# Compilation Techniques for Parallel Systems

Rajiv Gupta
Dept. of Computer Science, The Univ. of Arizona,
Tucson, AZ 85721

Santosh Pande
Dept. of Elect. and Comp. Engg & Computer Science, Univ. of Cincinnati,
ML 0030, Cincinnati, OH 45221-0030

Kleanthis Psarris
Division of Computer Science, Univ. of Texas at San Antonio,
San Antonio, TX 78249

Vivek Sarkar
IBM T.J. Watson Research Center,
P.O. Box 704, Yorktown Heights, NY 10598

**Abstract**

Over the past two decades tremendous progress has been made in both the design of parallel architectures and the compilers needed for exploiting parallelism on such architectures. In this paper we summarize the advances in compilation techniques for uncovering and effectively exploiting parallelism at various levels of granularity. We begin by describing the program analysis techniques through which parallelism is detected and expressed in form of a program representation. Next compilation techniques for scheduling instruction level parallelism are discussed along with the relationship between the nature of compiler support and type of processor architecture. Compilation techniques for exploiting loop and task level parallelism on shared memory multiprocessors are summarized. Locality optimizations that must be used in conjunction with parallelization techniques for achieving high performance on machines with complex memory hierarchies are also discussed. Finally we provide an overview of compilation techniques for distributed memory machines that must perform partitioning of both code and data for parallel execution. Communication optimization and code generation issues that are unique to such compilers are also briefly discussed.

**Keywords:** parallelism, dependence testing, instruction level parallelism, shared memory systems, distributed memory systems.

# 1 Introduction

Driven by the need to deliver high levels of performance for a wide array of applications, researchers have been developing parallel architectures and compiler based parallelization techniques for the past two decades. Parallelism at different levels of granularity, including instruction level parallelism, loop and task level parallelism, and data level parallelism has been considered by these techniques. In this paper we summarize the past developments in compilation technology and open issues that are being currently addressed by researchers.

The basis for the automatic detection of parallelism is dependence analysis. The results of the analysis enables the compiler to identify code fragments that can be executed in parallel. Thus, a great deal of effort has been spent on developing dependence tests of varying complexity and precision. The dependence tests used by compilers are conservative in nature, that is, if a test cannot prove the absence of a dependence it reports that a dependence is present. Since scientific applications have been a major focus of automatic parallelization, most of the work has considered programs involving array references within loops. The results of dependence testing for non-scientific programs involving pointer intensive computations have been far less encouraging.

Instruction level parallelism is exploited by modern day superscalars and long instruction word machines. These architectures depend greatly upon the compiler for uncovering parallelism. Studies have shown that although significant levels of instruction level parallelism is present in both scientific and non-scientific codes, its detection and exploitation requires two significant problems to be addressed. First it is essential to aggressively perform code reordering over long code sequences that extend across branches to uncover and schedule instruction level parallelism. Second aggressive memory disambiguation techniques must be employed to expose this form of parallelism. Often effective exploitation of instruction level parallelism requires a combination of architectural features and compiler support.

Shared-memory multiprocessors exploit parallelism by executing multiple threads of control that execute on independent processors and communicate via shared-memory. The global name space supported by such systems provides a uniform view of memory to all processors and simplify the task of writing or automatically generating parallel programs. Dependence analysis techniques are used to detect and schedule loop level parallelism on shared-memory machines. Experience with parallelization of non-numeric codes for shared-memory machines is quite limited. To achieve high levels of performance on such systems, in addition to the task of parallelization, it is important that the compiler pay attention be paid to data locality and placement. This is because of the multilevel memory hierarchies supported by such systems.

In an effort to scale multiprocessors to a large number of processors, distributed-memory systems have been developed. The task of the compiler for mapping programs written based upon the shared-memory model on to such machines is a complicated task. Not only is it necessary to detect parallelism and partition the computation for parallel execution, the shared data must also be partitioned and mapped to memories associated with individual processors. The complexity of the above tasks has led to the introduction of new constructs for data distribution in languages which shift some of the complexity from the compiler to the user. Since the cost of interprocessor communication is significant, the compiler must also employ a variety of optimization techniques to tolerate communication delays. Furthermore the distribution of arrays across memories requires new address generation techniques to be used.

The remainder of the paper is organized as follows. In section 2 dependence analysis techniques are surveyed. The combination of compiler support and accompanying architectural support required to exploit instruction level parallelism is described in section 3. The techniques for exploiting loop and task level parallelism on shared-memory multiprocessors are discussed in section 4. In section 5 the unique issues that the compiler faces in taking advantage of distributed memory machines is discussed. Concluding remarks are given in section 6.

# 2 Program Analysis and Representation

In a multiple processor system, traditional sequential computer programs must be redesigned to efficiently coordinate the parallel processors. In this way, a substantial gain in computational performance may be achieved. Since the tasks of redesigning a sequential program by hand or designing explicitly parallel programs can prove to be very difficult, one major approach is the automatic detection and incorporation of parallelism into the original sequential program. Optimizing compilers actually exploit limited forms of parallelism in sequential programs. By extending their capabilities, parallelizing compilers were designed to automatically restructure programs for execution on parallel architectures. Automatic detection of parallelism requires thorough, yet efficient, program analysis and optimization to identify independent tasks in a program that can be candidates for parallel execution.

Since early programming languages were developed for single processor machines, their semantics naturally reflected the underlying architecture that imposes an explicit order in which program statements are executed. This total ordering, however, is more restrictive than is necessary to preserve the semantics of a program. In fact it can be shown, using the concept of *dependence*, that only portions of the original ordering are absolutely necessary. Dependence is a partial order relation among the statements of a program. Statement $S_1$ is said to depend on $S_2$ if $S_1$ must be executed before $S_2$ in order to preserve the semantics of the original program. Statements in a program that are not dependent may be executed in any order and, therefore, in parallel. Dependence constraints within a program can arise from data considerations as well as from control considerations. In particular, two statements are *data dependent* if they both access the same memory location and at least one of them writes on it; whereas, two statements are *control dependent* whenever the execution of one statement is conditional on the results of the other.

In this section we discuss the program analysis techniques employed by compilers to detect dependences in sequential programs. We first provide some background about dependence analysis and intermediate program representation. Next we review the proposed data dependence analysis tests and techniques. Finally, we discuss current trends and future directions in dependence analysis.

## 2.1 Dependence Analysis

Three types of data dependence constraints are usually defined:

- *Flow dependence.* When a variable is assigned in one statement and used in a subsequently executed statement.
- *Anti-dependence.* When a variable is used in one statement and reassigned in a subsequently executed statement.
- *Output dependence.* When a variable is assigned in one statement and reassigned in a subsequently executed statement.

In the three cases above, the order of execution in the original program needs to be preserved. However, anti-dependence and output dependence arise from reassignment of variables and can be eliminated by variable renaming [177]. Flow dependence is inherent in the computation and can not be eliminated. Control dependences are derived from the usual control flow graph [4]. For scalar variables, traditional data-flow analysis [4] can find the exact data dependence relations. For arrays, the problem is more complicated and the compiler must examine the subscript expressions, as discussed later in this section.

An abstraction that compilers use to represent data dependence information is the *data dependence graph*. The nodes in the graph correspond to statements in the program and the edges correspond to data dependence constraints among the statements. A unified framework for data and control dependence representation is also introduced in the form of the *program dependence graph (PDG)* [51]. The PDG allows transformations such as vectorization, without special treatment of control dependence, to be performed in a manner that is uniform for both control and data dependences. A couple of other intermediate program representations used to represent both control and data dependences are proposed in [60, 133].

In loops, each statement can be executed many times. Data dependences can flow from any instance of execution of a statement to any other statement instance, and even back to the same statement. A data dependence relation between two statement instances in two different iterations of a loop is called *cross-iteration dependence*. A data dependence relation between two statement instances in the same iteration of a loop is called *intra-iteration dependence*. Since a compiler can not represent the repeated instance of each statement individually, an extended abstraction is needed. The data dependence graph is still constructed with one node for each statement. However in this case, each node may represent many instances of that statement. Similarly, the edges in the graph may represent many instance-to-instance data dependence constraints. Data dependence relations are annotated with information about the relative iterations in which the related (dependent) instances occur. Such information is stored in distance or direction vectors [177].

A major focus of automatic detection of parallelism has been at the loop level. This situation arose from the many important scientific computing applications that rely heavily on loop computations. In general a sequential loop can be transformed into a parallel one if it does not contain any cross-iteration dependences. Loop parallelism depends in part on the resolution of array aliases. The underlying problem is that of detecting multiple references to the same array element within a nest of loops. Data dependence analysis techniques compare every pair of references to the same array and attempt to determine if their subscript expressions, across different iterations, could possibly evaluate to the same value (i.e., same memory location). Existing techniques make the assumption that array subscripts are linear functions of the loop index variables. Computation of data dependences for arrays with non-linear subscripts is an extremely difficult problem.

Data dependence testing for multidimensional arrays reduces to determining if a system of linear equations (one equation for each dimension) has an integer solution that satisfies a set of linear inequality constraints. The variables of the linear equations are loop index variables and the inequality constraints arise from loop limits and direction vector relationships. For single dimensional arrays there is only one equation to be tested. When testing multidimensional arrays, we say that a subscript position is *separable* [61] if its loop indices do not occur in the other subscripts. If two different subscripts contain the same loop index, we say they are *coupled* [112]. When all subscripts are separable we can test each subscript separately, since the linear equations in the system are independent. This is known as *subscript-by-subscript testing* and results in testing, one at a time, a single linear equation for integer solutions that satisfy a set of linear inequality constraints. However, this method introduces a conservative approximation in the case of multidimensional arrays with coupled subscripts.

## 2.2 Data Dependence Tests

The data dependence problem is equivalent to integer linear programming, and thus, it can not be efficiently solved in general. A number of data dependence tests are proposed in the literature. In each test there are different tradeoffs between accuracy and efficiency. Data dependence tests always approximate on the conservative side; i.e., a dependence is assumed if independence cannot be proved. In this way, no unsafe parallel program is produced.

**GCD Test** [18]. The GCD test is arguably the most basic of the dependence tests, and is often incorporated into other dependence tests as an initial screen for dependences. It is based on a theorem of elementary number theory, which states that a linear equation has an integer solution if and only if the greatest common divisor of the coefficients on the left hand side (LHS) of the equation evenly divides the constant term on the right hand side. The test simply checks for this integer divisibility. If this condition does not hold, there can be no integer solutions to the linear equation, and hence, no dependence exists. However, if the condition does apply then a dependence does not necessarily exist. In the later case the GCD test returns a maybe answer.

The GCD test is limited in several major respects: Firstly, as noted above, it is an inexact test. It is a necessary but not sufficient condition for data dependence; i.e., it is incapable of proving dependences, only disproving them. Furthermore, it can only consider a single subscript of a multidimensional array reference at a time. Secondly, it is very likely that at least one of the terms in the LHS of the linear equation has a coefficient with an absolute value of 1. In such a case, the gcd of the coefficients of the LHS is 1 and, thus, the integer divisibility condition will always be met. Lastly, since inequality constraints are not taken into consideration, the test cannot provide any information regarding dependence distances or directions. In spite of these limitations, the speed and simplicity of the GCD test make it one of the most widely used tests in parallelizing compilers.

**Extreme Value Test** [18]. The extreme value test, often referred to as the Banerjee bounds test, is based on the Intermediate Value Theorem. It is another widely used test that calculates the possible minimum and maximum values an expression on the LHS of a linear equation can achieve, given bounds on each of the variables involved. Once the minimum and maximum of the expression is known, the test checks whether the constant on the RHS of the equation falls between these extreme values. If it does not, then no dependence exists. If it does fall in the range, we know only that a real solution to the linear equation exits. However, we cannot conclude that a dependence exists, since there may not in fact be an integer solution to the equation. This inability to distinguish between real and integer solutions makes the extreme value test, though highly efficient, an inexact test. In addition to providing a no or maybe answer to a dependence test, the extreme value test can be used to generate direction vector information.

Like the GCD test, the extreme value test is an inexact test that considers a single subscript of a multidimensional array reference at a time. It further requires that loop limit information be known at compile time. However, its efficiency and usefulness at disproving dependences make it one of the most common tests used in parallelizing compilers. In subsequent research [134, 135] it is shown that under certain conditions satisfied by the coefficients of the linear equation, the extreme value test becomes an exact test; i.e., a necessary and sufficient condition, for single dependence equations.

**I-Test** [105, 137]. The I-Test is based on and enhances the extreme value test. Whereas the extreme value test is unable to distinguish real from integer solutions, the I-Test can conclusively prove or disprove the existence of integer solutions (and hence dependences) in a large percentage of cases missed by the extreme value test. The I-Test arose from an observation that most real solutions predicted by the extreme value test are in fact integer solutions. This insight led to the development of a set of conditions, which if met, meant that a given linear expression actually achieved every integer value between the minimum and maximum values calculated by the extreme value test. These *accuracy conditions* state the necessary and sufficient relationship between the coefficients of loop iteration variables to the range of values they can assume in order to guarantee that every integer value between the extreme values is achievable. In practice [136], these conditions are met so frequently that the I-Test is in most cases a linear time exact test for single dimensional array references.

The I-Test can be thought of as performing the extreme value test one variable at a time. As each variable is considered, the accuracy conditions are tested. If they are met, the test continues. If all variables pass the test, as is most often the case, the I-Test will have conclusively disproved or proved a dependence. On the other hand, if one of the variables does not meet the conditions, the test simply reverts to the extreme value test, meaning that it can only disprove dependences, not prove them. The I-Test can be applied on a subscript by subscript basis in the case of multidimensional array references. If dependence is disproved when considering any of the subscripts then there can be no dependence. However, if all the individual tests produce yes answers, a dependence is known to exist only if the subscripts are separable. The I-Test cannot prove the existence of simultaneous integer solutions in the case of coupled subscripts, and thus returns a maybe answer. The I-Test inherits all of the benefits of the extreme value test, including efficiency and ability to provide direction vector information. It is always at least as accurate as a combination of the extreme value and GCD tests, and is almost always more accurate. However, it is not without its weaknesses; particularly, its inability to precisely handle multidimensional array references involving coupled subscripts, as well as its reliance on information known at compile time.

Lambda Test [112]. The Lambda test addresses coupled, multidimensional array references. It determines whether the intersection of the hyperplanes (that represent the individual subscripts within the array references) intersects the convex region formed by the loop limits and direction vector constraints. The test proceeds by examining a set of linear combinations of the intersection of these subscript hyperplanes, called "canonical solutions". For each such hyperplane combination, the intersection with the convex region is determined using the extreme value test. If any one of these canonical solutions is found not to intersect the region, a no answer is returned. Otherwise, if all canonical solutions do intersect the region, the Lambda test returns a maybe answer. This is because the Lambda test, like the extreme value test, does not distinguish between real and integer solutions.

The Lambda test is an efficient test. If no direction vectors are considered, it examines at most one canonical solution for every variable involved in the array reference. At most 3n/2 canonical solutions are examined if both loop bounds and direction vectors are present, where n is the number of variables in the array reference. The same reliance on compile-time information exists for the Lambda test as for the I-Test and extreme value test.

Generalized GCD Test [18]. The Generalized GCD test applies the ideas behind Euclid's GCD algorithm to determine if there are simultaneous integer solutions to a system of linear equations. Using linear algebra properties, the test begins by constructing a (n x m) coefficient matrix A and a (1 x m) coefficient matrix C, where n is the number of variables in the system and m is the number of equations. Next, an unimodular transformation matrix U is found that converts A into an upper triangular matrix D by a series of elementary row operations (as in Gaussian elimination). If there is an integer solution t such that tD = C, then integer solutions exist and dependence is assumed. Since D is an upper triangular matrix, simple back substitution can be used to easily determine if a solution t exists. Because of the echelon form of D, some of the coefficients of t will have zeros in them. These are referred to as "free variables". Taking the matrix product tU allows the calculation of additional information, such as constant dependence distances. Although the generalized GCD test considers only integer solutions to a set of linear equations, it does not take constraints such as loop limits into consideration. Thus, it is not useful for proving the existence of dependences, only disproving them.

Fourier-Motzkin Variable Elimination (FMVE) [45]. FMVE is a technique used in several dependence analysis tests, most notably the Omega and Power tests that are discussed later in this section. It is used to determine the existence of any real solutions to a system of linear inequalities, and simply treats constraints such as dependence vectors and loop limits as additional inequalities. FMVE proceeds by systematically eliminating variables from the system of linear inequalities. For example, once a variable x is selected for elimination, all inequalities in the system are rewritten in terms of upper and lower bounds on x. Next, each lower bound on x is compared to each upper bound on x, and a new inequality is derived that does not involve x. All the inequalities involving x are deleted, and the remaining inequalities are in terms of one less variable. This process continues until all variables have been eliminated, or until a contradiction is reached. If a contradiction is detected, FMVE announces that no dependence exists. Otherwise, a maybe answer is returned, indicating that the test is unsure if the solution is integer or not. In the worst case, the number of inequalities in the system could grow exponentially as variables are eliminated. However, in practice, many of the inequalities produced are redundant, and exponential growth usually does not occur.

Omega Test [138]. The Omega test is based on a combination of the least remainder algorithm and FMVE. It always produces exact yes/no answers, but has worst case exponential time complexity. The Omega test begins by employing a derivation of Knuth's [106] least remainder algorithm to convert a system of linear equalities and inequalities into a system involving only linear inequalities. During this initial conversion, bound normalization and the GCD test are employed to detect if the system is inconsistent. If so, the test reports that no dependences exist. Otherwise, an extension to standard FMVE is used to determine if the resulting system of linear inequalities has integer solutions (recall that FMVE can only prove or disprove the existence of real solutions). Intuitively, the elimination of a variable may be viewed as projecting an n- dimensional polyhedron onto an n-1 dimensional surface. If the resulting "real" shadow contains no integers, then the original object contains no integers, and the test reports that no solutions exist. The converse, however, is not necessarily true: the "real" shadow may contain integers, whereas, the original object actually contains no integers. The refinement added by the Omega Test consists in calculating a subset of this real shadow, called the "dark" shadow that corresponds to the area under the original object where integer solutions definitely exist. If this dark shadow is non-empty (i.e. contains integers), then the Omega Test reports that dependences exist. If the real shadow is non-empty, but the dark shadow is empty, the Omega Test begins an essentially exhaustive search of the solution space, recursively generating and solving integer programming problems until integer solutions are either found or disproved.

In addition, the Omega Test can be used to simplify integer programming problems via symbolic projection. Given an integer programming problem P and a set of protected variables V, P is reduced to one or more sub-problems involving only the variables in V. These sub- problems describe the values of the variables in V that produce integer solutions to P. Using this technique, the Omega Test can produce data dependence distance vectors that describe the relationship between loop iterations in which the data dependence occurs. Furthermore, the Omega test can handle certain cases in which loop bounds are unknown at compile time.

Power Test [179]. The Power test combines the Generalized GCD test with FMVE, producing a test that takes loop limits and direction vector constraints into consideration. Initially, the Generalized GCD algorithm is used to produce the transformation matrix U and upper triangular matrix D as described earlier and the integer matrix t is produced by back substitution. Loop limits and direction vector information is used to add upper and lower limits on the free variables of the t matrix to the system. Given this system of inequalities that describe upper and lower bounds, the Power test then performs FMVE on the free variables.

The Power test is applicable in many cases in which information is unknown at compile time; in particular, for those involving unknown loop limits. Since the Power test utilizes FMVE, it shares its exponential worst-case execution time; however, this is in terms of the number of free variables after the generalized GCD test has terminated. Additionally, in cases where the integer solutions occur near loop limits, or in cases where an imprecision is introduced through floor and ceiling calculations, the test can return a conservative yes answer when in fact no integer solutions exist.

Interprocedural Analysis. The presence of procedure calls raises some important practical issues in relation to data dependence analysis across interprocedural boundaries. A simple solution is to expand (inline) the procedure and then perform dependence analysis on the resulting program. The major technical difficulty in this case is that it is necessary to reflect in the resulting code the effect of aliasing between formal and actual parameters. The main drawback to this practice is that the size of the resulting code could become unmanageably large if all procedures are expanded. For this reason, several techniques for interprocedural data dependence analysis have been developed [31, 41, 111].

### 2.3 Future Directions

The limitation of program analysis is the inability to cope with statically unknown information. The existing dependence tests can not handle loop bounds or array subscripts that are symbolic or nonlinear expressions. In the presence of symbolic or nonlinear expressions, dependence is usually assumed. In certain cases, symbolic program analysis techniques [28, 73] can help to overcome this problem and enable the effective parallelization of a larger class of applications. But even symbolic analysis has its limitations, simply because the necessary information can not be obtained or inferred at compile time. In order to realize the full potential of automatic detection of parallelism, static program analysis techniques may be complemented by run-time dependence analysis and parallelization [144].

## 3 Instruction Level Parallelism

Parallelism studies have shown that both scientific and non-scientific codes contain high amounts of instruction level parallelism (ILP). Studies have shown that it is possible to achieve average instruction issue rates of as high as 100 instructions per cycle for some of the Spec95 benchmarks [79]. Uncovering high levels of ILP requires examination of instructions over a large instruction window containing hundreds and even thousands of instructions. In [124] it was shown that the relationship between extracted parallelism and the instruction window size is a quadratic one. In other words, in order to double the degree of ILP detected, the size of the instruction window must be quadrupled. A compiler constructs a large instruction window by examining instructions along program paths that extend across branches in acyclic code as well as cyclic code (i.e., across loop iterations). By reordering the instructions, and placing independent instructions in proximity of each other, the compiler enables generation of parallel schedules. In the context of statically scheduled very long instruction word (VLIW) machines, code reordering is performed so that independent operations can be explicitly scheduled to execute in parallel by their assignment to the same long instruction by the compiler. In the case of superscalar processors the actual schedule is determined dynamically. Instructions from within a small hardware instruction window, which is typically less than 100 instructions, are issued out-of-order to exploit ILP. However, the code reordering performed by the compiler over a large instruction window causes additional parallelism to become accessible within the small hardware instruction window of a superscalar processor.

In this section we first summarize the innovations in compiler technology that deal with global scheduling of *acyclic* program regions. We discuss the role of *speculation* and briefly outline the challenges posed by *predicated execution* supported by modern processors. The relationship of instruction scheduling to the task of register allocation is also briefly discussed. Next we discuss the innovations in the area of *cyclic* scheduling which is achieved by software pipelining algorithms. Various formulations to the cyclic scheduling problem are briefly discussed. The degree of ILP that can be uncovered statically by the compiler, or dynamically by the hardware, is greatly influenced by the effectiveness of the memory disambiguation techniques employed. We discuss some recent innovations in this area. Finally some current trends in processor design and their impact on compiler technology is discussed.

## 3.1 Acyclic Schedulers

In order to uncover significant levels of ILP the instruction window employed by the compiler must extend across branches. Acyclic schedulers employ an instruction window that extends across branches within an acyclic code segment. A variety of global scheduling algorithms continue to be developed to handle the task of scheduling acyclic code segments in a program. Some of the well known global scheduling techniques include trace scheduling [52], percolation scheduling [5], region scheduling [67], superblock scheduling [88], and critical path reduction [157]. These techniques can be characterized by the scope of the instruction window they employ, the program representation upon which they rely, and the nature of code reordering transformations that they incorporate. Each of these techniques is able to perform code motion across branches, including speculative code motion.

1. Trace Scheduling [52]: This is the earliest and the best known global instruction scheduling technique that was pioneered by Fisher. In this technique the instruction window is limited to a trace which is a sequence of consecutive basic blocks along a path in the program control flow graph. A trace cannot extend across loop boundaries. The data dependence graph for the trace is constructed and a scheduler (e.g., a list scheduler) is used to generate the instruction schedule. In the course of scheduling, instructions may be propagated across merge points and spilt points in the control flow graph. In this technique it is critical that the traces representing more frequently executed paths are constructed and scheduled prior to those that are executed less frequently. Doing so ensures that when speculative code motion is performed in order to generate a good schedule for the trace being processed, the paths that suffer from speculative code motion are less frequently executed than the current trace. Trace construction can be driven either by profile data or compiler time heuristics for identifying biased conditional branches. While the remainder of the scheduling techniques discussed here offer some improvements over trace scheduling, they also rely upon the core transformations first introduced by trace scheduling.

2. Percolation Scheduling [5]: This is a program transformation system that has certain advantages over a trace scheduler. A trace scheduler divides program transformation into two stages. The first stage reorders code along the trace while the second bookkeeping stage modifies the rest of the program to preserve program semantics. This separation of transformation process into two steps can lead to the generation of redundant code during the bookkeeping stage. By applying semantics preserving transformations in one step, percolation scheduling avoids this problem.

3. Region Scheduling [67]: This is also a program transformation system that operates upon the program dependence graph [51] representation of the program. Instruction scheduling is carried out by first reordering code within a control dependence region and then by performing code motions across control dependent

regions. Code reordering within a control dependence region is given preference because it results in less code growth and unlike speculative code motion it does not harm any program paths. Such code reordering, unlike trace scheduling, also enables instructions to be moved across loop boundaries. Much of the redundant code generated by a trace scheduler during bookkeeping is also avoided. The instruction schedule is progressively improved through code motion transformations until no more improvements in the schedule can be identified.

4. Superblock Scheduling [88]: This scheduling technique simplifies the complexity of compensation code generation associated with trace scheduling. In this technique, before instruction scheduling is carried out, superblocks are created through tail duplication that eliminates code following merge points in the program. Since code reordering performed on the resulting superblocks do not result in any code motion above merge points, the compensation code generation is simpler and the redundant code generated by the trace scheduler is avoided. However, this simplification comes at the cost of significant code growth.

5. Critical Path Reduction [157]: In this technique the program region being scheduled is a multiple entry multiple exit acyclic region. The region exits are classified into two categories, frequently taken and infrequently taken. Even if the paths leading to frequently taken exits does not include any delay slots, an attempt is made to reduce the schedule length by pushing statements off these paths and to the infrequently taken exits. This transformation is sometimes possible because a path to a frequently taken exit may contain statements that are dead along these paths even though they may be live along other paths. Essentially, this technique integrates the partial dead code elimination optimization into the instruction scheduler.

Predication. Each of the above techniques perform speculative code motion to aggressively reorder code. A mechanism for enabling additional code reorderings is *predicated execution*. The IA-64 architecture supports *predicated execution* in which an instruction is conditionally executed depending upon the value of its predicate input [87]. The control flow through an acyclic code fragment that is implemented through branches can be entirely eliminated by predicating the instructions. The resulting code may be viewed to simultaneously execute all the paths through the original acyclic code. There are some situations in which the use of *predication* can be quite useful. Predication may be used to eliminate unpredictable branches and consequently reduce the harmful effects of such branches on the execution. If a speculatively issued load frequently causes a cache miss along some program path then by predicating the load it may be possible to avoid the speculative issue along that path and hence minimize the cache misses. While in the above limited situations the benefits of *predication* are clear, its aggressive application is a far more challenging task. If the lengths of the paths through an acyclic code vary greatly, *predication* would extend the execution times of shorter paths [17]. Moreover the demand for register resources may be increased to a point that *predication* no longer yields superior instruction schedules. Finally the data flow analysis techniques used to analyze programs, for such basic tasks as uncovering data dependences and performing optimizations, is based upon explicit control flow. A new analysis framework must be developed to accurately handle the implicit control flow expressed in predicated code [95].

Interaction with register allocation. The interaction of instruction scheduling and register allocation is an important issue for VLIW and superscalar architectures that exploit significant degrees of ILP. Register allocation and instruction scheduling have somewhat conflicting goals. In order to keep the functional units busy, an instruction scheduler exploits ILP and thus requires that a large number of operand values be available in registers. On the other hand, a register allocator attempts to keep the register demand low by holding fewer values in registers so as to minimize the need for generating spill code.

If register allocation is performed first, it limits the amount of ILP available by introducing additional dependences between the instructions based on the temporal sharing of registers. If instruction scheduling is performed first, it can create a schedule demanding more registers than are available, causing more work for the register allocator. In addition, the spill code that is generated must be incorporated in the schedule by another scheduling pass, degrading the performance of the schedule. Thus, an effective solution should integrate register allocation and instruction scheduling. The significance of integration is greatly increased in programs where the register demands are high since the likelihood of spill code generation is high for such programs. Also, when compiling programs for wide issue machines, the need for integrating register allocation and instruction scheduling is the greatest.

A number of approaches have been proposed for integrating together instruction scheduling and register allocation. These approaches differ in the complexity of the techniques used for detecting excessive resource demands and the strategies for reducing resource demands.

1. On-the-fly Approach [63]: This approach performs local register allocation within extended basic blocks during instruction scheduling. It uses the on-the-fly measures of register needs to detect excessive register demands. The part of the program over which a value is used by the program is referred to as the *live range* of the value. In order to reduce register demands this algorithm employs *live range spilling* which places a value in memory for the entire duration of the value's *live range*. The main weakness of this approach is that it does not perform *live range splitting* which places a value in memory for only part of the time and keeps it in a register for the remaining duration of the live range. *Live range splitting* is far superior to *live range spilling*.

2. Parallel Interference Graph [123]: This approach uses an extended form of register interference graph to detect excessive register demands and guide *schedule sensitive* register allocation. The reduction in register demands is achieved through live range spilling.

3. Unified Resource Allocation [25]: This approach is based upon the *measure-and-reduce* paradigm for both registers and functional units. Using the *reuse dag* representation for registers and functional units, it identifies *excessive sets* that represent groups of instructions whose parallel scheduling requires more resources than are available. The excessive sets are used to drive reductions of the excessive demands for resources. Live range splitting is used to reduce register demands; whereas, serialization of otherwise independent instructions is performed to reduce excessive functional unit demands. Once excessive sets have been eliminated, a simple list scheduler is used to generate the final schedule.

## 3.2 Software Pipelining

The instruction window from which ILP is extracted by acyclic schedulers consists of paths that cannot extend across loop iterations. Thus, acyclic schedulers cannot exploit ILP present across code blocks from different loop iterations. One approach to uncover such parallelism is to unroll the loops to transform parallelism across loop iterations into parallelism that exists within a single loop iteration of the transformed loop. The transformed loop can then be scheduled by an acyclic scheduler. However, this approach can result in substantial code growth. Software pipelining exploits ILP across loop iterations without causing significant code growth.

Software pipelining is a technique which overlaps execution of operations from different loop iterations and thus exploits ILP across loop iteration boundaries. The objective of software pipelining is to generate a schedule which minimizes the interval at which iterations are initiated, that is, the *initiation interval*. A software pipelining algorithm must take into account the instruction latencies and resource availability while scheduling the operations from the loop. In addition, any increase in register demands must be met to avoid generation of spill code inside loops. Generally the techniques for software pipelining assume that the loop body of the pipelined loop contains no branches.

Rau and Glaser [143] introduced *modulo scheduling*, the most commonly used framework for software scheduling. In this approach a lower bound on the *initiation interval* is established based upon the data dependences in the loop and the resource demands of the loop. The modulo scheduler then searches for a schedule with the minimum *initiation interval*. If the search fails, the *initiation interval* is increased and the search is performed again. The above process is repeated until a schedule can be found. In the above manner the modulo scheduler finds a schedule with the minimum *initiation interval*. Since this technique was first introduced, numerous variants of modulo scheduling have been developed. The SGI's MIPSpro compiler employs a form of modulo scheduler. It sets both and upper and lower limit on the *initiation interval* and performs a binary search over the interval. It also employs a branch-and-bound algorithm in order to search for a schedule with a given *initiation interval*. Extensive pruning is performed in order to ensure that the search for a schedule is efficient in practice.

The algorithms based upon modulo scheduling are essentially heuristics. Motivated by the search for optimal algorithms, researchers formulated software pipelining as an integer linear programming problem [8]. The formulations allow for non-pipelined as well as pipelined functional units and arbitrary structural hazards. Although integer linear programming is an NP-complete problem, a number of standard packages for solving this problem exist; and therefore, can be employed to compute optimal schedules. In a recent study, when the results of the optimal algorithms are compared with the SGI's production compiler based upon a form modulo scheduling, it was observed that the heuristics for modulo scheduling employed by the SGI compiler are highly effective [148].

The solutions discussed above represent scheduling frameworks that are entirely distinct from those used by acyclic schedulers. In contrast the *circular scheduling* technique builds upon existing basic block schedulers to construct a software pipeliner [92]. Software pipelining is performed by code motions that move selected instructions from the top of the loop body to the bottom of the loop body. Doing so essentially causes an instruction from the current iteration to be moved out of the loop body; whereas, the same instruction from the next iteration is moved up into the loop body. Thus, the loop body is viewed as a circular list of instructions along which instructions can be moved. This techniques was implemented in a MIPS production compiler.

### 3.3 Memory Disambiguation

Serialization of memory load and store operations greatly limits the ILP that can be extracted from a program. In order to uncover significant levels of ILP, effective memory disambiguation techniques must be employed. Absence of dependences between store and load operations allows early issuing of load operations leading to better memory latency tolerance. Compile-time disambiguation techniques based upon dependence analysis were discussed earlier in this paper. Here we discuss two solutions to the memory disambiguation problem that require hardware support.

The *store set* mechanism is a highly effective hardware memory disambiguator that was recently proposed for superscalar processors [39]. The basic idea behind this technique is to initially assume that there are no dependences between stores and loads. Thus, loads are freely speculated past preceding stores. However, as mispredictions are observed and dependences are detected, the observed dependences are saved in a hardware structure. A group of stores that have been observed to reference the same address form a *store set* and each load which is dependent upon some store is associated with the store set containing that store. When a load that is linked to a *store set* is encountered again, it is not speculated pasts the stores belonging to the *store* *set*. Thus, in this approach *load speculation* is only inhibited by dependences that have been observed during program execution. In contrast, compile-time disambiguators are conservative in nature, that is, if they cannot prove the absence of a dependence they must assume that one is present.

Since ILP scheduled by statically scheduled long instruction word machines must be extracted at compile-time, their performance can be significantly limited due to conservative dependence analysis. As we discussed earlier in this paper, numerous dependence tests have been developed for handling arrays. However, research into dependence analysis techniques that work in the presence of pointers has not yielded equally positive results. Thus, extraction of ILP from non-scientific code requires that we look for alternatives to dependence analysis. The IA-64 architecture overcomes this problem using an innovative solution based upon a combination of hardware and software support. The instruction set provides a pair of new instructions, the *speculative load* instruction and the *load verify* instruction. If a load instruction is speculatively moved above earlier store instructions by the compiler, it is so indicated by using a *speculative load* instruction. When the store instructions are executed following the *speculative load*, the result of the load instruction is marked as invalid. Before the result of the load instruction is used, the *load verify* instruction is executed. This instruction checks the validity of the value loaded by the *speculative load* and if the result is invalid the load is reissued. If the loaded value is valid, the *load speculation* performed has successfully managed to hide the latency of the load operation. The candidates for *load speculation* may be found through address profiling.

### 3.4 Future Directions

Although extensive research has been performed on code optimizations, much of this research ignores the factors that are important in the ILP domain. Developing optimization algorithms that take into account path execution frequencies and machine characteristics poses a significant challenge.

The number of new processor architectures being developed today continues to increase at a rapid rate. In particular, new application domains have led the development of a wide array of special purpose embedded processors. At the same time general purpose architectures are being developed today to find increasingly effective means of detecting and exploiting high degrees of ILP present in non-scientific code. The above developments have also created new challenges for the compiler community.

Code optimizations and ILP. Effective application of code optimizations in the ILP domain requires two main issues to be addressed. First a suitable strategy for integrating the code optimizations with instruction scheduling must be developed. One approach that has been considered by researchers interleaves the application of optimizations with scheduling actions [74, 68]. Another approach suggests application of optimizations prior to instruction scheduling using new optimization algorithms that are aware of the effects that an optimization may have on register and functional unit demands [69, 70]. Second we must develop new optimization algorithms that take advantage of path execution frequencies and machine characteristics. For example, researchers have developed algorithms for eliminating redundant code [71, 29] and dead code [72, 30] along frequently executed program paths. These algorithms take advantage of path execution frequencies and utilize the *speculation* and *predication* features of the IA-64 architecture.

Researchers have also begun to realize that with the information available at run time, the compiler's ability to extract ILP and optimize code is greatly increased. Fisher [53] describes the notion of *walk-time* techniques which perform code optimizations at compile time as well as run time based upon profile data collected during program runs.

**General purpose architectures.** The design of a processor architecture significantly impacts the demands placed upon the compiler for exploiting ILP. Next we discuss some current trends in processor design and their impact on the complexity of the ILP compilers. Interestingly architectures being developed range for those that rely heavily on sophisticated compiler support to those that perform increasing number of traditional compile time tasks in hardware.

The IA-64 architecture supports several features for exploiting high degree of ILP and relies on a sophisticated compiler to exploit these features [48]. It allows the compiler to explicitly express instruction level parallelism in the machine code. An IA-64 instruction bundle contains three independent operations and a template which encodes dependence information allowing independent instructions to be placed across multiple bundles. IA-64 also supports *full predication* and *speculation* which are both exposed to the compiler. As mentioned earlier, IA-64 also supports a form of *load speculation* which can only be exploited through proper compiler support. Thus the compiler is responsible for exposing ILP using *speculation* and *predication* and then encoding the parallelism into instruction bundles.

Recently architectures have been proposed that on the one hand support instruction sets similar to conventional superscalars while on the other hand employ a long instruction word type *execute engine* internally [121, 49]. The long instruction word schedules are determined at run time and cached for reuse. This approach lowers the burden on the compiler and may lead to better instruction schedules as the schedules are based upon dynamic information. Moreover caching of instructions and repeatedly reusing them suggests that such architectures may be prime candidates for incorporating low level optimizations that are at present typically left up to the compiler [55]. Examples of such optimizations include copy propagation and constant propagation.

Architectures that combine characteristics of superscalar processors and long instruction word machines in creative ways are also being proposed [40]. An illustration of this approach is the recent attempt to incorporate load value prediction [116] in long instruction word machines [56, 122]. Aggressive *speculation* through *value prediction* has been proposed as an approach to scale the performance of superscalars to high issue widths [116]. A combination of compiler and hardware support is required to allow statically scheduled long instruction machines to take advantage of value prediction [56]. Value profiling is used to identify loads for which the values loaded can be accurately predicted. The compiler then schedules as early as possible the execution of instructions dependent upon these loads based upon values provided by the value predictor at run time. Explicit checks are also scheduled by the compiler to detect mispredictions. Recovery from mispredictions is achieved by executing compensation code. Although compensation code can be explicitly generated by the compiler [56], doing so results in higher misprediction penalty and a significant code growth. In order to avoid these drawbacks an architecture is proposed in [122] that automatically generates compensation code at run time which is executed on a dedicated execution unit.

**Special purpose architectures.** A wide array of products ranging from video games and cellular phones to automobiles are increasingly relying on specialized processors for embedded control. An embedded processor may be a simple controller or a very high performance DSP processor. Although the trend in this area is towards VLIW like processor designs, the issues that compiler must face here are significantly different and the compiler technology developed for exploiting ILP must be retargeted to handle certain important new complexities. The compilers must produce compact and efficient code that exploits ILP to meet high levels of real time performance. In addition, the compiler must contend with specialized data paths, addressing modes, and instructions for domain specific needs that are often found on embedded devices.

Although many applications executing on an embedded system may be able to meet the execution speed performance requirements by exploiting ILP, they must be able to deliver this performance under tight memory requirements. Thus the issue of *code density* is extremely important in such systems. The code growth that accompanies existing scheduling techniques may not be acceptable [54, 67, 43]. Jain et al. [93], developed a code motion framework to exploit ILP on TMS 320C2x. However, this framework is limited only to a basic block. Also the existing scheduling techniques do not deal with the issue of complex data paths between functional units that are typically found in DSPs [14] and they must be extended to incorporate instruction selection techniques that would lead to compact code. Liao et al. [114], have addressed the instruction selection issue for embedded DSP processors. Their method, based on a binate covering formulation, detects opportunities for combining neighboring simple instructions. However, limited opportunities exist if one were restricted to combining neighboring simple instructions. One could expose far more opportunities for combining such instructions through code motion obeying data dependencies.

Many architectures (e.g., the TI TMS320C2x DSP family) provide indirect addressing modes with auto-increment and auto-decrement arithmetic. By carefully laying out data variables in storage, the address arithmetic can be subsumed by auto-increment and auto-decrement resulting in faster and denser code. In order to increase opportunities to use efficient auto-increment and auto-decrement addressing modes, optimizing DSP compilers, such as the SPAM [161] compiler, delay storage allocation of variables to after the code selection phase. After code generation, when the sequence of variable accesses is known, *storage assignment* is performed as a separate code optimization pass. The *storage assignment* problem was first studied by Bartley [21] and Liao et al. [113]. Liao et al., formulated the *simple offset assignment problem* (SOA) that is a simplified *storage assignment* problem with a single address register. They modeled the problem as a graph theoretic optimization problem similar to Bartley and showed that the SOA problem is equivalent to the *maximum weighted path covering* (MWPC) problem and proved that it is NP-complete. Recently, Rao et al. [147], have derived a sequence of systematic program transformations when applied solves the problem efficiently for most cases. A number of approaches [43, 15, 14, 166, 165] based on instruction selection and scheduling have been proposed to improve code size but they do not address the problem of code compaction in relation to *storage assignment*.

# 4 Shared-Memory Parallelism

In this section, we summarize past work, the current state-of-the-art, and future directions in the area of compiler technologies for exploiting shared-memory parallelism. Issues related to distributed-memory parallelism are discussed in a later section. The compiler technologies discussed in the previous section enable exploitation of instruction-level parallelism in modern processors by grouping together independent instructions that belong to the same sequential thread of control. Shared-memory parallelism is an additional dimension in which parallelism is exploited by executing multiple threads of control on a shared-memory multiprocessor that contains multiple processors connected to a shared global memory. Over the last two decades, symmetric shared-memory multiprocessors (SMPs) have evolved from one-of-a-kind experimental machines to widely available commercial systems. In fact, because of the favorable price-performance characteristics of SMPs, future computer systems are more likely to be shipped as an SMP configuration rather than a uniprocessor configuration. Recent advances in VLSI technologies that enable an entire SMP to be integrated within a single chip will further strengthen this trend of making SMP configurations the default, compared to uniprocessor configurations.

Shared-memory parallelism is an attractive programming model for multiprocessors because it supports a global address space, that is, it enables any global data to be uniformly accessed by any processor. This greatly simplifies the task of writing a parallel program. To obtain a *correct* parallel program, the programmer needs to focus only on correct parallelization of the program's control logic; the global address space in a shared-memory multiprocessor ensures that all shared data will be uniformly accessible by all threads/tasks in the parallel program. However, this simplified view of shared-memory parallelism is often inadequate to obtain an *efficient* parallel program and it becomes important to also pay attention to data locality and placement for performance reasons. Many of these performance considerations apply to modern uniprocessors with deep memory hierarchies as well, so the effort required for tuning the performance of a shared-memory parallel program is mostly covered by the effort required for tuning the performance of the sequential version of the same program.

There are two ways of exploiting shared-memory parallelism on shared-memory multiprocessors:

1. Explicit parallelism. In this approach, the user writes an explicitly parallel program. Multithreading is the most common programming model for SMPs (e.g., Posix threads and Java threads). Other programming models for explicit parallelism build on language constructs at a higher level than threads, such as doall's and cobegin-coend's. The advantage of explicit parallelism is that the programmer can control exactly where the parallelization occurs. The disadvantage is that writing an explicitly parallel program requires more effort and is more error-prone than writing a sequential program, because there is the possibility of introducing errors in concurrent data access (e.g., errors due to data races) and in coordination of parallel threads (e.g., errors due to deadlocks). In general, explicit parallelism has been moderately successfully on non-numeric applications, but at the cost of programmer effort.

2. Implicit parallelism. In this approach, the user writes a sequential program and the compiler performs *automatic parallelization*, that is, the compiler automatically generates a shared-memory parallel program from the given sequential program. The advantage of implicit parallelism is that it reduces programming effort for parallelization. The disadvantage is that technologies for automatic parallelization are limited to certain classes of programs (typically, scientific programs with well-structured loops and dense matrix data accesses). In general, implicit parallelism has been a successful approach for shared-memory parallelization of many engineering and scientific programs written in Fortran and C.

The rest of this section is organized as follows. We first provide a brief summary of the history of loop parallelization techniques for shared-memory parallelism. Next the impact of data locality on loop transformations and shared-memory parallelization is discussed. We also outline approaches to statement-level or task-level parallelization for shared-memory multiprocessors, that go beyond loop parallelization. Finally, we discuss future directions in compiler technologies for shared-memory parallelism.

## 4.1 Loop Parallelization

Loop parallelization has been the primary focus of much of the past and current research on shared-memory parallelization. The goal of loop parallelization is to enable multiple iterations of a given loop to execute concurrently on multiple processors in a shared-memory multiprocessor. For programs in which most of the execution time is spent in loops, loop parallelization can be sufficient to fully utilize all the processors in the target multiprocessor. The bulk of past research on loop parallelization has been restricted to parallelization of *counted loops*, such as Fortran `DO` loops with no premature exits.

Perhaps the earliest work on loop parallelization was due to Lamport [108], who introduced the *hyperplane method* and the *coordinate method* for parallel execution of iterations in a set of perfectly nested counted loops. The framework used in [108] also introduced the notion of *dependence vectors*, and paved the way for later work on the use of linear algebra in describing loop transformations and legality conditions for loop parallelization [171, 90, 19, 176].

Though the framework in [108] provides a solid theoretical foundation for transforming and parallelizing multiple loops in a loop nest, the supercomputer hardware of that era (late 1970's and early 1980's) was geared to exploiting a very specific form of loop parallelism, namely *vector parallelism*. Hence, many optimizing compilers of that era focused their efforts on exposing vector parallelism [6, 46, 156]. The legality rules for vectorization follows easily from the more general legality rules for loop parallelization. However, the hard problem that remains is to compute dependence vectors with sufficient precision so as to increase the set of loops that could be automatically vectorized.

The path to improving the precision of dependence vectors lay in improved *data dependence testing*. A number of data dependence tests are introduced [18, 177] that result in more precise sets of dependence vectors, which in turn leads to improvements in vectorization. In addition, a number of new loop transformations are introduced (e.g., loop distribution, loop fusion, loop tiling) that went beyond the linear algebra framework introduced in [108], but are essential for obtaining good performance with vector instructions.

The next major step in the technology evolution from vector computers and vectorizing compilers was to shared-memory multiprocessors. The Cray architecture evolved into the Cray-XMP, a combination of vector processing within a processor and multiprocessing across processors. Many other experimental and commercial SMPs began to appear. Initially, it appeared that the vectorizing compiler technologies could simply be redirected to accomplish loop parallelization for SMP execution. While this was true for simple *loop nest*s, it became clear that the coarse grain nature of SMP parallelism led to at least two fundamental requirements that are not necessary for vectorization. First, it became important to parallelize outer loops that contain conditional control flow, unlike vectorization in which only the innermost loop was parallelized. Second, it became important to parallelize loops containing calls. The first problem was addressed by the introduction of the *control dependence* relationship [51], and the second problem was addressed by the advent of interprocedural analysis in parallelizing compilers [31, 42].

Until this stage, the primary metric for evaluating the effectiveness of shared-memory parallelization in a compiler was the set of loops parallelized and the granularity of the parallelized loops (coarse-grain parallelism is usually preferable to fine-grain parallelism on shared-memory multiprocessors because of its reduced synchronization cost). Many benchmark programs focused on this metric by comparing different vectorizing/parallelizing compilers based on the number of loops that are vectorized/parallelized. However, it soon became apparent that a simple count of parallelized loops was an inadequate predictor of performance in real applications, because it did not take into account the costs of data movement and data locality. Initially, the recognition of the importance of data locality came about when vectorizing compilers realized the importance of vector register allocation [7]. Later, this awareness crept into parallelizing compilers as they began to target more modern multiprocessors in which cache efficiency is as significant a performance factor as processor utilization. The next section discusses the impact of data locality on loop transformations and shared-memory parallelization.

## 4.2 Locality Optimizations

The early compiler work on shared-memory parallelization focused on speedup measurements rather than on absolute execution times. Most performance measurements in research publications are presented as speedup curves. The general belief was that the number of processors in shared-memory multiprocessors would keep increasing as hardware technology evolved, and thus scalable speedup should be the primary metric for evaluating the performance of shared-memory parallelization. This line of thinking was similar to the approach taken by designers of parallel algorithms during the 1980's, in which complexity analysis focused on analyzing the execution time of parallel algorithms as an asymptotic function of the number of processors without taking into account the cost of data movement.

The advent of processors with caches and deep memory hierarchies in the early 1990's showed that focusing solely on speedup was too simplistic a view of performance. The ratio of access time to main memory is ten times the access time to the first-level cache in these newer machines, and this ratio continues to increase with the widening performance gap between processor clock speeds and main memory access times. This implies that the performance difference due to locality optimizations can be a factor of ten or larger; thus, making it comparable to the performance difference between sequential code and parallelized code on many shared-memory multiprocessors. Hence, it became essential to consider locality optimization in conjunction with parallelization.

This realization set in as vectorizing and parallelizing compilers moved from targeting multiprocessors without caches (such as the Cray) to newer multiprocessors that contain caches. A program exhibits good locality when, for a majority of its memory instructions, the data location accessed is the same as, or in close proximity to, the data location accessed by a memory instruction executed in the recent past. The locality is said to be *temporal* if the data location accessed is the same as the location accessed by the previous instruction; otherwise, the locality is said to be *spatial*. To adapt to these locality constraints, vectorizing and parallelizing compilers built in the 1980's began extending their transformation heuristics so as to become more effective when targeting multiprocessors with caches. However, these extensions did not work well when there were conflicts between the transformations necessary for improving locality and the transformations necessary for improving parallelization. The next step was to develop approaches that could systematically express optimizations for locality and parallelism in a common framework [176, 97, 153].

The set of loop transformations that can be used to improve locality fall into two classes — *iteration-reordering* and *statement-reordering* transformations. An iteration-reordering loop transformation changes the order in which loop iterations are executed (when legal to do so) but leaves the loop body unchanged — each iteration of the loop is treated as an atomic entity. Some commonly used iteration-reordering loop transformations are *loop interchange* and *loop tiling*. When properly selected, these transformations can improve data locality by bringing together computations that access the same block of data. A statement-reordering loop transformation rearranges the statements within and across loop bodies. Some commonly used statement-reordering transformations are *loop distribution*, *loop fusion*, and *loop unrolling*.

Iteration-reordering and statement-reordering loop transformations are not mutually exclusive, and are often used in conjunction with each other. These transformations are referred to as "high-level" or "high-order" transformations because they can be performed at the source code level [107, 146] or at the level of an intermediate language that is close to the source code level [175]. In addition to performing high-level loop transformations automatically in optimizing compilers, it is also feasible for programmers to perform high level transformations by hand. In fact, these transformations have been used quite extensively in hand-coded implementations of computationally-intensive kernels [57, 89]. In contrast, hand-implementation is not a feasible approach for low level optimizations such as instruction scheduling because the programmer has little knowledge of the exact instruction sequence that would be generated by an optimizing compiler and of the hardware scheduling characteristics of these instructions.

Unfortunately, past research focused more attention on introducing new loop transformations than on the problem of automatically selecting combinations of these transformations to optimize locality and parallelism. Automatic selection of high level transformations is a hard problem because high level transformations are invertible — a poor choice of transformations can degrade performance just as effectively as a good choice can improve performance! In contrast, many classical compiler optimizations are non-invertible (e.g., in constant propagation, expressions are replaced by constants but not vice versa) so the difference between a poor optimization choice and a good optimization choice lies only in the amount of improvement that they deliver. Therefore, unlike classical optimizations, it is essential that selection of high level transformations be driven by cost models that can effectively determine when improvements or degradations in performance will occur. Since the best transformation choices may change as multiprocessors (and their cost models) evolve in the future, it is inadvisable in the long run to perform high level transformations by hand. Some notable examples of compilers and tools that perform automatic cost-driven selection of transformations for optimizing locality and parallelism on shared-memory multiprocessors are the KAP and VAST preprocessors [107, 146], IBM's XL Fortran compilers [154], and the Rice University Memoria system [118].

## 4.3 Statement-level/Task-level Parallelization

As mentioned earlier, the bulk of past compiler research for shared-memory multiprocessors focused on parallelization of loops. However, a few research projects also addressed the problem of automatically selecting "non-loop" (statement-level or task-level) parallelism in programs. Task-level parallelization can be viewed as an extension of the global instruction scheduling problem. In global instruction scheduling, the goal is to reorder instructions across multiple basic blocks so that independent instructions come together in close proximity — the unit of parallelism is a single instruction. In task-level parallelism, the goal is to reorder instructions and partition the program into independent tasks — the unit of parallelism is a single task which may contain multiple instructions with general sequential control flow (such as loops and conditionals). The goal of automatic task-level parallelization is to deliver the performance benefits of shared-memory parallelism to applications that are not loop-intensive or applications containing significant amounts of computation in non-parallelizable loops.

One of the key issues in task-level parallelization is the desired granularity of parallelism, which is determined by the task scheduling and synchronization overheads on the target multiprocessor. If these overheads are high, then it is desirable for tasks to be as large as possible; the primary drawback of large tasks is that the amount of parallelism may be small or (more generally) the division of work among tasks may not be well balanced. If the scheduling and synchronization overheads are low, then it is desirable to form smaller tasks that can exhibit better load balance. Since these considerations depend on the target multiprocessor, it is best if the task partitioning is performed automatically by the compiler or some other performance-enhancing tool. Examples of automatic task partitioning can be found in past work on partitioning programs written in functional languages such as Sisal [151], Id [170], and Haskell [32], and in imperative languages such as Fortran [152].

The data dependence analysis work developed for loop parallelization is also applicable to task-level parallelization. However, a key issue that is unique to task-level parallelization is dealing with global control flow. Sequential programs are written assuming a sequential flow of control, which lead to control dependences in addition to data dependences. For example, the decision to execute a region of code can depend on the runtime predicate values of prior conditional branches. The need to consider both control dependences and data dependences has motivated the use of the program dependence graph [51] and its variant in past work on task-level parallelization.

## 4.4 Future Directions

The future directions in compiler technologies for shared-memory parallelism are largely motivated by two important research problems:

1. Exploiting shared-memory parallelism for loops that have traditionally not been amenable to automatic parallelization.

This problem has been recently growing in importance due to the increased availability of current computer systems with SMP configurations; this trend will get even stronger when future computer systems are built out of single-chip SMPs. It is thus imperative to make SMP parallelization as widely applicable in the future as current optimizations (such as instruction scheduling and register allocation) that are necessary for exploiting the performance of today's high-end uniprocessors.

2. Exploiting shared-memory parallelism in new programming languages such as Java [16].

   Past work on shared-memory parallelization has largely been targeted to scientific programs written in procedural languages such as Fortran and C. However, object-oriented programming languages such as Java and C++ are now gaining in popularity for all programming tasks, including scientific and engineering applications. Therefore, it is essential for future parallelizing compiler technologies to target these programs as well.

One trend in expanding the set of loops that are amenable to automatic parallelization is to pursue more aggressive *storage duplication* transformations, such as array privatization [9] and construction of Array SSA form [102]. These transformations enable more parallelization by removing storage-related dependences, that is, anti and output dependences [177]. However, care is required when performing these data transformations because the costs of duplicating storage may sometimes outweigh the benefit of parallelization.

Another trend in enabling more loops to be executed in parallel is to perform *run-time parallelization.* A simple instance of run-time parallelization is when the compiler generates multi-version code for loops that it is unable to parallelize at compile-time. A conditional test is evaluated at run-time to choose, e.g., between the parallel version and the sequential version of the loop. This test includes all the conditions that need verification at run-time to ensure that the parallel version will be correct if selected. The *inspector-executor* framework [150] is a more general instance of run-time parallelization, suitable for sparse matrix computations. More recently, there are research efforts under way to explore the use of *dynamic compilation* in supporting even more general forms of run-time parallelization.

A logical extension to run-time parallelization is *speculative parallelization.* In run-time parallelization, the decision-making for loop parallelization is performed first, and then followed by parallel execution of the loop (if the decision made was to execute the loop in parallel). In speculative parallelization, iterations of the loop are executed in parallel without prior checking; instead, run-time checking is performed during the parallel execution to determine if the parallelization was indeed legal or not. If the parallelization is found to be legal, then the *speculation* is correct and no further action is needed. If the parallelization is found to be illegal, then some "fix-up" is necessary to ensure that any updates and side effects performed by iterations that were executed out of order are not "globally committed". This idea is analogous to optimistic concurrency control in transaction processing. Examples of systems that perform speculative loop parallelization can be found in [120, 125, 144].

The second important future research problem outlined above is that of exploiting shared-memory parallelism in new programming languages such as Java. Though current automatic parallelization techniques can also be applied to subsets of Java, a more pressing problem is dealing with the explicit parallelism of threads and locks present in most Java programs due to threads and locks. A Java programmer can explicitly create threads; a common example is to create a thread that performs some form of "asynchronous wait". More importantly, a Java programmer can easily introduce locks into a program by using the *synchronized* keyword for a method or for a block of statements. Even though the parallelism in Java is explicit, new compiler technologies are necessary to enable efficient parallel execution on shared-memory multiprocessors. The challenge for optimizing compilers in this area is to take a Java program with explicit threads and locks as input, and generate as output an equivalent, but more efficient, parallel program that uses fewer thread and locks. One promising direction is to use *escape analysis* [27] to identify unnecessary synchronization operations that can be eliminated. Another direction is optimize the structure of user-defined (virtual) threads in the Java programs so as to preserve the semantics of the program while using use fewer (physical) threads. This research direction is especially challenging because it requires the ability to analyze and optimize explicitly parallel programs as illustrated in [109, 155, 162].

# 5 Distributed Memory Parallelism

Shared memory systems typically do not scale well as the contention for memory increases with the number of processors. The distinctive characteristic of distributed memory parallel systems is their ability to scale. Therefore such systems are typically constructed around a scalable topology such as a 3D torus, mesh, or a hypercube. Each processor has its own local memory and the cost of interprocessor communication is higher than that of accessing local memories. The desired model supported on such machines is the popular shared memory model. Thus, the task of generating a parallel program whose threads execute in distributed address spaces falls on the compiler.

Although the potential for providing scalable shared-memory performance is possible, one main obstacle to realizing the full potential of distributed memory systems has been the complexity of programming such systems. The complexity arises from the requirement that not only must the parallelism be detected and managed effectively, the compiler must also simultaneously manage distributed address spaces. The latter task entails distribution of data across the local memories and orchestrating communication among the processors to execute an application. Effectively managing the communication is critical to obtaining high performance because the cost of interprocessor communication is higher than that of accessing data in local memories.

Spatial parallelism in applications such as fluid flow, weather modeling and image processing is perfectly decomposable and easy to map on to distributed memory systems. Speedups increase linearly with the number of processors for such applications because data accesses are mostly limited to local memory. However, due to the above complexities, for many applications the only practical success has been achieved through hand parallelization of codes with communication managed through message passing libraries. In spite of a tremendous amount of research in semiautomatic parallelization, applicability of many of the compiler techniques remains rather limited [130]. Therefore, in order to simplify the task of the compiler, there has also been significant use of languages in which the user provides the data distribution. This approach has worked quite well for regular numeric applications. While the success of automatic parallelization for distributed memory has been limited, nevertheless it appears that the compiler can play an important role in addressing some fundamental problems with regard to these machines.

The remainder of the section is organized as follows. We first discuss the language design aspects of compiling for distributed memory systems. It is important to discuss these approaches because, from a historical perspective, most of the compilation techniques have evolved around data parallel languages. We then look at the crucial issues of data and code partitioning. Many important techniques are examined along with open research directions. Next the issues relating to communication optimization are discussed. Some code generation problems that arise specifically in the context of distributed memory systems are discussed. Finally we present our perspective on potential general problems to be solved and their implication on effective use of these systems.

## 5.1 Language Support

In an attempt to reduce the burden of code and data distribution on the compiler, new constructs have been incorporated into variety of languages. We examine language support introduced to create imperative data parallel languages, high performance object-oriented languages, and functional languages.

Data parallel languages. The data parallel model derives its parallelism from the observation that updates to individual elements of large arrays are often independent of each other. This observation alone exposes significant degrees of parallelism in many problems. The data parallel languages require the programmer to specify data decomposition and allocate computation accordingly. Data parallel programming is one of the most significant approaches proposed to solve the tough problem of data distribution [80]. Languages of this style typically provide *aggregate* array operations to enable data level parallelism to be expressed and additional constructs for specifying data distributions. A number of research projects and commercial compilers have employed data parallel programming [82, 110, 59, 77, 149, 101] . The job of the compiler and run-time system for a data parallel language is to efficiently map programs onto parallel hardware. Typically, the implementation creates a parallel Single-Program Multiple-Data (SPMD) parallel program using the *owner computes* rule which assigns the computation in an assignment statement to the processor that owns the variable on the left-hand side.

Although this approach has been quite successful in the areas such as dense linear algebra codes that can be nicely mapped, it is unclear what fraction of scalable applications are data parallel. In the 1996 SPDP Workshop on Compiler Optimizations for Scalable Parallel Computing [132] it was suggested that 15 to 20% of scalable parallel applications are data parallel. Whether the other applications can be converted in data parallel domain through program transformations remains an open question. Limited forms of data distribution constructs were initially supported. New distribution patterns are being considered by researchers to broaden the class of applications that can be handled. Since with increase in parallelism the data demands go up, attention is being paid to the need to scale I/O [80].

Object oriented languages. The object oriented solution for distributed memory systems is mainly centered around providing library support for parallel programming in distributed memory space. HPC++ is one such important approach [24, 94]. The mode of program execution is an explicit SPMD model where copies of the same program are run with different contexts on multiple processors [34]. This programming model is similar to that of Split-C [44] in that the distribution of data that must be shared between contexts and the synchronization of accesses to that data must be managed by the programmer. HPC++ differs from Split-C in that the computation on each context can also be multi-threaded and the synchronization mechanisms for thread groups extends to sets of thread groups running in multiple contexts. In addition, HPC++ provides a template library [91] to support synchronization, collective parallel operations such as reductions, and remote memory references. In order to efficiently exploit parallelism new analysis techniques must be developed and approaches to limit run time overhead must be considered.

Functional languages. The functional model of computation is one attempt at providing an implicitly parallel programming paradigm. Because of the lack of state and its functionality, it allows the compiler to extract all available parallelism, fine and coarse grain, regular and irregular, and generate a partial evaluation order of the program. Sisal [50] is one of the attempts to provide implicitly parallel programming model. One of the key aspects of this model is the compiler flexibility to manage the memory binding of values. It leads to some interesting optimizations that can update arrays 'in-place' avoiding the communication that is otherwise needed.

However, the severe limitation of this model is very high operation counts that are typical of functional programs and run time overheads. Also, lack of memory binding could be a blessing or a problem. For example, it is generally not possible for a compiler to perform optimizations such as data layouts since the language semantics is rather strict and a lot of analysis is needed to ascertain the safety of such optimizations. These issues are significant since most of communication results due to non-local data accesses.

## 5.2 Data and Code Distribution

Since the communication overhead has a major impact on the performance, the code and data partitioning decisions are of critical importance. We already discussed language based solutions to partitioning that largely leave this problem in the hands of the programmer. In this section we discuss automatic techniques for determining data distribution patterns to minimize the communication overhead. Since code and data partitionings cannot in general be carried out independently, a tradeoff between increasing parallelism and decreasing interprocessor communication must be performed. Most of the research we have described assumes that following the data distribution stage, the compiler generates SPMD code with appropriate message passing operations.

We discuss three static partitioning methods for creating low communication distributions: communication-free partitioning, alignment and distribution, and tiling. Later we also briefly discuss some dynamic partitioning techniques. While our discussion is mainly limited to the manner in which automatic distribution techniques minimize communication overhead, it is worth mentioning that other important transformations for limiting the need for communication also play an important role. An example of such a transformation is privatization which leads to the localization of memory references [172].

Communication-free partitioning. This class of methods is based on the assumption that interprocessor communication always offsets the benefit of parallelization even if the parallelization exploits large amounts of parallelism. Thus, the goal of such techniques is to construct partitionings that completely eliminate interprocessor communication. Analysis of the relationship between iteration and data spaces of loops is performed to find communication-free partitionings. The key results based upon this approach are summarized below. In each of these cases the methods assume affine array references. Also these methods assume that there are no loop carried dependences since the iteration space then can be partitioned and traversed in any order. The methods are classified as loop-level or statement-level according to the level at which the partitioning is performed.

1. Loop-level partitioning: Huang and Sadayappan [85] derived the necessary and sufficient conditions for the feasibility of communication-free iteration space and data space partitioning. Ramanujam and Sadayappan [139] developed a method for communication-free hyperplane partitioning of data spaces. The computation partitioning is then achieved according to the *owner computes* rule. Chen and Sheu [38] proposed two communication-free data allocation strategies: one allows data duplication where as the other does not allow any data duplication.

2. Statement-Level partitioning: Shih, Sheu, and Huang [159] derived the necessary and sufficient conditions for the existence of communication-free statement-iteration space and data space partitionings. Lim and Lam's [115] proposed affine processor mappings for communication-free statement-iteration space partitioning and maximizing parallelism.

The main drawback of communication-free partitioning is that it is applicable to a narrow class of loops. A more general strategy, such as the ones proposed in [169, 13], must perform tradeoff between parallelism and communication.

**Alignment and distribution.** In order to expand the applicability of data and loop partitioning to a larger class of loops, another approach has been developed which consists of the two distinct phases of *alignment* and *distribution*. The alignment phase maps data and computations to a set of virtual processors organized as a Cartesian grid of some dimension. The distribution phase maps the virtual processors into the physical processors. The reason for separating alignment from distribution is that the alignment is dependent on program characteristics and independent of underlying architecture; whereas distribution involves balancing computation and communication for which their relative costs depend on the underlying architectural model.

A number of heuristic solutions have been suggested for the alignment problem. One approach trades parallelism for codes that allow decoupling of the issues of parallelism and communication by relaxing an appropriate constraint of the problem. Pingali et al. [22], have proposed a systematic three step procedure for determining alignment. First the constraints on data and computation placement are determined. Next constraints that will be left unsatisfied are selected. This step relaxes an overconstrained system so that non-trivial solution involving parallelism can be found. In the final step the remaining system of constraints is solved to determine data and computation placement. Other alignment heuristics based upon the above approach have also been developed [13, 36].

The relaxation step upon which the above techniques rely may not be feasible for many important problems such as image processing. In order to achieve communication-free parallel execution for such applications, one must replicate the data at all processors. Thus, in such situations, one must resort to a different partitioning solution based on relative costs of communication and computation. A new approach has been proposed to partition the iteration space by determining directions that maximally cover the communication and minimally sacrifice parallelism [129, 131]. The resulting partition of the iteration space is non-affine; and thus, results in approaches based on hyperplanes or systems of linear inequalities to be nonapplicable. There is no particular reason to confine solutions to affine partitions of iteration spaces as long as the iteration partitions can be spanned without much run time overhead. For limited cases, it is shown that the underlying non-affine iteration space partitions can be spanned by generating integer lattice vectors without much complexity [168].

While the above techniques primarily focus on the volume of communication, it is also important to orchestrate a structure of communication that utilizes *aggregation*. M. Dion, C. Randriamaro and Y. Robert [47] propose an approach in which the communication is minimized by taking advantage of macro communications such as broadcasts, gathers, scatters and reductions, or by decomposing the general affine communication into simpler components. Thus it addresses *communication aggregation* issues effectively. A linear algebraic framework is developed and a heuristic is proposed which first minimizes the non-local communication and then optimizes the residual affine communication using either macro communication or decomposition.

**Tiling.** This transformation essentially partitions the *iteration space* of a *loop nest* into tile like blocks. It has been the focus of loop partitioning work in compilers for increasing cache locality through data reuse [178]. Tiling is also very useful for reducing and *aggregating* communication in distributed memory systems. Agarwal et al. [2], address the problem of deriving the optimal tiling parameters for minimal communication in loops with general affine index expressions. They assume tiles to be atomic and only consider DOALL loops factoring out issues of dependencies and synchronizations that arise from the ordering of iterations of a loop. Ramanujam and Sadayappan [139] address the problem of compiling perfectly nested loops for multicomputers using tiling. However, they consider tiles to be atomic and therefore do not allow synchronization during the execution of a tile. They present an approach to partitioning the iteration and data space so that communication is minimized. D'Hollander [84] presents a partitioning algorithm to identify independent iterations in loops with constant dependence vectors. He also proposes a scheduling scheme for the dependent iterations.

Demands of *communication aggregation* and inter-tile latency can be conflicting. In order to combine communication, most of the above approaches consider tile execution to be atomic but in order to minimize latency, it may be better to schedule non-atomic execution within tiles. Rastello et al. [142], consider the non-atomic execution of tiles to minimize inter-tile latencies. They apply their technique to loops with dependence distances that are unknown at compile time. They also derive optimal tile sizes given the constraints of dependencies and inter-tile latencies to minimize loop completion time.

The main focus of tiling work in distributed memory systems has been to minimize communication latency. Tiling, is, however, a *control centric* transformation. Pingali et al. [103], in their recent paper, argue about the usefulness of *data centric* view in light of locality and propose a concept of data shackles to orient the loop partitioning around data rather than control. A similar argument could hold for tiling for distributed memory systems. In fact the concept of data shackles could be very useful to define *related references* that can be automatically discovered and localized by the compiler. Whether this would lead to more efficient loop partitioning methods remains to be seen.

Dynamic data partitioning. Static data alignment and distribution typically solves the problem of communication on a *loop nest* by *loop nest* basis but rarely in an intraprocedural scope. Most of the inter-*loop nest* level and interprocedural boundaries require dynamic data redistribution. Hudak and Abraham have proposed a method for selecting redistribution points based on locating significant control flow changes in a program [86]. Chapman, Fahringer, and Zima describe the design of a distribution tool that makes use of performance prediction methods when possible but also uses empirical performance data through a pattern matching process [35]. Anderson and Lam [13] approach the dynamic partitioning problem using a heuristic which combines *loop nest*s (with potentially different distributions) in such a way that the largest potential communication costs are eliminated first while still maintaining sufficient parallelism. Bixby, Kennedy and Kremer [26], as well as Garcia, Ayguadé, and Labarta [58], have formulated the dynamic data partitioning problem in the form of a 0-1 integer programming problem by selecting a number of candidate distributions for each loop nest and constructing constraints from the data relations. Also, Sheffler, Schreiber, Gilbert and Pugh [158] have applied graph contraction methods to the dynamic alignment problem to reduce the size of the problem space that must be examined. The work by Hall, Hiranandani, Kennedy, and Tseng [81] defined the term *reaching decompositions* for array sections which reach a function call site. D. J. Palermo and P. Banerjee [127, 128] develop techniques that can be used to automatically determine which data partitions are most beneficial over specific sections of the program by accounting for redistribution overhead. Initially a static data distribution is determined for the entire program and a communication graph is constructed based upon the static distribution. From this graph, a split point is determined by computing a maximal cut that splits the program into two parts for which better individual distributions may exist. The redistribution costs are then estimated for these parts to determine whether redistribution should be performed between the two parts.

## 5.3 Communication Optimizations

Since the message communication costs can be significant, and communication startup overheads tend to be very high on most distributed memory machines [160], compiler must attempt to reduce both the size and number of messages generated. Furthermore an attempt should be made to maximize the overlapping of computation and communication. A variety of communication optimizations, including the ones listed below, have been developed.

1. Message vectorization [82]: This optimization involves grouping together communication of different elements of the same array together and sending them in one large message. Typically this optimization is achieved by replacing the sending of shorter messages with a loop by larger messages in an outer loop. Message vectorization leads to reduction in message start-up costs and in hiding communication latency.

2. Collective communication [64, 110]: Collective communication involves replacing individual point-to-point messages by a *broad-cast* or *multi-cast* in order to achieve a more efficient communication mechanisms. This is especially beneficial for loops that transpose arrays between different alignments and for reductions.

3. Elimination of redundant communication [10]: This post-pass optimization eliminates redundant communication arising from alignments and the use of the *owner computes* rule. Array sections analysis is required to determine unmodified sections of arrays that need not be repeatedly communicated.

4. Overlapping of communication with computation [82, 100]: This technique involves hoisting communication at an appropriate point so that communication can be maximally overlapped with computation. By separating the send and receive points, blocking at the receives is minimized.

Frameworks based upon global array data-flow analysis that allow formulation of the above communication optimization problems have been proposed [33, 10, 62]. More sophisticated frameworks that handle complex situations have also been proposed. The *Give-N-Take* framework can generate communication in the presence of indirection arrays [76, 75]. This framework has also been extended to exploit information about array sections [99]. Agrawal's [3] framework works interprocedurally and also handles irregular communication.

While the above results are encouraging, none of the above techniques consider network properties in orchestrating communication. This is a crucial issue since communication-communication interaction can result in congestion. The SPMD programs generated by compilers typically operate in distinct computation and communication phases. Thus, the likelihood of congestion in such programs is even higher. Communication staggering optimizations should be developed to avoid congestion. A problem with current approaches to the important optimization of overlapping computation and communication is that they rely on communication latency information which cannot always be estimated statically. It may be better to rely on dynamic scheduling of communication guided by the compiler. Compiler can often determine the criticality of a message relative to other messages. For example, short, synchronizing messages are critical as their delay may delay all the processors. The compiler could prioritize the messages for more effective scheduling [164]. Finally more flexible communication models that exercise control over slack between producers and consumers should be developed [96].

## 5.4 Code Generation Issues

The final phase of compiling for distributed memory systems involves solving two main code generation problems: *communication generation* to satisfy references to non-local data by each processor and *address calculation* which maps references to global name space to local memory addresses.

Communication generation. In order to efficiently generate parallel SPMD code, the compiler computes the following quantities: array sections locally allocated to each processor, the set of iterations to be executed by each processor, the non-local data accessed from every other processor by each reference, the iterations that access non-local data, and the iterations that access local data only. The representation of above entities is an important issue and has been well researched. The following two representations have been proposed to represent the above entities during communication generation.

1. Symbolic. In this representation the above quantities are represented as sets of integer tuples or as mappings between integer tuples. An integer tuple may represent array indices, loop iterations, or processor indices. To compute the above sets and to generate code using them, the primary approach has been to focus on common special cases and precompute the iteration and communication sets symbolically for specific forms of data layouts and computation partitionings [23, 64, 66, 83, 110, 163]. Li and Chen describe algorithms to classify communication caused by more general reference patterns and generate code to realize these patterns efficiently on a target machine [110]. Typically the compilers focus on providing specific optimizations aimed at cases that are considered to be the most common and most important. The principal benefits of such case-based strategies are that they are conceptually simple, they have predictable and fast running times, and they can provide excellent performance in cases where they apply.

2. Linear inequalities. In this approach, each code generation or communication optimization problem is described by a collection of linear inequalities representing integer sets or mappings [11, 20]. Fourier-Motzkin elimination is used to simplify the resulting inequalities, and to compute a range of values for individual index variables that together enumerate the integer points described by these inequalities. Code generation then amounts to generating loops to iterate over these index ranges. These representations can be used to perform certain optimizations such as *message aggregation* as well as other index set related techniques. Adve et al. [1], have developed an approach based on integer sets that exploits the speed of case based approaches as well as preserving generality of inequalities approaches. Consequently more aggressive optimizations can be undertaken.

Address generation. While one important issue of code generation is communication generation, a very important issue is to map global address space to local address space efficiently. Address calculation mechanisms that work for different types of data distributions, access patterns (in terms of strides) and subscript expressions must be developed. In addition, the address calculation must be very efficient. A number of researchers have thoroughly investigated this problem and their approaches differ in terms of domains they solve (i.e., their applicability) and the efficacy of their solutions.

1. Applicability of techniques. A number of researchers have developed techniques which work for certain combinations of data distributions and strides. Koelbel [104] derived techniques for compile-time address calculation for `BLOCK` and `CYCLIC` distributions with non-unit stride accesses containing a single loop index variable. MacDonald et al. [117], provided a simple solution for a restricted case where the block sizes and the number of processors are powers of two. Gupta et al. [66], derived the *virtual-block* and the *virtual-cyclic* schemes. Banerjee et al. [20], discuss code generation for regular and irregular applications. In extending the applicability of these methods to broader range of distributions and access patterns, a run time mechanism for address calculations are needed. Chatterjee et al. [37], derived a run time technique that identifies a repeating access pattern, which is characterized as a finite-state machine. This $O(k \log k + \log \min(s, pk))$ algorithm involves a solution of $k$ linear Diophantine equations.

2. Efficiency of address calculations. Block-cyclic distribution and regular accesses are very common cases in most HPF programs and some researchers have focused on developing fast address calculation solutions for them. These techniques rely mostly on integer lattices, linear algebra or use of diophantine equations.

The solutions based on lattices tend to have closed form and are thus more efficient. Ancourt et al. [12], use a linear algebra framework to generate code for fully parallel loops in HPF. Midkiff [119] presented a technique that uses a linear algebra approach to enumerate local accesses on a processor; this technique is similar to the virtual block approach presented by Gupta et al. [66]. van Reeuwijk et al. [145], presented a technique which requires the solution of linear Diophantine equations. Kennedy, Nedeljkovic, and Sethi [98] derived an $O(k + \log \min(s, pk))$ algorithm for address generation. Wijshoff [174] describes access sequences for periodic skewing schemes using lattices and derived closed forms for the lattices. Finally, Ramanujam et al. [167, 168], present closed form expressions for basis vectors for several cases. Using the closed form expressions for the basis vectors, they derived a non-unimodular linear transformation. Wang et al. [173], discuss experiments with several address generation solutions and conclude that the strategy described by Ramanujam et al. [167], is the best strategy overall in terms of speed of address calculation.

So far most of the approaches have relied upon data distribution; and thus, address calculation problem has been mainly attempted for array local address calculation. As discussed earlier [131, 129], it is useful to generate *non-affine* iteration space partitions in some cases. However, the efficiency is highly dependent on how the iteration spaces can be spanned using lattice vectors. Code generation techniques could be possibly developed to address this issue and the efficiency of code generation will determine if non-affine iteration space partitioning approaches are viable or not.

### 5.5 Future Directions

The two important deficiencies that limit the use of distributed memory systems are: their applicability to limited application domains and less than satisfactory overall performance. By supporting a general run-time communication model, it may be possible to handle application domains beyond those requiring regular communication. Moreover to obtain superior performance, such a model may help in handling communication more effectively. The model should enable effective latency hiding by giving sufficient flexibility to the compiler and deferring the hard decisions to run time. It should still allow static application of other communication optimizations. As mentioned earlier, by allowing the compiler to communicate the criticality of messages to the run time system, the problem of statically estimating cost of communication may be avoided.

Probably the best reason to use distributed memory systems is the potential to benefit from their scalability even though the application domains and performance might be somewhat limited. Thus, research to develop scalable code generation techniques should not be ignored.

## 6 Concluding Remarks

In this paper we reviewed the developments in compilation technology for parallel systems and identified future directions for research. Since processors being used today and those being designed for future exploit increasing amounts of ILP, the research into compilation techniques is of great practical significance. Small scale shared-memory machines are common place and their use is expected to grow rapidly. Therefore the research into effective techniques for parallelization and memory hierarchy management is extremely important. The availability of powerful single processor machines connected by high speed networks in also common in today's computing environment. Thus, research into compilation techniques for distributed-memory machines is also highly relevant. The developments in compilation technology will continue to significantly impact the computing environments of the future.

## References


[1] V. Adve and J. Mellor-Crummey, "Using Integer Sets for Data-Parallel Analysis and Optimization," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 186-198, Montreal, Canada, 1998.

[2] A. Agarwal, D. Kranz, and V. Natrajan, "Automatic Partitioning of Parallel Loops and Data Arrays for Distributed Shared-Memory Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, 6(9):943–962, Sept. 1995.

[3] G. Agrawal, "Interprocedural Partial Redundancy Elimination with Application to Distributed Memory Compilation," *IEEE Transactions on Parallel and Distributed Systems*, Volume 9, Number 7, July 1998, pp. 609–625.

[4] A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1988.

[5] A. Aiken and A. Nicolau, "A Development Environment for Horizontal Microcode," *IEEE Transactions on Software Engineering*, Vol. 14, No. 5, pages 584-594, May 1988.

[6] R. Allen and K. Kennedy, "Automatic Translation of FORTRAN Programs to Vector Form," *ACM Transactions on Programming Languages and Systems*, pages 491-592, Vol. 9, No. 4, October 1987.

[7] R. Allen and K. Kennedy, "Vector Register Allocation," Technical Report TR86-45, Rice University, Houston, TX, December 1986.

[8] E.R. Altman, "Optimal Software Pipelining with Functional Unit and Register Constraints," Ph.D. Thesis, McGill University, Montreal, Quebec, 1995.

[9] S.P. Amarasinghe, "Parallelizing Compiler Techniques Based on Linear Inequalities," Computer Systems Laboratory, Stanford University, January 1997.

[10] S. P. Amarasinghe and M. S. Lam, "Communication Optimization and Code Generation for Distributed Memory Machines," *Proc. ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, New Mexico, June 1993.

[11] C. Ancourt and F. Irigoin, "Scanning Polyhedra with Do Loops," *Proc. of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[12] A. Ancourt, F. Coelho, F. Irigoin, and R. Keryell, "A Linear Algebra Framework for Static HPF Code Distribution," *Scientific Programming,* 6(1):3–28, Spring 1997.

[13] J. M. Anderson and M. S. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines," *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 112 – 125, June 1993.

[14] G. Araujo, S. Malik, and M. Lee, "Using Register-Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures," *Proc. of the* $33^{rd}$ *ACM/IEEE Design Automation Conference*, pages 591–596, June 1996.

[15] G. Araujo, A. Sudarsanam, and S. Malik, "Instruction Set Design and Optimization for Address Computation in DSP Architectures," *Proc. of the 9th International Symposium on System Synthesis*, pages 31–37, Nov. 1997.

[16] K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, 1996.

[17] D.I. August, W.W. Hwu, and S.A. Mahlke, "A Framework for Balancing Control Flow and Predication," *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 92-103, Research Triangle Park, North Carolina, December 1997.

[18] U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, Norwell, Massachusetts, 1988.

[19] U. Banerjee, *Loop Transformations for Restructuring Compilers: The Foundations*, Kluwer Academic Publishers, Boston, MA, 1993.

[20] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su, "The PARADIGM Compiler for Distributed-Memory Multicomputers," *IEEE Computer*, 28(10):37–47, Oct. 1995.

[21] D. Bartley, "Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes," *Software - Practice and Experience*, 22(2):101–110, Feb. 1992.

[22] D. Bau, I. Koduklula, V. Kotlyar, K. Pingali, and P. Stodghill, "Solving Alignment using Elementary Linear Algebra," *Proc. of the 7th Workshop on Languages and Compilers for Parallel Computing*, volume 892 of *Lecture Notes in Computer Science*, 46–60, Ithica, NY, 1994. Springer-Verlag, 1995.

[23] S. Benkner, B. Chapman, and H. Zima, "Vienna Fortran 90," *Proc. of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[24] P. Beckman and D. Gannon, "Tulip: A Portable Run-time System for Object-parallel Systems," *Proc. of the 10th International Parallel Processing Symposium*, April 1996.

[25] D.A. Berson, R. Gupta, and M.L. Soffa, "Resource Spackling: A Framework for Integrating Register Allocation in Local and Global Schedulers," *Proc. of IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques*, pages 135-146, 1994.

[26] R. Bixby, K. Kennedy, and U. Kremer, "Automatic Data Layout using 0-1 Integer Programming," *Proc. of the 1994 Int'l Conf. on Parallel Architectures and Compilation Techniques*, 111–122, Montréal, Canada, Aug. 1994.

[27] B. Blanchet, "Escape Analysis: Correctness, Proof, Implementation and Experimental Results," *Proc. of the 25th Annual ACM Symposium on Principles of Programming Languages*, pages 25-37, San Diego, CA, January 1998.

[28] W. Blume, R. Eigenmann, "Nonlinear and Symbolic Data Dependence Testing", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, No. 12, December 1998.

[29] R. Bodik, R. Gupta, and M.L. Soffa, "Complete Removal of Redundant Expressions," *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1-14, Montreal, Canada, June 1998.

[30] R. Bodik and R. Gupta, "Partial Dead Code Elimination using Slicing Transformations," *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 159-170,Las Vegas, Nevada, June 1997.

[31] M. Burke and R. Cytron, "Interprocedural Dependence Analysis and Parallelization," *Proc. of the SIGPLAN Symposium on Compiler Construction*, pages 162-175, July 1986.

[32] A. Caro, "Generating Multithreaded Code from Parallel Haskell for Symmetric Multiprocessors," Ph.D. Thesis, Massachussetts Institute of Technology, 1999.

[33] S. Chakrabarti, M. Gupta, and J.-D. Choi, "Global Communication Analysis and Optimization," *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996.

[34] K. M. Chandy and C. Kesselman, "CC++: A Declarative Concurrent Object-oriented Programming Notation," *Research Directions in Concurrent Object Oriented Programming*, MIT Press. 1993.

[35] B. Chapman, T. Fahringer, and H. Zima, "Automatic Support for Data Distribution on Distributed Memory Multiprocessor Systems," *Proc. of the 6th Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, 184–199, Portland, OR, Aug. 1993. Springer-Verlag, 1994.

[36] S. Chatterjee, J. Gilbert, and R. Schreiber, "The Alignment-Distribution Graph," *Languages and Compilers for Parallel Computing. Sixth International Workshop.*, number 768 in LNCS. Springer-Verlag, 1993.

[37] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng, "Generating Local Addresses and Communication Sets for Data Parallel Programs," *Journal of Parallel and Distributed Computing*, 26(1):72–84, 1995.

[38] T. S. Chen and J. P. Sheu, "Communication-free Data Allocation Techniques for Parallelizing Compilers on Multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, 5(9):924–938, September 1994.

[39] G. Chrysos and J. Emer, "Memory Dependence Prediction using Store Sets," *Proc. of the ACM/IEEE 25th International Symposium on Computer Architecture*, pages 142-154, Barcelona, Spain, July 1998.

[40] T.M. Conte, "Evolutionary Compilation to Long Instruction Superscalar Microarchitectures for Exploiting Parallelism at all Levels," *ASPLOS Wild and Crazy Idea Session*, 1998.

[41] K. Cooper and K. Kennedy, "Efficient Computation of Flow Insensitive Interprocedural Summary Information", *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, June 1984.

[42] K.D. Cooper, K. Kennedy and L. Torczon, "The Impact of Interprocedural Analysis and Optimization in the Rn Programming Environment," *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 4, pages 491-523, October 1986.

[43] K. Cooper and P. Schielke, "Non-Local Instruction Scheduling with Limited Code Growth," *Proc. of Languages, Compilers and Tools for Embedded Systems*, pages 193–207, June 1998.

[44] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. "Parallel Programming in Split-C," *Proc. of Supercomputing'93*, 1993.

[45] G. Dantzig and B. Eaves, "Fourier-Motzkin Elimination and its Dual", *Journal of Combinatorial Theory (A)*,Vol. 14, 1973.

[46] J. Davies, C. Huson, T. Macke, B. Leasure, and M. Wolfe, "The KAP/S-1: An Advanced Source-to-Source Vectorizer for the S-1 Mark IIa Supercomputer," *Proc. of the 1986 Int'l Conf. on Parallel Processing*, pages 833-835, St. Charles, Illinois, August 1986.

[47] M. Dion, C. Randriamaro and Y. Robert, "How to optimize residual communications?". *Special Issue of Journal of Parallel and Distributed Computing on 'Compilation Techniques for Distributed Memory Systems'*, **38**, Nov. 1996.

[48] C. Dulong, "The IA-64 Architecture at Work," *IEEE Computer*, pages 24-32, July 1998.

[49] K. Ebcioglu and E. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," *Proc. of the International Symposium on Computer Architecture*, pages 26-37, Denver, Colorado, June 1997.

[50] J. T. Feo, D. C. Cann and R. R. Oldehoeft, "A Report on Sisal Language Project", *Journal of Parallel and Distributed Computing*, Vol. 10, no. 4, October 199 0, pp. 349-366.

[51] J. Ferrante, K. Ottenstein, and J. Warren, "The Program Dependence Graph and its Use in Optimization," *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, pages 319-349, July 1987.

[52] J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, Vol. 30, No. 7, pages 478-490, July 1981.

[53] J.A. Fisher, "Walk-time Techniques: Catalyst for Architectural Change," *IEEE Computer*, 30(9):40–42, September 1997.

[54] S. Freudenberger, T. Gross and P. G. Lowney, "Avoidance and Suppression of Compensation Code in a Trace Scheduling Compiler," *ACM Transactions on Programming Languages and Systems*,16(4):1156–1214, July 1994.

[55] D.H. Friendly, S.J. Patel, and Y.N. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors," *Proc. of the 31st Annual ACM/IEEE Symposium on Microarchitecture*, pages 173-181, November 1998.

[56] C. Fu, M.D. Jennings, S.Y. Larin, and T.M. Conte, "Value Speculation Scheduling for High Performance Processors," *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 262-271, October 1998.

[57] K. Gallivan, W. Jalby, and U. Meier, "The Use of BLAS3 in Linear Algebra on a Parallel Processor with a Hierarchical Memory," Technical Report CSRD Rpt. No. 610, Center for Supercomputing Res. and Dev., U. of Illinois, October 1986.

[58] J. Garcia, E. Ayguadé, and J. Labarta, "A Novel Approach Towards Automatic Data Distribution," *Proc. of the Workshop on Automatic Data Layout and Performance Prediction*, Houston, TX, Apr. 1995.

[59] M. Gerndt and H. Zima, "SUPERB: Experiences and Future Research," *Proc. of the Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*, North-Holland, Amsterdam, The Netherlands, 1992.

[60] M. Girkar and C. Polychronopoulos, "The HTG: An intermediate Representation for Programs based on Control and Data Dependences", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 2, March 1992.

[61] G. Golf, K. Kennedy, and C. W. Tseng, "Practical Dependence Testing", *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[62] C. Gong, R. Gupta, and R. Melhem, "Compilation Techniques for Optimizing Communication in Distributed-memory Systems," *Proc. 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993.

[63] J.R. Goodman and W-C. Hsu, "Code Scheduling and Register Allocation in Large Basic Blocks," *Proc. of ACM Supercomputing Conf.*, pages 442-452, 1988.

[64] M. Gupta and P. Banerjee, "A Methodology for High-level Synthesis of Communication for Multicomputers," *Proc. of the ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[65] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Wang, W. Ching, and T. Ngo, "An HPF Compiler for the IBM SP2," *Proc. of Supercomputing '95*, San Diego, CA, December 1995.

[66] S. K. S. Gupta, S. D. Kaushik, C.-H. Huang, and P. Sadayappan, "On Compiling Array Expressions for Efficient Execution on Distributed-memory Machines," *Journal of Parallel and Distributed Computing*, 32(2):155-172, February 1996.

[67] R. Gupta and M.L. Soffa, "Region Scheduling: An Approach for Detecting and Redistributing Parallelism," *IEEE Transactions on Software Engineering*, Vol. 16, No. 4, pages 421-431, April 1990.

[68] R. Gupta, "Code Optimization as a Side Effect of Instruction Scheduling," *Proc. of the International Conference on High Performance Computing*, pages 370-377, Bangalore, India, December 1997.

[69] R. Gupta and R. Bodik, "Register Pressure Sensitive Redundancy Elimination," *Proc. of the International Conference on Compiler Construction, LNCS 1575, Springer Verlag*, pages 107-121, Amsterdam, Netherlands,

[70] R. Gupta, D. Berson, and J.Z. Fang, "Resource-Sensitive Profile-Directed Data Flow Analysis for Code Optimization," *Proc. of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 358-368, Research Triangle Park, North Carolina, December 1997.

[71] R. Gupta, D. Berson, and J.Z. Fang, "Path Profile Guided Partial Redundancy Elimination Using Speculation," *Proc. of the IEEE International Conference on Computer Languages*, pages 230-239, Chicago, Illinois, May 1998.

[72] R. Gupta, D. Berson, and J.Z. Fang, "Path Profile Guided Partial Dead Code Elimination Using Predication," *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, pages 102-115, San Francisco, California, November 1997.

[73] M. Haghighat and C. Polychronopoulos, "Symbolic Analysis for Parallelizing Compilers", *ACM Transactions on Programming Languages and Systems*, Vol. 18, No. 4, July 1996.

[74] R.E. Hank, W-M.W. Hwu and B.R. Rau, "Region-Based Compilation: An Introduction and Motivation," *Proc. of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, 1995.

[75] R. v. Hanxleden and K. Kennedy, "Give-n-take – a Balanced Code Placement Framework," *Proc. of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, Florida, June 1994.

[76] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz, "Compiler Analysis for Irregular Problems in Fortran D," *Proc. 5th Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[77] P. Hatcher and M. Quinn, *Data-parallel Programming on MIMD Computers*, The MIT Press, Cambridge, MA, 1991.

[78] P. Havlak and K. Kennedy, "An Implementation of Interprocedural Bounded Regular Section Analysis," *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[79] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1990.

[80] High Performance Fortran Forum, High Performance Fortran language specification, version 2.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, January 1997.

[81] M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng, "Interprocedural Compilation of Fortran D for MIMD Distributed-Memory Machines," *Proc. of Supercomputing '92*, 522–534, Minneapolis, MN, Nov. 1992.

[82] S. Hiranandani, K. Kennedy, and C. W. Tseng, "Compiling Fortran D for MIMD Distributed-Memory Machines," *Communications of the ACM*, 35(8):66–80, August 1992.

[83] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Preliminary Experiences with the Fortran D Compiler," *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[84] E. D'Hollander, "Partitioning and Labeling of Loops by Unimodular Transformations," *IEEE Transactions on Parallel and Distributed Systems*, 3(4):465–476, July 1992.

[85] C.-H. Huang and P. Sadayappan, "Communication-free Hyperplane Partitioning of Nested Loops," *Journal of Parallel and Distributed Computing*, 19:90–102, 1993.

[86] D. E. Hudak and S. G. Abraham, *Compiling Parallel Loops for High Performance Computers – Partitioning, Data Assignment and Remapping*, Kluwer Academic Pub., Boston, MA, 1993.

[87] W-m. Hwu, "Technology Outlook: Introduction to Predicated Execution," *IEEE Computer*, Vol. 31, No. 1, pages 49-50, January 1998.

[88] W-M. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J. Warter, R.A. Bringmann, R.G. Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, J.G. Holm, and D.M. Lavery, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *Journal of Supercomputing*, A:229-248, May 1993.

[89] IBM, "Engineering and Scientific Subroutine Library (ESSL), Guide and Reference," Document SC23-0526-01, 1994.

[90] F. Irigoin and R. Triolet, "Supernode Partitioning," *Proc. of the Fifteenth ACM Symposium on Principles of Programming Languages*, 1988.

[91] Y. Ishikawa, "Multiple Threads Template Library," Technical Report TR-96-012, Real World Computing Partnership, September 1996.

[92] S. Jain, "Circular Scheduling: A New technique to Perform Software Pipelining," *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 219-228, Toronto, Canada, June 1991.

[93] V. Jain and S. Pande, 'Code Motion for Generating Compact Code on Embedded DSPs', *1998 Workshop on Compiler and Architecture Support for Embedded Systems*, Washington, D.C, Dec. 4–6 '98. Available under publications link at http://www.ececs.uc.edu/ compiler

[94] E. Johnson and D. Gannon," "HPC++: Experiments with the Parallel Standard Template Library," Technical Report TR-96-51, Indiana University, Department of Computer Science, December 1996.

[95] R. Johnson and M. Schlansker, "Analysis Techniques for Predicated Code," *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 100-113, December 1996.

[96] M. Kandemir, N. Shenoy, P. Banerjee, J. Ramanujam and A.Choudhary, "Minimizing Data and Synchronization Costs in One-Way Communication", *1998 International Conference on Parallel Processing*, pp. 180–188.

[97] K. Kennedy and K. S. McKinley, "Optimizing for Parallelism and Data Locality," *Proc. of the ACM 1992 International Conference on Supercomputing*, July 1992.

[98] K. Kennedy, N. Nedeljkovic, and A. Sethi, "A Linear-Time Algorithm for Computing the Memory Access Sequence in Data-Parallel Programs," *Proc. of Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, pages 102–111, July 1995.

[99] K. Kennedy and N. Nedeljkovic, "Combining Dependence and Data-flow Analyses to Optimize Communication," *Proc. 9th International Parallel Processing Symposium*, Santa Barbara, CA, April 1995.

[100] K. Kennedy and A. Sethi, "Resource-based Communication Placement Analysis," *Proc. Ninth Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996.

[101] K. Knobe, J. Lukas, and M. Weiss, "Optimization Techniques for SIMD Fortran Compilers," *Concurrency: Practice and Experience*, 5(7):527–552, October 1993.

[102] K. Knobe and V. Sarkar, "Array SSA form and its use in Parallelization," *Proc. of the Twenty-fifth ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1998.

[103] I. Kodukula, N. Ahmed and K. Pingali, "Data Centric Multi-level Blocking," *Proc. ofthe SIGPLAN ACM Conference on Programming Language Design and Implementation*, 1997.

[104] C. Koelbel, "Compile-time Generation of Communication for Scientific Programs," *Proc. of Supercomputing '91*, Albuquerque, NM, pages 101–110, November 1991.

[105] X. Kong, D. Klappholz, and K. Psarris, "The I-Test: An Improved Dependence Test for Automatic Parallelization and Vectorization", *IEEE Transactions on Parallel and Distributed Systems*, Special Issue on Parallel Languages and Compilers, Vol. 2, No. 3, July 1991.

[106] D. Knuth, *The Art of Computer Programming*, Vol. 2, Seminumerical Algorithms, Addison-Wesley, 1981.

[107] Kuck and Associates, Inc., "KAP for IBM Fortran, User's Guide Version 3.3," Document #9603001, Champaign, IL, 1996.

[108] L. Lamport, "The Parallel Execution of DO Loops," *Communications of the ACM*, pages 83-93, Vol. 17, No. 2, February 1974.

[109] J. Lee, S.P. Midkiff, and D.A. Padua, "Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs," *Proc. of the 10th International Workshop on Languages and Compilers for Parallel Computing*, LNCS Springer-Verlag, Minneapolis, MN, August 1997.

[110] J. Li and M. Chen, "Compiling Communication-Efficient Programs for Massively Parallel Machines," *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[111] Z. Li and P. Yew, "Efficient Interprocedural Analysis for Program Parallelization and Restructuring", *Proceedings of the ACM/SIGPLAN Symposium on Parallel Programming*, New Haven, CT, July 1998.

[112] Z. Li, P. Yew, and C. Zhu, "An Efficient Data Dependence Analysis for Parallelizing Compilers", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 1, January 1990.

[113] S. Liao et al., "Storage Assignment to Decrease Code Size," *ACM Transactions on Programming Languages and Systems*, Vol. 18, No. 3, pages 235–253, May 1996.

[114] S. Liao et al., "Instruction Selection Using Binate Covering for Code Size Optimization," *Proc. of the 1995 International Conference on Computer-Aided Design*, 1995.

[115] A. W. Lim and M. S. Lam, "Communication-free Parallelization via Affine Transformations," *Proc. of the $7^{th}$ Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[116] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen, "Value Locality and Load Value Prediction," *Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138-149, Cambridge, Massachusetts, 1996.

[117] T. MacDonald, D. Pase, and A. Meltzer, "Addressing in Cray Research's MPP Fortran," *Proceedings of the 3rd Workshop on Compilers for Parallel Computers*, Vienna, Austria, pages 161–172, July 1992.

[118] K.S. McKinley, S. Carr, and C-W. Tseng, "Improving Data Locality with Loop Transformations," *ACM Transactions on Programming Languages and Systems*, Vol. 18, pages 423-453, July 1996.

[119] S. Midkiff, "Local Iteration Set Computation for Block-cyclic Distributions," *Proc. International Conference on Parallel Processing*. Vol. II, pages 77–84, August 1995.

[120] A.I. Moshovos, S.E. Breach, T.N. Vijaykumar, and G.S. Sohi, "Dynamic Speculation and Synchronization of Data Dependences," *Proc. of the 24th International Symposium on Computer Architecture*, 1997.

[121] R. Nair and M.E. Hopkins, "Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups," *Proc. of the International Symposium on Computer Architecture*, pages 13-25, Denver, Colorado, June 1997.

[122] T. Nakra, R. Gupta, and M.L. Soffa, "Value Prediction in VLIW Machines," *Proc. of the ACM/IEEE 26th International Symposium on Computer Architecture*, Atlanta, Georgia, May 1999.

[123] C. Norris and L.L. Pollock, "A Scheduler-Sensitive Global Register Allocator," *Proceedings of Supercomputing'93*, pages 804-813, Portland, Oregon, 1993.

[124] S. Onder and R. Gupta, "Superscalar Execution with Direct Data Forwarding," *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, pages 130-135, Paris, France, October 1998.

[125] J. Oplinger, D. Heine, S-W. Liao, B.A. Nayfeh, M.S. Lam, and K. Olukotun, "Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor," Stanford University Computer Systems Lab, Technical Report CSL-TR-97-715, February 1997.

[126] D. A. Padua and M. J. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Communications of the ACM*, 29:1184–1201, December 1986.

[127] D. J. Palermo, "Compiler Techniques for Optimizing Communication and Data Distribution for Distributed-Memory Multicomputers," Ph.D. thesis, Dept. of Electrical and Computer Eng., Univ. of llinois, Urbana, IL, June 1996.

[128] D. J. Palermo, E. W. Hodges IV, and P. Banerjee, "Dynamic Data Partitioning for Distributed-Memory Multicomputers," *Journal of Parallel and Distributed Computing*, 38(2):158–175, Nov. 1996.

[129] S. Pande and T. Bali, "A Computation+Communication Load Balanced Loop Partitioning Method for Distributed Memory Systems", *Journal of Parallel and Distributed Computing* (to appear).

[130] S. Pande and D. P. Agrawal, "Compilation Techniques for Distributed Memory Systems : Guest Editorial Introduction", *Special Issue of Journal of Parallel and Distributed Computing on 'Compilation Techniques for Distributed Memory Systems'*, **38**, pages 107-113, November 1996.

[131] S.Pande, "A Compile Time Partitioning Method for DOALL Loops on Distributed Memory Systems", *International Conference on Parallel Processing*, IEEE Computer Society Press. Vol. III, pages 35-44, 1996.

[132] S. Pande and J. Ramanujam and Y. Robert, "Workshop on Challenges Compiling for Scalable Parallel Systems", 8th IEEE Symposium on Parallel and Distributed Systems, October 1996.

[133] K. Pingali, M. Beck, R. Johnson, M. Moudgill and P. Stodghill, "Dependence Flow Graphs: An Algebraic Approach to Program Dependences", *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1991.

[134] K. Psarris, "The Banerjee-Wolfe and GCD Tests on Exact Data Dependence Information", *Journal of Parallel and Distributed Computing*, Vol. 32, No. 2, February 1996.

[135] K. Psarris, D. Klappholz, and X. Kong, "On the Accuracy of the Banerjee Test", *Journal of Parallel and Distributed Computing*, Special Issue on Shared Memory Multiprocessors, Vol. 12, No. 2, June 1991.

[136] K. Psarris and S. Pande, "An Empirical Study of the I Test for Exact Data Dependence", *Proceedings of the 1994 International Conference on Parallel Processing*, St. Charles, IL, August 1994.

[137] K. Psarris, X. Kong, and D. Klappholz, "The Direction Vector I Test", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 11, November 1993.

[138] W. Pugh, "A Practical Algorithm for Exact Array Dependence Analysis", *Communications of the ACM*, Vol. 35, No. 8, August 1992.

[139] J. Ramanujam and P. Sadayappan, "Compile-time Techniques for Data Distribution in Distributed Memory Machines," *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, October 1991.

[140] J. Ramanujam and P. Sadayappan, "Tiling Multidimensional Iteration Spaces for Multicomputers," *Journal of Parallel and Distributed Computing*, 16:108–120, 1992.

[141] J. Ramanujam, S. Dutta, and A. Venkatachar, "Code Generation for Complex Subscripts in Data-parallel Programs," *Proc. 10th Workshop on Languages and Compilers for Parallel Computing*, Minneapolis, MN, Springer-Verlag, 1997.

[142] F. Rastello, A. Rao and S. Pande, "Optimal Task Scheduling to Minimize Inter-Tile Latencies," *International Conference on Parallel Processing*, pages 172-179, 1998.

[143] B.R. Rau and C.D. Glaser, "Some Scheduling Techniques and an easily Schedulable Horizontal Architecture for High Performance Scientific Computing," *Proc. of the 14th Annual Microprogramming Workshop*, pages 183-198, Chatham, Mass., October 1981.

[144] L. Rauchwerger and D. Padua, "The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization," *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1995 June.

[145] C. van Reeuwijk, H. Sips, W. Denissen, and E. Paalvast, "An Implementation Framework for HPF Distributed Arrays on Message-Passing Parallel Computer Systems," *IEEE Transactions on Parallel and Distributed Systems*, 7(9):897–914, September 1996.

[146] Pacific-Sierra Research Corporation, "VAST-2 for XL Fortran, User's Guide, Edition 1.2," Document Number VA061, Santa Monica, CA, 1994.

[147] A. Rao and S. Pande, "Storage Assignment Optimizations to Generate Compact and Efficient Code on Embedded DSPs," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, 1999.

[148] J. Ruttenberg, G.R. Gao, A. Stoutchinin, and W. Lichtenstein, "Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler," *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1-11, Philadelphia, Pennsylvania, May 1996.

[149] G. Sabot, "Optimizing CM Fortran Compiler for Connection Machine Computers," *Journal of Parallel and Distributed Computing*, 23(1):224–238, November 1994.

[150] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman, "Run-time Scheduling and Execution of Loops on Message Passing Machines," *Journal of Parallel and Distributed Computing*, Vol. 8, No. 4, April 1990.

[151] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989.

[152] V. Sarkar, "Automatic Partitioning of a Program Dependence Graph into Parallel Tasks," *IBM Journal of Research and Development*, Vol. 35, No. 5/6, 1991.

[153] V. Sarkar and R. Thekkath, "A General Framework for Iteration-Reordering Loop Transformations," *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 175-187, San Francisco, California, June 1992.

[154] V. Sarkar, "Automatic Selection of High Order Transformations in the IBM XL Fortran Compilers," *IBM Journal of Research and Development*, Vol. 41, No. 3, May 1997.

[155] V. Sarkar, "Analysis and Optimization of Explicitly Parallel Programs using the Parallel Program Graph Representation," *Proc. of the 10th International Workshop on Languages and Compilers for Parallel Computing*, LNCS Springer-Verlag, Minneapolis, MN, August 1997.

[156] R.G. Scarborough and H.G. Kolsky, "A vectorizing Fortran compiler," *IBM Journal of Research and Development*, Vol. 30, No. 2, pages 163-171, March 1986.

[157] M. Schlansker and V. Kathail, "Critical Path Reduction for Scalar Programs," *28th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 1995.

[158] T. J. Sheffler, R. Schreiber, J. R. Gilbert, and W. Pugh, "Efficient Distribution Analysis via Graph Contraction," *Proc. of the 8th Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science 1033*, pages 377–391, Columbus, OH, Aug. 1995. Springer-Verlag, 1996.

[159] K.-P. Shih, J.-P. Sheu, and C.-H. Huang, "Statement-level Communication-free Partitioning Techniques for Parallelizing Compilers," *Proc. of the $9^{th}$ Workshop on Languages and Compilers for Parallel Computing*, August 1996.

[160] M. Snir et al., "The Communication Software and Parallel Environment of the IBM SP2," *IBM Systems Journal*, 34(2):205–221, 1995.

[161] SPAM Research Group. *SPAM Compiler User's Manual*, Sept. 1997. http://www.ee.princeton.edu/spam.

[162] H. Srinivasan, "Optimizing Explicitly Parallel Programs," Ph.D. Thesis, Department of Computer Science, University of Colorado, Denver, Colorado, 1994.

[163] J. Stichnoth, D. O'Hallaron, and T. Gross, "Generating Communication for Array Statements: Design, Implementation, and Evaluation," *Journal of Parallel and Distributed Computing*, 21(1):150–159, April 1994.

[164] R. Subramanian and S. Pande, "Efficient Program Partitioning based on Compiler Controlled Communication", *Proc. of the Fourth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, to appear, 1999.

[165] A. Sudarsanam et al., "Optimization of Embedded DSP Programs Using Post-pass Data-flow Analysis," *Proc. of International Conference on Acoustics, Speech, and Signal Processing*, Apr. 1997.

[166] A. Sudarsanam and S. Malik. "Memory Bank and Register Allocation in Software Synthesis for ASIPs," *Proc. of International Conference on Computer Aided Design*, pages 388–392, 1995.

[167] A. Thirumalai and J. Ramanujam, "Fast Address Sequence Generation for Data-parallel Programs using Integer Lattices," *Proc. of the Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science 1033, pages 191-208, Springer-Verlag, 1996.

[168] A. Thirumalai and J. Ramanujam, "Efficient Computation of Address Sequences in Data-parallel Programs using Closed Forms for Basis Vectors," *Journal of Parallel and Distributed Computing*, 38(2):188-203, November 1996.

[169] J. Tims, R. Gupta, and M.L. Soffa, "Dataflow Analysis Driven Dynamic Data Partitioning," *Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers, LNCS 1511, Springer Verlag*, pages 75-90, Pittsburgh, PA, May 1998.

[170] K.R. Traub, D.E. Culler, and K.E. Schauser, "Global Analysis for Partitioning Non-Strict Programs into Sequential Threads," *ACM Conference on Lisp and Functional Programming*, San Francisco, CA, June 1992.

[171] R. Triolet, F. Irigoin, and P. Feautrier, "Direct Parallelization of Call Statements," *Proceedings of the Sigplan Symposium on Compiler Construction*, pages 176-185, July 1986.

[172] P. Tu and D. Padua, "Array Privatization for Shared and Distributed Memory Machines," *Proc. 2nd Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Machines, in ACM SIGPLAN Notices*, January 1993.

[173] L. Wang, J. Stichnoth, and S. Chatterjee, "Runtime Performance of Parallel Array Assignment: An Empirical Study," *Proc. Supercomputing 96*, Pittsburgh, PA, November 1996.

[174] H. Wijshoff, *Data Organization in Parallel Computers*, Kluwer Academic Publishers, 1989.

[175] R. Wilson et al., "SUIF: A Parallelizing and Optimizing Research Compiler," *SIGPLAN Notices*, Vol. 29, No. 12, pages 31-37, December 1994.

[176] M.E. Wolf and M.S. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 4, pages 452-471, October 1991.

[177] M.J. Wolfe, *Optimizing Supercompilers for Supercomputers*, Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989.

[178] M.J. Wolfe, "Iteration Space Tiling for Memory Hierarchies," *Proc. of the 3rd SIAM Conf. on Parallel Processing for Scientific Computing*, pages 357–361, 1987.

[179] M. Wolfe and C. Tseng, "The Power Test for Data Dependence", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 5, September 1992.

[180] H. Zima, H.-J. Bast, and M. Gerndt, "Superb: A Tool for Semi-Automatic MIMD/SIMD Parallelization," *Parallel Computing*, 6:1–18, 1988.