

Speculative Parallelization of Partially Parallel Loops

Francis Dang and Lawrence Rauchwerger *

Dept. of Computer Science
Texas A&M University
College Station, TX 77843-3112
{fhd4244,rwgerger}@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. We have previously proposed a framework for their identification. We 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. In this paper we propose a generalization of our speculative `doall` parallelization technique, named 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, i.e., removes the time lost due to incorrect parallel execution. The asymptotic time-complexity is, for fully serial loops, equal to the sequential execution time. We present the base algorithm and an analysis of the different heuristics for its practical application. Some preliminary experimental results on loops from Track will show the performance of this new technique.

1 Efficient Run-Time Parallelization Needed for All Loops

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 address this problem by detecting and exploiting parallelism in sequential programs written in conventional languages as well as parallel languages (e.g., HPC). Although compiler techniques for the automatic detection of parallelism have been studied extensively over the last two decades (see, e.g., [7, 12]), 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 [6], DYNA-3D [11], GAUSSIAN [4], and CHARMM [1]. Run-time techniques can succeed where

* 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

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.

In previous work we have taken two different approaches to run time parallelization. First, we have employed the LRPD test [9], 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 will experience a slow-down equal to the speculative parallel execution time (the loop has to be re-executed sequentially). Second, for loops which were presumed to be partially parallel we have developed an inspector/executor technique [8] in which we record the relevant memory references and then employ a sorting based technique to construct the iteration dependence graph of the loop. Then the iterations are scheduled 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).

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 our 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 (those before the first detected dependence) 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. The negative impact of this limitation can be reduced through dynamic feedback guided scheduling, a dynamic load balancing technique described in Section 4.1.

An additional benefit of this technique is the overall reduction in potential slow-downs 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.

The remainder of this paper will first present the technique as an extension of the LRPD test and several implementation issues. We will introduce a performance model that guides our strategy for applying the various flavors of the technique. Finally we will validate the model and present some experimental results on a real code.

2 The Recursive LRPD Test (R-LRPD)

In our previous work [9] we have described the LRPD test as a technique for detecting `doall` loops. When the compiler cannot perform classical data dependence analysis it can speculatively transform a loop for parallel execution. At run-time, it 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.¹ For simplicity, we will not present reduction parallelization in the following discussion; it is tested in a similar manner as independence and privatization. We have also previously shown that by using a *processor-wise* test we can reduce the overhead of the test as well as qualify more loops as parallel by checking only for cross-processor dependences rather than loop carried dependences (as classical data dependence does). We have further shown that we can increase the number of loops found parallel by testing the *the copy-in* condition in combination with privatization. Privatization testing will detect if a read memory location is referenced by (*Write—Read*) sequence in every iteration (and therefore remove this type of dependence by allocating private storage on each processor). However, if a memory location is first read (any number of times in any number of iterations) before it becomes privatizable, i.e., it has a reference pattern of the form $(Read^*|(Write|Read)^*)$, then the memory location can be transformed for safe parallel execution by privatizing it and initializing it with the original shared data previous to the start of the loop. More formally, in addition to the privatization condition, we need to test at run-time if the highest consecutive read iteration (maximum read) is lower than the earliest (minimum) writing iteration - for all references of the loop. 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). It follows that 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) – all other dependences have been removed through privatization and copy-in.

We now make the crucial remark that in any block-scheduled loop executed under the processor-wise LRPD test, the chunks of iterations that are less than or equal to the source of the first detected dependence arc are always executed correctly. Only the processors executing iterations larger or equal to the earliest sink of any dependence arc need to re-execute their portion of work. This leads to the conclusion that only the remainder of the work (of the loop) needs to be re-executed, which can represent a significant saving over the previously presented LRPD test method (which would

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 [10, 5, 3].

re-execute the whole loop sequentially). To re-execute the fraction of the iterations assigned to the processors that may have worked off erroneous data we need to repair the unsatisfied dependences. This can be accomplished 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. Furthermore, we do not need to re-execute the remainder of the loop serially. If we re-apply the LRPD test on the remaining processors we can in fact speculatively re-execute in parallel. This procedure is applied recursively until all processors have finished correctly their work. For loops with cross-processor dependences we can expect to finish in only a few parallel steps.

To better understand the technique 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: If on a processor the Read occurs before the Write then both bits will remain set – which means 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 will not cause a change in 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`). Finally, after re-initializing the shadow arrays on processors 3 and 4 a new parallel loop is started on the last two processors for the remainder of the iterations (5-8). The final state is shown in Fig. 1(d). At this point all data can be committed and the loop finishes in a total of two steps of two iterations each.

In Fig. 1(e) we adopt a different strategy (RD): Instead of re-executing only on the processors that have incorrect data and leaving the rest of them idle (NRD), we redistribute at every stage the remainder of the work across all processors (while keeping the rest of the procedure the same). There are pros and cons for this approach. Through redistribution of the work we employ all processors all the time and thus the execution time of every stage decreases (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, there is a 'hidden' but potentially large cost associated with work redistribution: more remote misses during loop execution due to data redistribution between the stages of the test. In the next section we will model these two strategies and devise a method to decide between them.

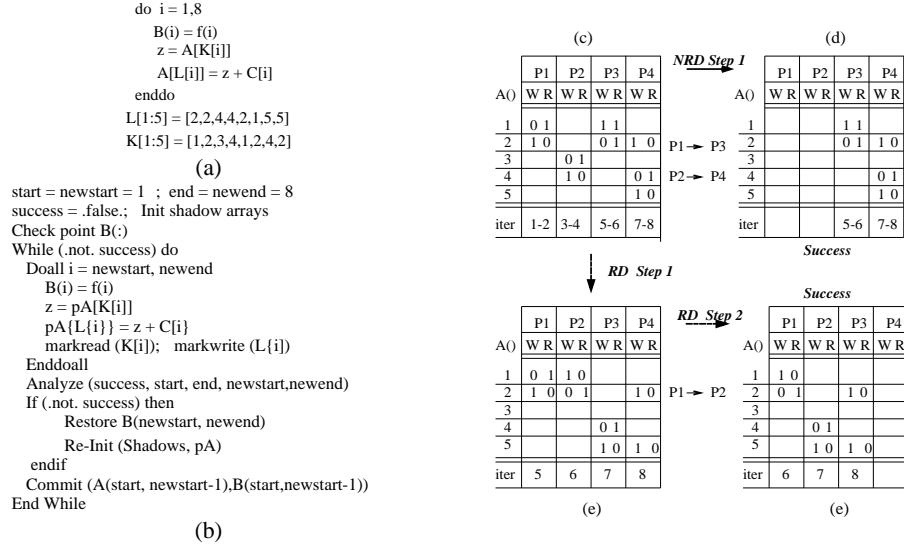


Fig. 1. Do loop (a) transformed for recursive speculative execution, (b) the `markwrite` and `markread` operations update the appropriate shadow arrays. The test is repeated until *success* becomes true. B is an independent array that is checkpointed and partially restored at every stage. pA is the privatized 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).

3 The Model

We have previously shown [9] that if the LRPD test passes, i.e., the loop is in fact fully parallel, then the speedups obtained range from nearly 100% of the ideal in the best case, to *at least* 25% of the ideal in the worst case. 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 the dimension of the shadow structures. For dense access patterns we initialize shadow arrays dimensioned to conform 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. For dense access patterns it is proportional to the dimension of all shared arrays that may be modified during loop execution. The actual time spent saving the state of the loop at every stage will depend on how the checkpointing is implemented as a separate step before loop execution or 'on-the-fly', during loop execution, before modifying a shared variable.

The overhead associated with the execution of the *speculative loop* is equal to the time spent marking (recording) relevant data references. It is proportional to the dynamic count of these memory references. For dense access patterns it can be approximated by the number of references to the tested arrays.

The final *analysis of the marked shadow structures* will be, in the worst case, proportional to the number of distinct memory references marked in 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 components which must be accounted for in the performance analysis. The following breakdown will always depend on the fraction of the successfully completed work which in turn depends on the data dependence structure of the loop. It is important to note that in dynamic programs the data dependence structure of a loop is extremely input dependent and varies during program execution.

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. It is time 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. These issues will be explained in more detail in Section 4.

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. As mentioned earlier, when we do not redistribute work (NRD), the time complexity of the technique is the cost of a sequential execution in the 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 RD (with redistribution) case we will take progressively less time because we execute in p processors decreasing amount of work. We are always guaranteed to finish in a finite number of steps because we are guaranteed that the first processor is always executing correctly. Let us now model more carefully the tradeoff between these two strategies.

Initially, there are n iterations which are equally distributed among the processors. The computation time for each iteration is ω , yielding a total amount of (useful) work in the loop as ωn . In the following discussion we assume that we know ω , the cost of useful computation in an iteration, ℓ , 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** (α) loops where a constant fraction $(1 - \alpha)$ of the current *remaining* iterations are com-

pleted during each speculative parallelization (step), and (b) **Linear** (β) loops where a constant fraction $(1 - \beta)$ of the *original* iterations are completed during each speculative parallelization (step).

No Redistribution of Data Between Speculative Parallelizations (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 α and β loops) and determine the value of k_s for each.

For the α 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 β 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 $T_{\text{static}}(n) = n\omega + ps$.

Redistribution of Data Between Speculative Parallelizations (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. The difference here as opposed to the no redistribution case is that in each subsequent step the processors will have a smaller number of iterations assigned to them. In this case, the total time required by the parallel execution is

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

where n_i is the number of iterations remaining to be completed at the start of the i -th speculative parallelization step, and k_d is the number of speculative parallelization 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, or equivalently, as long as

$$n_{k_d} \geq \frac{ps}{\omega - \ell}. \quad (3)$$

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 , ω , and ℓ , which we assume are known *a priori*, and can be estimated through both static analysis and experimental measurements.

Thus, 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_{static} above, but starting from $n' = n_{k_d}$ instead of n . Thus, the total time required will be

$$T(n) = T_{\text{dyn}}(n) + T_{\text{static}}(n_{k_d}) = \frac{(\omega + \ell)}{p} \left(\sum_{i=0}^{k_d} n_i \right) + \frac{n_{k_d} \omega k_s}{p} + (k_d + k_s)s \quad (4)$$

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. 4. For example, consider the geometric loops in which a constant fraction $(1 - \alpha)$ of the current work is completed during each speculative parallelization attempt.² 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. 3, 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

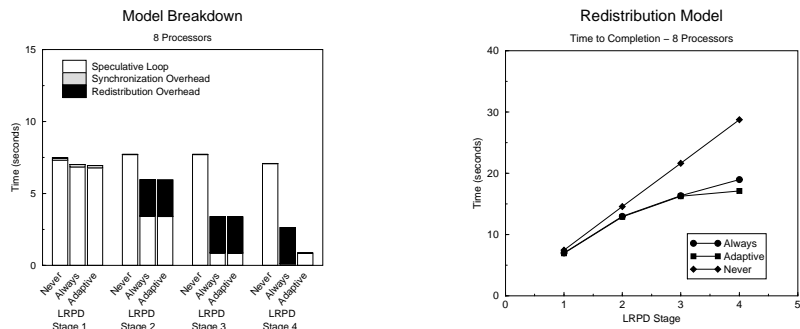
$$T(n) = \frac{n}{p} (\omega + \ell) \left(\frac{\alpha^{k_d+1} - 1}{1 - \alpha} \right) + \frac{n\alpha^{k_d} \omega k_s}{p} + (k_d + k_s)s$$

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

Experimental Model Validation The graph in Fig. 2 illustrates the loop, testing overhead, and redistribution overhead time (mostly due to remote cache misses) for each restart of Recursive LRPD test of a synthetic loop executed on 8 processors of HP-V2200 system. We assume that the fraction of remaining iterations is 1/2. The initial

² 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.

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, i.e., we 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. 2(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. 2(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 (never redistribute) performs the worst, by a wide margin. It should be noted however that our synthetic loop assumes, for simplicity, that α and β are constant. In practice we would have to adjust the model parameters at every stage of the R-LRPD test.



(a) Execution time breakdown for three strategies (b) Time to completion for three strategies

Fig. 2. Selection strategy between RD and NRD re-execution technique.

4 Implementation and Optimizations

We have implemented the Recursive LRPD test in both RD and NRD flavors and applied several optimization techniques to reduce the run-time overhead of checkpointing and reduce the load imbalance caused by the block scheduling of the parallelized irregular loops. As previously mentioned block scheduling is a requirement of the R-LRPD test and thus load balancing is an important issue. 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, a Perfect code. In the remainder of this section we will present two optimizations which we have found to be the most effective in reducing the run-time overhead of our technique.

On-demand Checkpointing and Commit In Section 3 we have already mentioned the need to optimize checkpointing because its work is approximately proportional to the working set of the loop. At every stage of the test we find a contiguous number of processors (processors executing a contiguous block of iterations) that have executed without uncovering any dependences between them and a remainder block of processors which have to re-execute their work. Thus we need to save the data residing in the shared arrays before it is modified by the speculative execution. There are two types of shared variables: Variables that are under test because the compiler cannot analyze them and variables proven by the compiler to be either independent (accessed in only one iteration (processor) or read-only) or privatizable. Saving state or preserving a safe state can be done in two ways: (a) We can write into un-committed private storage which we later either commit by copying it out to the shared area or delete. (b) We can copy the data that *may* be modified by the speculative loop to another, safe, memory storage and then either delete it (if we commit the results of the speculation) or copy back from the original variables (in case we have to restore state).

Both the copy-in/copy-out mechanism and the copying to a safe area can be done in two ways: (i) Before the speculative loop the *entire* working set of the loop is saved or copied-in or (ii) *On-demand*, during loop execution. Performing this activity before the loop always adds to the critical path-length of the program and, in the case of sparse reference patterns, generate more work and consume more memory than necessary. It is however fully parallel and the per operation cost is small (block copy). The on-demand strategy has many advantages: It performs the copy operations only when and if they are needed, which in case of sparse codes, can be orders of magnitude less than a 'wholesale' approach. Moreover, because it is done during loop execution, it may not actually add to the critical path of the program due to the exploitation of low level parallelism. However, each operation has to be initiated separately and may have to be guarded. We need to save data (or copy-in) only at the first write (or read) reference. To accomplish this 'first access' filter we have to distinguish between *variables under test*, i.e., those variables that cannot be analyzed by the compiler and which are shadowed during execution and shared variables that have been analyzed statically. From these variables only the independent ones need attention (read-only and privatized variables don't modify state and don't need to be restored). An independent variable references its location in only one iteration (or processor) and its location can be extracted by the compiler. The 'referenced first' filter can be generated also by the compiler either through peeling it off (in case of nested loops) or using a guard and a very simple shadow (or tag). If the code is such that there is only statement per distinct reference in an iteration then the filter becomes trivial.

The commit and restoration phase needed after the analysis region of each stage of the R-LRPD test depends on the strategy used for checkpointing. For Committing data we need to copy out the last value written (in the sequential semantics). For independent arrays (not under test) this is either accomplished by a compiler generated loop (in case we used copy-in) or by simply deleting the corresponding saved data (if the wholesale copy before the loop strategy is used).

In the experiments shown in Section 4.1 we have implemented on demand copy-in, last-value-out for the arrays under test and on-demand checkpointing with release of

back-up storage at commit phase because it proved to be the most cost-effective for the application studied.

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. Due to the fact that 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 [2] that adapts the size of the blocks of iterations assigned to a processor such that load balancing is achieved at every stage of the R-LRPD test. Briefly this how are 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', perfect balance, execution time per processor, i.e., the average execution time per processor ($\frac{total_time}{number_of_processors}$). 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.

4.1 Experimental Results

Our experimental test-bed is a 16 processor ccUMA HP-V2200 system running HPUX11. 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. Track is a missile tracking code that simulates the capability of tracking many boosters from several sites simultaneously. The main loops in this program are DO 400 in subroutine EXTEND and DO 300 in NLFILT and DO 300 in FPTRAK. They 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. The loops under study are instantiated ≈ 56 times. 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*

$$PR = \frac{total\ number\ of\ instantiations}{total\ number\ of\ restarts + total\ number\ of\ instantiations}$$

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.

We will now briefly analyze the loop from subroutine NLFILT and show the effect of the various optimizations we have applied.

NLFILT DO 300. The compiler un-analyzable array that can cause dependences is NUSED. Its write reference is guarded by a loop variant condition. We have shadowed and marked NUSED according to the rules previously explained. The dependences are mostly of short distance. Fig. 3(a) presents the effect of the input sets on the resulting execution ratio (PR) when the number of processors is varied. It is important to remark that the PR is dependent on the number of processors because only interprocessor dependences affect the number of restarts (stages) of the R-LRPD test. Furthermore, when feedback guided scheduling is performed the length of iterations blocks assigned to processors is variable which can lead to a variable PR. Fig. 3(b) shows the best obtained speedups (all optimizations turned on) for the tested input sets. The speedup numbers include all associated overhead.

The next figures, present the importance of our optimizations to the quality of our parallelization. Figs 4 compare the execution time breakdown of our method when the checkpointing is done (a) before the speculative loop and (b) on-demand, i.e., during the speculative loop. It is quite obvious that the on-demand strategy generates much less overhead and drastically reduces the overall execution time. Fig. 5(a) compares the execution time per processor when the iteration space is equally distributed to the processors with the time per processor when feedback guided scheduling is employed. We can clearly see that our loop balancing technique 'flattens' the execution profile and thus balances the irregular loop. Fig. 5(b) compares the effectiveness of the various optimizations techniques. The input set is 16-400, i.e., a moderate number of dependences are uncovered almost independent of the number of processors used in the experiment. 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.

EXTEND DO 400. This loop has mostly independent array references. It 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 a counter (LSTTRK) that is incremented conditionally. It is in fact a conditionally incremented induction variable and thus does not have a closed form. Because it is used as an index into the arrays, data dependence analysis is difficult. It cannot be pre-computed through loop distribution because its guarding condition is loop variant. Our solution was to speculatively let all processors compute it from a zero offset. At the same time we privatize and shadow the arrays in question and collect their reference ranges [13]. 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 with any of the writes, i.e., maximum read index < minimum write (which occurs in the first iteration). All other write references are indexed higher by LSTTRK because it is a (not strictly) monotonic induction variable. Finally, in the second doall

we repeat the execution using the correct offsets for LSTTRK. Last value assignment commits the arrays to their shared storage. In a future implementation we will process the loop only once - only the last value needs to be committed after computing the actual values of the induction variables. In Figures 6(a) and (b) we show the PR and the best obtained speedup for these inputs. We obtain about 60% of the speedup obtainable through hand-parallelization because our compiler cannot yet recognize and deal with conditional induction variable.

FPTRAK DO 300. This loop is very similar to yet simpler than EXTEND DO 400. The array under test has a read-only front section which is conditionally extended by appending a new element. 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. Figures 7(a) and (b) show the PR and the best obtained speedup for these inputs.

PROGRAM TRACK. The execution profile of the **entire TRACK code** for different input sets given in Fig. 8(b) shows how input sensitive this program is. However, regardless of input, almost all the execution time is spent in the previously discussed loops. The overall speedup for input 16-400 and shown in Fig. 8(a) is scalable and is quite impressive, especially given the fact that these are only preliminary results with minimal compiler support.

5 Conclusion

In this paper we have shown how to exploit parallelism in loops that are less than fully parallel and thus cannot be parallelized with either compile time analysis nor with the original LRPD test. We have also shown how to overcome some of the overheads associated with this method. Moreover, some of these optimizations have general applicability, e.g., load balancing of irregular applications and checkpointing for various applications.

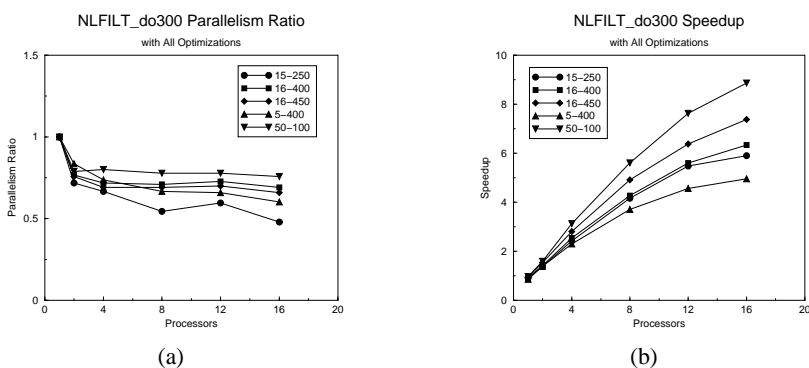


Fig. 3. . NLFILT DO 300: (a) Execution ratio for different inputs. (b) Obtained speedups.

References

1. Charmm: A program for macromolecular energy, minimization, and dynamics calculations. *J. of Computational Chemistry*, 4(6), 1983.

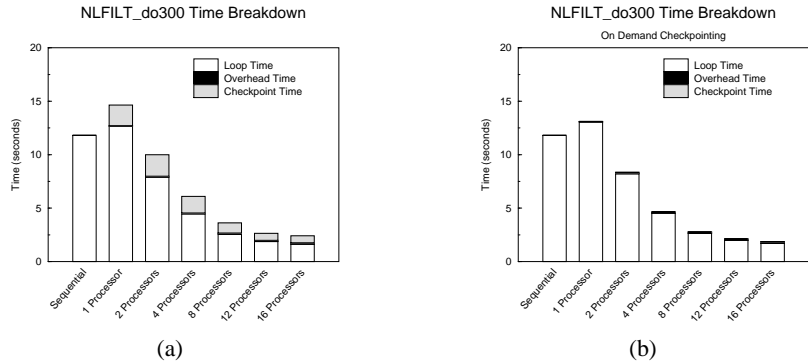


Fig. 4. . NLFILT DO 300: Checkpointing before loop vs. on-demand .

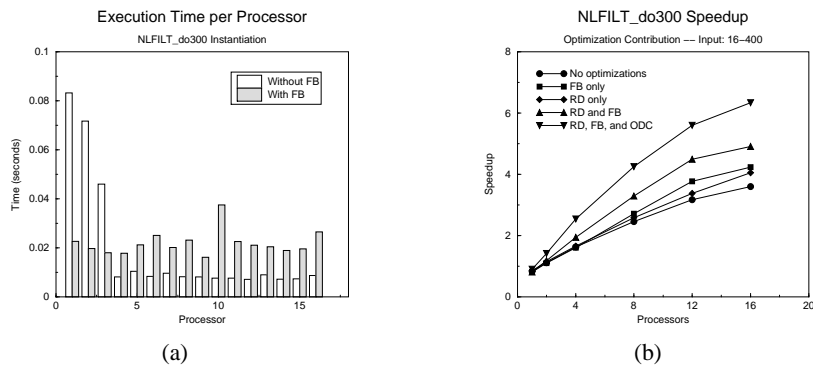


Fig. 5. . NLFILT DO 300: (a) Feedback guided load balancing, (b) Optimization contribution .

2. J. Mark Bull. Feedback guided dynamic loop scheduling: Algorithms and experiments. In *EUROPAR98*, Sept., 1998.
3. Zhiyuan Li. Array privatization for parallel execution of loops. In *Proc. of the 19th Int. Symp. on Computer Architecture*, pp. 313-322, 1992.

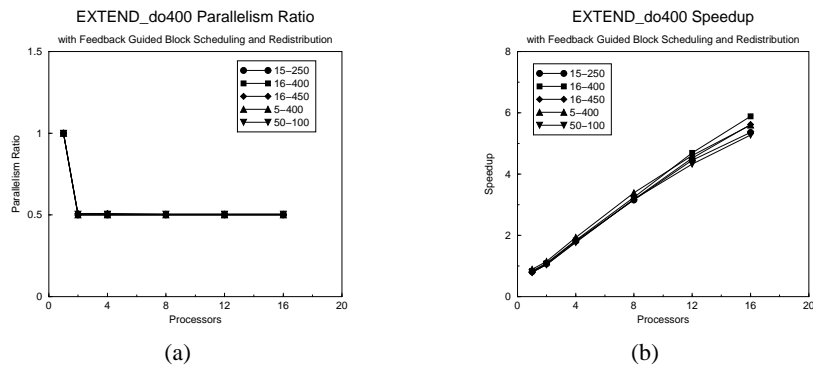


Fig. 6. . EXTEND DO 400: Execution ratio and speedup.

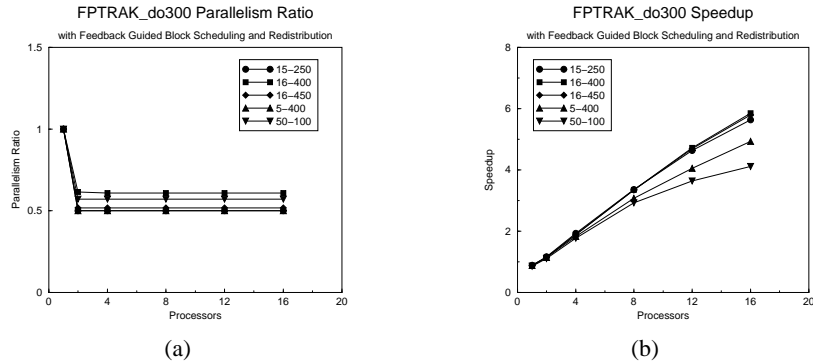


Fig. 7. . FPTRAK DO 300: Execution ratio and speedup.

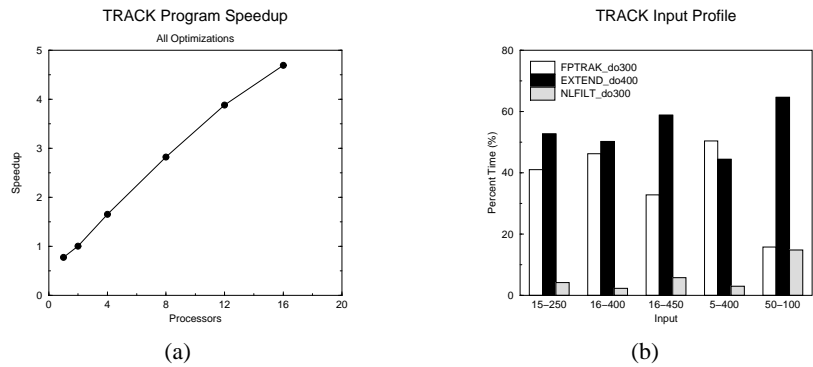


Fig. 8. . TRACK (a)Speedup and (b) Execution Profile for entire program

4. M. J. Frisch et. al. *Gaussian 94, Revision B.1*. Gaussian, Inc., Pittsburgh PA, 1995.
5. D. Maydan, S. Amarasinghe, and M. Lam. Data dependence and data-flow analysis of arrays. In *Proc. 5th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1992.
6. L. Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. PhD thesis, University of California, May 1975.
7. D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29:1184–1201, Dec. 1986.
8. L. Rauchwerger, N. Amato, and D. Padua. A scalable method for run-time loop parallelization. *Int. J. Parallel Programming*, 26(6):537–576, July 1995.
9. 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.
10. P. Tu and D. Padua. Automatic array privatization. In *Proc. 6th Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, Aug. 1993.
11. R. Whirley and B. Engelmann. *DYNA3D: A Nonlinear, Explicit, Three-Dimensional Finite Element Code For Solid and Structural Mechanics*. L. Livermore National Lab., Nov., 1993.
12. M. Wolfe. *Optimizing Compilers for Supercomputers*. The MIT Press, Boston, MA, 1989.
13. Hao Yu and L. Rauchwerger. Run-time parallelization overhead reduction techniques. In *Proc. of the 9th Int. Conf. on Compiler Construction (CC2000), Berlin, Germany*. Lecture Notes in Computer Science, Springer-Verlag, March 2000.