Techniques for Reducing the Overhead of Run-Time Parallelization

Hao Yu and Lawrence Rauchwerger*

Dept. of Computer Science
Texas A&M University
College Station, TX 77843-3112
{h0y8494,rwerger}@cs.tamu.edu

Abstract. Current parallelizing compilers cannot identify a significant fraction of parallelizable loops because they have complex or statically insufficiently defined access patterns. As parallelizable loops arise frequently in practice, we have introduced a novel framework for their identification: speculative parallelization. While we have previously shown that this method is inherently scalable its practical success depends on the fraction of ideal speedup that can be obtained on modest to moderately large parallel machines. Maximum parallelism can be obtained only through a minimization of the run-time overhead of the method, which in turn depends on its level of integration within a classic restructuring compiler and on its adaptation to characteristics of the parallelized application. We present several compiler and run-time techniques designed specifically for optimizing the run-time parallelization of sparse applications. We show how we minimize the run-time overhead associated with the speculative parallelization of sparse applications by using static control flow information to reduce the number of memory references that have to be collected at run-time. We then present heuristics to speculate on the type and data structures used by the program and thus reduce the memory requirements needed for tracing the sparse access patterns. We present an implementation in the Polaris infrastructure and experimental results.

1 Run-Time Parallelization Requires Static Compiler Analysis

To achieve a high level of performance for a particular program on today's supercomputers, software developers are often forced to tediously hand–code optimizations tailored to a specific machine. Such hand–coding is difficult, increases the possibility of error over sequential programming, and the resulting code may not be portable to other machines. The large human effort that is involved in

* Research supported in part by NSF CAREER Award CCR-9734471, NSF Grant ACI-9872126, NSF Grant EIA-9975018, DOE ASCI ASAP Level 2 Grant B347886 and a Hewlett-Packard Equipment Grant

A. Watt (Ed.): CC/ETAPS 2000, LNCS 1781, pp. 232–248, 2000.

parallelizing code makes parallel programming a task for highly qualified scientists and has kept it from entering mainstream computing. The only avenue for bringing parallel processing to every desktop is to make parallel programming as easy (or as difficult) as programming current uniprocessor systems. This can be achieved through good programming languages and, mainly, through automatic compilation.

Restructuring, or parallelizing, compilers address this need by detecting and exploiting parallelism in sequential programs written in conventional languages as well as parallel languages (e.g., HPF). Although compiler techniques for the automatic detection of parallelism have been studied extensively over the last two decades (see, e.g., [10,19]), current parallelizing compilers cannot extract a significant fraction of the available parallelism in a loop if it has a complex and/or statically insufficiently defined access pattern. Typical examples are complex simulations such as SPICE [9], DYNA–3D [18], GAUSSIAN [7], CHARMM [1].

In previous work [14] we have shown that a viable method to improve the results of classic, static automatic parallelization is to employ run-time techniques that can trace 'relevant' memory references and decide whether a loop is parallel or not. Run–time techniques can succeed where static compilation fails because they have access to the input data. For example, input dependent or dynamic data distribution, memory accesses guarded by run–time dependent conditions, and subscript expressions can all be analyzed unambiguously at run–time. In contrast, at compile–time the access pattern of some programs cannot be determined, sometimes due to limitations in the current analysis algorithms but most often because the necessary information is just not available, i.e., the access pattern is a function of the input data. For example, compilers usually conservatively assume data dependences in the presence of subscripted subscripts. Although more powerful analysis techniques could remove this last limitation when the index arrays are computed using only statically–known values, nothing can be done at compile–time when the index arrays are a function of the input data [5,16,20].

In [12] we have presented the general principles of run-time parallelization implementation. Briefly, such run-time parallelization can be effective, i.e., obtain a large fraction of the available speedup, by reducing the associated run-time overhead. This can be achieved through a careful exploitation of *all or most* available *partial* static information by the compiler to generate a minimal run-time activity (for reference tracing and subsequent analysis). To achieve significant performance gains both compiler and run-time techniques need to take into account the specific characteristic of the applications. While it is difficult and may be, for now, impractical to specialize the compilation technology to each individual code, we have found two important classes of reference patterns that need to be treated quite differently: dense and sparse accesses.

In our previous work we have mostly discussed how to efficiently implement the LRPD test for the dense case. In this paper we will emphasize the compiler and run-time techniques required by sparse applications. As we will show later, the run-time disambiguation of sparse reference patterns requires a rather

different new implementation and presents serious challenges in obtaining good speedups.

We will first present some generally applicable techniques to reduce the run-time overhead of run-time testing through shadow reference aggregation. More specifically we will show how we can reduce the number and instances of memory references traced during execution by using statically available control- and data-flow information. Then we will present specific shadow structures for sparse access patterns. Finally we will present experimental results obtained through implementation in the Polaris infrastructure to illustrate the benefits of our techniques.

2 Foundational Work - The LRPD Test for Dense Problems

We have developed several techniques [13,14,15] that can detect and exploit loop level parallelism in various cases encountered in irregular applications: (i) a speculative method to detect fully parallel loops (The LRPD Test), (ii) an inspector/executor technique to compute wavefronts (sequences of mutually independent sets of iterations that can be executed in parallel) and (iii) a technique for parallelizing `while` loops (`do` loops with an unknown number of iterations and/or containing linked list traversals). In this paper we will mostly refer to the LRPD test and how it is used to detect fully parallel loops. To make this paper self-contained we will now briefly describe a simplified version of the speculative LRPD test.

2.1 The LRPD Test

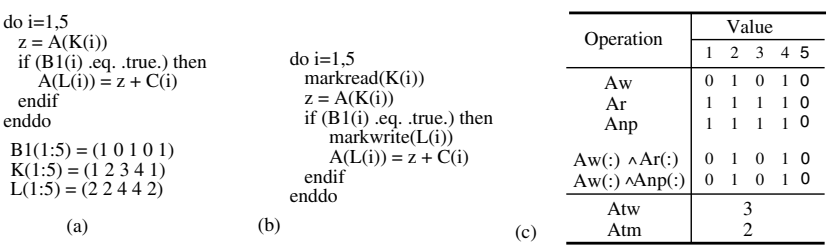
The LRPD test speculatively executes a loop in parallel and tests subsequently if any data dependences could have occurred. If the test fails, the loop is re–executed in a safe manner, e.g., sequentially. To qualify more parallel loops, *array privatization* and *reduction parallelization* can be speculatively applied and their validity tested after loop termination.[1] For simplicity, reduction parallelization is not shown in the example below; it is tested in a similar manner as independence and privatization.

Consider a `do` loop for which the compiler cannot statically determine the access pattern of a shared array `A` (Fig. 1(a)). We allocate the shadow arrays for marking the write accesses, $\mathtt{A_w}$, and the read accesses, $\mathtt{A_r}$, and an array $\mathtt{A_{np}}$, for flagging non-privatizable elements. The loop is augmented with code (Fig. 1(b))

[1] *Privatization* creates, for each processor cooperating on the execution of the loop, private copies of the program variables. A shared variable is privatizable if it is always written in an iteration before it is read, e.g., many temporary variables. A *reduction variable* is a variable used in one operation of the form $x = x \otimes exp$, where $\otimes$ is an associative and commutative operator and x does not occur in exp or anywhere else in the loop. There are known transformations for implementing reductions in parallel [6,17,8].

that will mark during speculative execution the shadow arrays every time A is referenced (based on specific rules). The result of the marking can be seen in Fig. 1(c). The first time an element of A is written during an iteration, the corresponding element in the write shadow array $\mathtt{A_w}$ is marked. If, during any iteration, an element in A is read, but never written, then the corresponding element in the read shadow array $\mathtt{A_r}$ is marked. Another shadow array $\mathtt{A_{np}}$ is used to flag the elements of A that *cannot* be privatized: an element in $\mathtt{A_{np}}$ is marked if the corresponding element in A is both read and written, and is read first, in any iteration.

A post-execution analysis, illustrated in Fig. 1(c), determines whether there were any cross-iteration dependencies between statements referencing A as follows. If $\mathtt{any}(\mathtt{A_w}(:) \wedge \mathtt{A_r}(:))$[2] is true, then there is at least one flow- or anti-dependence that was not removed by privatizing A (some element is read and written in different iterations). If $\mathtt{any}(\mathtt{A_{np}}(:))$ is true, then A is not privatizable (some element is read before being written in an iteration). If Atw, the total number of writes marked during the parallel execution, is not equal to Atm, the total number of marks computed after the parallel execution, then there is at least one output dependence (some element is overwritten); however, if A is privatizable (i.e., if $\mathtt{any}(\mathtt{A_{np}}(:))$ is false), then these dependencies were removed by privatizing A.



Operation	Value				
	1	2	3	4	5
Aw	0	1	0	1	0
Ar	1	1	1	1	0
Anp	1	1	1	1	0
Aw(:) ∧Ar(:)	0	1	0	1	0
Aw(:) ∧Anp(:)	0	1	0	1	0
Atw	3				
Atm	2				

Fig. 1. Do loop (a) transformed for speculative execution, (b) the markwrite and markread operations update the appropriate shadow arrays, (c) shadow arrays after loop execution. In this example, the test fails.

2.2 Overhead of the LRPD Test for Dense Access Patterns

The overhead spent performing the LRPD test scales well with the number of processors and data set size of the parallelized loop. For dense access patterns the best choice for the shadow structures are *shadow arrays* conformable to the arrays under test because they provide fast random access to its elements and can be readily analyzed in parallel during the post-execution phase. The efficiency

[2] any returns the "OR" of its vector operand's elements, i.e., $\mathtt{any}(\mathtt{v}(1:\mathtt{n})) = (\mathtt{v}(1) \vee \mathtt{v}(2) \vee \ldots \vee \mathtt{v}(\mathtt{n}))$.

of the algorithm will be high because (almost) all allocated shadow space will be used. We can break down the time spent testing a loop with the LRPD test into the following components:

1. The *initialization of shadow structures* - takes time proportional to the dimension of the shadow structures (arrays).
2. *Checkpointing* the state of the program before entering speculation takes time proportional to the number of distinct shared data structures that may be modified by the loop. The work involved is approximately equal to saving all modified shared arrays and thus very program dependent.
3. The overhead associated with the execution of the *speculative loop* is equal to the time spent marking (recording) the references to the arrays under test, i.e., proportional with their dynamic count.
4. The final *analysis of the marked shadow structures* will be, in the worst case, proportional to the number of distinct memory references marked on each processor and to the (logarithm of the) number of processors. For dense access patterns this phase is equivalent to the parallel merge of p shadow arrays.
5. If the speculation fails, the *safe re-execution of the loop* may cost as much as the restoration of the checkpointed variables and a sequential re-execution of the original loop.

Each of these steps is fully parallel and scales with the number of processors. Another important measure of performance of run-time parallelization is its *relative efficiency*. We define this efficiency as the ratio between the speedup obtained through our techniques and the speedup obtained through hand-parallelization. In case hand-parallelization is not possible due to the dynamic nature of the code then we measure an ideal speedup. Another measure of performance is the *potential slowdown*, i.e., the ratio between sequential, un-parallelized execution time and the time it takes to speculate, fail, and re-execute. Our goal is to simultaneously maximize these two measures (equal to 1) and thus obtain an optimized application with good performance.

While we do not consider increasing efficiency and reducing potential slowdown as being orthogonal, in this paper we will present, to a large extent, avenues to improve *relative efficiency*, i.e., how to increase speedups obtained for successful speculation.

2.3 Some Specific Problems in Sparse Code Parallelization

The run-time overhead associated with loops exhibiting a sparse access pattern has the same break-down as the one described in the previous section. However the scalability and relative efficiency of the technique is, for practical purposes, jeopardized if we use the same implementation as the one used for dense problems

The essential difficulty in sparse codes is that the dimension of the array tested may be orders of magnitude larger than the number of distinct elements referenced by the parallelized loop. Therefore the use of shadow arrays can become prohibitively expensive: In addition to allocating much more memory than

necessary (causing all the associated problems) the work of the initialization, analysis and checkpointing phases would not scale with data size and/or number of processors. We would have to traverse many more elements than have been actually referenced by the loop and thus drastically reduce the relative efficiency of our general technique.

For these reasons we have concluded that sparse codes need a compacted shadow structure. However, such data structures (e.g., hash tables, linked lists, etc) do not have, in general, the desirable random, fast access time of arrays. This in turn will increase the overhead represented by the actual marking (tracing) of references under test, during the execution of the speculative loop.

Another important optimization specific to the sparse codes is the parallelization of reductions. This is a very common operation in scientific codes and has also to be specialized for the case of sparse codes. We have developed such techniques [3] but they will not constitute the focus of this paper.

Sparse codes rely almost exclusively on indirect, often multi–level addressing. Furthermore, such loops may traverse linked lists (implemented with arrays) and use equivalenced offset arrays to build C-like structures. These characteristics, as we will show later, result in a statically completely un-analyzable situation in which even the most standard transformations like loop distribution and constant propagation, cannot be performed (all statements end up in one strongly connected component). It is therefore clear that different, more aggressive techniques are needed. We will further show that a possible solution to these problems is the use of compiler heuristics to speculate on the type of data structures used by the original code, which can be verified at run-time.

A representative and complex example can be found in SPICE 2G6, a well known and much used circuit simulation code, in subroutine BJT. The unstructured loop (implemented with `goto` statements) traverses a linked list and evaluates the model of a transistor. Then it updates the global circuit matrix (a sparse reduction). All shared memory references are to arrays that are equivalenced to the same name (`value`) and use several levels of indirection. Because almost all references may be aliased, no classic compiler analysis can be directly applied.

3 Overhead Minimization

Our simple performance model of the LRPD test gives us general directions for performance improvement. To reduce slowdown we need to improve the probability of successful parallelization and reduce the time it takes to fail a speculation. The techniques handling this problem are important but will not be detailed in this paper. Instead, we will now present several methods to reduce the run-time overhead associated with run-time parallelization: First we will present a generally applicable technique that uses compile time (static) information to reduce the number of references that need to be traced (marked) during speculative execution. Then in Section 4 we will present a method for sparse codes that speculates about the data structures and reference patterns of the original loop and customizes the shape and size of the shadow structures.

3.1 Redundant Marking Elimination

Same-Address Type Based Aggregation While in previous implementations we have traced every reference to the arrays under test we have found that such an approach incorporates significant redundancy. We only need to detect attributes of the reference pattern that will insure correct parallelization of loops. For this purpose memory references can be classified, similar to [4] as: (1) Read only (RO), (2) Write-first (WF), (3) Read-first-write (RW), (4) Not referenced (NO).

NO or RO references can never introduce data dependences. WF references can always be privatized. RW accesses must occur in only one iteration (or processor) otherwise they will cause flow-dependences and invalidate the speculative parallelization. The overall goal of the algorithm is to mark only the necessary and sufficient sites to unambiguously establish the type of reference: WF,RO,RW or NO by using the dominance (on the control graph) relationship.

Based on the control flow graph of the loop we can aggregate the marking of read and/or write references (*to the same address*) into one of the categories listed above and replace them with a single marking instruction. The intuitive and elementary rule for combining Reads and Writes *to the same address* is shown in Figure 2.

The algorithm relies on a DFS traversal of the control dependence graph (CDG) and the recursive combination of the elementary constructs (elementary CDG's) shown in Figure 2. First all Read and Write references are initialized to RO and WF respectively. Then, at every step of the CDG traversal we attempt to aggregate the siblings with the parent of the subgraph, remove the original marks and add the new one at the root of the subgraph. When marks of siblings cannot be directly replaced with a mark of the parent (because they are not of the same type) then references (marks) and their predicates are OR'ed together and passed to the next level. Simplification of boolean expressions will enhance the chance of success. The final output of the algorithm is a loop with fewer marks than the number of memory references under test. Of course, the effectiveness of this method is program dependent and thus does not always lead to significant improvement (fewer marks).

It is important to remark that if predicates of references are loop invariant then the access pattern can be completely analyzed before the loop execution in an inspector phase. This inspector would be equivalent to a LRPD test (or simpler run-time check) of a generalized address descriptor. Such address descriptors have been implemented in a more restricted form (for structured control-flow graphs) in [11].

Grouping of Related References We say that **two memory addresses are related** if they can be expressed as a function of the same base pointer. For example, when subscripts are of the form *ptr +affine function*, then all addresses starting at the pointer *ptr* are related. For example, in SPICE, we find many indices to be of the form *ptr* + *const*, where *const* takes values from 1 to 50. In fact they are constructed through offset `equivalence` declarations for the

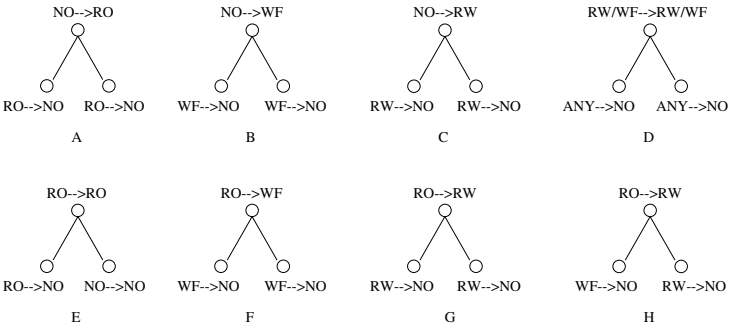


Fig. 2. Simple aggregation situations. The currently visited node is root of an elementary *cdg*. XX before aggregation is transformed into YY after aggregation. ANY denotes R or W. In (D), if the root is RW or WF, then it remains that way and the previous marks of the children, if any, are removed.

purpose of building C-like structures (`struct`). The *ptr* takes a different value at every iteration.

Intuitively, two related references of the same type can be aggregated for the purpose of marking if they are executed under the same control flow conditions, or more aggressively, if the predicates guarding one reference imply the other reference.

More formally, we will define a *marking group* as set of subscript expressions of references to an array under run-time test that satisfies the following conditions:

- The addresses are derived from the same base pointer.
- For every path from the entry of the considered block to its exit all *related* array references are of the same type (same attribute from the list WF, RO, RW, NO).

The *grouping algorithm* tries to find a minimum number of disjoint sets of references of maximum cardinality (subscript expressions) to the array under test. These groups can then be marked as a single abstract reference. The net result is:

- A reduced number of marking instructions (because we mark several individual references at once) and
- A reduced size (dimension) of the shadow structure that needs to be allocated because we map several distinct references into a single marking point.

Algorithm Outline A. CDG and colorCDG construction. We represent control dependence relationships in a control dependence graph, with the same vertices as the CFG and an edge $(X - cd \rightarrow Y)$ whenever Y is control dependent on X. Figure 4(a) shows the CDG for the loop example in Figure 5. In Figure 4(a), each edge is marked as a predicate expression. For multiple nodes that are control dependent on one node with the same predicate expression, (e.g., Node S2, S3, S4 are control dependent on node S1 with predicate expression A) we put a

branch node between S1 and S2, S3, S4 with label A. We name the resulting graph a colorCDG: The white node is the original CDG node and the black node is a branch node. The corresponding colorCDG for example in Figure 5 is shown in Figure 4(b), where node S1 represents an `IF` statement which leads two branch nodes. Each of these two nodes leads to multiple cdg nodes which are control dependent on the edge (S1,A) and (S1,NOT A).

B. Recursive Grouping. For each CDG node, in DFS order, the `extract_grp` function returns the group sets of the current child colorCDG. Siblings are visited in control flow order. In our example, the grouping heuristic is applied in three places: S1, S2, S3. Since references in one CDG node have the same predicate, the `compute_local_grp` function only needs to put subscripts with same base pointer and access type into one group. In `grp_union`, we do the same work as that in `compute_local_grp` except the operators are groups of subscripts. When two groups with common elements (same subscript expressions) cannot be combined, we compute their intersection (a set operation) which can generate three new groups:

$out\_group1 = group1 - group2$
$out\_group2 = group1 \cap group2$
$out\_group3 = group2 - group1$

The access type and predicate properties of out_group1 and out_group3 retain those of group1 and group2. The access type and predicate properties of out_group2 are the union of that of group1 and group2. This algorithm is sketched in Figure 3.

```
extract_grp (N, Bcond)

Input:
  CdgNode       N
  Predicate     Bcond
Output:
  Grouping      localGrp
```

```
Begin
S1 localGrp = compute_local_grp(N, Bcond)
if (N leads branch nodes) then
  for (each branch node B leaded from N)
    Grouping  branchGrp
    Predicate new_Bcond = Bcond AND
                          (Predicate of branch B)
    for (each cdg node N1 rooted in B) do
      subGrp = extract_grp(N1, new_Bcond)
S2    branchGrp = grp_union(branchGrp, subGrp)
    end
S3  localGrp = grp_intersect(localGrp, branchGrp)
  end
return localGrp
End
```

Fig. 3. Recursive grouping algorithm

C. Marking the Groups. In this step, we simply mark the groups where the first element of a group is referenced.

Global Reference Aggregation With the same rules as before we can even group references to addresses formed with different base pointers. In other words, we need not analyze references to each array individually but can group them together as long as they follow the same access pattern. This straight forward extension will not lead to smaller shadow structures (because we have to collect all pointer values) but may reduce the calls to marking routines.

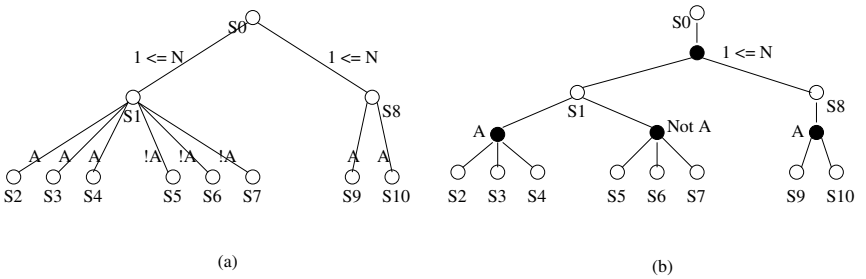


Fig. 4. CDG and colorCDG of the loop example

```
Before marking

S0      DO i = 1,N
S1        IF (A) THEN
S2          A(B(i)+1) = ...
S3          A(B(i)+2) = ...
S4          A(B(i)+3) = ...
          ELSE
S5          .. = A(B(i)+2)
S6          .. = A(B(i)+3)
S7          .. = A(B(i)+4)
          ENDIF
S8        IF (A) THEN
S9          A(B(i)+5) = ...
S10         A(B(i)+6) = ...
          ENDIF
        ENDDO

Groups:

        grp1: {B(i)+i | i=1,5,6}
        grp2: {B(i)+i | i=2,3}
        grp3: {B(i)+i | i=4}
```

```
After grouping and marking

S0      DO i = 1,N
S1        IF (A) THEN
            MARK_WRITE( grp1 )
S2          A(B(i)+1) = ...
            MARK_WRITE( grp2 )
S3          A(B(i)+2) = ...
S4          A(B(i)+3) = ...
          ELSE
            MARK_READ( grp2 )
S5          .. = A(B(i)+2)
S6          .. = A(B(i)+3)
            MARK_READ( grp3 )
S7          .. = A(B(i)+4)
          ENDIF
S8        IF (A) THEN
S9          A(B(i)+5) = ...
S10         A(B(i)+6) = ...
          ENDIF
        ENDDO
```

Fig. 5. Example of loop, obtained groups and resulting loop marked for speculative execution

A different possibility is that different base pointer groups follow the exact same reference pattern, e.g., two completely different arrays are traversed in the same manner. In this case only one of the arrays will be marked, the tracing of the second one being redundant.

For example, in the program SPICE this global aggregation is made somewhat more difficult because the different base pointers point into the same global array. So even if the access to different base pointers can be marked at the same time we cannot merge their shadow representation. Each pointer will have its own stride even if they can be marked together. Still, this optimization can lead to performance improvements.

The situation is more favorable in P3M where several arrays under test have the same access pattern and are referenced under the same conditions. The different arrays can be mapped to a single shadow array that, when analyzed after loop execution, can qualify the correctness of the parallelization. Furthermore,

even if only some of the references to different arrays can be grouped together we can still significantly reduce the run-time overhead.[3]

4 Shadow Structures for Sparse Codes

Many sparse codes use linked structure traversals when processing their data structures. The referenced pointers can, in principle, take any value (in address space) and give the overall 'impression' of being very sparse and random. For example, in SPICE 2G6 the device evaluation loops (in subroutine `load` and its descendants, e.g., BJT) traverse linked lists and process C-like structures pointed to by each node in the list. Because the program does its own memory management out of a large statically allocated array, all pointers index into the same space (the code uses different array names but they are overlaid). This makes the task of efficiently shadowing and representing memory references seem extremely difficult.

However a static analysis reveals a single statement strongly connected component, a recurrence between address and data, that is initialized before the loop and whose values are used as indices in the loop body. It is of the form `loc = NODPLC(loc)`. Furthermore, we can find more such recurrences in the loop body, with the difference that they are initialized within loop.

After this type of static analysis we can speculate with a high degree of confidence that the code traverses a linked list and that the addresses it references are in some 'advantageous' order which is amenable to optimization.

We have therefore identified the base-pointers used by the loop (the various names of overlaid names) and classified their accesses as:

(A) monotonic accesses with constant stride, **(B)** monotonic accesses with variable stride and **(C)** random access patterns.

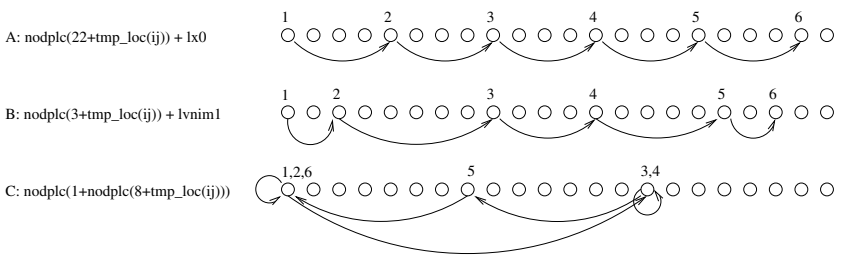


Fig. 6. (A)access region with constant stride, (B) access region with non-constant stride but monotonic, (C) access region with non-constant stride and not monotonic (random). Examples are from the BJT loop in SPICE.

Figures 6 and 7 show examples of such accesses. For each of these possible reference patterns we have adopted a specialized representation.

[3] This part of the algorithm has not yet been implemented.

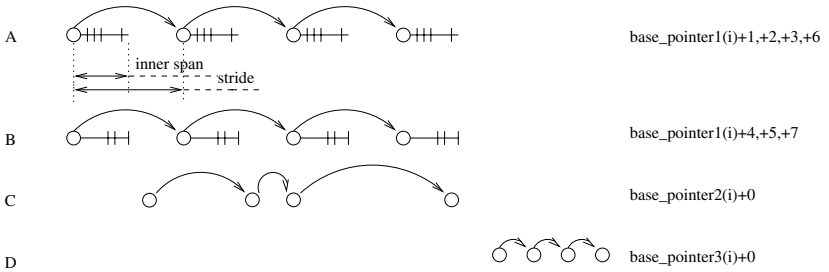


Fig. 7. (A) and (B) have the same base pointers, inner span, and stride. Actual array reference indices are different, because they are in two groups. To verify no overlap between A and B, only check whether 'stride > inner span'; (A) and (C) have different base pointers, and C doesn't have constant stride. To verify for no overlap between A and C, merge A and C and check for collisions; (C) and (D) To verify for no overlap between C and D, compare ranges. Examples have been abstracted from loop BJT in SPICE.

- monotonic constant strides can be recorded in a triplet [offset,stride,count]
- monotonic addresses with variable stride can be recorded in an array with the additional fields [min,max] of their value
- random addresses can be stored in hash tables (if we expect a large number of them) or simple lists which are to be sorted later. Range information will also be maintained and recorded.

The run-time marking routines are adaptive, i.e., they will verify the class of the access pattern and use the simplest possible form of representation. Ideally all references can be stored as a triplet, dramatically reducing the space requirements. In the worst case, the shadow structures will be proportional to the number of marked references. **The type of reference**, i.e., WF, RO, RW and NO will be recorded in a **bit vector** which could be as long as the number recorded references.

After loop execution the analysis of the recorded references will again use algorithms that range from the simplest to the most time consuming. We will test the data dependence conditions by detecting if pointers (and their associated groups, as defined in Section 3.1) collide through the following hierarchical procedure:

- Check for overlap of address ranges traversed by the base pointers (linked lists) using min/max information.
- If there is overlap then check (analytically) triplets for collisions; Check collision of monotonic stride lists by merging them into one array
- Sort random accesses stored in lists (if they exist) and merge into other the previous arrays. (Self collisions will be detected during sorting)
- Merge hash tables (if they exist) into the previous arrays. (Self collisions will be detected at insertion time)

When (if) a collision is detected, then the type of reference will be read from the bit vector for that particular address and any possible data dependence will be detected.

This scheme uses shadow data structures that are, in general, more expensive (no random access) to access and analyze than the shadow arrays used in dense problems. However, if the speculation about the code's reference pattern is correct then storage requirements are minimized and only inexpensive operations will be performed. Of course, should the speculation fail then the only advantage of this technique is its compact storage. As we will show in Section 5.1, we have devised reasonably accurate compile time heuristics for a successful speculation.

5 Experimental Results

5.1 Run-Time Overhead Reduction

We have implemented the previously presented method of reducing marking points in a program through the grouping algorithm in the POLARIS compiler infrastructure [2].

The grouping algorithm has been implemented as part of our run-time parallelization pass, the last optimization/transformation step before the code generation pass in Polaris. We ran it on several important loops from the Perfect Benchmarks (SPICE2G6, Ocean), SPEC (TFFT2) and a N-body code from NCSA (P3M). In Figure 8 we compare the number of references to the arrays under run-time test in the original code, the number of references that were marked in a previous implementation of the LRPD test (that already had some optimizations based on simple dominator relation between references) and the resulting number of static marking after applying the grouping technique. The reduction is significant in all cases and does indeed contribute to improved performance. The actual performance improvement is more impacted by the *dynamic* counts of the marking code and is also quite significant.

Program : Loop		Static			Dynamic		
	# of references	# of marks before group	# of marks after group	reduction %	# of marks before group	# of ma rks after group	reduction %
SPICE2G6:BJT_do	259	150	13	91.3%	68	11	83.88%
P3M:PP_do100	24	24	12	50%	5081	3020	40.57%
OCEAN:Ftrvmt_do9109	18	6	3	50%	258/128	129/64	50%
TFFT2:Cfftz_do#1	18	18	8	55%	98304	30720	68.75%

Fig. 8. Reduction of static/dynamic marking points using the grouping al gorithm

We have applied the technique to several loops from SPICE, OCEAN (PERFECT codes), P3M (an NCSA benchmark) and TFFT2, a SPEC benchmark.

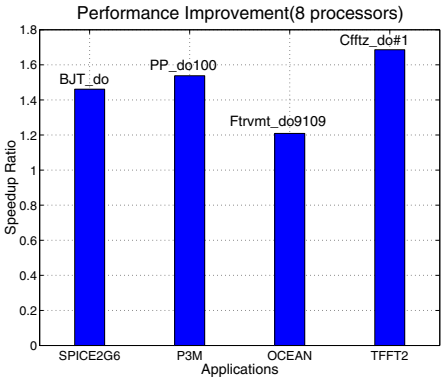


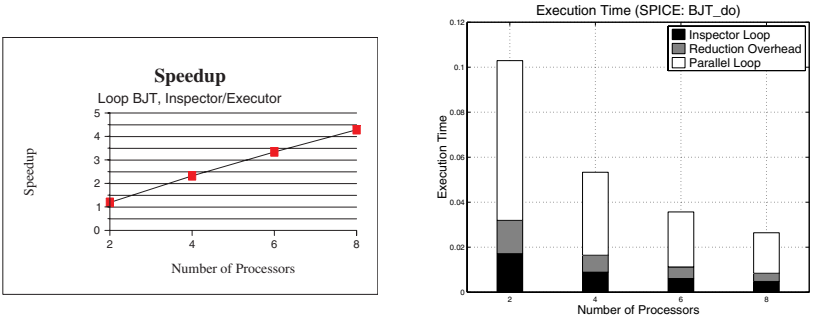
Fig. 9. Performance improvement through redundant marking elimination using grouping. $Speedup - Ratio = \frac{Execution-time-of-loop(before-grouping)}{Execution-time-of-loop(after-grouping)}$

We will now use the main loop in subroutine BJT from SPICE as our case study and give statistical results for all the other loops.

A Case Study: SPICE 2G6 We have chosen as the target of our detailed experiment the loop in subroutine BJT of the SPICE 2G6 code. This loop has an almost identical access pattern as most of the device evaluation step and represents between 31% and 57% of the total execution time of the code. The SPICE2G6 program is a very sparse code and thus offered us the opportunity to evaluate both our grouping methods (which are also applicable to dense codes) as well as the choice of shadow structures and sparse reduction validation and optimized parallelization.

The unstructured loop has first been brought to a structured `do` loop form (a separate pass we have developed in Polaris). Then, through a different technique we have recently developed, we have distributed the dominating recurrence outside the loop: This is in fact the loop containing the linked list traversal that controls the traversal of all data structures of the loop and has the form `LOC = NODPLC(LOC)`. This first loop is executed sequentially and all pointers are collected in a temporary array of pointers that is used by the remainder of the BJT loop (and has random access).

Then we have used the run-time pass of the compiler to instrument the minimal number of reference groups for run-time marking. The loop invariant part of the marked addresses has been hoisted outside the loop and set up as an inspector loop. It represents the flow insensitive traversal of all base pointers (13 of them) that the loop can reference. These are the base pointers of all marking groups. The predicates guarding their actual execution are loop variant and had to be left for marking inside the loop itself. The traversal and analysis of the inspector loop gives us a conservative result about the existence of any cross-processor collisions (overlaps) between the references. The shadow data structures used by our **Run-time library** for reference tracing are *triplets* for 7 pointers, list of values for 3 other the pointers and hash tables for the reduction



(a) Speedup - Loop BJT (b) Breakdown of Execution Time - Loop BJT

Fig. 10. The input data is extended from a 8 bits adder. the execution time of loop BJT is about 31% of total execution time of SPICE.

operand addresses. Had our 'guess' been incorrect, then our adaptive run-time library would have automatically 'demoted' the triplets (for linked lists with constant, monotonic stride) to lists and then hash tables. The run-time library also collects range information on the fly (min/max values of specific base pointers). Then we have generated four versions of the loop that represent a combination of four situations:

1. Conservative test (inspector) is sufficient to qualify the loop as parallel
2. Speculative execution is needed in order to mark the dynamic existence of the groups (based on the actual control flow) and qualify/disqualify the loop as parallel after execution
3. The reduction parallelization needs to be verified (not described in this paper)
4. The parallelization is known to be valid because it has been proven in a previous instantiation and no modifications of addresses has been found in the outer loop.

Finally we have instrumented (with the help of the same grouping algorithms) the remainder of the loop containing BJT to flag any shared integer variable (potential address modification). Depending on the dynamic situation, simple code generated by the compiler decides which version to run.

In our experiments with two different input sets we have had to run the conservative inspector and and validate the reduction parallelization only three times: The first time and two other times when address modification outside the loop have been flagged. (For the reduction validation it was sufficient to show that the range of the reduction operand addresses did not overlap with the rest of the references.)

The experimental setup for our speedup measurement consisted of a 16 processor HP-V class system with 4Gb memory, running the HPUX11 operating system.

Figure 10(a) reports overall actual obtained speedup. The results seem to scale up to 8 processors. We have not reported numbers for larger number of

processors because our input set was fairly small. (Forking overhead is 5% of the serial time - very significant). Execution time breakdown per phase is presented in this Figure 10(b).

6 Conclusion

The paper presents several techniques to increase the potential speedup and efficiency of run-time parallelized loops. Great emphasis has been put on efficiently applying the run-time parallelization for sparse codes. The detailed case study, SPICE, is one of the most difficult codes and our techniques have proven themselves to be quite useful. We believe that other sparse codes will behave similarly or better. SPICE is an interesting case study because it requires all the above methods (and more) and - more importantly - is the most similar to the problems arising in C codes: memory management, extensive use of pointers, linked structure traversals, etc. So by parallelizing SPICE we hope to gain valuable experience applicable to C programs.

References

1. Charmm: A program for macromolecular energy, minimization, and dynamics calculations. *J. of Computational Chemistry*, 4(6), 1983. 233
2. W. Blume et. al. Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, 29(12):78–82, December 1996. 244
3. Y. Hao and L. Rauchwerger Adaptive Reduction Parallelization Techniques. Tech. Rept., Dept. of Computer Science, Texas A&M Univ., Dec. 1999. 237
4. J. Hoeflinger. *Interprocedural Parallelization Using Memory Classification Analysis.* PhD thesis, University of Illinois, Urbana-Champaign, August, 1998. 238
5. S. Leung and J. Zahorjan. Improving the performance of runtime parallelization. In *4th PPOPP*, pages 83–91, May 1993. 233
6. Z. Li. Array privatization for parallel execution of loops. In *Proc. of the 19th Int. Symposium on Computer Architecture*, pages 313–322, 1992. 234
7. M. J. Frisch et. al. *Gaussian 94, Revision B.1.* Gaussian, Inc., Pittsburgh PA, 1995. 233
8. D. E. Maydan et. al. Data dependence and data-flow analysis of arrays. In *Proc. 5th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1992. 234
9. L. Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits.* PhD thesis, University of California, May 1975. 233
10. D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29:1184–1201, Dec. 1986. 233
11. Y. Paek, J. Hoeflinger, and D. Padua. Simplification of Array Access Patterns for Compiler Optimizat ions. In *Proc. of the SIGPLAN 1998 Conference on Programming Language Design and Implementation, Montreal, Canada*, June 1998. 238
12. D. Patel and L. Rauchwerger. Implementation issues of loop-level speculative runtime parallelization. In *Proc. of the 8th Int. Conference on Compiler Construction (CC'99), Amsterdam, The Netherlands.* Lecture Notes in Computer Science, Springer-Verlag, March 1999. 233

13. L. Rauchwerger, N. Amato, and D. Padua. A scalable method for run-time loop parallelization. *Int. J. Paral. Prog.*, 26(6):537–576, July 1995. 234
14. L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Trans. on Parallel and Distributed Systems*, 10(2), 1999. 233, 234
15. L. Rauchwerger and D. Padua. Parallelizing WHILE Loops for Multiprocessor Systems. In *Proc. of 9th Int. Parallel Processing Symposium*, April 1995. 234
16. J. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.*, 40(5), May 1991. 233
17. P. Tu and D. Padua. Automatic array privatization. In *Proc. 6th Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, Aug. 1993. 234
18. R. G. Whirley and B. Engelmann. *DYNA3D: A Nonlinear, Explicit, Three-Dimensional Finite Element Code For Solid and Structural Mechanics.* Lawrence Livermore Labs, Nov., 1993. 233
19. M. Wolfe. *Optimizing Compilers for Supercomputers.* The MIT Press, Boston, MA, 1989. 233
20. C. Zhu and P. C. Yew. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Trans. Softw. Eng.*, 13(6):726–739, June 1987. 233