

# The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops \*

Francis Dang, Hao Yu, Lawrence Rauchwerger  
Dept. of Computer Science, Texas A&M University  
College Station, TX 77843-3112  
{fhd4244,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. In our previously proposed framework we have speculatively executed a loop as a `doall`, and applied a fully parallel data dependence test to determine if it had any cross-processor dependences; If the test failed, then the loop was re-executed serially. While this method exploits `doall` parallelism well, it can cause slowdowns for loops with even one cross-processor flow dependence because we have to re-execute sequentially. Moreover, the existing, partial parallelism of loops is not exploited. We now propose a generalization of our speculative `doall` parallelization technique, called the Recursive LRPD test, that can extract and exploit the maximum available parallelism of any loop and that limits potential slowdowns to the overhead of the run-time dependence test itself. In this paper we present the base algorithm and an analysis of the different heuristics for its practical application and a few experimental results on loops from *Track*, *Spice*, and *FMA3D*.*

## 1. Improving Loop Parallelization Coverage

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. Restructuring, or parallelizing, compilers detect and exploit parallelism in programs written in conventional sequential languages or 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., [12, 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 [11], DYNAS3D [18], GAUSSIAN [9], and CHARMM [1]. 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 input dependent.

In previous work we have taken two different approaches to run time parallelization. First, we have employed the LRPD test [14], to speculatively execute a loop as a `doall` and subsequently test whether the execution was correct. If not, the loop was re-executed sequentially. While for fully parallel loops the method performs very well, partially parallel loops experience a slow-down equal to the speculative parallel execution time. Second, for loops which were presumed to be partially parallel we have developed an inspector/executor technique [13] in which we record the relevant memory references and then employ a sorting based technique to construct the iteration dependence graph of the loop and schedule the iterations in topological order. The major limitation of this method is its assumption that a proper inspector loop exists. If there is a dependence cycle between data and address computation of the shared arrays then a proper, side-effect free inspector of the traversed address space cannot be obtained. (It would be most of the analyzed loop itself.) Furthermore, the technique requires large additional data structures (proportional to the reference trace). Kazi and Lilja [6] parallelize partially parallel

---

\*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

loops by using a DOACROSS mechanism for enforcing dependencies, executing in private storage and committing in order, after any possibility of further dependence violation has passed. This method requires a set-up phase for every iteration during which all potential dependence causing addresses are pre-computed and then broadcast to all processors. This information is used to set tags for future advance/await type synchronizations. The method can never properly exploit large amounts of parallelism and does not remove the need for address pre-computation, i.e., an inspector per iteration. Thus, it cannot parallelize loops in which address and data depend upon one another.

In this paper we will present a new technique to extract the maximum available parallelism from a partially parallel loop that removes the limitations of previous techniques, i.e., it can be applied to any loop and requires less memory overhead. We propose to transform a partially parallel loop into a sequence of fully parallel loops. At each stage, we speculatively execute all remaining iterations in parallel and the LRPD test is applied to detect the potential dependences. All correctly executed iterations are committed, and the process recurses on the remaining iterations. The only limitation is that the loop has to be statically block scheduled in increasing order of iteration and thus may become imbalanced. This negative effect can be reduced through *dynamic feedback guided scheduling*, a dynamic load balancing technique described in Section 5.2. Furthermore, we show how the new technique can be used to extract the full data dependence graph (DDG) for any loop and then used for 'optimal' scheduling.

An additional benefit of this technique is the overall reduction in potential slowdowns that simple `doall` speculation can incur when the compiler and/or user guesses wrong. In effect, by applying this new method exclusively we can remove the uncertainty or unpredictability of execution time – we can guarantee that a speculatively parallelized program will run at least as fast as its sequential version and with some additional testing overhead.

## 2. The Recursive LRPD Test (R-LRPD)

In our previous work [14] we have described the LRPD test as a technique for detecting `doall` loops. At run-time, it 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 sequentially. To qualify more parallel loops, *array privatization* and *reduction parallelization* can be speculatively applied and their validity tested after loop termination.<sup>1</sup>

<sup>1</sup>*Privatization* creates, for each processor executing 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

To reduce overhead and qualify even dependent loops as parallel we test the *the copy-in* condition instead of the privatization condition. That is, instead of checking that every Read is covered by a Write to a memory location, i.e., a  $(Write|Read)^*$  pattern we check for a reference pattern of the form  $(Read^*|(Write|Read)^*)$ . If the condition holds then the memory location can be transformed for safe parallel execution by initializing its private storage with the original shared data. In practice we need to test at run-time if the latest consecutive reading iteration (maximum read) is before the earliest writing iteration (minimum write) – for all references of the loop.

In addition, we use a *processor-wise* test which checks only for cross-processor dependences rather than loop carried dependences. In a processor-wise test (always preferable) we have to schedule the loop statically (blocked). While this is a limitation it also simplifies the tested conditions: Highest reading processor  $\leq$  lowest writing processor. The initialization of the private arrays can be done either before the start of the speculative loop or, preferably, as an 'on-demand copy-in' (read-in if the memory element has not been written before).

Thus the only reference pattern that can still invalidate a speculative parallelization is a flow dependence between processors (a write on a lower processor matched by a read from a higher processor) We now make the crucial observation that in any block-scheduled loop executed under the processor-wise LRPD test, the iterations that are less than or equal to the source of the first detected dependence arc are always executed correctly. Only iterations larger or equal to the earliest sink of any dependence arc need to be re-executed. This means that only the remainder of the work (of the loop) needs to be re-executed, as opposed to the original LRPD test which would re-execute the entire loop sequentially.

To re-execute the fraction of the iterations assigned to the processors that may have worked off erroneous data we repair the unsatisfied dependences by initializing their privatized memory with the data produced by the lower ranked processors. Alternatively, we can commit (i.e., copy-out) the correctly computed data from private to shared storage and use on-demand copy-in during re-execution. We can then re-apply the LRPD test recursively on the remaining processors, until all processors have correctly finished their work. We call this application of the test the Recursive LRPD test.

There are several options for implementing this basic strategy. They differ in the manner in which the iterations are assigned to the processors. We begin with the simplest

$x = x \otimes exp$ , where  $\otimes$  is an associative and commutative operator and  $x$  does not occur in  $exp$  or anywhere else in the loop. Transformations are known for implementing reductions in parallel [17, 10, 8].

For clarity reduction parallelization is not presented in the following discussion; it is tested in a similar manner as independence and privatization.

and then describe potential optimizations.

**The Non-Redistribution (NRD) Strategy.** Let us 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,  $A_w$ , and the read accesses,  $A_r$ . The loop is augmented with marking code (Fig. 1(b)) and enclosed in a `while` loop that repeats the speculative parallelization until the loop completes successfully. We use two bits for Read and Write. On a processor, if the Read occurs before the Write, then both bits will remain set – indicating the reference is not privatizable. If the Write occurs first, then any subsequent Read will not set the read bit. Repeated references of the same type to an element on a processor do not change the shadow arrays. The array A is first privatized. Read-first references will copy-in on-demand the content of the shared array A. Array B, which is not tested (it is statically analyzable), is checkpointed. The result of the marking after the first speculative `doall` can be seen in Fig. 1(c). After the analysis phase we copy (commit) the elements of A that have been computed on processors 1 and 2 to their shared counterpart (by taking their last written value). This step also insures that flow-dependences will be satisfied during the next stage of parallel execution (we will read-in data produced in the previous stage). We further need to restore the section of array B that is modified/used in processors 3 and 4 so that a correct state is established for all arrays. (In our simple example this is not really necessary because we would overwrite B).

In the non-redistribution strategy (NRD), the `Re-Init` step in Fig. 1(b) re-initializes the shadow arrays on all processors that have not successfully completed their assigned iterations yet, (processors 3 and 4). Then, a new parallel loop is started on these processors for the remainder of the iterations (5-8 in this case). In the final state (Fig. 1(d)) all data can be committed. The loop finishes in a total of two steps of two iterations each.

**The Redistribution (RD) Strategy.** In Fig. 1(e) we adopt the redistribution (RD) strategy. Instead of re-executing only on the processors that have incorrect data and leaving the rest of them idle (NRD), at every stage we redistribute the remainder of the work across all processors. There are pros and cons for this approach. Through redistribution of the work we employ all processors all the time and thus decrease the execution time of each stage (instead of staying constant, as in the NRD case). The disadvantage is that we may uncover new dependences across processors which were satisfied before by executing on the same processor. Moreover, with redistribution there is the potentially large cost of more remote misses.

```

do i = 1,8
  B(i) = f(i)
  z = A[K[i]]
  A[L[i]] = z + C[i]
enddo
L[1:8] = [2,2,4,4,2,1,5,5]
K[1:8] = [1,2,3,4,1,2,4,2]

```

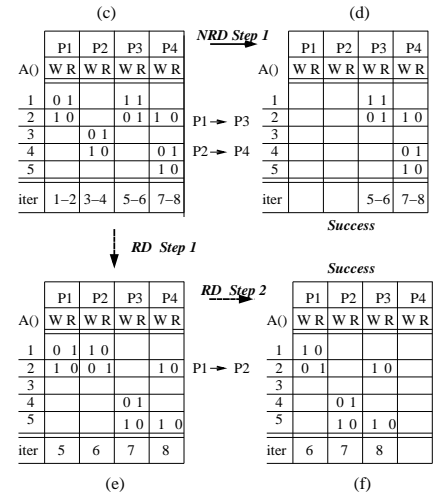
(a)

```

start = newstart = 1 ; end = newend = 8
success = .false.; Init shadow arrays
Checkpoint B(:)
While (.not. success) do
  Doall i = newstart, newend
    B(i) = f(i)
    z = pA[K[i]]
    pA[L{i}] = z + C{i}
    markread (K{i}); markwrite (L{i})
  Enddoall
  Analyze (success, start, end, newstart,newend)
  If (.not. success) then
    Restore B(newstart, newend)
    Re-Init (Shadows, pA)
  endif
  Commit (A(start, newstart-1),B(start,newstart-1))
End While

```

(b)



**Figure 1.** `do` loop (a) transformed for recursive speculative execution, (b) the `markwrite` and `markread` operations update the appropriate shadow arrays. The test is repeated until `success` becomes true. B is an independent array that is checkpointed and partially restored at every stage. pA is the privatized array A that is initialized to A and partially committed at every stage. (c) State of shadow arrays after first LRPD test. Note the cross-processor dependences. (d) State of shadow arrays after second (and successful) LRPD test on processors 3 and 4 only (NRD). (e) State of shadow arrays after second LRPD test when remainder of work is redistributed (RD) on all processors. Note the newly uncovered dependences. (f) Final state of shadow arrays after the second (and successful) LRPD test with work redistribution (RD).

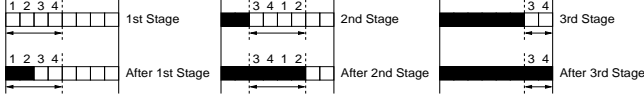


Figure 2. Sliding Window Strategy - An Example.

**The Sliding Window (SW) Strategy.** The performance of the R-LRPD test is very dependent on the distribution and type of data dependences encountered. For codes with long distance data dependences we have devised the *sliding window* strategy. Instead of distributing the *entire* iteration space over all the available processors, we strip-mine the entire speculative execution process and apply the R-LRPD test on each strip (contiguous set) of iterations. In the example shown in Fig. 2 we first speculatively execute the first 4 contiguous iterations (1–4) on the four processors. The subsequent analysis phase commits iteration blocks 1 and 2, re-schedules iterations 3 and 4 (due to a dependence between processors two and three) and advances the *commit point* to iteration 3. Then the higher iterations (5 and 6) are scheduled and the R-LRPD test is applied again. All 4 iterations can now be committed (3-6). Finally the last two iterations are speculatively executed on processors 3 and 4.

To increase memory reference locality we organize the sliding window in a circular manner such that iterations are re-executed (if necessary) on their originally assigned processor. There are trade-offs to be made between the Sliding Window (SW) and the previously presented strategies. For a fully parallel loop (N)RD methods execute in one stage, i.e., with one global synchronization, while SW will have one synchronization per strip. If dependences are present, it is possible that (N)RD techniques need to re-execute many more iterations than SW. The SW strategy has potentially more analysis overhead because it may have to go over the shadows of the memory elements that are reused in every iteration. So far we have not devised a strategy to choose between the two techniques except through the use of history based predictions.

The window size, i.e., the size of the block of contiguous iterations (super-iteration) assigned to one processor affects the number of global synchronizations (a larger window needs fewer global synchronizations) and the number of uncovered dependences. So far we have been able to tune our technique only experimentally (empirically). The scheduled block sizes can be dynamically adjusted by applying history based prediction. When many close dependences are encountered, the block size is increased. Alternatively, we can start with a very large block, equivalent to (N)RD and, if dependences are uncovered, reduce it until no re-executions are needed.

### 3. Extracting Data Dependence Graphs (DDG)

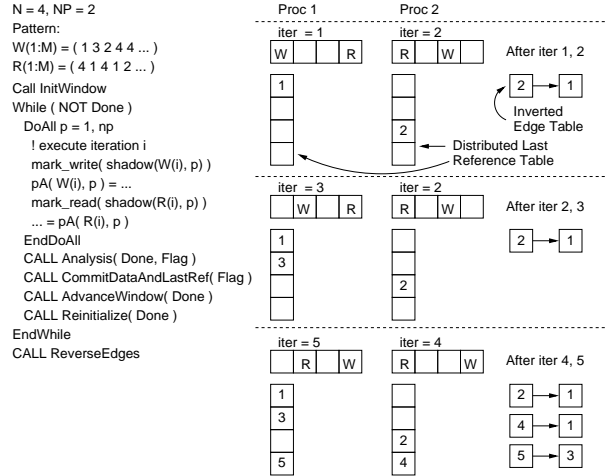


Figure 3. Instrumented code to generate DD graph using the Sliding Window R-LRPD test. "InitializeWindow" and "AdvanceWindow" control the iteration window. "Analysis" applies the LRPD test for iterations within the window and generates the DD graph edges. **STEP 1** After iterations 1 and 2 are executed in parallel, one cross-processor flow dependence is found and recorded in the inverted edge table. **STEP 2** No cross-processor flow dependences are found. **STEP 3** Two *cross-window* flow dependences are detected and recorded in the inverted edge table.

Although the R-LRPD test can extract parallelism from loops for which a proper inspector does not exist, it cannot always extract the maximum available parallelism. For some loops with complex dependence graphs but significant intrinsic parallelism the R-LRPD test may generate an almost sequential execution schedule. In such cases it would be beneficial to extract the (iteration) data dependence graph (DDG) and, and generate an optimized schedule. Somewhat similar techniques have been previously presented in the literature [7, 16, 22, 13, 6], but apply only to loops from which a proper inspector can be extracted.

We propose to use the Sliding Window R-LRPD test to detect, window by window, the edges of the DDG and store them as (Read, Write) pairs. The shadow arrays are organized as n-level mark list where n is the number of iterations assigned to each processor. A distributed last reference table maintains the last valid write for each memory address. This is used to detect cross-window flow dependences between a successfully completed iteration and an iteration inside the current window. After each stage (a doall), we perform the SW R-LRPD test. Every cross-processor dependence between multiply referenced memory elements is logged into the inverted edge table. We also log each intra-

processor flow dependence and each cross-window flow dependence into the inverted edge table. We save the last valid write reference of each memory address into the distributed last reference table in order to record the latest write reference occurring before the current window of iterations. At the next stage we record the next set of references into the shadow array. After execution is completed, we obtain the DDG by inverting the edges in the graph stored in the inverted edge table. Fig. 3 illustrates an application of this method for two processors and one iteration assigned to each processor.

The so extracted DDG is then reduced by removing edges representing non-essential dependences (through privatization, reduction recognition, etc). We can also insert synchronizations in the DDG followed by copy-out or accumulate and copy-out operations. Finally, the iteration space scheduled using the DDG information. These last optimizations have not yet been implemented. Instead we have used the 'un-processed' DDG to obtain a wavefront schedule (sets of independent iterations separated by global synchronizations) and used them to parallelize some important loops in SPICE 2G6 (Section 5.2).

The complexity of this method is essentially the same as that of the SW R-LRPD test with some additional, constant overhead. It is important to note that in the case of sparse reference patterns (e.g. in SPICE) we have to use shadow hash-tables instead of shadow arrays. The result is an increased time per logging operation but a much more compact representation which allows faster analysis.

#### 4. Modeling the RD and NRD strategies

In [14] we have shown that if the LRPD test passes (fully parallel loop), then the obtained speedups range from nearly 100% to *at least* 25% of the ideal. The overhead spent performing the single stage (original) LRPD test scales well with the number of processors and data set size of the parallelized loop. We can break down the time spent testing and running a loop with the LRPD (single stage) test in the following *fully parallel* phases:

The *initialization of shadow structures* is proportional to their dimension. For dense access patterns the shadow arrays are conformable to the tested arrays.

The work associated with *checkpointing* the state of the program before entering speculation is proportional to the number of distinct shared data structures that may be modified by the loop. Checkpointing can be implemented before loop execution or 'on-the-fly', before the modification of a shared variable.

The overhead associated with the execution of the *speculative loop* is proportional to the cost of marking relevant data references. For dense access patterns it can be approximated by the number of distinct references under test.

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 that have participated in the speculative parallel execution. For dense access patterns this phase may involve the merge operation of  $p$  (number of processors) shadow arrays.

The recursive application of the LRPD test adds some additional overhead components which depend on the fraction of the successfully completed work which in turn depends on the data dependence structure of the loop. If cross-processor dependences are detected then a *Data Restoration* phase will restore the state of the shared arrays that were modified by the processors whose work cannot be committed. Its time is proportional to the number of elements of the shared arrays that need to be copied from their checkpointed values. If dependences are detected and re-execution is needed, then the shadow arrays will be *re-initialized*. The *Commit* phase transfers the last data computed (last value) by the earlier processors from private to shared memory. Its cost is proportional to the number of written array elements. Each of these steps is *fully parallel* and scales with the number of processors and data size. Furthermore, the commit, re-initialization of shadow arrays and restoration of modified arrays can be done concurrently as two tasks on the two disjoint groups of processors, i.e., those that performed a successful computation and those that have to restart.

The number of times re-execution is performed, as well as the work performed during each of them, depends on the strategy adopted: with or without work redistribution. When we do not redistribute work (NRD), the time complexity equals the cost of a sequential execution (worst case). We will have at most  $p$  steps performing  $n/p$  work, where  $p$  is the number of processors and  $n$  is the number of iterations. In the redistribution case (RD), each step will take progressively less time because we execute in  $p$  processors a decreasing amount of work. Completion is guaranteed in a finite number of steps because the first processor always executes correctly. Let us now model more carefully the tradeoff between these two strategies.

Initially, there are  $n$  iterations equally distributed among the processors. The computation time for each iteration is  $\omega$ , yielding a total amount of (useful) work in the loop as  $\omega n$ . In the following discussion we assume that we know  $\omega$ , the cost of useful computation in an iteration,  $\ell$ , the cost of redistributing the data for one iteration to another processor, and  $s$ , the cost of a barrier synchronization.

For the purpose of an efficient speculative parallelization we classify loop types based on their dependence distribution in the following two classes: (a) **geometric** ( $\alpha$ ) loops where a constant fraction  $(1 - \alpha)$  of the current *remaining* iterations are completed during each speculative parallelization (step), and (b) **Linear** ( $\beta$ ) loops where a constant frac-

tion  $(1 - \beta)$  of the *original* iterations are completed during each speculative parallelization (step).

**No Data Redistribution (NRD).** If  $\omega \leq \ell + s$ , then it does not pay to redistribute the remaining iterations among the  $p$  processors after a dependence is detected during a speculative parallelization attempt. That is, the overhead of the redistribution (per iteration) is larger than work of the iteration. In this case, the total time required by the parallel execution is simply

$$T_{\text{static}}(n) = \sum_{i=0}^{k_s} \left( \frac{n\omega}{p} + s \right) = \frac{n\omega k_s}{p} + k_s s \quad (1)$$

where  $k_s \leq p$  is the number of steps required to complete the speculative parallelization. Thus, to determine the time  $T_{\text{static}}(n)$  we need to compute the number of steps  $k_s$  (the number of speculative parallelization attempts needed to execute the loop). We consider two cases (the  $\alpha$  and  $\beta$  loops) and determine the value of  $k_s$  for each.

For the  $\alpha$  loops, we assume a constant fraction  $(1 - \alpha)$  of the *remaining work* is completed during each speculative parallelization step. In this case,  $n\omega\alpha^i$  work remains to be completed after  $i$  steps. Thus, the final ( $k_s$ -th) step will occur when  $n\omega\alpha^{k_s} = \frac{n\omega}{p}$  (since then all remaining iterations reside on one processor because we do not redistribute). So, solving for  $k_s$ , we get  $k_s = \log_{\frac{1}{\alpha}} p$ . For example, if  $\alpha = \frac{1}{c}$ , then  $k_s = \log_c p$ , for constant  $c$ .

For the  $\beta$  loops, we assume a constant fraction  $(1 - \beta)$  of the *original work* is completed successfully in each speculative parallelization step (i.e., a constant number of processors successfully complete their assigned iterations). In this case,  $n\omega(1 - \beta)i$  work is completed after  $i$  steps. Thus, all the work will be completed when  $n\omega(1 - \beta)k_s = n\omega$ , or when  $k_s = \frac{1}{(1 - \beta)}$ . For example, for a fully parallel loop,  $\beta = 0$  and so  $k_s = 1$  and  $T_{\text{static}}(n) = \frac{n\omega}{p} + s$ , and for a sequential loop,  $\beta = \frac{p-1}{p}$  and so  $k_s = p$  and  $T_{\text{static}}(n) = n\omega + ps$ .

**Data Redistribution (RD).** If  $\omega > \ell + s$ , then it may pay to redistribute the remaining iterations among the  $p$  processors after a dependence is detected during a speculative parallelization attempt. In this case, as opposed to the **NRD** case, in each subsequent step the processors will have a smaller number of iterations assigned to them. Thus the total time required by the parallel execution is

$$T_{\text{dyn}}(n) = \sum_{i=0}^{k_d} \left( \frac{n_i\omega}{p} + \frac{n_i\ell}{p} + s \right) \quad (2)$$

$$= \frac{(\omega + \ell)}{p} \left( \sum_{i=0}^{k_d} n_i \right) + k_d s \quad (3)$$

where  $n_i$  is the number of iterations remaining to be completed at the start of the  $i$ -th step, and  $k_d$  is the number of steps completed to this point using redistribution.

Even if redistribution is initially useful, there comes a point when it should be discontinued. In particular, it should occur only as long as the time spent (per processor) on useful computation is larger than the overhead of redistribution and synchronization. That is, redistribution should occur as long as the first term in the first sum in Eq. 2 is larger than the sum of the last two terms, i.e., as long as

$$n_{k_d} \geq \frac{ps}{\omega - \ell}. \quad (4)$$

Note that this condition can be tested at run-time since it only involves the number of uncompleted iterations which is known at run-time and  $p$ ,  $s$ ,  $\omega$ , and  $\ell$ , which we assume are known *a priori*, and can be estimated through both static analysis and experimental measurements.

In summary, for the first  $k_d$  steps, the remaining iterations should be redistributed among the processors. After that, no redistribution should occur. From this point on, we are in the case described as  $T_{\text{static}}$  above, but starting from  $n' = n_{k_d}$  instead of  $n$ . Thus, the total time required will be

$$T(n) = T_{\text{dyn}}(n) + T_{\text{static}}(n_{k_d}) \quad (5)$$

$$= \frac{(\omega + \ell)}{p} \left( \sum_{i=0}^{k_d} n_i \right) + \frac{n_{k_d}\omega k_s}{p} + (k_d + k_s)s \quad (6)$$

where  $n_i$ ,  $k_d$  and  $k_s$  are as defined above.

To compute an actual value for  $T(n)$ , we need to determine  $n_i$ ,  $k_d$ , and  $k_s$ , and substitute them in Eq. 6. For example, consider the geometric loops in which a constant fraction  $(1 - \alpha)$  of the current work is completed during each speculative parallelization attempt.<sup>2</sup> In this case,  $n_i = n\alpha^i$ , and  $\sum_{i=0}^{k_d} n_i = \sum_{i=0}^{k_d} n\alpha^i = n \left( \frac{\alpha^{k_d+1} - 1}{1 - \alpha} \right)$ . Using  $n_{k_d} = n\alpha^{k_d}$  in Eq. 4, and solving for  $k_d$  we obtain  $k_d = \log_{\alpha} \left[ \left( \frac{s}{\omega - \ell} \right) \frac{p}{n} \right]$ . Finally,  $k_s = \log_{\frac{1}{\alpha}} p$  as described above. Thus, the total time required will be

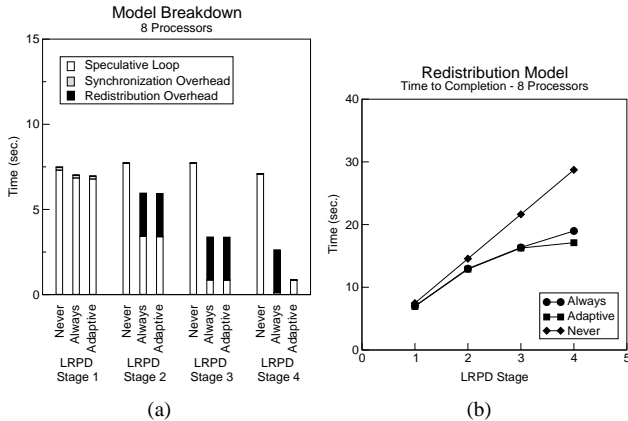
$$T(n) = \frac{n}{p} (\omega + \ell) \left( \frac{\alpha^{k_d+1} - 1}{1 - \alpha} \right) + \frac{n\alpha^{k_d}\omega k_s}{p} + (k_d + k_s)s$$

where  $k_d$  and  $k_s$  are computed as defined above based on the known values of  $n$ ,  $\omega$ ,  $\ell$ ,  $s$ , and  $\alpha$ . In general, one may not know  $\alpha$  exactly, however, in many cases reasonable estimates can be made in advance, and recomputed during execution (e.g., as an average of the  $\alpha$  values observed so far).

**Experimental Model Validation.** The graph in Fig. 4 illustrates the loop, testing overhead, and redistribution overhead time (mostly due to remote cache misses) for each

<sup>2</sup>The case in which a constant fraction of the original work is completed during each speculative parallelization is not realistic here since the number of iterations each processor is assigned varies from one speculative parallelization to another.

restart of the R-LRPD test of a synthetic loop executed on 8 processors of an HP-V2200 system. We assume that the fraction of remaining iterations is 1/2. The initial speculative run is assumed not to incur a redistribution overhead. We have performed three experiments to illustrate the performance of the following three strategies: The *never* case means that we use the NRD strategy (never redistribute the remaining work). *Adaptive* redistribution means that redistribution is done as long as the previous speculative loop time is greater than the sum of the overhead and incurred delay times of the previous run. *Always* redistribution means 'always' redistribute. Fig. 4(a) shows the execution time breakdown of our experiment. At each stage of the R-LRPD test we measure the time spent in the actual loop and the synchronization and redistribution overhead. In Fig. 4(b) we show the cumulative times spent by the test during its four stages. The "adaptive" redistribution method begins to have shorter overall execution times compared to the "always" redistribution method after the failure on processor 8. The NRD method performs the worst, by a wide margin. It should be noted that our synthetic loop assumes, for simplicity, that  $\alpha$  and  $\beta$  are constant. In practice we would have to adjust the model parameters at every stage of the R-LRPD test.



**Figure 4.** Selection strategy between RD and NRD re-execution technique. (a) Execution time breakdown for three strategies. (b) Time to completion for three strategies

## 5. Implementation and Experimental Results

We have implemented the Recursive LRPD test in the NRD, RD, and SW flavors. We have also applied several optimization techniques to reduce the run-time overhead of checkpointing and the load imbalance caused by the required block scheduling. The implementation (code transformations) is mostly done by our run-time pass in Polaris (it can automatically apply the simple LRPD test)

and additional manually inserted code for the commit phase and execution of the `while` loop shown in Fig. 1(b). We have then applied our technique to the most important loops in TRACK, SPICE2G6, and FMA3D. In the remainder of this section we will present a novel technique used to reduce load imbalance. Further optimizations related to *On-demand Checkpointing and Commit* and *Shadow Data Structures* are documented in [5].

### 5.1 Feedback-Guided Load Balancing

One of the drawbacks of the R-LRPD test is the requirement that the speculative loop needs to be statically block scheduled in order to commit partial work. Since the target of our techniques are irregular codes, load balancing does indeed pose some performance problems. We have independently developed and implemented a new technique similar to [3] that adapts the size of the blocks of iterations assigned to a processor such that load imbalance is minimal at every stage of the R-LRPD test.

Briefly this is how our technique works. At every instantiation of the loop, we measure the execution time of each iteration. After the loop finishes, we compute the prefix sums of the total execution time of the loop as well as the 'ideal', perfectly balanced, execution time per processor, i.e., the average execution time per processor ( $\frac{total\_time}{\#processors}$ ). Using the prefix sums we can then compute a block distribution of iterations that would have achieved perfect load balance. We then save this result and use it as a first order predictor for the next instantiation of the loop. When the iteration space changes from one instantiation to another, we scale the block distribution accordingly. The implementation is rather simple: We instrument the loop with low overhead timers and then use a parallel prefix routine to compute the iteration assignments to the processors. In the near future we will improve this technique by using higher order derivatives to better predict trends in the distribution of the execution time of the iterations. The overhead of the technique is relatively small and can be further decreased. Another advantage of the method is its tendency to preserve locality.

### 5.2 Experimental Results

Our experimental test-bed is a 16 processor ccUMA HP-V2200 system running HP-UX11. It has 4Gb of main memory and 4Mb single level caches. We have applied our techniques to the most important loops in TRACK, a PERFECT code, SPICE 2G6, a PERFECT and SPEC code, and FMA3D, a SPEC 2000 code. The codes (with the exception of Loop 15 in SPICE\_DCDCMP have been instrumented with our run-time pass which was (and is) developed in the Polaris infra-structure [2].

**TRACK** is a missile tracking code that simulates the capability of tracking many boosters from several sites simultaneously. Its main loops, 400 in subroutine **EXTEND**, 300 in **NLFILT** and 300 in **FPTRAK**, account for  $\approx 95\%$  of sequential execution time. We have modified the original inputs which were too small for any meaningful measurement. We have also created several input files to vary the degree of parallelism of some of its loops. To better gauge the obtained speedups we define a measure of the parallelism available in a loop over the life of the program as the *parallelism ratio*  $PR = \frac{\#instantiations}{\#restarts + \#instantiations}$ . For example, a fully parallel loop has a  $PR = 1$  and a partially parallel loop has a  $PR < 1$ . In the case of the **NRD** strategy, a fully sequential loop has a  $PR = 1/p$ , while the **RD** case it can be much lower.

**NLFILT 300.** The compiler un-analyzable array that can cause dependences (mostly short distances) is **NUSED**. Its write reference is guarded by a loop variant condition. Fig. 7(a) presents the effect of the input sets on **PR** for different number of processors. **PR** is dependent on the number of processors because only interprocessor dependences affect the number of restarts (stages) of the **R-LRPD** test. With feedback guided scheduling, the length of iteration blocks assigned to processors is variable which can lead to a variable **PR**. Fig. 7(b) shows the best obtained speedups (all optimizations turned on) for the tested input sets. The speedup numbers include all associated overhead.

Fig. 12(a) compares the effectiveness of the various optimization techniques. Clearly, due to the large state of the loop and its conditional modification the on-demand-checkpointing is the most important optimization. The load balancing technique is very important when redistribution (**RD**) is used. **RD** vs. **NRD** strategy has here a lesser impact because we use only 16 processors.

In Figs. 8 and 9 we show that both **SW** and (**N**)**RD** strategies are useful, depending on the actual dependence structure of the loop. The resulting **PR** is also strategy dependent. The effect on the obtained speedup is a bit skewed due to the different levels of optimizations that could be applied. Scaling the analysis phase for the **SW** technique is not always possible. The graphs also show how the **PR** and speedup varies with the window size. Ideally, we want the largest window size for which there is minimum number of failures (restarts); this size can be adapted based on previous loop instantiations.

**EXTEND 400.** This loop reads data from a read-only part of an array and always writes at the end of the same arrays that are being extended at every iteration. It first extends them in a temporary manner by one slot. If some loop variant condition does not materialize then the newly created slot (track) is re-used (overwritten) in the next iteration. This implies that at most one element of the track arrays needs to be privatized. These arrays are indexed by

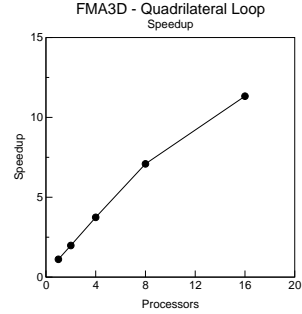


Figure 5. Speedup of FMA3D Quadrilateral Loop

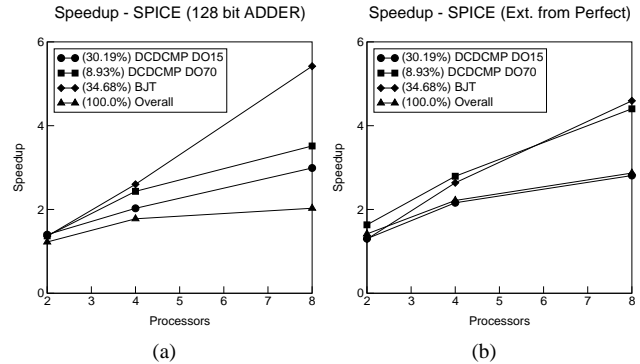


Figure 6. SPICE 2G6 Speedup for important loops and entire program for (a) adder128, and (b) extended PERFECT input decks.

a counter (**LSTTRK**) that is incremented conditionally and whose values cannot be precomputed.

We have all processors speculatively compute **LSTTRK** from a zero offset and collect the array reference ranges [21]. After the first parallel execution we obtain the per processor offsets of the induction variable (the prefix sums of **LSTTRK**) and show that all read references to the array do not intersect any writes, i.e., maximum read index < minimum write. In the second `doall` we repeat the execution using the correct offsets for **LSTTRK**. Last value assignment commits the arrays to their shared storage. Figs. 10(a) and (b) show the **PR** and the best obtained speedup for these inputs, which represents about 60% of the speedup obtainable through hand-parallelization.

**FPTRAK 300.** This loop is very similar to, yet simpler than, **EXTEND 400**. The array under test is privatized with the copy-in/last-value out method and shadowed. The same two stage approach as in **EXTEND** is employed here. Figs. 11(a) and (b) show the **PR** and the best obtained speedup for these inputs.

The overall program speedup of the **entire TRACK** code shown in Fig. 12(b) is scalable and is quite impressive.

**SPICE 2G6** is a circuit simulator that spends most of its time in two distinct loops: A loop in subroutine **DCDCMP** which implements a sparse solver (the decomposition part) and several similar loops in subroutine **LOAD**, **BJT**, **MOS-**

FET, etc., which update the Y matrix of a circuit with the current evaluation of the device models. None of the arrays can be compiler analyzed because they are all equivalenced to one large array (VALUE), the working space of the program and all references have multiple levels of indirection. It is a 'total' workspace aliasing problem.

To make parallelization profitable we have chosen larger input decks than the ones available in the original PERFECT and SPEC codes, i.e., the circuit of a 128 bit adder in BJT technology and a scaled up input deck from PERFECT codes. We have parallelized the three most important loops in SPICE: Loops 70 and 15 in subroutine DCDCMP and the main loop in BJT (which is similar to all the loops called from the model evaluation routine LOAD). The technique to parallelize the main loop in BJT (speculative linked list traversal distribution, sparse LRPD test on the remainder couple with sparse reduction optimization) has been presented in [21, 20]. Loop 70 in DCDCMP is fully parallel with a premature exit and has been parallelized with our techniques described in [15, 4]. Loop 15 in DCDCMP (LU decomposition) is partially parallel due to the sparse nature of the circuit topology. We employ a sparse version of the R-LRPD test that can extract the Data Dependence Graph. Based on the DDG we generate a wavefront schedule of the iterations which can then be reused throughout the remainder of the program execution. For the adder .128 input deck the parallelized loop in DCDCMP has 14337 iterations with a critical path length of 334 (number of wavefronts). Figs. 6 show the obtained speedups for each loop and the for the *entire code*.

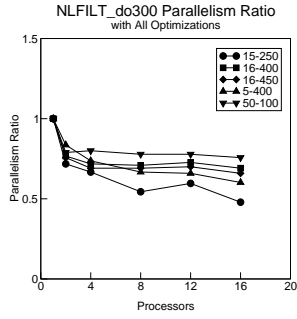
**FMA3D** is a finite element method computer program designed to simulate the inelastic, transient dynamic response of three-dimensional solids and structures subjected to impulsively or suddenly applied loads. Its most important loop, (56% of the sequential execution time), contains array references (to *stress* and *state* arrays) using indirection and its call graph is several levels deep. This complexity makes the 'Quad' loop statically un-analyzable (Theoretically this loop can be statically parallelized because it is input independent). As it turns out the loop is fully parallel and thus the R-LRPD test has only one stage. Fig. 5 shows the overall speedup of this loop.

## 6. Conclusion

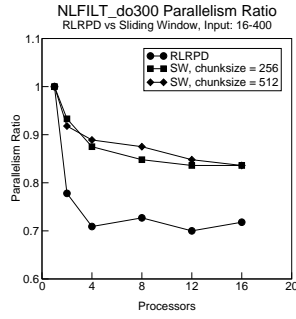
We have shown how to exploit the performance of partially parallel loops which had resisted compiler analysis and the LRPD test. We have presented techniques to lower some of the overheads associated with this method, e.g., feedback guided load balancing of irregular loops. We have further shown how to use our R-LRPD test to build precise data dependence graphs from loops from which a proper inspector cannot be extracted. Experimental results confirm the power of the method.

## References

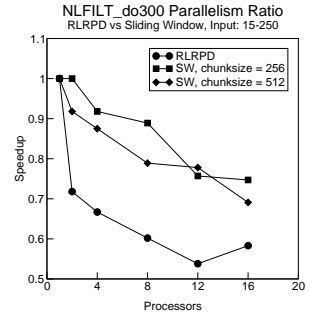
- [1] Charmm: A program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, 4(6), 1983.
- [2] W. Blume, et. al. Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, 29(12):78–82, Dec 1996.
- [3] J. Mark Bull. Feedback guided dynamic loop scheduling: Algorithms and experiments. In *EUROPAR98*, Sept 1998.
- [4] J. A. Carvallo de Ochoa. Optimizations enabling transformations and code generation for the HP V Class. MS Thesis, Texas A&M University, Dept. of Computer Science, Aug 2000.
- [5] F. Dang, H. Yu and L. Rauchwerger. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. TR02-001, PARASOL Lab, Dept. of Computer Science, Texas A&M Univ., Jan., 2002.
- [6] I.H. Kazi and D. Lilja. Coarse-grained speculative execution in shared-memory multiprocessors. In *Proc. of the 12th ACM Int. Conf. on Supercomputing*, pp. 93–100, July 1998.
- [7] S. Leung and J. Zahorjan. Improving the performance of runtime parallelization. In *4th PPOPP*, pp. 83–91, May 1993.
- [8] Z. Li. Array privatization for parallel execution of loops. In *Proc. of the 19th Int. Symp. on Computer Architecture*, pp. 313–322, 1992.
- [9] et. al M. J. Frisch. *Gaussian 94, Revision B.1*. Gaussian, Inc., Pittsburgh PA, 1995.
- [10] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Data dependence and data-flow analysis of arrays. In *Proc. 5th Workshop on Languages and Compilers for Parallel Computing*, Aug 1992.
- [11] Laurence Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. PhD thesis, University of California, May 1975.
- [12] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29:1184–1201, Dec 1986.
- [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.
- [14] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Trans. on Paral. and Dist. Systems*, 10(2), 1999.
- [15] L. Rauchwerger and D. Padua. Parallelizing WHILE Loops for Multiprocessor Systems. In *Proc. of 9th Int. Parallel Processing Symposium*, April 1995.
- [16] J. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.*, 40(5), May 1991.
- [17] P. Tu and D. Padua. Automatic array privatization. In *Proc. 6th Ann. Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, Aug 1993.
- [18] R. Whirley and B. Engelmann. *DYNA3D: A Nonlinear, Explicit, Three-Dimensional Finite Element Code For Solid and Structural Mechanics*. Lawrence Livermore National Laboratory, Nov 1993.
- [19] M. Wolfe. *Optimizing Compilers for Supercomputers*. The MIT Press, Boston, MA, 1989.
- [20] H. Yu and L. Rauchwerger. Adaptive reduction parallelization. In *Proc. of the 14th ACM Int. Conf. on Supercomputing*, Santa Fe, NM, May 2000.
- [21] H. Yu and L. Rauchwerger. Run-time parallelization overhead reduction techniques. In *Proc. of the 9th Int. Conf. on Compiler Construction, Berlin, Germany*. LNCS, Springer-Verlag, March 2000.
- [22] 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.



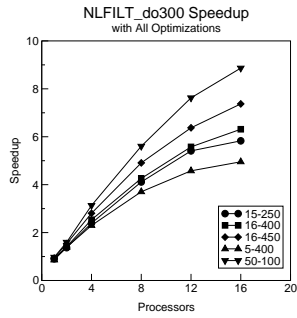
(a)



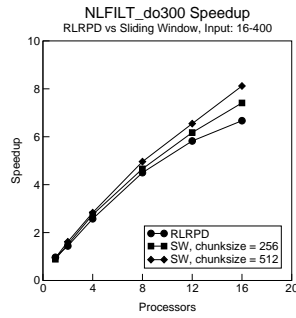
(a)



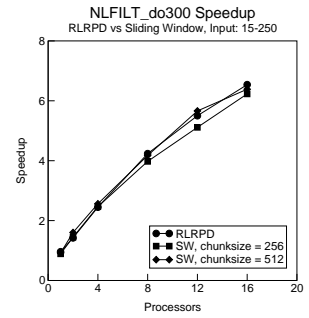
(a)



(b)



(b)

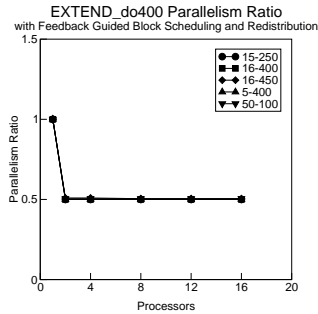


(b)

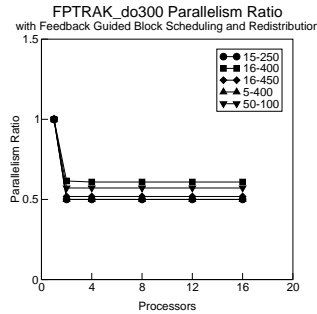
**Figure 7.** NLFILT 300: (a) Parallelism ratio and (b) Speedup.

**Figure 8.** NLFILT 300: Sliding Window vs (N)RD strategy. Input: 16-400

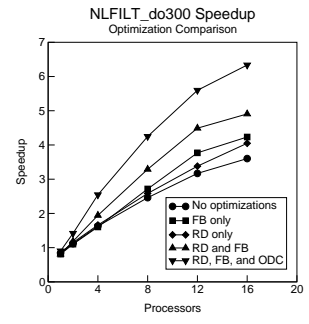
**Figure 9.** NLFILT 300: Sliding Window vs (N)RD strategy. Input: 15-250



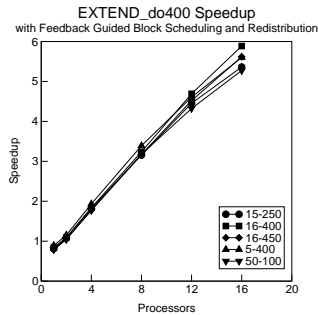
(a)



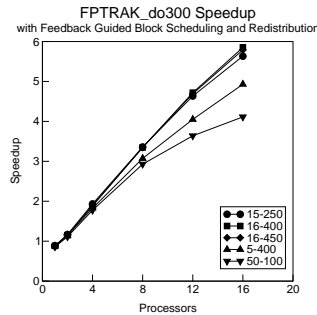
(a)



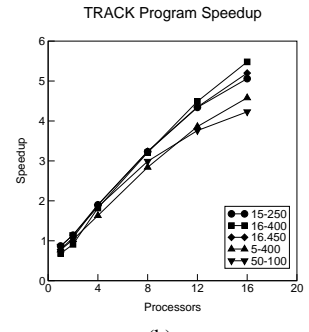
(a)



(b)



(b)



(b)

**Figure 10.** EXTEND 400: (a) Parallelism ratio and (b) Speedup.

**Figure 11.** FPTRAK 300: (a) Parallelism ratio and (b) Speedup.

**Figure 12.** (a) NLFILT 300: Optimization contributions and (b) TRACK Program speedup