

# Principles of Speculative Run-time Parallelization

Devang Patel and Lawrence Rauchwerger \*

Dept. of Computer Science

Texas A&M University

College Station, TX 77843-3112

<http://www.cs.tamu.edu/faculty/rwerger>, {dpatel,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. We advocate a novel framework for the identification of parallel loops. It speculatively executes a loop as a `doall` and applies a fully parallel data dependence test to check for any unsatisfied data dependencies; if the test fails, then the loop is re-executed serially. We will present the principles of the design and implementation of a compiler that employs both run-time and static techniques to parallelize dynamic applications. Run-time optimizations always represent a tradeoff between a *speculated* potential benefit and a *certain* (sure) overhead that must be paid. We will introduce techniques that take advantage of classic compiler methods to reduce the cost of run-time optimization thus tilting the outcome of *speculation* in favor of significant performance gains. Experimental results from the PERFECT, SPEC and NCSA Benchmark suites show that these techniques yield speedups not obtainable by any other known method.

## 1 Run-time Optimization Is Necessary

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 these problems by detecting and exploiting parallelism in sequential programs written in conventional languages. Although compiler techniques for the automatic detection of parallelism have been studied extensively over the last two decades, 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 [16], DYNA-3D [27], GAUSSIAN [14], CHARMM [1].

---

\* Research supported in part by NSF CAREER Award CCR-9734471 and utilized the SGI systems at the NCSA, University of Illinois under grant#ASC980006N.

It has become clear that static (compile-time) analysis must be complemented by new methods capable of automatically extracting parallelism at *run-time* [6]. 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 often because the necessary information is just not available, i.e., the access pattern is a function of the input data. For example, most dependence analysis algorithms can only deal with subscript expressions that are affine in the loop indices. In the presence of non-linear expressions, or of subscripted subscripts, compilers generally conservatively assume data dependences. Although more powerful analysis techniques could remove this last limitation when the index arrays are computed using only statically-known values, nothing can be done at compile-time when the index arrays are a function of the input data [12, 25, 28].

We will present the principles of the design and implementation of a compiling system that employs run-time and classic techniques in tandem to automatically parallelize irregular, dynamic applications. We will show that run-time optimizations always represent a tradeoff between a *speculated* potential benefit and a *certain* (sure) overhead that must be paid. This work models the competing factors of this optimization technique and outlines a guiding strategy for increasing performance. We will introduce techniques that take advantage of classic compiler methods to reduce the cost of run-time optimization thus tilting the outcome of *speculation* in favor of significant performance gains.

The scope of presented work will be initially limited to loop level parallelization and optimization of Fortran programs in a shared memory environment using the SPMD programming paradigm. The run-time techniques described here are designed to be used in the automatic parallelization of 'legacy' Fortran applications as well as in explicit parallel coding of new, dynamic codes, where concurrency is a function of the input data.

## 2 Run-time Optimization

Maximizing the performance of an application executing on a specific parallel system can be derived from three fundamental optimization principles: (i) maximizing parallelism while minimizing overhead and redundant computation, (ii) minimizing wait-time due to load imbalance, and (iii) minimizing wait-time due to memory latency.

Maximizing the parallelism in a program is probably the most important factor affecting parallel, scalable performance. It allows full concurrent use of all the resources of any given architecture without any idle (wait) time. At the limit, full parallelism also allows perfect scalability with the number of processors and can be efficiently used to improve memory latency and load balancing.

The most effective vehicle for improving multiprocessor performance has been the restructuring compiler [5, 10, 18, 9]. Compilers have incorporated sophisticated data dependence analysis techniques (e.g., [3, 19]) to detect intrinsic parallelism in codes and transform them for parallel execution. These techniques usually rely on a static (compile time) analysis of the memory access pattern (array subscripts in the case of Fortran programs) and on parallelism enabling transformations like privatization, reduction parallelization, induction variable substitution, etc. [7]. When static information is insufficient to safely perform an optimizing transformation the classic compiler emits conservative code. Alternatively it might delay the decision to execution time, when sufficient information becomes available. This strategy implies that a certain amount of code analysis has to be performed during the time which was initially allocated to useful data processing. This shift of activity will inherently account for *a priori* performance degradation. Only when the outcome of the run-time analysis is a safe optimization can we hope that the overall execution time will decrease. For example, if the parallelization of a loop depends on the value of a parameter that is statically unavailable the compiler can generate a two-version loop (one parallel and one sequential) and code that will test the parameter at run-time and decide which version to execute. While this is a very simple case it shows that time will be 'lost' testing the parameter and, depending on the outcome, may or may not lead to an optimization. Furthermore, even if the loop under question is executed in parallel, performance gains are not certain. All this implies that run-time optimizations always represent a tradeoff which needs a guiding strategy; they represent a *speculation* about a potential benefit for which a *certain* (sure) overhead will have to be paid.

## 2.1 Principles of Run-Time Optimization

Loop parallelization is the most effective and far reaching optimization for scientific applications. Briefly stated, a loop can be safely executed in parallel if and only if its later iterations do not use data computed in its earlier iterations, i.e., there are no flow dependences. The safety of this and other related transformations (e.g., privatization, reduction parallelization) is checked at compile time through data dependence analysis (i.e., analyzing array subscript functions). When static analysis is not possible the access pattern is analyzed at run-time through various techniques which we will shortly introduce and analyze.

**Inspector/Executor vs. Speculative run-time testing.** All run-time optimizations in general, and parallelization in particular, consist of at least two activities: (a) a test of a set of run-time values (e.g., the values taken by array subscripts) and (b) the execution of one of the compiler generated options (e.g., multi-version loops).

If the test phase is performed before the execution of the loop and has no side effects, i.e., it does not modify the state of the original program (shared) variables then this technique is called *inspector/executor* [4]. Its run-time overhead consists only of the time to execute the inspection phase.

If the test phase is done at the same time as the execution of the aggressively optimized loop and, in general, the state of the program is modified during this process, then the technique is called *speculative execution*. Its associated overhead consists at least of the test itself and the saving of the program state (checkpointing). If the optimization test fails, then extra overhead is paid during a program *ante loop* state restoration phase before the conservative version of the code can be executed. In this scenario the the initial optimized loop execution time becomes additional overhead too.

Although it might appear that the more 'sedate' *inspector/executor* method is a better overall strategy than the *speculative* technique, there is in fact a much more subtle trade-off between the two. An inspector loop represents a segment of code that must always be executed before any decision can be made and always adds to the program's critical path. However, if the test is executed concurrently with the actual computation (it is always quasi-independent of it – computation cannot possibly depend on the test) then some of the overhead may not add additional wall-clock time. The same is true if checkpointing is done 'on-the-fly', just before a variable is about to be modified. In other words *with respect to performance alone* the two methods are competitive.

A potential negative effect of speculative execution is that the optimization test's data structures are used concurrently with those of the original program, which could increase the working set of the loop and degrade its cache performance.

The previous comparison assumes an important principle: *any run-time parallelization technique must be fully parallel* to scale with the number of processors. For the speculative method this is always implicitly true – we test during a speculative parallel execution. Inspectors may be executed sequentially or in parallel – but, with the exception of simple cases, only parallel execution can lead to scalable performance. Inspector loops cannot always be parallelized. If there exists a data or control dependence cycle between shared data and its address computation then it is not possible to extract an address inspector that can be safely parallelized and/or that is side effect free. In fact the inspector would contain most of the original loop, in effect degenerating into a *speculative* technique (will need checkpointing) without its benefits.

In summary we conclude that both run-time techniques are generally competitive but that the speculative method is the only generally applicable one.

## 2.2 Obtaining Performance

Run-time optimization can produce performance gains only if the associated overhead for its validation is outweighed by the obtained speedup or,

$$Speedup = SuccessRate \times (Optimization\_Speedup - Testing\_Overhead) > 0$$

This *Speedup* function can be maximized by increasing the power of the intended optimization and decreasing the time it takes to validate it. Because run-time optimizations are speculative, their success is not guaranteed and therefore, their *SuccessRate*, needs to be maximized.

**Performance through Run-time Overhead Reduction.** The optimization representing the focus of this paper is loop parallelization within a SPMD computation. This transformation generally scales with data size and number of processors and its overall potential for speedup is unquestionable. Its general profitability (when and where to apply it) has been the topic of previous research and its conclusions remain valid in our context.

Thus, the task at hand is to decrease the second term of our performance objective function, the testing-overhead. Regardless of the adopted testing strategy (inspector/executor or aggressive speculation) this overhead can be broken down into (a) the time it takes to extract data dependence information about the statically un-analyzable access pattern, and (b) the time to perform an analysis of the collected data dependence information.

The first rule we have adopted is that all run-time processing (access tracing and analysis) must be performed in parallel — otherwise it may become the sequential bottleneck of the application. The access pattern tracing will be performed within a parallel region either before the loop in case of the inspector approach or during the speculative execution of the transformed loop. The amount of work can be upper bounded by the length of the trace but (see Section 5) can be further reduced (at times dramatically) through reference aggregation and elimination of duplicated (redundant) address records. This type of optimization can be achieved through the use of static, i.e., compile time information. Usually, when a compiler cannot prove independence for all referenced variables, the partial information obtained during static analysis is discarded. In such a case our run-time compiler phase will retrieve all previously considered useless, but valid information and complement it with only the really dynamic data. This tight integration of the run-time technique with the classic compiler methods is the key to the reduction of tracing overhead.

Another important tool in reducing overhead is the development of static heuristics for uncovering the algorithms and data structures used in the original program. For example, pattern matching a reduction can encourage the use of a run-time reduction validation technique. An inference about the use of structures may reduce the number of addresses shadowed.

**Increasing the Success Rate of Speculation.** Collecting the outcome of every speculation and using this data in the computation of a *statistic* could drastically alter the success rate of speculation. The use of *meaningful* statistics about the parallelism profile of dynamic programs will require some evidence that different experiments on one application with different input sets produces similar results (with respect to parallelism). Feeding back the results of speculative parallelization during the same execution of a code may be, for the moment, a more practical approach. For example, after failing speculation on loop several consecutive times a more conservative approach can adopted 'on-the-fly'.

A more difficult but more effective strategy in enhancing both the success rate of speculation as well as lowering run-time overhead is to find heuristics that can 'guess' the algorithmic approach and/or data structure used by the original program and drive the speculation in the right direction. A simple example

is reduction recognition: if a statement ‘looks’ like a reduction then it can be verified by generating a speculative test for it – the chances of success are very high. Making the correct assumption at compile time whether an access pattern is sparse or dense or whether we use linked lists or arrays (regardless of their implementation) can go a long way in making run-time optimization profitable (see Section 5).

### 3 Foundational Work: Run-Time Parallelization

We have developed several techniques [20–24] that can detect and exploit loop level parallelism in various cases encountered in irregular applications: (i) a speculative method to detect fully parallel loops (The LRPD Test), (ii) an inspector/executor technique to compute wavefronts (sequences of mutually independent sets of iterations that can be executed in parallel) and (iii) a technique for parallelizing `while` loops (`do` loops with an unknown number of iterations and/or containing linked list traversals). We now briefly describe a simplified version of the speculative LRPD test (complete details can be found in [20, 22]).

**The LRPD Test.** The LRPD test speculatively executes a loop in parallel and tests subsequently if any data dependences could have occurred. If the test fails, the loop is re-executed 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> For simplicity, reduction parallelization is not shown in the example below; it is tested in a similar manner as independence and privatization. The LRPD test is fully parallel and requires time  $O(a/p + \log p)$ , where  $p$  is the number of processors, and  $a$  is the total number of accesses made to  $A$  in the loop.

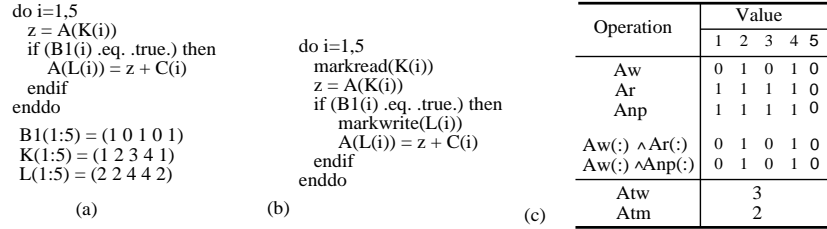
Consider a `do` loop for which the compiler cannot statically determine the access pattern of a shared array  $A$  (Fig. 1(a)). We allocate the shadow arrays for marking the write accesses,  $A_w$ , and the read accesses,  $A_r$ , and an array  $A_{np}$ , for flagging non-privatizable elements. The loop is augmented with code (Fig. 1(b)) that will mark during speculative execution the shadow arrays every time  $A$  is referenced (based on specific rules). The result of the marking can be seen in Fig. 1(c). The first time an element of  $A$  is written during an iteration, the corresponding element in the write shadow array  $A_w$  is marked. If, during any iteration, an element in  $A$  is read, but never written, then the corresponding element in the read shadow array  $A_r$  is marked. Another shadow array  $A_{np}$  is used to flag the elements of  $A$  that *cannot* be privatized: an element in  $A_{np}$  is

---

<sup>1</sup> *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 [26, 15, 13].

marked if the corresponding element in **A** is both read and written, and is read first, in any iteration.

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



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

## 4 Variations of the LRPD Test

Static compilation can generate a wealth of incomplete information that, by itself, is insufficient to decide whether parallelization is safe but can be exploited to reduce run-time overhead. When we can establish statically that, for example, all iterations of a loop first read and then write a shared array (but nothing else) then we can conclude that privatization is not possible, and therefore should not test for it. This approach of using partial information has led to the development of simplified variants of the LRPD test. The overall purpose of the various specialized forms of the LRPD test presented in this section is (a) to reduce the overhead of run-time processing to the minimum necessary and sufficient to achieve safe parallelization (but without becoming conservative), and (b) to extend the number of access patterns that can be recognized as parallelizable. We will now enumerate some of the more frequently used variants of the LRPD

<sup>2</sup> `any` returns the “OR” of its vector operand’s elements, i.e.,  $\text{any}(v(1 : n)) = (v(1) \vee v(2) \vee \dots \vee v(n))$ .

test that we have developed and elaborate on those that have not been presented elsewhere [24, 22]. Further refinements and related issues (such as choice of marking data structures) are discussed in Section 5.

- Processor-wise LRPD test for testing cross-processor instead of cross-iteration dependences, qualifying more parallel loops with less overhead.
- A test supporting copy-in of external values to allow loops that first read-in a value to be executed in parallel.
- Early failure detection test to reduce the overhead of failed speculation.
- Early success detection test with on-demand cross-processor analysis.
- A test that can distinguish between fully independent and privatizable accesses to reduce private storage replication for privatized arrays.
- An Aggregate LRPD test – aggregates individual memory the references in contiguous intervals or sets of points.

#### 4.1 Early Failure Detection

Note that it can be detected *during* the execution of the LRPD test, almost without any additional cost, if a reference under test causes a failure condition (read and written in different iterations, or, within the same iteration, read before write or just referenced outside a reduction statement) among those iterations executing on the same processor. If this happens we can fail the test immediately without actually finishing the rest of the `un`-executed code and without any cross-processor analysis. Of course, if we decide to apply the processor-wise version of the test then cross-iteration dependences between iterations scheduled on the same processor may not necessarily cause failure. A more sophisticated approach is to establish a correlation between the number of cross-iteration dependences and the probability of finding cross-processor dependences after a complete loop execution. If a loop has dependences it is highly likely that these dependences will appear well before the last iteration is executed.

#### 4.2 Faster Analysis and Early Success Detection

The analysis phase of the processor-wise version of the LRPD test can be reduced by maintaining a flag per processor (each representing an entire array with one scalar), per possible cause of failure (see Section 4.1). If, after loop execution, these flags have not been set on any processor then no further analysis is necessary and speculation was successful. If only some of the processors show a potential problem then only their shadows will be analyzed, thus reducing the overall work. If the flags show problems everywhere then classic parallel merge is performed. For example, we can keep a per-processor reduction flag (a bit) and mark any occurrence of an array reference outside the reduction statement. After loop execution only the processors with flags set will participate in the analysis phase because they are the only potential cause for dependences – if no processor has the flag set then the test passes without any further overhead.

We have modified the cross-processor analysis phase itself in an optimistic manner to reduce cross-processor communication in the following way: each processor traverses its own shadow array and 'looks' at the contents of the other ones only if it detects a cause for failure. If, for example, we test reduction and/or privatization the processor(s) that find the flag set will 'look' in the same position of the shadow across processors and merge their information. This technique could be named 'on demand merging'.

### 4.3 Fully Independent and Privatizable Accesses

Since privatization generally implies replication of program variables (i.e., an increase in memory requirements), this transformation should be avoided when there is no benefit to be gained. Instead of privatizing entire arrays, the LRPD test can identify and privatize only elements that are actually written. However, if an element is written only once in the loop, then there is no need for it to be privatized and replicated on multiple processors.

The LRPD test can easily be augmented to determine whether a element is written more than once. One simple approach is to use another shadow structure  $A_{mw}^p$  to flag the array elements which have been written multiple times. On a write to an element during the marking phase, the corresponding entry of  $A_{mw}^p$  is marked if the corresponding entry of  $A_w$  is already marked. The global shadow structure  $A_{mw}$  is now constructed from the processors' shadow structures  $A_{mw}^p$  and  $A_w$ . First, the marked elements in the processors'  $A_{mw}^p$  are transferred directly, without synchronization, into  $A_{mw}$ . If the private structures  $A_w$  are merged pairwise into the global shadow structure  $A_w$ , then during this process it can be determined if an element is marked in more than one  $A_w$ , i.e., if it was written more than once. Note that the pairwise merges can be eliminated if the accesses to the global shadow structure are placed in critical sections.

It is simple to see that the need for the additional structure  $A_{mw}^p$  can be eliminated by using three states for the structure  $A_{np}$ , e.g., negative values for multiple writes and privatizable, 0 for at most one write and privatizable (initial value), and positive values for not privatizable (once set, never reset).

If the processor-wise version of the LRPD test is used then the elements that are written more than once can be identified in essentially the same manner. Note that for the processor-wise version it is possible that the number of private variables could be reduced even more since only the processors that actually write the elements need copies, and the private structures identify these elements.

### 4.4 Aggregate LRPD test

The simple, and rather naive way to insert the marking code into a loop is to simply add a `markwrite`, `markread`, `markredux` macro for every occurrence of a write and read access to the shadowed array.

There are however many programs that although irregular in general have a specific 'local' or partial regularity. These types of access patterns can be classified in the following manner:

- Arrays in nested loops accessed by an index of the form (*affine\_fcn, ptr*). The innermost loop index generates points for the affine function, and the outermost loop for the pointer. Generally the first dimension of the array is relatively small and is often traversed in an innermost loop or, for very small loop bounds, completely unrolled. It usually represents the access to the components of an  $n$ -dimensional vector. The bounds of this inner loop never change throughout the program.
- Multi-dimensional access patterns described by complex but statically determined functions, but where one more of the inner dimensions are simple functions.

A commonality in these cases is the fact that they all perform portion-wise contiguous accesses. The length of this regular interval can be either fixed (vectors with  $n$ -components, structures) or of variable length (e.g., in sparse matrix solvers). This characteristic can be exploited by marking contiguous intervals rather than every element accessed. Depending on the actual length of the intervals this technique can lead to major performance improvements.

In the case of fixed-size intervals the information is kept implicitly (not stored in the shadow structures themselves) and the analysis phase needs only minor adjustment to the generic LRPD test. When intervals are variable in size within the context of the tested loop, their length will be kept explicitly and the shadow structures adapted accordingly into *shadow interval structures* (e.g., interval trees). The analysis phase will change to a more complex algorithm to reflect the parallel merge of complex data structures. While the asymptotic complexity increases the problem size can decrease dramatically (depending on the average length of intervals).

In our implementation the compile time detection of these types of semi-irregular access patterns is obtained using recently developed array region analysis techniques [17].

It is important to mention here the possibility of applying the run-time test somewhat conservatively by always marking whole intervals even if only some of the addresses within the interval have actually been referenced during loop execution. This will reduce overhead and, in the case of fixed size intervals, rarely result in overly conservative loss of parallelism. For example, if a program is using 2 dimensional arrays to simulate structures we can map the *entire* structure into one shadow point. While there is the possibility that such a structure is accessed in more than one iteration at different offsets (and therefore would conservatively fail the test) we have not found such an occurrence.

## 5 Some Strategy and Implementation Issues

In this section we mention some techniques for optimizing the implementation of the LRPD test. Due to space constraints, complete details are omitted here.

**Merging the Phases of LRPD Test.** Some of the steps of a speculative execution can be merged, i.e., executed concurrently, thereby increasing fine grain

parallelism without increasing the working set. For example, cross-processor last value assignment, merge-out and reduction can be done concurrently after the analysis phase, if necessary. Also, operations such as copy-in of shared values, checkpointing, or initialization of shadow structures can be performed *on-demand*, which decreases the critical path and also the total operations required (e.g., checkpointing only the modified elements of an array).

**Choosing Shadow Structures.** The choice of shadow structures is dictated by the characteristics of the access pattern and the data structures used (implicitly or explicitly) of the original program. If the access pattern is dense (inferred from the ratio of the number of references to the array size), we choose shadow arrays, and if it is sparse (regular or irregular) we choose specialized shadow structures such as hash-tables. If the application performs portion-wise contiguous regular accesses, then shadow arrays are used for fixed-size intervals, and interval trees are used if the intervals are loop variant. More sophisticated data structures such as linked lists are currently under development.

**Schedule reuse, inspector decoupling, two-process solution.** If the speculatively executed loop is re-executed during the program with the same data access pattern, then the results of the first LRPD test can be reused (this is an instance of *schedule reuse* [25]). If the defining parameters of an inspector loop are available well before the loop will be executed, then the test code can be executed early, perhaps during a portion of the program that does not have enough (or any) parallelism. A way to ensure that the program's critical path is never increased, is to fork two processes: one processor executes the original sequential loop and the remaining processors proceed on the more aggressive parallel path.

## 6 Current Implementation of Run-time Pass in Polaris

Based on the previously presented techniques and the early work described in [22] and [11] we have implemented a first version of run-time parallelization in the Polaris compiler infrastructure [8]. Here is a very general overview of this 'run-time pass'.

Currently, candidate loops for run-time parallelization are marked by a special directive in the Fortran source code. Alternatively, all loops that Polaris leaves sequential are run-time parallelized. As a statistical model of loop parallelism in irregular applications will be developed we will be able to automatically select the candidates which have the highest possibility of success.

The bulk of the run-time pass is placed after all other static analysis has been completed and just before the post-pass (code generation). It can therefore use all the information uncovered by the existing Polaris analysis.

## 7 Experimental Results of Run-time Test in Polaris

We will now present experimental results obtained on several important loops from three applications that Polaris could not parallelize, namely, **TFFT2**,

**P3M** and **TRACK**. After inserting run-time test directives before the loop, the codes have been automatically transformed by the compiler and executed in dedicated mode on an SGI Power Challenge with R10K processors at NCSA, University of Illinois. All test variant selections and other optimizations are automatic and no special, application specific compiler switches have been used.

**TFFT2**, a SPEC code has a fairly simple structure and all access patterns are statically defined, i.e., they are not dynamic or input dependent. Nevertheless, difficulties in its analysis arise due to (1) five levels of subroutine calls within a loop, (2) array reshaping between subroutine calls, (3) exponential relations between inner and outer loop bounds, and (4) array index offsets that depend on outer loop index. We have transformed all important loops of this program for speculative execution.

The speedups shown in Figure 3 reflect the application of the speculative LRPD test to the five most important loops of the program: CFFTZ\_DO#1, CFFTZ\_DO#2, CFFTZ\_DO\_#3, RCFFTZ\_DO\_110, CRFFTZ\_DO\_100. While speedups are generally good Loop CFFTZ\_DO\_#2 performs poorly because we allocated a shadow array four times larger than the actual access region (allocation based on dimension rather than access region) and because the loop itself is relatively small. The overall speedup of the TFFT2 program is 2.2 on 8 processors.

From the **P3M**, NCSA benchmark, a N-body simulation we have considered the triply nested loop in subroutine `pp` which takes about 50% of the actual sequential execution time. For better load balancing we have coalesced the loop nest and then applied speculative parallelization to several arrays that could not be proven privatizable by Polaris. For the best result we have employed the processor-wise privatization test (with dynamic scheduling) with shadow arrays and early success detection. No checkpointing was necessary because all arrays are privatized and the final reduction is performed on private arrays that are merged after loop execution. Figure 4 shows good speedup and scalability. The obtained speedup is significantly less than the manually parallelized version because the initialization phase, though short, has a cache flushing effect, thus causing the speculative loop to slow down; misses are experienced on all read-only arrays.

**TRACK**, a PERFECT code that simulates missile tracking, is one of the more interesting programs we have encountered. The tested loop, NLFILT\_DO\_300, has cross-iteration dependences in some of its instantiations and their frequency is input dependent. For the data set presented in Figure 2 the loop fails the cross-iteration dependence once in its 60 instantiations. However, the processor-wise test 'hides' the dependences and passes every time. The checkpointing overhead is quite important when array sizes are large with respect to the actual work that the loops performs. We believe that an improved on-demand checkpointing scheme will reduce this overhead. Note: The hand-parallel speedup in Figure 2 is in fact an *ideal* speedup because the loop cannot be manually parallelized (because its parallelism is input dependent). The value shown is still correct

because the hand-parallel version has been statically scheduled and there are no cross-processor dependences.

## 8 Conclusion

While the general LRPD algorithm has been extensively presented in [24] and briefly shown here for clarity of the presentation, this paper emphasizes the practical aspects of its application and integration in a compiler. In essence we advocate a very tight connection between static information obtained through classical compiler methods and the run-time system. This resulting optimized code will make use of all available static information and test only the necessary and sufficient conditions for safe parallelization. This interplay between compiler and run-time system results in testing methods that are tailored to a particular application (within limits) and that perform better.

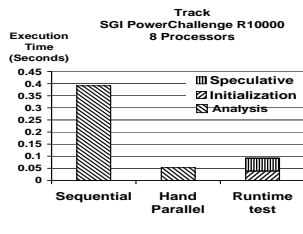
A major source of optimization in speculative parallelization is the use of heuristics for inferring the data structure and access pattern characteristics of the program. Once a hypothesis is made, it can be tested at run-time much faster than a general method. For example, guessing the use of linked list or a structure and testing accordingly can improve performance dramatically.

Reducing run-time overhead may also require the speculative application of known code transformations, e.g., loop distribution, forward substitution. Their validity will be checked simultaneously with the previously presented run-time data dependence test, without incurring any additional overhead.

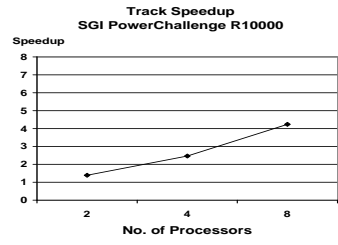
## References

1. Charmm: A program for macromolecular energy, minimization, and dynamics calculations. *J. of Computational Chemistry*, 4(6), 1983.
2. Santosh Abraham. *Private Communication*. Hewlett Packard Laboratories, 1994.
3. Utpal Banerjee. *Loop Parallelization*. Norwell, MA: Kluwer Publishers, 1994.
4. H. Berryman and J. Saltz. A manual for PARTI runtime primitives. Interim Report 90-13, ICASE, 1990.
5. W. Blume, et. al. Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.
6. W. Blume and R. Eigenmann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks<sup>TM</sup> Programs. *IEEE Trans. on Parallel and Distributed Systems*, 3(6):643–656, November 1992.
7. W. Blume et. al. Effective automatic parallelization with polaris. *Int. J. Paral. Prog.*, May 1995.
8. W. Blume et al. Polaris: The next generation in parallelizing compilers, In *Proc. of the 7-th Workshop on Languages and Compilers for Parallel Computing*, 1994.
9. K. Cooper et al. The parascope parallel programming environment. *Proc. of IEEE*, pp. 84–89, February 1993.
10. M. Hall et. al. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*, 29(12):84–89, December 1996.
11. T. Lawrence. Implementation of run time techniques in the polaris fortran restructurer. TR 1501, CSRD, Univ. of Illinois at Urbana-Champaign, July 1995.

12. S. Leung and J. Zahorjan. Improving the performance of runtime parallelization. In *4th PPOPP*, pp. 83–91, May 1993.
13. Z. Li. Array privatization for parallel execution of loops. In *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 313–322, 1992.
14. M. J. Frisch et. al. *Gaussian 94*. Gaussian, Inc., Pittsburgh PA, 1995.
15. D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Data dependence and data-flow analysis of arrays. In *Proc. 5th Workshop on Programming Languages and Compilers for Parallel Computing*, August 1992.
16. L. Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. PhD thesis, University of California, May 1975.
17. Y. Paek, J. Hoeflinger, and D. Padua. Simplification of Array Access Patterns for Compiler Optimizations. In *Proc. of the SIGPLAN 1998 Conf. on Programming Language Design and Implementation, Montreal, Canada*, June 1998.
18. C. Polychronopoulos et. al. Parafraze-2: A New Generation Parallelizing Compiler. *Proc. of 1989 Int. Conf. on Parallel Processing, St. Charles, IL*, II:39–48, August 1989.
19. W. Pugh. A practical algorithm for exact array dependence analysis. *Comm. of the ACM*, 35(8):102–114, August 1992.
20. L. Rauchwerger, N. Amato, and D. Padua. A scalable method for run-time loop parallelization. *IJPP*, 26(6):537–576, July 1995.
21. L. Rauchwerger and D. Padua. The privatizing doall test: A run-time technique for doall loop identification and array privatization. In *Proc. of the 1994 International Conf. on Supercomputing*, pp. 33–43, July 1994.
22. L. Rauchwerger. Run-time parallelization: A framework for parallel computation. TR UIUCDCS-R-95-1926, Dept. of Computer Science, University of Illinois, Urbana, IL, September 1995.
23. L. Rauchwerger and D. Padua. Parallelizing WHILE Loops for Multiprocessor Systems. In *Proc. of 9th International Parallel Processing Symposium*, April 1995.
24. L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *Proc. of the SIGPLAN 1995 Conf. on Programming Language Design and Implementation, La Jolla, CA*, pp. 218–232, June 1995.
25. J. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.*, 40(5), May 1991.
26. P. Tu and D. Padua. Array privatization for shared and distributed memory machines. In *Proc. 2nd Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*, September 1992.
27. 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.
28. 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.

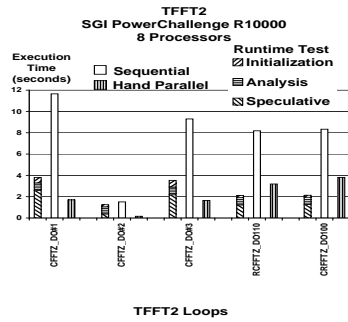


(a)

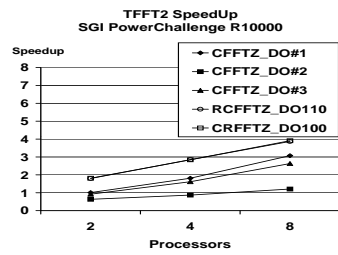


(b)

**Fig. 2.** Loop TRACK\_NLFITL\_DO\_300: (a) Timing of test phases, (b) Speedup

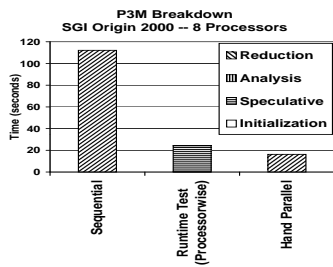


(a)

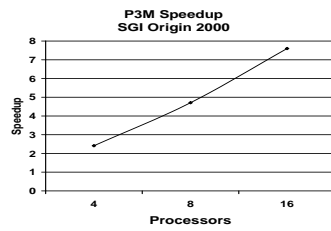


(b)

**Fig. 3.** Major Loops in TFFT2: (a) Timing of test phases, (b) Speedup



(a)



(b)

**Fig. 4.** Loop P3M\_PP\_DO\_100: (a) Timing of test phases, (b) Speedup