are defined. In case the evaluation method is an inspector, we clone the program slice [23] that computes the values referenced by the RT_LMAD.

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 ag-

gregations is that the accumulation code is inserted at the very end of the loop 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. 12.

Parallel Execution Code

This section presents the code that manages parallel execution. We assume that at this point the dependence tests were passed with success (independence was established). We use the already computed per-iteration descriptors *WF*, *RO*, and *RW*. For every processor *p*, we compute RT_LMADs *WF_p*, *RO_p*, *RW_p* by aggregating the descriptors within its domain. The RT_LMADs evaluation strategy has been biased towards re-usable (hoisted) inspectors. For every run-time parallelized loop we have generated three code sections: an initialization, the loop itself (to be executed as a DOALL), and a postprocessing phase. If private storage is required (for reductions and to eliminate memory related dependencies), only the part of the *RW_p* section of reduction variables needs to be initialized (zero-ed). If copy-in of read-only variables is desired, the array section(s) corresponding to *RO_p* are copied in during *initialization*.

In the *post-processing* phase, *WF_p* and *RW_p* are known and thus minimum accumulation (for reductions) and dynamic last value assignments (for privatized, live array sections) need to be performed. The usually high cost of 'classic' dynamic last value assignment (equivalent to a cross-processor *max* reduction operation for all privatized arrays) is reduced because the identity of the last writing iteration can be computed from the *WF_p* through RT_LMAD operations.

In general, the generated code is faster because it can use the precise information given by the run-time evaluated LMADs rather than rely on conservative, compile time estimates. Instead of operating on whole arrays we can operate on precise array sections. For instance, for an array `A(1:10)`, and an LMAD `WF[3:7]`, we can call the routine `initialize(A, WF, 0)` that will set elements from 3 to 7 in array `A` to 0. Routines like `initialize` are implemented as generic, fully parallel loops that follow the structure of the given LMAD.

Optimizations

We use several methods to reduce the run-time evaluation of RT_LMADs: based on dataflow analysis (such as loop invariant hoisting), based on control dependence analysis (such as AND-ing mutually exclusive predicates), based on set identities (such as $(A - B) - A = \emptyset$), and based on lattice identities (such as $A - T = \emptyset$). Loop invariant RT_LMAD hoisting is similar to the inspector re-use technique in [21].

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 choice between inspector/executor and speculative execution is either dictated by the data dependence relations or by a performance model. When an array `A` is written based on an index or conditional that contains references to `A`, 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 are in the cycle or to use the spec-

ulative 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 choices 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 lists of LMADs used at run-time to evaluate RT_LMADs. Our RT_LMAD evaluation scheme is similar to a register-based evaluation scheme for an arithmetic expression. Instead of machine registers we use lists of LMADs. In the worst case the number of our 'registers' grows linearly with the number of memory reference statements in the original code. However, throughout our experiments, this number did not exceed 50. Also, this number is known at compile-time and they can be statically allocated. The small number on top of some of the nodes in Fig. 12 represents the ID of the 'register' assigned to store the result of the RT_LMAD evaluation at that particular node. If the access pattern is found linear at compile-time (as in direct indexing), then the size of a 'register' is input data invariant (the size of an LMAD is proportional to the number of linear dimensions in the space it represents). 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 the 'register' will still be constant. The size of the 'register' increases only when a recurrence has a non-linear access pattern that cannot be aggregated using LMADs even at run-time. In the worst case, the size of a 'register' can be the same as the size of the data tested for dependences. We have, so far, not encountered the worst case in our experiments.

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 magnitude 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 RT_LMADs that take non-constant time to evaluate, our framework can easily take advantage of value reuse (a.k.a. schedule reuse) through aggressive hoisting.

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 time complexity of our analysis of a single symbol assuming that the symbolic forward propagation, range dictionary, and inter-procedural SSA passes have already run.

The overall memory budget is composed of the storage needed to keep the RT_LMAD internal nodes and the memory needed to store the primary representation objects (the LMADs). We allow the parse trees for different RT_LMADs to overlap in memory. This way the number of additional internal nodes needed to represent the result of any RT_LMAD operation is constant. Every summary set update can create 15 more RT_LMAD nodes, every recurrence 9 more, every conditional can create 9 nodes, and every routine call 3 more (the number of nodes can be counted as the number of operators on the right hand side in Fig. 11). The number of RT_LMAD parse tree nodes is upper bounded by $(15 * S + 9 * L + 9 * I + 3 * C)$, when there are S statements, L loops, I IF statements, C call sites that may have effect on the access pattern. Storage for the primary representation (LMADs) 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 pattern, which in practice is < 4 . Thus, total memory usage for computing the access pattern on an array A is upper bounded by $(12 * S + 9 * L + 9 * I + 3 * C) * \text{sizeof}(RT_LMAD) + S * 50 * 4 * \text{sizeof}(1D_LMAD)$, i.e., it is linear in the number of program statements that may have effect on the access pattern.

Some of the optimizing transformations we apply to RT_LMADs require bottom-up traversals of the associated parse 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 not reached in practice because the optimizing transformations performed reduce the sizes of RT_LMADs as they are aggregated.

3. EXPERIMENTAL RESULTS

We have integrated our **HA** analysis in the Polaris compiler and compiled four difficult PERFECT benchmarks: **ADM**, **TRACK**, **DYFESM** and **MDG**. We are not aware of any other automatic parallelization technique that can deal with all these codes uniformly. [14] reported 70% coverage on **ADM** based on run-time tests. However, their approach cannot deal with memory access patterns that are not linear with respect to the index of the loop under analysis. Within the set of loops they found parallel in **ADM:RUN**, the apparent dependences on array **SAVEX** caused by the "flip variable" **NA** were eliminated using loop peeling. Even though it helped in this particular case, there is no mention to what triggered the use of peeling and whether it is generally profitable. [13] reported 20% parallel coverage on **DYFESM**. Their technique has the advantage of being performed at compile-time. The description of

| Program | Loops | Cause of Difficulties | (C—R)T | %Seq |
|---------|--|---|--------|------|
| 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 | 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/do_10 | 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 based on cond. induct. var. | RT | 46 |
| | NLFILT/do_300 | Partially parallel- input data sensitive | RT | 2 |
| | EXTEND/do_400 | Possible dependencies based on cond. induct. var. | RT | 50 |

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

their experimental results show that the index array property that they exploit (Closed-Form Distance) does not show in any of the other programs analyzed.

Our previous work [11] could parallelize only 13% of the sequential execution at the same level of granularity at which we obtain now 99%. The LRPD test obtains the same coverage we do, but its run-time overhead is generally higher.

The four selected benchmarks contain large, deeply nested loops with a wide variety of memory access patterns and complex control and data dependencies 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 dependencies 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.

In DYFESM, most major loops exhibit an offset/length memory access pattern based on subscripted subscripts. Loops SOLVH/do_20, BLCKMX/do_10 and BLCKR0/do_10 have also 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). Loops SOLXDD/do_4, do_10, do_30, do_50 use subscripted subscripts but they have been proven parallel at compile-time ([13]). We solved them all at run-time using an inspector of the index arrays, hoisted to the main program level.

In MDG, loop INTERF/do_1000 takes 90% of the execution time. Even though the access pattern is input independent, a deductive system would be required to make the necessary inferences. 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 and fully parallel. 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.

In loop FPTRAK/do_300, array IHITS is read from 1 to NTROLD and written at offset LSTTRK in every iteration. This access pattern presents two apparent dependences: a flow dependence from the write at offset LSTTRK to the read at any offset between 1 and NTROLD, and an output dependence between any two writes at offset LSTTRK within different loop iterations. We eliminated the flow dependence at compile-time by proving that $LSTTRK \geq NTROLD + 1$. The proof is based on computing the shortest path in a graph having as nodes the GSA names in the subprogram. The edges were weighted by the difference between related values. For instance, $LSTTRK_3 = LSTTRK_2 + 1$ will add an edge between $LSTTRK_2$, $LSTTRK_3$ of weight 1. The shortest path gives the minimum distance between two values, in this case 1. The shortest path computation is done on demand and is triggered by comparisons of values within the subprogram range dictionary, being thus fully integrated in Polaris. The output dependence can be solved at compile time by proving that the shortest nontrivial path from $LSTTRK_3$ to itself is positive. This analysis, though already implemented, is not yet fully integrated in our framework. Therefore, we left this dependence to be tested at run-time. The results show that the overhead induced was very small.

Similarly loop EXTEND/do_400 presents a potential output dependence on several arrays (IHITS, XH1, ...) with essentially the same access pattern. The analysis reduces formally to a last-value computation of the arrays that show possible output dependences. The last-value computation is performed inexpensively by using RTLMADs to aggregate the private writes of every processor. A cross-processor merge is performed to find the RTLMADs that represent the memory regions that must be committed by every processor. The complexity of this operation was $O(p)$ since the RTLMADs representing private writes turned out at run-time to be contiguous disjoint 1-dimensional LMADs.

Table 2 presents the codes that were given as input to our optimizing pass and their compilation time (seconds). The second column contains the number of lines of code, and the third one the static compilation time. The fourth column presents the percentage of the sequential execution time that was parallelized using only compile-time methods, and the last column using HA methods. Parallel coverage was measured at the highest level of granularity (whole data

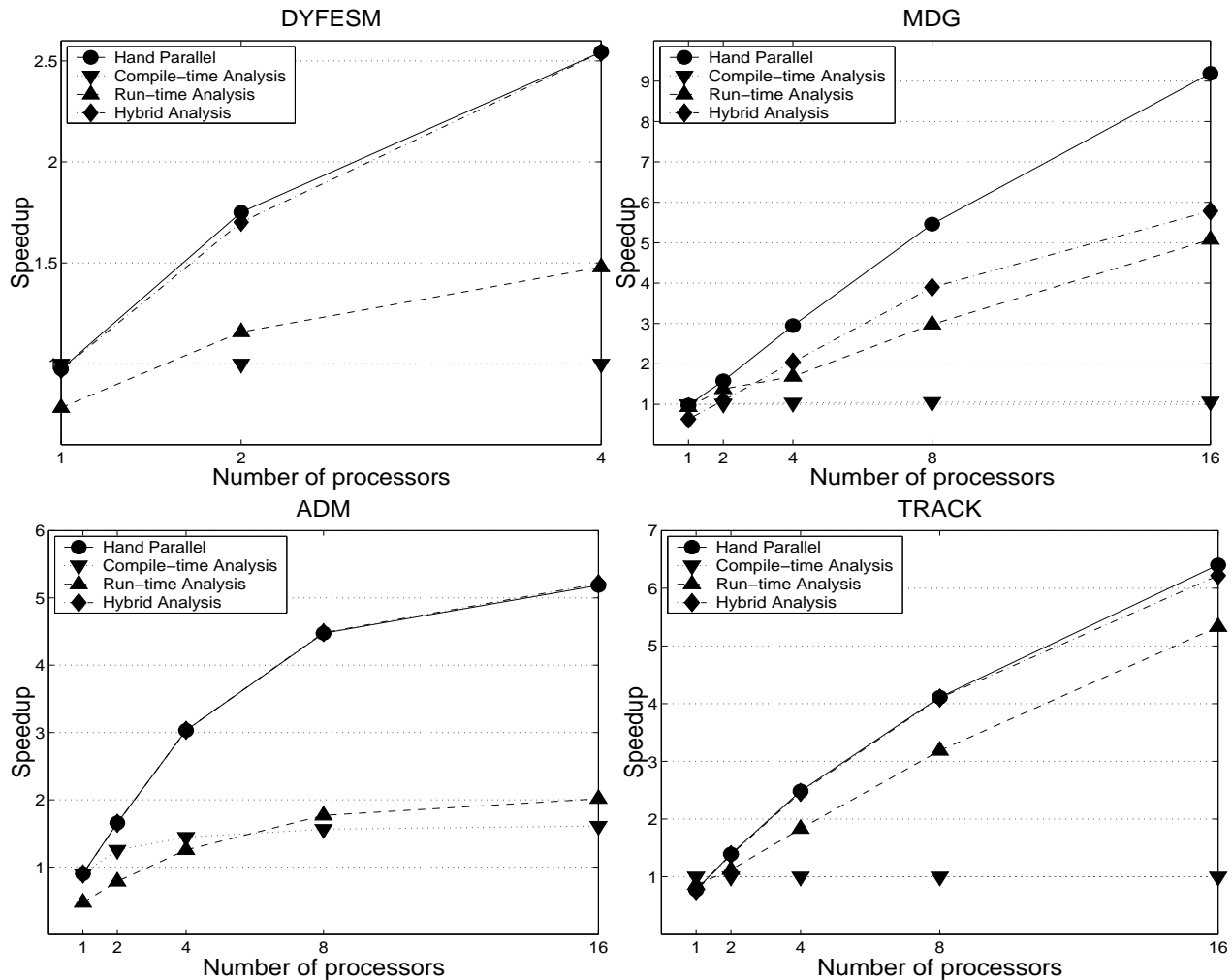


Figure 14: Speedups obtained for each application using four parallelization methods: by hand, based on Compile-time Analysis only, based on Run-time Analysis only (LRPD for DYFESM, MDG and ADM, and R-LRPD for TRACK), and based on Hybrid Analysis respectively.

| Benchmark | Lines | Comp. time | CT Cov. | RT Cov. |
|-----------|-------|------------|---------|---------|
| ADM | 4227 | 291 | 44 | 99.63 |
| DYFESM | 3435 | 45 | 0 | 99.72 |
| MDG | 938 | 47 | 8 | 98.87 |
| TRACK | 2157 | 143 | 0 | 98.00 |

Table 2: Compilation results

set). Polaris was run on a Pentium III PC, 800Mhz, with 512 MB RAM. As we can see, the compilation times 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. We compared them against three sets of results: parallelization by hand (done by programmer analysis and coding), parallelization based on compile-time analysis (using LMADs only), and parallelization based on run-time analysis (using (R-)LRPD only). In DYFESM and ADM our run-time tests are inexpensive because they are hoisted outside time step

loops with large iteration counts. Also, they are partially aggregated at compile-time so that at run-time they are proportional to just a fraction of the data set size. The reference-by-reference LRPD test leads to smaller speedups primarily due to initialization and analysis overhead proportional to the whole data set. In addition, the static analysis on which it relies (as it was implemented Polaris) lacks interprocedural analysis and the inspector reuse technology. In MDG and TRACK the compile-time aggregation of memory references is not significant. However, our analysis generates light-weight tests by isolating the array region with possible dependencies. Their overhead is very small in TRACK. In MDG, the overhead is comparable to that of the reference-by-reference LRPD test. There is a trade-off between operations with RTLMADs and simple shadow arrays: Individual RTLMADs operations are more expensive than references to shadow elements (dense LRPD test) but a high degree of reference aggregation can reduce the overall instruction count and memory consumption dramatically.

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

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 most data data dependence analysis and can generate necessary information for 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 expensive run-time techniques through a continuum of simple to complex solutions.

We have implemented **HA** 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. To achieve good speedups we will need to improve on a performance model that can better evaluate the tradeoffs we are facing, e.g., LMADs vs. shadow arrays, speculative execution vs. inspector/executor model, level at which to test for parallelism, etc.

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] B. Creusillet and F. Irigoien. *Interprocedural array region analyses*. Springer-Verlag, August 1995.
- [3] 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.
- [4] 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.
- [5] 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.
- [6] Keith Cooper et al. The parascope parallel programming environment. *Proceedings of IEEE*, pages 84–89, February 1993.
- [7] 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.
- [8] Mohammad R. Haghghat 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.
- [9] 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.
- [10] Mary Wolcott Hall. Managing interprocedural optimization. Technical Report TR91-157, Rice University – Computer Science Department, 28, 1998.
- [11] J. Hoeflinger. *Interprocedural Parallelization Using Memory Classification Analysis*. PhD thesis, University of Illinois, Urbana-Champaign, August, 1998.
- [12] J. Knoop. *Optimal interprocedural program optimization: A new framework and its application*. PhD thesis, Department of Computer Science, University of Kiel, 1993.
- [13] Yuan Lin and David Padua. Compiler analysis of irregular memory accesses. In *ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*.
- [14] Sungdo Moon, Mary W. Hall, and Brian R. Murphy. Predicated array data-flow analysis for run-time parallelization. In *Proceedings of the 12th ACM International Conference on Supercomputing*, July 1988.
- [15] Yunheung Paek, Jay Hoeflinger, and David Padua. Simplification of Array Access Patterns for Compiler Optimizations. 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] Rémi Triolet, François Irigoien, and Paul Feautrier. Direct parallelization of Call statements. In *ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 175–185, Palo Alto, Calif., June 1986.
- [23] Mark Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, July 1984.
- [24] 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.