# Automatic selection of high-order transformations in the IBM XL FORTRAN compilers

by V. Sarkar

**The IBM ASTI optimizer provides the foundation for high-order transformations and automatic shared-memory parallelization in the latest IBM XL FORTRAN (XLF) compilers for RS/6000™ and PowerPC® uniprocessors and symmetric multiprocessors (SMPs), and for automatic distributed-memory parallelization in the IBM XL High-Performance FORTRAN (XLHPF) compiler for the SP™ distributed-memory multiprocessor. In this paper, we describe how the transformer component of the ASTI optimizer automatically selects high-order transformations for a given input program and a target uniprocessor, so as to improve utilization of the memory hierarchy (including cache and registers) and instruction-level parallelism. Our solution is centered on a quantitative approach in which optimization problems are formulated using quantitative cost models. The loop and data transformations currently employed by the ASTI transformer for optimizing uniprocessor performance are loop distribution, loop interchange, loop reversal, loop skewing, loop tiling/blocking (with compiler-selected tile sizes), loop fusion, unrolling of multiple loops (with compiler-selected unroll factors), and scalar replacement of selected array references. The design and initial implementation of the ASTI optimizer were completed during the 1991–1993 time period. To the best of our knowledge, the ASTI transformer is the first system to perform automatic selection of this wide range of transformations using a cost-based framework.**

## 1. Introduction

Major changes in processor architecture over the last decade have created a demand for new compiler optimization technologies. Optimizing compilers have risen to this challenge by steadily increasing the performance gap between optimized compiled and unoptimized compiled code to a level that already exceeds



**0018-8646/97/$5.00 © 1997 IBM**

the performance gap between two successive generations of processor hardware. One class of compiler optimizations that has recently made significant contributions to improving the performance of optimized code is *high-order transformations*. High-order (or high-level) transformations operate on an intermediate representation of the program that is close to the source program level, as opposed to traditional compiler optimizations that operate on an intermediate representation that is close to the machine level. Examples of high-level transformations include loop transformations such as loop interchange and loop tiling, data transformations such as alignment and padding, and in-line expansion of procedure calls.

While a judicious use of high-order transformations can significantly improve performance, improperly selected high-order transformations can also degrade performance to levels that are worse than unoptimized code. For example, the loop interchange transformation can improve the cache locality of a loop nest with a poor input loop ordering, but can degrade the performance of a well-tuned loop nest. The loop distribution and loop fusion transformations show a similar behavior as well. In contrast, traditional optimizations (e.g., constant folding, register allocation) rarely degrade performance. This characteristic highlights an important distinction between high-order transformations and traditional optimizations. The impact of changing a compiler algorithm for performing a traditional optimization is usually an increase or decrease in the amount of performance improvement obtained, but rarely a performance degradation compared to unoptimized code. Thus, automatic selection of program transformations has to be performed much more carefully for high-order transformations than for traditional optimizations.

The IBM ASTI optimizer provides the foundation for high-order transformations and automatic shared-memory parallelization in the latest IBM XL FORTRAN (XLF) compilers for RS/6000* and PowerPC* uniprocessors and symmetric multiprocessors (SMPs), and for automatic distributed-memory parallelization in the IBM XL High-Performance FORTRAN (XLHPF) compiler for the SP* distributed-memory multiprocessor. In this paper, we describe how the transformer component of the ASTI optimizer automatically selects high-order transformations for a given input program and a target uniprocessor, so as to improve utilization of the memory hierarchy (including cache and registers) and instruction-level parallelism. The loop and data transformations currently employed by the ASTI transformer for optimizing uniprocessor performance are loop distribution, loop interchange, loop reversal, loop skewing, loop tiling/blocking (with compiler-selected tile sizes), loop fusion, unrolling of multiple loops (with compiler-selected unroll factors), and scalar replacement of array references. The design and initial implementation of the ASTI optimizer were completed during the 1991–1993 time period. To the best of our knowledge, the ASTI transformer is the first system to perform automatic selection of this wide range of transformations using a cost-based framework.

The transformations implemented in the ASTI transformer have all been proposed in past work by other researchers with motivating examples showing cases in which the transformations can be used to improve performance (e.g., see [1]). Many of these transformations were first proposed in the context of vectorizing and parallelizing compilers. However, there has been little attention paid in the research literature to the problem of how a compiler should automatically select these high-order transformations for optimizing uniprocessor performance, especially for the wide set of transformations employed by the ASTI transformer. Our solution is centered on a quantitative approach in which optimization problems are formulated using quantitative cost models, which are built on target hardware parameters and on compiler estimates of memory costs, execution time costs, and execution frequencies. In general, there is a many-to-many mapping between high-order transformations and the hardware resources that they optimize. Multiple transformations may be used to optimize a single resource (e.g., the use of loop interchange, tiling, and fusion to improve cache locality), and multiple resources may be optimized by a single transformation (e.g., the use of loop unrolling to improve both register locality and instruction-level parallelism).

In addition to improving the execution time of the optimized program, great care has been taken to ensure that the flexibility in the ASTI transformer does not come at the cost of high compile-time overhead. Efficient compile times in the ASTI transformer are obtained by avoiding modification of the program after each transformation (and thus avoiding repeated reanalysis to obtain updated control and dataflow information). Instead, the set of transformations to be applied on a procedure is accumulated as updates to the loop structure graph data structure defined in Section 4. The intermediate language is updated only after all of the encompassed transformations are finalized, after which reanalysis is not required in ASTI. This approach also allows "what-if" analysis of different loop transformation scenarios. Compile-time performance is further improved by decomposing the input procedure into regions and performing transformations on a region-by-region basis. Since the array data dependence analysis required for high-order transformations can have a worst-case execution time that is quadratic in the size of the region, this decomposition can be very effective in reducing compile time. The current decomposition approach used

by the transformer is to place in a single region each sequence of loops that are contiguous at the outermost nesting level. Compile-time performance is further improved by the use of *demand-driven* data dependence analysis, in which transformation profitability is computed before data dependence analysis so as to avoid the overhead of dependence analysis for loop nests that are already well-tuned in the input program (and thus would not benefit from transformation).

The rest of the paper is organized as follows. Section 2 gives an overview of the ASTI transformer component. Section 3 introduces a simple matrix multiply–transpose program that is used as a running example in many of the following sections. Section 4 describes the loop structure graph (LSG) data structure, which provides the foundation for the transformer. Section 5 describes how the loop distribution transformation is performed on the LSG using demand-driven data dependence analysis. Section 6 contains the details of the memory cost analysis performed by the transformer. Section 7 outlines how the transformer selects iteration-reordering loop transformations for locality optimization of the input loop nest, including automatic selection of tile sizes for the tiling transformation. Section 8 outlines the algorithm used by the transformer to automatically perform loop fusion. Section 9 describes the loop-invariant scalar replacement step, and how loop transformations are selected to increase the opportunity for loop-invariant scalar replacement. Section 10 outlines how the transformer decides which loops to unroll and what the unroll factors should be. Section 11 describes the transformer's demand-driven data dependence tester. Finally, Section 12 discusses related work, and Section 13 contains our conclusions.

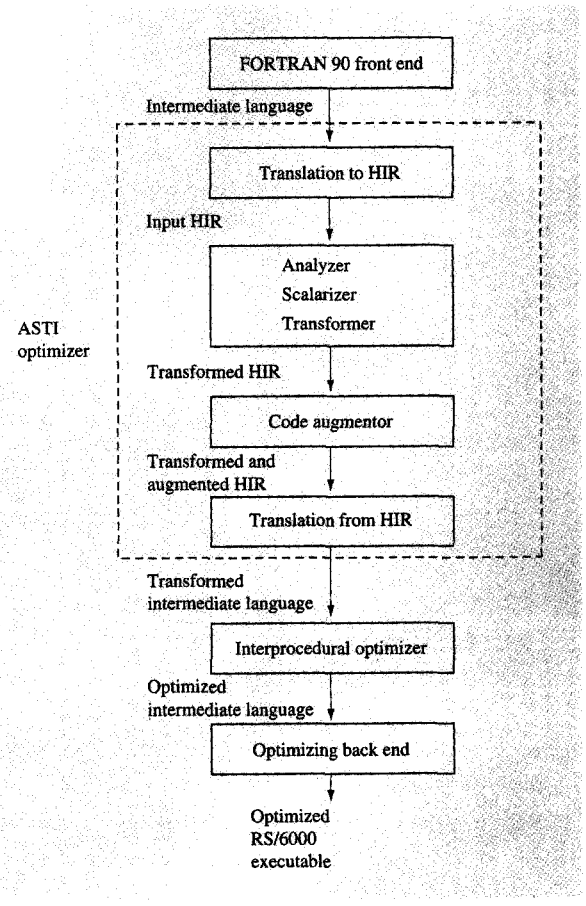



**Figure 1**

Compiler structure.

## 2. Overview of the ASTI transformer

In this section, we give an overview of the transformer component of the ASTI optimizer; this is the component that selects and implements high-order transformations. **Figure 1** shows how the ASTI optimizer fits into the overall structure of the IBM XL FORTRAN compilers. The ASTI optimizer accepts input and generates output in a high-level compiler intermediate language used to connect ASTI to the front-end and back-end components of the compiler. ASTI translates the intermediate language into a high-level intermediate representation, HIR, designed to represent the constructs of the FORTRAN and C programming languages in data structures that are suitable for high-level program analysis and transformation. This translation involves converting flat intermediate language representations of program statements and expressions into hierarchical linked-list and tree data structures that are suitable for traversal and modification by the ASTI optimizer. The HIR intermediate representation is actually referred to as HIL in the product compiler (for "high-level intermediate language"), but we refer to it as HIR in this paper to emphasize that it is an intermediate representation used internally within the ASTI optimizer and is distinct from the intermediate language used to connect ASTI to the front end and back end.

The ASTI optimizer consists of three major components:

1. *Analyzer* Performs global intraprocedural control and dataflow analysis (including static single assignment form (SSA) construction [2], induction variable analysis [3], and value numbering [4]) together with a small set of global optimizations: dead-branch and dead-code elimination [3], constant propagation [5], and invariant IF code motion out of loops [3].
2. *Scalarizer* Converts FORTRAN 90 array language statements [6] into equivalent sequential loops,

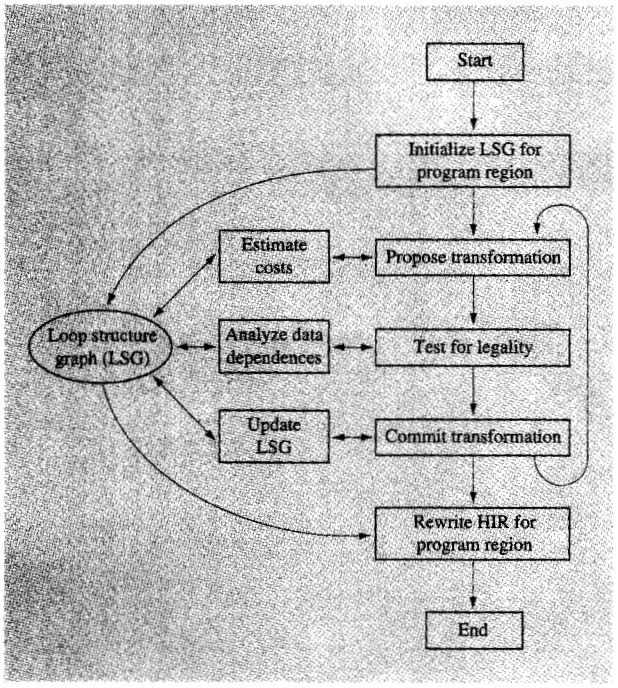



**Figure 2**

High-level structure of the ASTI transformer.

optimizing the conversion to maintain array statement semantics (all values on right-hand side evaluated before storage into the left-hand side) while limiting the number of array temporaries required. It also performs optimized expansion of array intrinsic functions to in-line code.

3. *Transformer* Performs a sequence of loop transformations which, for the scope of this paper, focus on obtaining increased performance on uniprocessor systems. The primary goal is to make optimal use of the memory hierarchy (including registers) and instruction-level hardware parallelism via transformations such as loop distribution, loop permutation, loop skewing, loop blocking, loop fusion, loop unrolling, and scalar replacement. This is the component that is described in this paper.

The acronym ASTI was derived from the functionalities provided by these three components and the interprocedural optimizer shown in Figure 1. The interprocedural optimizer is currently invoked only as a post-pass to the transformer. The transformer will benefit from automatic interprocedural analysis information and in-line expansion of procedure calls when the interprocedural optimizer becomes available as a pre-pass. In the interim, the user can communicate interprocedural information to the transformer by using compiler directives and INTENT attributes and statements in the FORTRAN 90 language [6].

**Figure 2** shows the high-level structure of the ASTI transformer. The key aspect of the design of the ASTI transformer that distinguishes it from other optimizers and makes it extensible and retargetable is a strong separation among the following procedures:

- *Initialize LSG*—Build the loop structure graph (LSG) data structure for specified single-entry single-exit regions from the HIR and related analysis information.
- *Estimate costs*—Estimate the profitability of transformations, guided by parameterized hardware costs.
- *Propose transformations*—Select a set of transformations on the basis of estimated costs.
- *Analyze data dependences*—Perform demand-driven data dependence testing on pairs of array references.
- *Test for legality*—Use data dependence and other analysis information to test for legality of proposed transformations.
- *Commit transformations*—Select transformations to be applied to the input program.
- *Update LSG*—Incrementally update the LSG data structure according to selected transformations.
- *Rewrite HIR for program region*—Update the HIR in accordance with the selected transformations recorded in the LSG.

In addition to uniprocessor target machines, this overall structure of the ASTI transformer is used for automatic shared-memory parallelization [7] in the latest IBM XL FORTRAN (XLF) compiler for PowerPC-based symmetric multiprocessors (SMPs), and is also used in the IBM XL High-Performance FORTRAN (XLHPF) compiler for performing transformations before and after generating a single-program multiple-data (SPMD) parallel program [8] for the SP distributed-memory multiprocessor.

Specifically, the ASTI transformer performs the following ten steps by default when optimizing for a uniprocessor target machine:

1. LSG initialization.
2. Loop distribution.
3. Identification of perfect loop nests.
4. Reduction recognition.
5. Locality optimization.
6. Loop fusion.
7. Loop-invariant scalar replacement.
8. Loop unrolling and interleaving.
9. Local scalar replacement.
10. Transcription.

The cost estimation, dependence analysis, and LSG update procedures shown in Figure 2 can be performed on demand by any of the steps listed above.

Transformations can be turned on and off with ease within the default sequence of ten steps listed above, either by internal switch settings or by user options and directives (which can be changed over complete programs or all or part of a loop nest). It is also relatively easy to extend the transformer by adding new transformations and modifying the default sequence. In addition to the use of the transformer in a range of product compilers for sequential and parallel machines, several prototype extensions have also been made in order to experiment with transformations such as array padding and alignment [9] and data-cache prefetching.

As a concluding note to this section, we present some measurements comparing the performance of the four kernel computations (Btrix, Gmtry, Emit, Vpenta) from the SPECfp92** Dnasa7 benchmark that were studied in [10], when compiled with and without use of the ASTI transformer. Figure 2 summarizes the user execution time (in seconds) measured for the four kernels on a single 133-MHz PowerPC 604* processor in an IBM RS/6000 Model J30 SMP workstation. The performance measurements were made using Version 4.1 of the IBM XL FORTRAN compiler. The bar chart labels refer to the two different compiler optimization options that were used to obtain the performance measurements, as follows (the `-qarch=604` option directs the compiler to generate code for the PowerPC 604 processor):

*-O3* *Compile command*: `xlf -O3 -qarch=604...`
The `-O3` option directs the compiler to perform the highest level of back-end optimization, even though it may come at the cost of a larger compile time or a larger memory utilization by the compiler, compared to `-O`.

*-qhot* *Compile command*: `xlf -O3 -qhot -qarch=604...`
The `-qhot` option enables high-order transformations in the XL FORTRAN compiler, using the ASTI optimizer described in this paper[1]. The `-O3 -qhot` combination can be viewed as the next (and highest) level of optimization beyond `-O3` that is supported by the compiler.

**Figure 3** shows that the automatic high-order transformations implemented in the ASTI transformer (i.e., the `-qhot` option) provided significant speedup (up to 3.4×) for three of the four kernels. Even larger

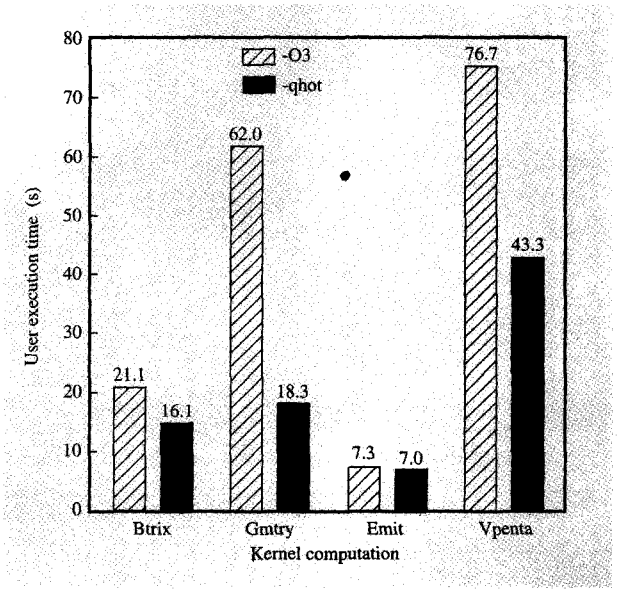



**Figure 3**

Performance measurements on a 133-MHz PowerPC 604 processor for four Dnasa7 kernels.

speedups can be obtained for other kernel computations, as discussed in Section 3. Since most high-order transformations can be implemented by hand in the source program, the speedup that can be obtained by automatic selection of high-order transformations will depend on how much effort has already been spent by the programmer on tuning the code.

## 3. Matrix multiply–transpose example

In this section, we introduce a simple matrix multiply–transpose example that is used as a running example in many of the following sections. The FORTRAN 77 code for this example is shown in **Figure 4**. Subroutine `mmt` effectively computes the product of input matrices `b` and `c` and stores the transpose of the result in matrix `a`. Array variables `a`, `b`, and `c` are declared as two-dimensional $n \times n$-sized matrices, and the dimension size $n$ is unknown at compile time.

**Figure 5** summarizes some performance measurements for this example program on a single 133-MHz PowerPC 604 processor in an IBM RS/6000 Model J30 SMP workstation. These performance measurements were made for a single call to subroutine `mmt()` for $n = 500$. The performance measurements were made using Version 4.1 of the IBM XL FORTRAN compiler. The labels on the $x$-axis in the bar charts in Figure 5 refer to different compiler optimization options that were used to obtain the performance measurements, as follows (the `-qarch=604`

[1] In Release 3.2 of the XL FORTRAN compiler, the `-qhot` option invoked a back-end phase [11] for performing high-order transformations. That phase is no longer supported, and `-qhot` is now used to invoke the ASTI optimizer as described in this paper.

```
[1]    subroutine mmt(a, b, c, n)
       real*8 a(n,n), b(n,n), c(n,n)

[2]    do i1 = 1, n
[3]       do i2 = 1, n
[4]          a(i1,i2) = 0
[5]          do i3 = 1, n
[6]             a(i1,i2) = a(i1,i2) + b(i2,i3) * c(i3,i1)
[7]          end do
[8]       end do
[9]    end do
[10]   end
```

**Figure 4**

Matrix multiply–transpose example.

option directs the compiler to generate code for the PowerPC 604 processor):

*unopt* *Compile command*: `xlf -qarch=604...`
The compiler generates unoptimized code in the absence of any optimization option.

*-O2* *Compile command*: `xlf -O2 -qarch=604...`
This is the default level of optimization performed by the XL FORTRAN compiler; i.e., `-O2` is the same as `-O`.

*-O3* *Compile command*: `xlf -O3 -qarch=604...`
The `-O3` option directs the compiler to perform additional back-end optimizations that may come at the cost of a larger compile time or a larger memory utilization by the compiler, compared to `-O2`.

*-Pk* *Compile command*: `xlf -O3 -Pk -Wp, -optimize=5, -scalaropt=3 -qarch=604...`
The `-Pk` option directs the compiler to invoke the "KAP for IBM XL FORTRAN" source-to-source preprocessor [13] before compilation, for additional optimization. The `-Wp, -optimize=5, -scalaropt=3` command string enables the highest optimization levels supported by this preprocessor.

*-Pv* *Compile command*: `xlf -O3 -Pv -Wp, -eavx -qarch=604...`
The `-Pv` option directs the compiler to invoke the "VAST-2 for XL FORTRAN" source-to-source preprocessor [14] before compilation, for additional optimization. The `-Wp, -eavx` command string enables the highest optimization levels supported by this preprocessor.

*-qhot* *Compile command*: `xlf -O3 -qhot -qarch=604...`
The `-qhot` option enables high-order transformations in the XL FORTRAN compiler, using the ASTI optimizer described in this paper. The `-O3 -qhot` combination can be viewed as the next (and highest) level of optimization beyond `-O3` that is supported by the compiler.

*-essl* *Compile command*: `xlf -O3 -lessl -qarch=604...`
Since the matrix multiply–transpose example uses standard dense-matrix operations, we include performance measurements obtained by rewriting the program to call the following IBM Engineering Scientific Subroutine Library (ESSL) routines for matrix multiplication and transpose in place of subroutine `mmt`:

```
call dgemm ('N', 'N', n, n, n, 1.DO,
b, n, c, n, 0.DO, a, n)
call dgetmo (a, n, n, n, a,n)
```

The `-lessl` option directs the compiler to search the `libessl.a` library file to find definitions of the `dgemm` and `dgetmo` routines [15, 16]. This option is technically not an optimization option, since it involves a rewrite of the source program. However, we include this case in the performance measurements, since it represents an ideal performance goal for optimizing compilers, viz., to match or beat the performance of well-tuned handcrafted code for a given processor.

Figure 5 is a scatter plot illustrating the *user + system execution times* (in seconds) measured for the different optimization options. The execution time was dominated by the user time component—the system execution time was almost negligible (less than 0.1 seconds in all cases except `-lessl`, for which it was 0.4 seconds). The measurements were repeated ten times, and the scatter plot shows significant variation (up to 20%) in execution times for the first five cases (all cases except `-qhot` and `-lessl`). The most likely source of execution time variation in the first five cases is the fact that the PowerPC 604 (like other modern microprocessors) has caches that are indexed by physical memory addresses rather than virtual addresses. The last two cases (`-qhot` and `-lessl`) did not exhibit this execution time variation because they incur significantly fewer cache misses than the first five cases. In general, some form of operating system support (e.g., as in [17]) is required to avoid the execution time variations in the first five cases. However, the general performance trend for the different compiler

options can easily be observed in Figure 5, despite the execution time variations.

We see that a speedup of approximately 1.3× was obtained by turning on optimization at the `O2` level. An extra speedup of approximately 1.1× was then obtained by increasing the optimization level from `-O2` to `-O3`, resulting in an execution time of approximately 60 seconds. Adding the `-qhot` option dramatically reduced the execution time to 4.1 seconds, which is about the same as the execution time of 4.3 seconds obtained by using the ESSL routines. Thus, automatic selection of high-order transformations using the `-qhot` option for this simple program delivered a 15× performance improvement over the `-O3` case! In contrast, the `-Pk` and `-Pv` options resulted in speedups of approximately 1.1× and 2.2× respectively, compared to the `-O3` case. The 15× performance improvement delivered by the `-qhot` option came about because of a large reduction in both the number of data-cache misses incurred by the program and the number of memory (load/store) instructions executed by the program.

Speedups in the 5–15× range are usually only observed for kernel computations such as matrix multiply–transpose. The performance improvement for entire programs, while still significant, is typically less than 2×. However, this speedup factor is likely to increase in the future as the performance gap between processor hardware and memory systems continues to widen.

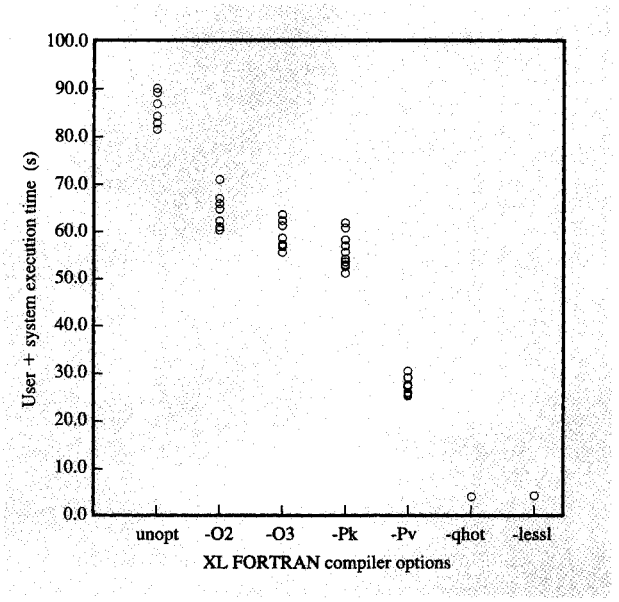



**Figure 5**

Performance measurements on a 133-MHz PowerPC 604 processor for matrix multiply–transpose example with different compiler optimization options.

## 4. Loop structure graph

Key questions that arise when building an optimizer for performing high-order transformations are "What structures should be used to represent the internal state of the program, and how should these structures be updated after each transformation?" We believe that the choice of internal representation is a critical issue in determining a compiler's ability and effectiveness in dealing with multiple transformations. It is important for the internal representation to be flexible and general enough to accommodate new transformations and different orderings of existing transformations as the optimizer evolves to target new processor architectures.

Many internal program representations have been proposed in the past, including the control flow graph [3], the interval structure [18, 19], the program dependence graph [20], the static single assignment form [21, 22], the forward control dependence graph [23–25], the hierarchical structured control flow graph [26], and the hierarchical task graph [27]. These representations simplify analysis and code generation, but updating these representations after transformation is often tedious, error-prone, and costly. This is particularly true for loop transformations, which can dramatically alter the program form. The problem is exacerbated when compilers simultaneously use more than one of these representations, thus enhancing analysis at the expense of transformation complexity.

In this section, we describe a new internal program representation called the loop structure graph (LSG), which is designed to facilitate loop transformations and statement-reordering transformations without sacrificing the analysis efficiency of other representations. All of the high-order transformations performed by the ASTI transformer use the LSG in such a way that only a small amount of updating is required after each transformation. We show how the LSG can be used to perform both iteration-reordering loop transformations such as interchange and tiling, and statement-reordering loop transformations such as distribution and fusion of loop nests.

The backbone of the LSG is a loop structure tree (LST), in which each internal node corresponds to a loop (i.e., a single-entry strongly connected region) in the control flow graph for the program being transformed, and each leaf node corresponds to a statement. The LST is initialized from the interval structure tree [19] of the input program, and is updated by various loop transformations. The control flow in each loop body is captured by a loop-level control flow graph (LCFG), and the data references in each loop body are summarized in the input/output lists. In addition, a loop-level dependence graph (LDG) is

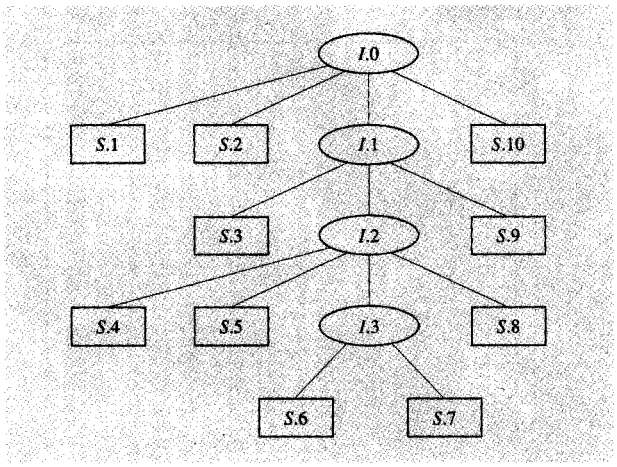



**Figure 6**

Loop structure tree for matrix multiply–transpose example.

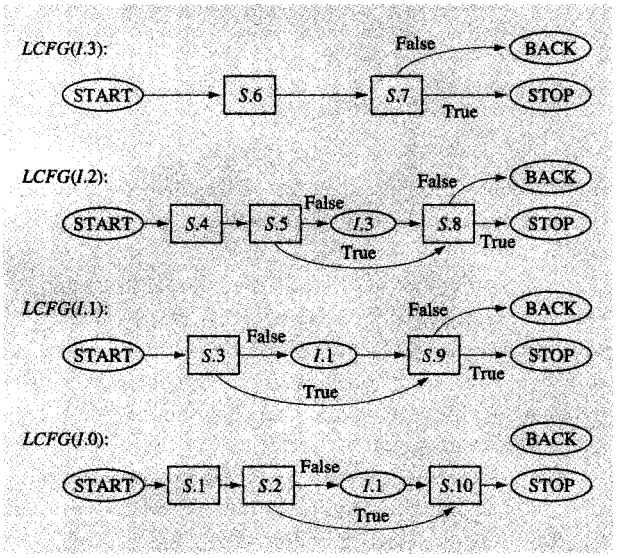



**Figure 7**

Loop-level control flow graphs for matrix multiply–transpose example.

computed on demand for any loop body, when the transformer has to enumerate its control and data dependences. Together, the LST and the individual LCFGs, input/output lists, and LDGs comprise the LSG representation introduced in this paper. As a whole, the LSG is initialized from the information provided by the ASTI analyzer component (control flow graph, intervals, SSA, constant propagation, value numbering, induction variable analysis), but subsequent transformations of the LSG update only the LSG without reinvoking any of the analysis algorithms.

The program's loop structure is self-evident in the hierarchical structure of the LSG. In contrast to the forward control dependence graph [23–25], the LSG avoids the creation of pseudo-control-flow edges that can potentially weaken analysis by representing control flow paths that cannot be executed in the original program. An important strength of the LSG is its robust handling of *irreducible regions* [3]. An irreducible region is merged, with the smallest single-entry region in which it is contained, into a single-loop node, thus isolating it from other (containing, contained, or unrelated) loops in the LST which remain eligible for all transformations and optimizations.

The following sections describe the four major data structures in the LSG: the loop structure tree, the loop-level control flow graph, the input/output lists, and the loop-level dependence graph.

- *Loop structure tree (LST)*

The loop structure tree represents the loop-nesting structure of the input program. It is derived from the interval structure used in the modified interval analysis formulated by Schwartz and Sharir [19]. If the flow graph contains an irreducible region (a strongly connected region with multiple entries), we include that irreducible region within its smallest containing single-entry region (which we refer to as a *loop*) and mark that loop as irreducible. An irreducible loop is not eligible for any loop transformation, but all other loops are eligible for loop transformations. Thus, each interior node of the interval structure tree represents a single-entry loop, and each leaf node of the interval structure tree corresponds to a node in the control flow graph. Usually, a node in the control flow graph is a *basic block* which may contain multiple statements. Since many program transformations are based on single statements, we expand each basic block into its individual statements when constructing the loop structure tree. Thus, each interior node (I-node) of the LST represents a (structured or unstructured) loop, and each leaf node (S-node) of the LST represents a statement.

The I-node serves as a useful anchor for all information related to its loop. For example, the loop-level control flow graph (LCFG), input/output lists, loop dependence graph (LDG), loop-carried dependence vectors, and the transformation sequence are all anchored in the I-node for the appropriate loop.

**Figure 6** shows the loop structure tree built for the matrix multiply–transpose example program in Figure 4. For convenience, we create a special I-node, *I*.0, that represents the entire subroutine as a dummy loop. *I*.0 is

**Table 1** Local input/output lists built for the matrix multiply–transpose example of Figure 3.

| *Loop* | *Scalar-defs* | *Scalar-uses* | *Array-defs* | *Array-uses* |
|---|---|---|---|---|
| $I.0$ | $i_1$ | $n$ | | |
| $I.1$ | $i_1, i_2$ | $n, i_1$ | | |
| $I.2$ | $i_2, i_3$ | $n, i_1, i_2$ | $a(i_1, i_2)$ | |
| $I.3$ | $i_3$ | $n, i_1, i_2, i_3$ | $a(i_1, i_2)$ | $a(i_1, i_2), b(i_2, i_3), c(i_3, i_1)$ |

thus the root of the LST. The remaining I-nodes, $I.1$, $I.2$, $I.3$, correspond to the DO loops with index variables $i_1$, $i_2$, and $i_3$, respectively. There are ten S-nodes in the LST, corresponding to the ten statements in the input program. For a given DO loop, the DO statement corresponds to its initial zero-trip test, and the ENDDO statement corresponds to the increment-and-test operation performed in each iteration. That is why, for example, the LST shows the DO statement $S.2$ outside the $I.1$ loop and the ENDDO statement $S.9$ contained inside the $I.1$ loop.

- *Loop-level control flow graph (LCFG)*

For each I-node (loop), L, in LST, we build a loop-level control flow graph, LCFG(L), that defines the control flow for L's immediate children in the LST. Each child LST node (statement or loop) is a node in L's LCFG. We also add three pseudo-nodes to each LCFG: START, BACK, and STOP. The pseudo-nodes have the following interpretations:

- START node is the loop entry node.
- BACK node is the target of all backward control flow edges.
- STOP node is the target of all loop exit branches.

LCFG edges represent control flow within the loop. Each LCFG edge is annotated with an ordered pair (BranchStmt, BranchLabel) as follows:

- BranchStmt is an index to the HIR representation of the branch statement that caused this control flow branch.
- BranchLabel is the label value which is the target of this branch from BranchStmt.

The LCFG is acyclic for all loops that are reducible. For each exit from loop L, there is an edge in L's LCFG with target STOP, and there is an edge in the LCFG of L's outer loop from L's I-node to the exit target. If multiple nested loops are being exited, there are additional edges in the LCFG edges for the outer loops.

**Figure 7** shows the four loop-level control flow graphs, $LCFG(I.3)$, $LCFG(I.2)$, $LCFG(I.1)$, and $LCFG(I.0)$, built for the matrix multiply–transpose example program in Figure 4. All LCFGs are acyclic, since all loops are reducible in this example. $LCFG(I.3)$ is the loop-level control flow graph for the innermost $i_3$ DO loop, $I.3$, and thus contains only S-nodes and pseudo-nodes. A true branch from the ENDDO statement $S.7$ represents a loop exit, as shown by the branch to the STOP pseudo-node; the false branch is connected to the BACK node because it represents a continuation of the loop. $LCFG(I.2)$ is the loop-level control flow graph for the middle $i_2$ DO loop. Since the $i_3$ loop is nested inside the $i_2$ loop, we see that $I.3$ is a child of $I.2$ in the LST; hence, there is also a node for $I.3$ in $LCFG(I.2)$. Similarly, $LCFG(I.1)$ is the loop-level control flow graph for the outer $i_1$ DO loop, and $LCFG(I.0)$ is the loop-level control flow graph for the outermost level of control flow in the subroutine. One distinguishing feature of $LCFG(I.0)$ is that it does not contain an edge to the BACK pseudo-node, since there is no loop at this outermost level of control flow.

- *Input/output lists*

Input/output lists are used to collect variable references[2] on a loop-by-loop basis. The input/output lists for a given loop are anchored in the LSG I-node corresponding to that loop, and are updated when the loop body changes, e.g., due to a loop distribution or loop fusion transformation. Input/output lists are used to enumerate data dependences in a loop and also to estimate memory costs for cache/TLB locality and register locality. There are four separate input/output lists—scalar-defs, scalar-uses, array-defs, and array-uses. Each input/output list has two levels and is structured as a linked list of variables at the top level. For each variable in the top-level list, there is a second-level list of references to the variable in the loop.

The input/output lists are *local* lists; they contain only references that are immediately contained within the loop. When needed, the transformer builds temporary *global* input/output lists on demand by merging local lists in a bottom-up traversal of the LST.

**Table 1** shows the (local) input/output lists built for the matrix multiply–transpose example program in Figure 4.

- *Loop-level dependence graph (LDG)*

For each loop, L, in LST, we can compute a loop-level data dependence graph, LDG(L), on demand that contains

[2] A variable *reference* is a definition or a use of a variable [3]. The analyzer component of the ASTI optimizer identifies variable references in the HIR.

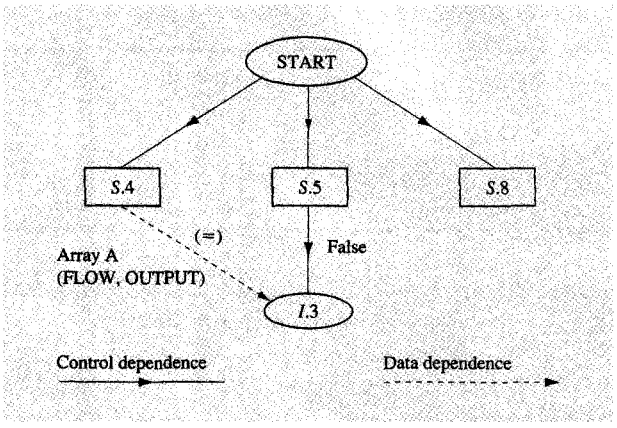



**Figure 8**

Loop-level dependence graph for $i_2$ loop in matrix multiply–transpose example.

the control dependence [20] and data dependence [1] edges for L's immediate children in the LST. Each LDG node in the graph is either a child node of loop L in LST, or the START pseudo-node. BACK and STOP nodes do not appear in the LDG. Each LDG edge is either a control dependence edge or data dependence edge. A control dependence edge is labeled with a (BranchStmt,BranchLabel) pair identifying the LCFG edge that caused this control dependence. A data dependence edge is labeled with a tag identifying the nature of the dependence (FLOW, ANTI, OUTPUT), the variable(s) that cause the dependence, and additional information, as described later in Section 11. Control dependence edges are computed from LCFG(L) using the standard algorithm [20]. Data dependence edges are computed for scalar variables using analysis information (SSA links [2], induction variable information, etc.) and for array variables by performing array data dependence testing on demand.

**Figure 8** shows the loop-level dependence graph that is constructed on demand for loop *I*.2. Except for STOP and BACK, it contains the same set of nodes as in *LCFG*(*I*.2) in Figure 7. Note that *I*.3 is control-dependent on *S*.5 (with the False label), but *S*.8 is not control-dependent on *S*.5. Since *S*.5 and *S*.8 are DO and ENDDO statements, the only nodes in *LDG*(*I*.2) with interesting data accesses are *S*.4 and *I*.3. There is a data dependence from *S*.4 to *I*.3 due to array variable `a`, but the data dependence tester can tell that it is a loop-independent dependence [28], and that there is no dependence from *I*.3 back to *S*.4. We do not include any data dependences on loop index variables $i_1$, $i_2$, $i_3$ in Figure 8, because they are all induction variables that impose no restriction on transformations.

- *Sequence representation of iteration-reordering loop transformations*

Using the framework described in [29], we represent an iteration-reordering loop transformation as a sequence of template instantiations from a small but extensible kernel set of transformation templates. The sequence is anchored in the LST I-node of the outermost loop in the perfect nest being transformed. The three loop-transformation templates supported by the ASTI transformer for uniprocessor optimizations are `ReversePermute`, `Unimodular`, and `Tile/Block`. The `Parallel` and `Coalesce` templates described in [29] are used for SMP parallelization. The transformer selects templates from this kernel set and instantiates them with specific values so as to build a desired iteration-reordering transformation as a sequence of template instantiations.

A transformation template specifies rules for mapping dependence vectors, mapping loop-bound expressions, and creating initialization statements for a transformed loop nest. The dependence vectors of a loop nest are used to test the legality of applying a transformation, with respect to the data dependence constraints of the original loop nest. When the loop nest is transformed, its dependence vectors also change, as specified by the dependence vector mapping rules. Details on the mapping rules for dependence vectors and loop bounds for individual transformations are provided in [29].

- *Transcription from the LSG*

One of the major advantages of using the LSG is that it simplifies the HIR rewrite step (transcription, step 10 in Figure 1). During transformation, the LSG is updated to facilitate the final code generation. For example, the updated loop-bound expressions and index variable mappings are stored in the LSG after the loop transformations have been performed [29]. Also, all nodes in the same loop body (i.e., the same LCFG) are linked in a total order that is guaranteed to be a topological sort of the LDG. This provides a legal order in which the statements can be emitted in the output code. Transcription is performed via a depth-first traversal of the LST analogous to code generation from the forward control dependence graph [25]. The main extra piece of work that must be done during code generation is to update the branch statements in the HIR to reflect the changes made to the LSG.

## 5. Loop distribution

In this section, we describe how the loop distribution step is performed in the ASTI transformer, and how the LSG is updated after loop distribution. Loop distribution is a program transformation that converts a single loop into multiple loops, each of which iterates over a subset of statements in the original loop. Loop distribution can

uncover more perfectly nested loops, thus increasing the opportunity for applying iteration-reordering loop transformations. When applied to large loop bodies, it can also improve performance by reducing register spills.

Loop distribution is performed early (step 2) in the transformer because it increases the opportunity for applying other transformations. For example, disabling loop distribution in the transformer increases the execution time of the matrix multiply–transpose example for the `-qhot` case from 4.1 seconds to 60 seconds (the same as the execution time for the `-O3` case). However, loop distribution also has the potential to degrade performance compared to unoptimized code by worsening data locality, and by creating extra loop increment-and-test instructions. We say that a loop distribution transformation is *necessary* if it enables a later iteration-reordering loop transformation that could otherwise not have been performed. We assume that the performance gain obtained by the later iteration-reordering loop transformation always outweighs the overhead of a necessary loop distribution transformation.

Loop fusion [1] is a well-known loop transformation that fuses (combines) multiple conformable loop nests into a single loop nest and is thus the inverse of loop distribution. The loop fusion step described later, in Section 8, can thus remove the overhead of unnecessary loop distribution by fusing the loops back together again. Weighted loop fusion is an NP-hard problem; the transformer currently uses a greedy clustering algorithm to select an optimized loop fusion configuration. It is important to perform loop fusion late in the transformer, so that the flexibility provided by loop distribution can be exploited by the loop transformations for enhancing locality which must be performed before loop fusion.

To better understand the interaction between loop distribution and loop fusion, we observe that the result of any sequence of fusion and distribution transformations is a regrouping of the statements in the bodies of the loop nests in the original program. All sequences of fusion and distribution transformations that result in the same regrouping of statements and in the same ordering of regrouped loop nests are equivalent. The goal of combining distribution and fusion is to automatically select an optimized fusion/distribution configuration, i.e., an optimized regrouping of statements. Therefore, without any loss of generality, we can assume that all loop distribution transformations are performed before any loop fusion transformation, and the problem of selecting an optimized fusion/distribution configuration thus becomes equivalent to selecting an optimized fusion partition of the statements after loop distribution. That is why we do not need to use costs to guide selection of loop distribution transformations, but we must use costs to guide the loop fusion step in the transformer.

The ASTI transformer can perform loop distribution at the following granularities:

1. *Maximal*—Distribute loops across strongly connected components ($\pi$-blocks) [1] of the dependence graph.[3]
2. *Affinity*—Restrict distribution of innermost loops so that two statements that access the same variable in the same loop are not split into separate loops. Non-innermost loops are maximally distributed.
3. *Outer*—Restrict loop distribution so that innermost loops are not distributed at all. Non-innermost loops are maximally distributed.
4. *None*—No loop distribution.

Other phases in the transformer make no assumption about the degree of distribution performed; they simply operate on the loop nests present in the LSG. The current loop distribution default for uniprocessor optimization is *Affinity*. Affinity loop distribution is used as the default to ensure that there is no possibility for a serious performance degradation after loop distribution, since a greedy heuristic algorithm is used in the loop fusion step. The use of an optimal weighted loop fusion algorithm (such as the integer programming formulation in [30]) instead would remove the need for this precaution, and allow *Maximal* loop distribution to be used as a suitable default.

The distribution of statements into loops must preserve the data and control dependences of the original loop. A partition violates the original dependence relations if and only if a dependence cycle is distributed across more than one loop. Allen and Kennedy [31] presented an algorithm to maximally distribute loops from the outer level to the inner level on the basis of a comprehensive data dependence graph. We obtain the same result by distributing the loops from the inner level to the outer level using a demand-driven approach to data dependence analysis. The inner-to-outer traversal is well suited to demand-driven loop distribution for uniprocessor optimization, because outer levels of loop distribution can be skipped once we reach a boundary of an innermost perfect loop nest.

A key feature of the loop distribution performed by the ASTI transformer is its use of control dependence information. Analogous to data dependences, a control dependence may be loop-carried or loop-independent. A control dependence is loop-carried if its control path passes through a back edge of a surrounding loop; otherwise, it is loop-independent. Control dependences are labeled with direction vectors in the same manner as

[3] We define "maximal" loop distribution in the absence of other transformations. There may be cases in which additional opportunities for loop distribution can be revealed by first performing iteration-reordering transformations such as loop interchange.

```
Before distribution
--------
      subroutine foo(a,b,c,q,r,s,n)
      real*8 a(n), b(n), c(n), q(n), r(n), s(n)

      do i = 1, n
        if ( q(i) - r(i) ) 10, 20, 30
10      s(i)= -1.0
        goto 40
20      s(i)= 0.0
        goto 40
30      s(i)= 1.0
40      a(i)= b(i) * c(i)
      end do

      end

After distribution
--------
      subroutine foo(a,b,c,q,r,s,n)
      real*8 a(n), b(n), c(n), q(n), r(n), s(n)

      do i= 1, n
        if ( q(i)- r(i) ) 10, 20, 30
10      s(i)= -1.0
        goto 40
20      s(i)= 0.0
        goto 40
30      s(i)= 1.0
40      continue
      end do

      do i= 1, n
        a(i)= b(i) * c(i)
      end do
      end
```

**Figure 9**

Example of loop distribution with control flow.

data dependences, so that a loop-independent control dependence will have a $(=, \cdots, =)$ direction vector. Loop distribution is then performed across strongly connected components of the combined control and data dependence graph. The use of control dependence information enables the loop distribution algorithm to work even in the presence of control flow arising from GOTO statements in the loop body, as in the example program shown in **Figure 9**. (Though the ASTI transformer updates the LST, we instead show the equivalent transformed source code in the figure for the sake of simplicity.) This robustness in handling control flow is important for generality in optimization of handwritten programs and computer-generated programs.

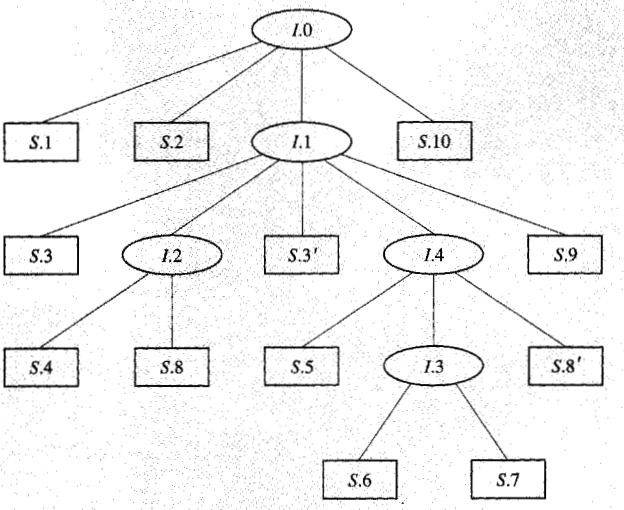
Demand-driven data dependence testing for loop distribution is based on the observation that the output of loop distribution is a partition of dependence graph nodes into components such that there is no cycle among components. Strongly connected components define the finest such partition. As we add edges to the dependence graph (based on control and data dependences), we do not need to test for a data dependence between statements A and B, in either direction, if we have already determined that there is a cycle containing both A and B at the current level of loop distribution. This observation eliminates the need to perform array data dependence tests between statements that belong to a cycle built from control dependences and scalar data dependences. Also note that loop distribution is concerned only with the set of *levels* that may carry a data or control dependence, but not with other details of the direction or distance vectors. Once the set of levels has been established, we do not need to further refine the set of direction vectors into a more precise set, thus avoiding the overhead of full data dependence testing.

Let us examine how loop distribution proceeds inside-out for the matrix multiply–transpose example program in Figure 4. There is no scope for distribution in the innermost $i_3$ loop because it contains only one nontrivial statement. To perform loop distribution on the $i_2$ loop, the transformer first computed $LDG(I.2)$ on demand (see Figure 8). Since the LDG is acyclic, it is legal to distribute the $i_2$ loop across $\{S.4\}$ and $\{S.5, I.3\}$ (we keep $S.5$ together with $I.3$ because $S.5$ is the DO statement for loop $I.3$). The LST resulting from this loop distribution is shown in **Figure 10(a)**. Though the ASTI transformer only updates the LST, we also show the transformed source code corresponding to the new LST in **Figure 10(b)** for convenience in understanding the transformation. Note that distributing the $i_2$ loop resulted in a new I-node, $I.4$, and two new S-nodes, $S.3'$ and $S.8'$.

Finally, loop distribution is performed on the outermost $i_1$ loop. This results in distributing the $i_1$ loop across $\{S.3, I.2\}$ and $\{S.3', I.4\}$; the updated LST and corresponding transformed code are shown in **Figure 11**. It results in the creation of a new I-node, $I.5$, and two new S-nodes, $S.2'$ and $S.9'$. Figure 11 also highlights the two perfect loop nests, $I.1$–$I.2$ and $I.5$–$I.4$–$I.3$, identified in step 3 of the transformer.

## 6. Memory cost analysis

Consider an innermost loop nest containing $h \geq 1$ loops, numbered $1 \cdots h$ from outer to inner. The job of memory cost analysis in the transformer is to build symbolic functions for the estimated number of *distinct cache lines* [32], $DL_{\text{total}}(t_1, \cdots, t_h)$, and the estimated number of *distinct pages*, $DP_{\text{total}}(t_1, \cdots, t_h)$, accessed by a single *tile* [1] of $t_1 \times \cdots \times t_h$ iterations of loops $1 \cdots h$,




(a)

```
[1]     subroutine mmt(a, b, c, n)
        real*8 a(n,n), b(n,n), c(n,n)
[2]     do i1 = 1, n
[3]        do i2 = 1, n
[4]           a(i1,i2) = 0
[8]        end do
[3']       do i2 = 1, n
[5]           do i3 = 1, n
[6]              a(i1,i2) = a(i1,i2) + b(i2,i3) * c(i3,i1)
[7]           end do
[8']       end do
[9]     end do
[10]    end
```

(b)

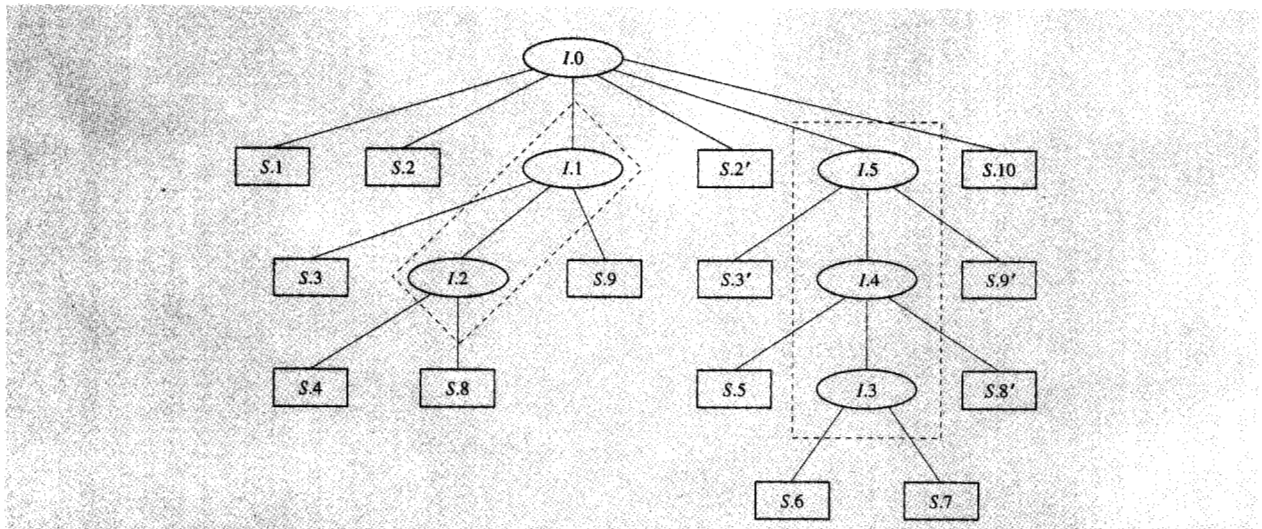
**Figure 10**

(a) Transformed loop structure tree and (b) corresponding source code after distributing loop $i_2$ in matrix multiply–transpose example.

respectively. The memory cost analysis techniques presented in this paper apply to cost estimations for the data cache and the data TLB (translation lookaside buffer) in the processor. These techniques can easily be adapted for cost analysis of other levels of the memory hierarchy, such as the L2 cache.

$DL_{\text{total}}(t_1, \cdots, t_h)$ and $DP_{\text{total}}(t_1, \cdots, t_h)$ are symbolic functions of hypothetical tile size variables, $t_1, \cdots, t_h$. The tile size variables, $t_1, \cdots, t_h$, are used to provide a flexible interface for memory cost analysis. Their presence does not mean that the tiling transformation will necessarily be performed or that memory cost analysis is restricted to tiled loops. Instead, setting different tile size values in memory cost analysis is a convenient way of using the same $DL_{\text{total}}$ and $DP_{\text{total}}$ cost functions to evaluate the memory costs of different loop configurations.

As an example, let us consider how the $DL_{\text{total}}$ memory cost function might be used to decide whether or not to perform a loop interchange/permutation [1] transformation that moves the outermost loop to the




(a)

```
[1]     subroutine mmt(a, b, c, n)
        real*8 a(n,n), b(n,n), c(n,n)
[2]     do i1 = 1, n
[3]        do i2 = 1, n
[4]           a(i1,i2) = 0
[8]        end do
[9]     end do
[2']    do i1 = 1, n
[3']       do i2 = 1, n
[5]           do i3 = 1, n
[6]              a(i1,i2) = a(i1,i2) + b(i2,i3) * c(i3,i1)
[7]           end do
[8']       end do
[9']    end do
[10]    end
```

(b)

**Figure 11**

(a) Transformed loop structure tree and (b) corresponding source code after distributing loops $i_2$ and $i_1$ in matrix multiply–transpose example.

innermost position. Let the iteration counts for loops $1 \cdots h$ be $n_1, \cdots, n_h$. Then, the total number of distinct cache lines accessed by the innermost loop can be evaluated as $DL_{\text{total}}(1, \cdots, 1, n_h)$, which yields an average of $DL_{\text{total}}(1, \cdots, 1, n_h)/n_h$ cache lines per iteration. Similarly, the number of distinct cache lines accessed by the outermost loop when moved to the innermost position can be evaluated as $DL_{\text{total}}(n_1, 1, \cdots, 1)$, which yields an average of $DL_{\text{total}}(n_1, 1, \cdots, 1)/n_1$ cache lines per iteration. The loop interchange is likely to be beneficial when this cost is smaller than the original per-iteration memory cost. Section 7 describes how the locality optimization step in the ASTI transformer uses these memory cost functions to automatically select loop permutation and loop tiling transformations. The following subsections describe how the $DL_{\text{total}}$ and $DP_{\text{total}}$ memory

cost functions are constructed by the transformer at compile time.

• *Estimating memory cost for a single array reference*

We start by describing how the ASTI transformer estimates the memory cost of a single array reference by extending the approach outlined in [32]. The goal of this analysis is to obtain a symbolic expression for the number of distinct lines accessed by the array reference as a function of the tile size variables, $t_1, \cdots, t_h$. An important use of cost functions for individual array references is in building the $DL_{\text{total}}$ cost function for an entire loop body as described in the next subsection. In addition, the cost function for a single array reference can be used to characterize the *self-reference locality* exhibited by the reference, as follows. Let

$$F(t_1, \cdots, t_h) = \frac{\textit{Number of distinct lines accessed by array reference}}{t_1 \times \cdots \times t_h}$$

be the average number of cache lines accessed by the array reference per iteration in the tile. We can then determine whether the array reference exhibits cache locality for loop $k$ by evaluating the partial derivative $\delta F/\delta t_k$ at $t_1 = \cdots = t_h = 1$ and checking whether $\delta F/\delta t_k < 0$. A negative value of $\delta F/\delta t_k$ indicates that increasing $t_k$ (the number of loop $i_k$ iterations in the tile) causes a decrease in the average number of lines accessed per iteration by the array reference.

Consider a reference of the form $\mathbf{A}[f_1(i_1, \cdots, i_h), \cdots, f_m(i_1, \cdots, i_h)]$ to an $m$-dimensional array variable called **A**, enclosed in $h$ inner perfectly nested loops with index variables $i_1, \cdots, i_h$. The subscript expressions for the $m$ dimensions of the array reference are denoted by the functions $f_1, \cdots, f_m$. An array reference is said to be *analyzable* if its subscript expressions can be written as

$$f_1(i_1, \cdots, i_h) = c[1, 0] + \sum_{k=1}^{h} c[1, k] i_k$$

$$f_m(i_1, \cdots, i_h) = c[m, 0] + \sum_{k=1}^{h} c[m, k] i_k ,$$

such that all coefficients $c[j, k]$ are *compile-time constants* for $1 \le j \le m$, $1 \le k \le h$, and the $c[j, 0]$ terms are *invariant* in loops $i_1, \cdots, i_h$; i.e., the $c[j, 0]$ terms need not be compile-time constants but they must not depend on the values of $i_1, \cdots, i_h$. Otherwise, the array reference is said to be *nonanalyzable*. The memory cost analysis in the ASTI transformer makes a worst-case assumption of one cache miss and one TLB miss per iteration for nonanalyzable array references. For analyzable references, we define $\mathbf{C}[j] = (c[j, 1], \cdots, c[j, h])$ to be the coefficient vector for dimension $j$. All elements of $\mathbf{C}[j]$ are constant in an analyzable reference. We also denote the entire coefficient matrix consisting of all $c[*, *]$ elements by **C**.

This definition of analyzable array references is similar to the definition of affine subscript expressions used in the literature. The only distinction is that our definition of analyzable array references allows a $c[j, 0]$ term to take on any (unknown) value that is invariant in loops $i_1, \cdots, i_h$. For example, a $c[j, 0]$ term is permitted to be a function of the index variable of a loop that encloses the innermost nest of loops $i_1, \cdots, i_h$.

Let $L$ equal the cache line size (in bytes) for the target architecture. The following discussion is presented for estimating the number of distinct *cache lines* accessed by an array reference. However, it can just as well be used to estimate the number of distinct *pages* accessed, by setting the value of $L$ to the page size (also in bytes).

*Computing the index range spanned by an analyzable subscript expression*

We first address the problem of estimating $\mathit{RANGE}(\mathbf{C}[j], \mathbf{T}) = \mathit{MaxValue}(\mathbf{C}[j], \mathbf{T}) - \mathit{MinValue}(\mathbf{C}[j], \mathbf{T})$, for a given coefficient vector $\mathbf{C}[j]$ and a tile of $t_1 \times \cdots \times t_h$ iterations specified by tile size vector $\mathbf{T} = (t_1, \cdots, t_h)$, where $\mathit{MaxValue}(\mathbf{C}[j], \mathbf{T})$ and $\mathit{MinValue}(\mathbf{C}[j], \mathbf{T})$ are the maximum and minimum values taken by the subscript expression $f_j$ in the iteration tile. $\mathit{RANGE}(\mathbf{C}[j], \mathbf{T})$ can be viewed as the length of the range of distinct *values* spanned in the tile by subscript expression $f_j$. For convenience in developing the memory cost functions, $\mathit{RANGE}(\mathbf{C}[j], \mathbf{T})$ is defined so that it equals zero when function $f_j$ takes on a single value across the tile [in which case $\mathit{MinValue}(\mathbf{C}[j], \mathbf{T}) = \mathit{MaxValue}(\mathbf{C}[j], \mathbf{T})$].

We use an approach similar to the rectangular form of Banerjee's inequality [33] to compute a symbolic expression for $\mathit{RANGE}(\mathbf{C}[j], \mathbf{T})$. Initially, assume that loops $i_1, \cdots, i_h$ all have a step of +1. Let $\mathit{LBound}_k$ and $\mathit{UBound}_k$ denote the lower and upper bounds of loop $i_k$ in a single tile of iterations. This means that tile size $t_k$ must equal $(\mathit{UBound}_k - \mathit{LBound}_k + 1)$. Then, $\mathit{RANGE}(\mathbf{C}[j], \mathbf{T})$ can be derived as follows (the notation $|x|$ denotes the absolute value of $x$, and the notation *cond ? true-expr:false-expr* denotes a conditional operator as in the C programming language):

$$\mathit{MaxValue}(\mathbf{C}[j], \mathbf{T}) = \sum_{k=1}^{h} (c[j, k] \ge 0 \ ? \ c[j, k] \times \mathit{UBound}_k : c[j, k] \times \mathit{LBound}_k),$$

$$MinValue(\mathbf{C}[j], \mathbf{T}) = \sum_{k=1}^{h} (c[j,k] \geq 0 \; ? \; c[j,k] \times LBound_k : c[j,k] \times UBound_k)$$

$$\Rightarrow RANGE(\mathbf{C}[j], \mathbf{T}) = MaxValue(\mathbf{C}, LoopSet) - MinValue(\mathbf{C}, LoopSet)$$

$$= \sum_{k=1}^{h} [c[j,k] \geq 0 \; ? \; c[j,k] \times (UBound_k - LBound_k) : c[j,k] \times (LBound_k UBound_k)]$$

$$= \sum_{k=1}^{h} |c[j,k]| \times (t_k - 1).$$

Interestingly, the value of $RANGE(\mathbf{C}[j], \mathbf{T})$ does not depend on the specific values of $LBound_k$ and $UBound_k$, and can therefore be represented as a linear polynomial function of only the $t_k$ tile sizes. Note that the value of *RANGE* is zero when all tile sizes equal one, since the subscript expression has only one distinct value in this case. To relax our assumption that all loops have a step of +1, we can extend the above expression for *RANGE* to $RANGE(\mathbf{C}[j], \mathbf{T}) = \sum_{k=1}^{h} |c[j,k] \times Step_k| \times (t_k - 1)$, where $Step_k$ is the (constant-valued) step for loop $i_k$, assuming that the tile sizes $t_1, \cdots, t_h$ are still defined as iteration counts.

*Computing the memory range spanned by an analyzable array reference*

Recall that an array reference, $\mathbf{A}[f_1, \cdots, f_m]$, is translated by a compiler to a linearized address [3] of the form

$$addr(\mathbf{A}[f_1, \cdots, f_m]) = StartAddr(\mathbf{A}) + \sum_{d=1}^{m} [f_d(i_1, \cdots, i_h) - LO_d(\mathbf{A})] DimStride(d),$$

where $LO_d(\mathbf{A})$ is the lower bound for indexing into dimension $d$ of array **A**, and $DimStride(d)$ is the *stride* in bytes for dimension $d$, i.e., the address increment for the array reference when the subscript value for dimension $d$ is increased by +1. Specifically, $DimStride(1)$ is the size of a single element of array **A**, and $DimStride(d) = DimStride(d-1) \times DimSize(d-1)$ when $d > 1$, where $DimSize(d-1)$ is the size of dimension $d-1$.

We now address the problem of estimating $MEMRANGE(j, \mathbf{C}, \mathbf{T})$, the number of bytes spanned by the first $j$ dimensions of an array reference with coefficient matrix **C** across a tile of $t_1 \times \cdots \times t_h$ iterations specified by tile size vector $\mathbf{T} = (t_1, \cdots, t_h)$. Specifically, $MEMRANGE(j, \mathbf{C}, \mathbf{T}) = MaxAddress(j, \mathbf{C}, \mathbf{T}) - MinAddress(j, \mathbf{C}, \mathbf{T})$, where $MaxAddress(j, \mathbf{C}, \mathbf{T})$ and $MinAddress(j, \mathbf{C}, \mathbf{T})$ are the maximum and minimum address values taken by $Address(\mathbf{A}[f_1, \cdots, f_j, LO_{j+1}, \cdots, LO_m])$ across the tile. As in the definition of *RANGE*, $MEMRANGE(j, \mathbf{C}, \mathbf{T})$ equals zero when functions $f_1, \cdots, f_j$ each take on a single value across the tile (this is the case when the memory range consists of a single array element). Note that the effect of parameter $j$ is to restrict *MEMRANGE* to the size of the memory range spanned by the first $j$ dimensions by ignoring subscript expressions $f_{j+1}, \cdots, f_m$.

Given the previous definition of *RANGE*, we can use the following identity to compute *MEMRANGE:*

$$MEMRANGE(j, \mathbf{C}, \mathbf{T}) = \sum_{d=1}^{j} RANGE(\mathbf{C}[d], \mathbf{T}) DimStride(d).$$

Recall that each $RANGE(\mathbf{C}[d], \mathbf{T})$ can be expressed as a linear polynomial of tile size variables. If, as is usually the case, the $DimStride(d)$ values are compile-time constants, we obtain a linear polynomial expression for $MEMRANGE(j, \mathbf{C}, \mathbf{T})$ as well. If $DimStride(d)$ is not a compile-time constant for some dimension $d$ (e.g., when **A** is a dynamically sized array), a default value such as 1000 is used for each unknown *DimSize* value when estimating $DimStride(d)$ using the identity $DimStride(d) = DimStride(d-1) \times DimSize(d-1)$. We can therefore assume that the actual/estimated *DimStride* values are compile-time constants; hence, $MEMRANGE(j, \mathbf{C}, \mathbf{T})$ can be treated as a linear polynomial in the tile sizes $t_1, \cdots, t_h$ for the purpose of memory cost analysis. Note that $DimStride(1)$ is always a compile-time constant because it is the size of a single array element.

*Estimating the number of distinct lines spanned by an analyzable array reference*

We first briefly review the approach from [32] to estimate the number of *distinct lines* for a single array reference. The upper bound estimate given in [32] for a one-dimensional subscript expression $f$ is

$$DL(f) \leq \min\left[\frac{(f^{hi} - f^{lo})}{g} + 1, \left\lceil\frac{(f^{hi} - f^{lo})}{L'}\right\rceil + 1\right],$$

where $g$ is the greatest common divisor of the linear coefficients in $f$, $f^{hi}$ and $f^{lo}$ are the maximum and minimum values taken by subscript expression $f$ across the entire loop nest, and $L' = L/DimStride(1)$ is the line size in units of array element size. For the special case when $L' = 1$, $DL(f) = [(f^{hi} - f^{lo})/g] + 1$ becomes an estimate of the number of *distinct accesses* made by the array reference. There are some special cases for which this estimate can be proved to be exact [34]. In practice,

the relative error of this estimation is small when, as is usually the case, the size of the $(f^{hi} - f^{lo})$ range is much larger than the size of the individual $c[j, k]$ coefficients.

For a multidimensional array reference, $\mathbf{A}[f_1, \cdots, f_m]$, the upper bound estimate given in [32] is

$$DL(f_1, \cdots, f_m) \leq DL(f_1) \times \prod_{j=2}^{m} \left( \frac{f_j^{hi} - f_j^{lo}}{g_j} + 1 \right).$$

This bound provides a reasonable estimate when $DimStride(2)$ is $\geq L$.

We can now address the problem of estimating $DL(j, \mathbf{C}, \mathbf{T})$, the number of distinct lines spanned by the first $j$ dimensions of an array reference with coefficient matrix $\mathbf{C}$ across a tile of $t_1 \times \cdots \times t_h$ iterations specified by tile size vector $\mathbf{T} = (t_1, \cdots, t_h)$. For the $j = 1$ case, we can rewrite the one-dimensional solution from [32] as

$$DL(1, \mathbf{C}, \mathbf{T}) \approx \min \left[ 1 + \frac{RANGE(\mathbf{C}[1], \mathbf{T})}{GCD(|c[1, 1]|, \cdots, |c[1, h]|)}, 1 + \frac{MEMRANGE(1, \mathbf{C}, \mathbf{T})}{L} \right],$$

since $RANGE(\mathbf{C}[1], \mathbf{T}) = f_1^{hi} - f_1^{lo}$ and $MEMRANGE(1, \mathbf{C}, \mathbf{T}) = (f_1^{hi} - f_1^{lo}) \times DimStride(1)$. The main difference is that $DL(1, \mathbf{C}, \mathbf{T})$ is a function of tile size variables, but $DL$ in [32] was defined for the entire loop nest. Note that we also replaced the ceiling ($\lceil \quad \rceil$) function with a continuous approximation; in essence, we approximated an $\lceil x/y \rceil$ term by $[1 + (x - 1)/y]$.

For multidimensional arrays, our approach extends the $DL$ solution from [32] by not relying on the $DimStride(2) \geq L$ assumption. This extension is important because we want to use the same memory cost function for counting lines and pages, and the $DimStride(2) \geq L$ assumption is less likely to hold when $L$ is set to the page size. Our solution is to estimate $DL(j, \mathbf{C}, \mathbf{T})$ for the first $j$ dimensions by using the following recurrence when $j > 1$:

$$DL(j, \mathbf{C}, \mathbf{T}) \approx \min \left[ 1 + \frac{RANGE(\mathbf{C}[j], \mathbf{T})}{GCD(|c[j, 1]|, \cdots, |c[j, h]|)} \times DL(j - 1, \mathbf{C}, \mathbf{T}), 1 + \frac{MEMRANGE(j, \mathbf{C}, \mathbf{T})}{L} \right].$$

For the sake of efficiency, we would like to simplify the above expression for $DL(j, \mathbf{C}, \mathbf{T})$ to a linear polynomial structure. In general this is hard to do, because the evaluation of the min function depends on the values of the $RANGE$ and $MEMRANGE$ terms, which in turn depend on the tile size variables, $t_1, \cdots, t_h$. However, we observe that the min function for the $j = 1$ case can be rewritten as

$$DL(1, \mathbf{C}, \mathbf{T}) \approx \min \left[ 1 + \frac{RANGE(\mathbf{C}[1], \mathbf{T})}{GCD(|c[1, 1]|, \cdots, |c[1, h]|)}, 1 + \frac{DimStride(1) \times RANGE(\mathbf{C}[1], \mathbf{T})}{L} \right],$$

and can be resolved at compile time by comparing $1/GCD(|c[1, 1]|, \cdots, |c[1, h]|)$ with $DimStride(1)/L$ and choosing the term with the smaller value, even though the value of $RANGE(\mathbf{C}[1], \mathbf{T})$ is unknown at compile time. This idea can be extended to the multidimensional case by choosing the first ($RANGE$) term in the recurrence if $1/GCD(\cdots)$ is smaller than $DimStride(j)/L$. Note that this approach will always select the $1/GCD(\cdots)$ term when $DimStride(j) > L$, which is consistent with the observation that there is no spatial locality in dimension $j$ in this case.

The above technique essentially specifies an algorithm for resolving the min function in the $DL$ recurrence at compile time, and building a symbolic expression for $DL(m, \mathbf{C}, \mathbf{T})$, the number of distinct lines accessed by the array reference. The symbolic expression is a linear polynomial in the tile size variables. This is the algorithm implemented in the ASTI transformer for building linear polynomial expressions for $DL_{\text{total}}(t_1, \cdots, t_h)$ and $DP_{\text{total}}(t_1, \cdots, t_h)$ for a given array reference; an example is provided later.

• *Estimating the memory cost for the entire loop nest*

In the previous section, we showed how to estimate the number of distinct lines accessed by a single array reference. To obtain the total number of (distinct) lines accessed by the entire loop body, we build a symbolic expression representing the sum of the individual $DL$ expressions. This sum of $DL$ terms is denoted by $DL_{\text{total}}$.

An important issue in computing $DL_{\text{total}}$ is that of considering cross-reference locality [35] for multiple references to the same array. If we ignore the potential overlap across multiple references and simply add in the contribution for each reference separately to $DL_{\text{total}}$, our memory cost estimation may become conservatively too large. Consider the following one-dimensional relaxation loop as an example:

```
do i1 = 2, n-1
    x(i1) = 0.3333 * (x(i1-1) + x(i1)
          + x(i1+1))
end do
```

This loop has substantial cross-reference locality, and the number of distinct lines accessed by the entire loop body ($\approx n/L$, for large $n$) is approximately the same as the number of distinct lines accessed by a single reference. Ignoring cross-reference locality in this case would lead to a memory cost estimation that is three times as large as the actual memory cost.

An algorithm for doing a precise estimation of the memory cost of multiple references to an array is given in [32]. However, in the worst case, the execution time of this precise estimation can be exponential in the number of references to the same array variable. We use a simpler estimation in the ASTI transformer that is efficient but still more precise than the conservative approach of ignoring overlap among multiple references to the same array. It is based on the idea of assuming 100% overlap among array references that are *uniformly generated* [36] (i.e., array references that have the same coefficient matrix, **C**), and zero overlap otherwise. This approach can be viewed as partitioning references to the same array variable into equivalence classes, and computing $DL_{\text{total}}$ by adding in the memory cost of only one representative array reference from each equivalence class. One refinement implemented in the ASTI transformer is to place two array references in the same equivalence class if and only if they have the same coefficient matrix and the difference between the invariant terms is less than some compiler-specified constant threshold. A future refinement that would be easy to implement is to add constant offsets to the terms in the $DL$ expression for the representative array reference, so as to reflect the contribution of other array references in the same equivalence class.

For the one-dimensional relaxation example, all three array references are placed in the same equivalence class because they have the same coefficient matrix and the difference between their invariant terms is at most two. The current approach computes the cost for only one reference in the equivalence class, thus yielding $DL_{\text{total}}(t_1) \approx 1 + (t_1 - 1)/L$, which is more precise than the cost that we would obtain by ignoring cross-reference locality. The additional refinement of extending the *RANGE* and *MEMRANGE* terms to include the extent of the invariant terms of other references in the same equivalence class would instead yield $DL_{\text{total}}(t_1) \approx 1 + (t_1 + 1)/L$. While this is closer to the actual value, the difference is not significant, especially for large $L$.

- *An example*

In this section, we illustrate the memory cost analysis procedure for the triply nested loop obtained after performing loop distribution on the matrix multiply–transpose example shown in Figure 4. The transformer uses the array-defs and array-uses input/output lists defined in Section 4 to enumerate all array references. After partitioning the array references into equivalence classes as described earlier, the memory cost analysis step builds a list of "representative" array references containing one reference from each equivalence class, and then builds a symbolic representation of the $DL_{\text{total}}$ cost function.

For the triply nested loop *I5–I4–I3* obtained after loop distribution on the matrix multiply–transpose example (Figure 11), the representative list of array references consists of `a(i1,i2)`, `b(i2,i3)`, and `c(i3,i1)`. Note that `a(i1,i2)` is only counted once. Using the techniques outlined in the previous sections, the memory cost analysis step then traverses the representative list and builds the following symbolic expression for $DL_{\text{total}}$, the numbers of distinct cache lines accessed by a tile of $t_1 \times t_2 \times t_3$ iterations,

$$\begin{aligned} DL_{\text{total}} &\approx \lceil 8t_1/L \rceil t_2 + \lceil 8t_2/L \rceil t_3 + \lceil 8t_3/L \rceil t_1 \\ &\approx (1 + 8(t_1 - 1)/L)t_2 + (1 + 8(t_2 - 1)/L)t_3 \\ &\quad + (1 + 8(t_3 - 1)/L)t_1 \\ &= (0.25t_1 + 0.75)t_2 + (0.25t_2 + 0.75)t_3 \\ &\quad + (0.25t_3 + 0.75)t_1 , \end{aligned}$$

since $L = 32$ bytes is the cache line size for the PowerPC 604 processor, and each array element is 8 bytes long (the array variables are declared with a real*8 base type).

A symbolic expression for the number of distinct pages, $DP_{\text{total}}$ can be constructed similarly to $DL_{\text{total}}$ by using $P = 4096$ bytes in place of $L$. For this example program, $DL_{\text{total}}$ and $DP_{\text{total}}$ are completely symmetric in $i_1$, $i_2$, $i_3$. However, this need not be true in general.

- *Memory cost analysis for a given unimodular transformation*

In this section, we briefly outline how memory cost analysis can be performed efficiently for a given unimodular loop transformation [37] such as loop permutation, reversal, or skewing [1]. This capability can be used to estimate memory costs of different unimodular transformations without requiring that the transformations be performed on the intermediate language prior to memory cost analysis.

Let **M** be the transformation matrix for a unimodular loop transformation that we want to consider applying to a set of perfectly nested loops with index variables $i_1, \cdots, i_h$, and let $i'_1, \cdots, i'_h$ denote the index variables of the loop nest that would be obtained if the transformation were performed. To perform memory cost analysis on the transformed loop nest, we must first rewrite the subscript expression for each dimension $j$ of an array reference, $f_j(i_1, \cdots, i_h) = c[j, 0] + \Sigma_{k=1}^{h} c[j, k]i_k$, into an equivalent function on the index variables of the transformed loop nest, $f'_j(i'_1, \cdots, i'_h) = c'[j, 0] + \Sigma_{k=1}^{h} c'[j, k]i'_k$. The unimodular transformation matrix **M** can be used to derive the transformed coefficients as follows:

$$c'[j, 0] = c[j, 0],$$

$$(c'[j, 1], \cdots, c'[j, h]) = (c[j, 1], \cdots, c[j, h]) \times \mathbf{M}^{-1},$$

where $\mathbf{M}^{-1}$ is the inverse of transformation matrix $\mathbf{M}$. Memory cost analysis can then be performed using the transformed coefficients $c'[*, *]$.

## 7. Locality optimization

This section gives a brief outline of the locality optimization step in the transformer and how it uses the memory cost functions described in Section 6. The locality optimization step performs transformations to minimize memory costs, in particular cache misses and TLB misses. Our algorithm uses iteration-reordering loop transformations (interchange, reversal, skewing, tiling) [29, 35] to move loops carrying locality inward, and is guided by the memory cost functions in selecting the optimized loop configuration. Locality optimization is attempted only on the innermost perfect loop nests of the program (including those revealed after loop distribution). Outer loop nests and loops with subroutine calls are currently not considered, since such loops often overflow the cache in a single iteration. Maximal innermost perfect loop nests are easily identified by a depth-first traversal of the LST.

The outputs of the locality optimization phase are a sequence of iteration-reordering transformations (typically, interchange and/or tiling, prefixed by enabling transformations such as reversal and skewing as needed) and a subset of inner transformed loops called the *locality group*. The locality group is defined to be the largest innermost subspace of the transformed iteration space that is guaranteed to incur no capacity or collision misses if it starts execution with a clean (empty) cache and a clean TLB. The locality group can be specified by two parameters, $(m, B)$, as follows:

1. $m \geq 0$, the number of innermost loops in the locality group. The parameter $m$ can be determined by estimating the number of distinct lines accessed by progressively larger subspaces of the iteration space (innermost loop, inner two loops, and so on) till we reach a case where no loops remain or one iteration of the $m$ + 1st loop overflows cache.
2. $B \geq 1$, the largest number of iterations (block size) of the outermost loop in the locality group such that no capacity or collision misses occur. Note that $B$ may, in general, only be a fraction of the total number of iterations in the outermost loop of the locality group. In other words, only the first $m - 1$ inner loops in the group span their complete iteration ranges without incurring capacity or collision misses. For a given $m$, $B$ can be estimated by solving a simple linear equation obtained by setting the estimated number of distinct lines to the number of lines available in cache, and applying the floor ($\lfloor\ \rfloor$) function to convert the interpolated $B$ value to an integer value.

Thus, the locality group consists of the $m$ innermost loops, with $B$ iterations of the outermost loop in the locality group and complete iterations of all other loops in the locality group. $B$ is properly defined only when $m \geq 1$; $m = 0$ indicates that a single iteration of the loop nest overflows the cache. The locality group can also be viewed as an innermost "tile" of the iteration space such that no capacity or collision misses occur within the tile. In fact, if the tiling transformation is performed on the loop nest, the locality group will be identical to be identical to a single tile from the tiling transformation.

Later transformations are not permitted to change the grouping of iterations in the locality group. However, they are free to reorder iterations within each locality group (e.g., during loop-invariant scalar replacement and loop unrolling) and/or reorder the execution sequence of locality groups (e.g., when parallelizing loops outside the locality group).

• *Algorithm for selecting an optimized loop ordering*

In this section, we describe how the transformer uses memory cost functions to automatically select an optimized loop ordering for a perfect loop nest. Given the polynomial expressions for $DL_{\text{total}}$ and $DP_{\text{total}}$ derived in Section 6, the total memory cost of compulsory misses for a hypothetical tile of $t_1 \times \cdots \times t_h$ is estimated as follows:

$$\begin{aligned} &COST_{\text{total}}(t_1, \cdots, t_h) \\ &\quad = (\textit{cache miss penalty}) * DL_{\text{total}}(t_1, \cdots, t_h) \\ &\qquad + (\textit{TLB miss penalty}) * DP_{\text{total}}(t_1, \cdots, t_h). \end{aligned}$$

Our objective is to minimize the *memory cost per iteration*, which is given by the function

$$F(t_1, \cdots, t_h) = \frac{COST_{\text{total}}(t_1, \cdots, t_h)}{t_1 \times \cdots \times t_h}.$$

In general, the minimization has to be done subject to the constraint that $DL_{\text{total}}$ not exceed the effective cache size and $DP_{\text{total}}$ not exceed the effective TLB size.

We have designed the following simple and efficient algorithm to choose a loop ordering that is optimized for locality:

1. Examine $(t_1, \cdots, t_h) = (1, \cdots, 1)$ as an initial solution. If $DL(1, \cdots, 1)$ exceeds the effective number of lines available in cache, or $DP(1, \cdots, 1)$ exceeds the effective number of page entries available in the TLB, no loop restructuring is performed, because each iteration will overflow the cache or TLB.
2. Otherwise, evaluate the $h$ partial derivatives of function $F$, $\delta F/\delta t_k$ at $(t_1, \cdots, t_h) = (1, \cdots, 1)$.
3. Return a suggested loop ordering in decreasing order of the partial derivative (slope) values. The loop with

the most negative slope should be placed in the innermost position, since it is likely to yield the largest reduction in memory cost, and so on. This is the ideal loop ordering based on memory costs. If the ideal ordering differs from the input ordering, dependence analysis is invoked to determine how closely the input loop ordering can be made to approach the ideal loop ordering.

As shown in Figure 11, there are two perfect loop nests in the matrix multiply–transpose example after the loop distribution step. Let us consider the first perfect loop nest, which corresponds to the following transformed code:

```
[2] do i1 = 1, n
[3]    do i2 = 1, n
[4]       a(i1,i2) = 0
[8]    end do
[9] end do
```

The memory cost functions are very simple for this single array reference, and are computed using estimated penalties of 17 cycles and 21 cycles for a cache miss and a TLB miss, respectively, on the PowerPC 604 processor (note that 0.001953 and 0.998047 are numerical approximations of 1/512 and 511/512):

$$DL_{\text{total}}(t_1, t_2) = (0.25t_1 + 0.75)t_2,$$

$$DP_{\text{total}}(t_1, t_2) = (0.001953t_1 + 0.998047)t_2$$

$$\Rightarrow COST_{\text{total}}(t_1, t_2) = 17 \times DL_{\text{total}}(t_1, t_2) + 21 \times DP_{\text{total}}(t_1, t_2)$$

$$= (4.25t_1t_2 + 12.75t_2) + (0.04t_1t_2 + 20.96t_2)$$

$$\Rightarrow F(t_1, t_2) = \frac{COST_{\text{total}}(t_1, t_2)}{t_1t_2} = \left(4.25 + \frac{12.75}{t_1}\right) + \left(0.04 + \frac{20.96}{t_1}\right).$$

Thus $\delta F/\delta t_1$ has a negative slope, but $\delta F/\delta t_2$ equals zero. Therefore, locality optimization will perform a loop interchange to bring the loop with negative slope into the innermost position, resulting in

```
[3] do i2 = 1, n
[2]    do i1 = 1, n
[4]       a(i1,i2) = 0
[8]    end do
[9] end do
```

The locality cost function for the triply nested loop in Figure 11 is completely symmetric in $t_1$, $t_2$, $t_3$, so there is no difference in the slopes for the three loops, and the original loop ordering is preserved. Since all three loops have negative slopes, this example is an ideal candidate for the tiling transformation discussed later.

• *Effective cache and TLB sizes*

We now outline how the transformer estimates effective cache and TLB sizes for use by the algorithm for selecting optimized tile sizes. For a fully associative cache, the effective cache size is identical to the actual cache size. However, caches in real processors are not fully associative, but instead have a limited degree of set associativity. Our solution is to compute a cache utilization efficiency that estimates the effective shrinking of a set-associative cache ith $S$ lines to an "equivalent" fully associative cache with $S' \leq S$ lines, where $S'$ is the effective cache size. The TLB utilization efficiency can be estimated by following the same approach with TLB parameters in place of cache parameters.

To start with, consider a simple array reference with a single stride $T$ as follows:

```
  DO 10 i = . . .
10 A(T*i + c) = . . .
```

We are interested in estimating

$\eta(T)$ = *cache utilization efficiency of stride T*
= *fraction of sets accessed over a large number of iterations.*

For example, the array reference, $A(i)$, has 100% efficiency; i.e., $\eta(1) = 1.0$, because it will access all of the sets in the cache. However, it is well known that a stride value that is a power of two leads to poor efficiency. For example, the reference $A(32768*i)$ will repeatedly access a single set in the cache. Since the (first-level) cache in the PowerPC 604 processor has 512 sets, the efficiency for this reference is only $\eta(32768) = 1/512$.

We estimate $\eta(T)$ by considering three possible cases, as outlined below. More precise estimates are possible if the cache block alignment offset and number of loop iterations are also known at compile time.

*Case 1*: $T \leq L \Rightarrow \eta(T) = 1.0$.

*Case 2*: $T$ is a multiple of $L \Rightarrow \eta(T) = 1/\text{GCD}(T/L, S)$.

*Case 3*: Otherwise, we find the smallest $n \geq 1$ that satisfies [32],

$$\text{mod}(n \times T, L \times S) < L \text{ or } \text{mod}(n \times T, L \times S) > L \times S - L.$$

In this case, every $n$th access can map to the same set, so we conservatively estimate $\eta(T) = n/S$.

Our overall approach to estimating effective cache size is as follows, assuming that the actual cache is a $k$-way *set-associative* cache with $S$ sets (we set $k = 1$ to do the analysis for a direct-mapped cache):

1. Compute the locality group, $(m, B)$, for the loop nest, assuming a fully associative cache containing $k \times S$ lines.
2. For each loop index variable $i_j$, such that $h - m < j < m$, examine each array reference that contains $i_j$, and estimate $\eta(T)$ for the stride $T$ of $i_j$ in that array reference. This step restricts the estimation of cache utilization efficiency to the $m$ innermost loops, where $m$ is the number of loops in the locality group computed in step 1.
3. For each array variable, choose the *minimum* among the estimated $\eta(T)$ values from step 2 as the cache utilization efficiency for the array variable.
4. Estimate effective number of sets as $S' = \lfloor \eta_{avg/min} S \rfloor$, where $\eta_{avg/min}$ is the average value taken over all array variables of the minimal cache utilization efficiency obtained in step 3.
5. Set effective cache size to $k \times S' \times L$ bytes.

In the above steps, the minimum $\eta(T)$ value is used for each array variable, so that the estimation is conservative; $\eta_{avg/min}$ is estimated as the average of the min values based on the assumption that cache usage is divided equally among all of the array variables. If needed, the average can be refined to a weighted mean by considering a nonuniform partitioning of the cache for different array variables.

- *Algorithm for selecting optimized tile sizes*

If multiple loops are found to have a negative slope ($\delta F/\delta t_k < 0$) in the algorithm for selecting an optimized loop ordering, the locality of the loop nest can often be further improved by tiling the loops that have negative slope. The key problem that must then be addressed by the transformer is the selection of tile sizes, which we formulate as a constrained optimization problem:

- As before, the objective function to be minimized is $F(t_1, \cdots, t_h) = COST_{total}/(t_1 \times \cdots \times t_h)$, the average cache and TLB miss overhead per iteration of the tiled loop.
- The constraints to be satisfied by the solution are the following:
  - Each $t_k$ must be integer-valued and must be in the range $1 \leq t_k \leq Ubound_k$. A default value such as 1000 is used for $Ubound_k$ if the number of loop iterations is not known at compile time.
  - $DL(t_1, \cdots, t_h) \leq ECS$. The number of distinct cache lines accessed in a tile must not exceed the effective cache size.
  - $DP(t_1, \cdots, t_h) \leq ETS$. The number of distinct virtual pages accessed in a tile must not exceed the effective TLB size.

We have designed an efficient constant-time algorithm [38] to solve this constrained optimization problem for the case of two loops with negative slope. If there are $N > 2$ loops with negative slope that are eligible for tiling, standard logarithmic search techniques are used on $N - 2$ variables by invoking the two-variable solution at each search point. In practice, $N$ is rarely >3, and the $N = 3$ case is solved efficiently by searching on a single variable.

When the locality optimization step was performed on the matrix multiply–transpose example for the PowerPC 604 processor, the binding constraint was the data cache size constraint. The compiler optimistically assumes that $\eta_{avg/min} = 1.0$ when the dimension size $n$ is not known at compile time. (A safer approach would be to assume a smaller value of $\eta_{avg/min}$ for unknown dimension sizes, but that is not in the current implementation.) Therefore, the effective cache size is estimated as being the actual cache size, $4 \times 512 = 2048$ lines. This yields $DL_{total}(t_1, t_2, t_3) \leq 2048$ as the cache size constraint. Our procedure for selecting tile sizes returned $t_1 = 50$, $t_2 = 51$, $t_3 = 51$ as the optimized solution for this example. We can verify that this solution satisfies the cache size constraint by using the expression described previously for $DL_{total}(t_1, t_2, t_3)$ and computing $DL_{total}(50, 51, 51) = 2039.25$, which is <2048. We can also see that $DL_{total}(51, 51, 51) = 2065.50$, which shows that a slightly large tile overflows the cache. The optimized tile sizes are (almost) equal for this example because the cost function is completely symmetric in $i_1, i_2, i_3$. Other cost functions can lead to different tile sizes for different loops.

Therefore, the cumulative transformations performed by the locality optimization step for the matrix multiply–transpose example is a loop interchange for the first loop nest and loop tiling for the second loop nest, as shown in **Figure 12**. (As before, the figure shows the transformed source for convenience, though the transformer only updates the LSG.)

## 8. Loop fusion

As discussed in Section 5, the loop fusion transformation is important for removing the overhead of unnecessary loop distribution. In addition, it can be useful for fusing loops that were not distributed, e.g., conformable loop nests arising from FORTRAN 90 array language statements [6]. Just as with loop distribution, the key data structure that is used to guide loop fusion is the loop dependence graph (LDG) in the LSG. However, one important difference is that loop distribution is performed on loops from the inside out, and loop fusion is performed from the outside in.

For simplicity, the implementation of loop fusion in the transformer focused on optimizing the common case of a program region that represents a set of $k$ adjacent conformable and identically control-dependent perfect

```
do i2=1,n,1
  do i1=1,n,1
    a(i1,i2) = 0d0
  end do
end do


do bb$_12=1,n,50
  do bb$_13=1,n,51
    do bb$_14=1,n,51
      do i1=MAX0(1,bb$_12),MIN0(n,49 + bb$_12)
        do i2=MAX0(1,bb$_13),MIN0(n,50 + bb$_13)
          do i3=MAX0(1,bb$_14),MIN0(n,50 + bb$_14),1
            a(i1,i2) = a(i1,i2) + b(i2,i3) * c(i3,i1)
          end do
        end do
      end do
    end do
  end do
end do
```

**Figure 12**

Locality optimization for matrix multiply–transpose example.

loop nests. Two loop nests are said to be conformable if their corresponding loops have identical iteration lengths (loop bounds). Two loop nests (or, more generally, statements) are said to be identically control-dependent if they have the same set of control conditions [20], i.e., the same set of (node, label) pairs as control dependence predecessors. However, it is straightforward to extend this approach to a more general region which includes nonconformable loop nests and the presence of control flow across LDG nodes.

Each edge in the LDG is marked as being contractable or noncontractable. The source and destination loop nests of a noncontractable LDG edge cannot be fused, because this would violate the data dependence test for loop fusion [1]. This test is formally specified by the algorithm *Contractable*($L$, $M$) in **Figure 13**, which outlines this data dependence test for nodes (loop nests) $L$ and $M$ such that there is at least one LDG edge from $L$ to $M$. If the algorithm returns a *false* value, nodes $L$ and $M$ cannot be fused, and all edges from $L$ to $M$ are marked as noncontractable. It is not necessary to call the algorithm *Contractable*($L$, $M$) when there is no LDG edge from $L$ to $M$, because in that case there cannot possibly be any data interference between loop nests $L$ and $M$.

A *fusion partition* of an LDG is a partition of the set of nodes into disjoint *fusion clusters*; each fusion cluster represents a set of loop nests to be fused. A fusion partition is legal if and only if 1) for each noncontractable edge, the source and destination nodes belong to distinct fusion clusters; and 2) the reduced graph defined by the fusion partition is acyclic. Given an LDG and a legal fusion partition, the output code configuration can be obtained by fusing all loops that belong to the same fusion cluster and by ordering the fused loops according to some topological sort defined by the edges in the reduced cluster graph.

We also assume that there is a weight $w_{ij} = w_{ji}$ associated with each pair of nodes $i$ and $j$, representing the cost savings that would be obtained if loops $i$ and $j$ were fused. For convenience, we assume that $w_{ii} = 0$ for all $i$, and that $w_{ij} = w_{ji} = 0$ for each noncontractable edge

**Algorithm** *Contractable*($L$,$M$):

**Input:** Two input loop nests, $L$ and $M$, with at least one LDG edge from $L$ to $M$.

**Output:** True/false value indicating if the LDG edges from $L$ to $M$ are contractable.

**begin**

1. **if** loop nests $L$ and $M$ are not conformable
   **then**
   **return false**

2. /* Test for loop-carried control dependence */
   **if** loop nest $L$ or loop nest $M$ contains a premature loop exit
   **then**
   **return false**

3. /* Test for illegal scalar loop-carried data dependence (it is recommended that the optimizer perform its scalar variable renaming and privatization transformations prior to this test). */
   **if** $\exists$ scalar variable $S$ accessed in both loop nest $L$ and loop nest $M$ such that at least one access is a def
   **then**
   **return false**

4. /* Check for illegal array data dependence or loop-carried array data dependence. */
   **for each** array variable $A$ accessed in loop nest $L$ and loop nest $M$ such that at least one access is a def
   **do**

   4.1. **let** $Def_L$, $Use_L$, $Def_M$, $Use_M$ be the sets of defs and uses to array variable $A$ in loop nests $L$ and $M$

   4.2. /* Compute $V$, the set of fusion loop-carried data dependence vectors = union of output, flow, and anti dependence vector sets. We assume that *DDtest(Ref1,Ref2)* returns the union of dependence vectors obtained from all pairs of source array references $r1 \in Ref1$ and destination array references $r2 \in Ref2$. */

   $$V := DDtest(Def_L, Def_M) \cup DDtest(Def_L, Use_M) \cup DDtest(Use_L, Def_M)$$

   4.3. /* Test for illegal array data dependence. */
   **if** $V$ contains a lexicographically negative dependence vector
   **then**
   **return false**

   **end for**

5. **return true**

**end**

**Figure 13**

The algorithm *Contractable*.

$(i, j)$[4]. Weights may also take into account execution probabilities for conditionals in the loop bodies [39].

We can now model weighted loop fusion as the following optimization problem: *Given an LDG, find a legal fusion partition that has minimum total intercluster weight, i.e., minimum total weight of node-pairs that cross cluster boundaries.*

It has been shown in past work that the weighted loop fusion problem is NP-hard [40]. The implementation in the ASTI transformer adds the constraint that no fusion cluster should lead to a fused loop in which a single

[4]Note that weights of noncontractable edges have no impact on the selection of optimal fusion partitions, since these weights are always included in the total intercluster weight.

```
do bb$_12=1,n,50
  do bb$_13=1,n,51
    do bb$_14=1,n,51
      do i1=MAX0(1,bb$_12),MIN0(n,49 + bb$_12)
        do i2=MAX0(1,bb$_13),MIN0(n,50 + bb$_13)
          ScRep_19 = a(i1,i2)
          do i3=MAX0(1,bb$_14),MIN0(n,50 + bb$_14),1
            ScRep_19 = ScRep_19 + b(i2,i3) * c(i3,i1)
          end do
          a(i1,i2) = ScRep_19
        end do
      end do
    end do
  end do
end do
return
end
```

**Figure 14**

Source-level listing of transformed matrix multiply–transpose program after scalar replacement.

iteration spills registers if none of the original loops in the fusion cluster spilled registers, and uses a greedy merge heuristic to solve this problem [41]. A new algorithm for optimal weighted loop fusion has recently been designed [30]. It is based on an integer programming formulation that is efficient enough for use in a production-quality compiler. We plan to experiment with this new approach to see how much more effective the optimal weighted loop fusion is compared to the greedy heuristic.

## 9. Loop-invariant scalar replacement

The ASTI transformer performs scalar replacement for loop-invariant array references so as to reduce the number of memory accesses in a loop nest. The scalar replacement transformation [42] uses array analysis information to selectively replace array references with compiler-generated scalar temporaries. This transformation is performed so as to enable more effective register allocation of array references by the compiler back end. The ASTI transformer takes this idea one step further by also performing loop interchange within the locality group so as to move the loop with the largest number of loop-invariant array references and operations to the innermost position.

The outputs of the loop-invariant scalar replacement phase are as follows:

1. A new sequence of iteration-reordering loop transformations that is appended to the existing sequence.
2. A scalar replacement interface data structure with one entry per scalar temporary containing the following information:

- List of array references to be replaced by this scalar variable.
- Number of transformed inner loops in which the array references are invariant.
- Information on whether the scalar temporary must be loaded/initialized from the array element location on loop entry (liveness on entry), and/or stored into the array element location on loop exit (liveness on exit).

Continuing with the matrix multiply–transpose example from the previous sections, we see that the two references to `a(i1,i2)` are invariant in loop `i3`, reference `b(i2,i3)` is invariant in loop `i1`, and `c(i3,i1)` is invariant in loop `i2`. Since the scalar replacement savings for `a(i1,i2)` consists of a load and a store instruction, as opposed to just a load instruction for the other array references, this phase decides to keep loop `i3` in the innermost position in the locality group, and then creates a scalar replacement entry for the `a(i1,i2)` references. **Figure 14** shows a source-level listing of the transformed code for the `i1±i2±i3` loop nest. The transformations performed include the interchange and tiling from locality optimization as well as scalar replacement from this phase. The loop-unrolling transformation described in Section 10 is used later by the transformer to exploit the loop-invariance of the other array references.

The ASTI transformer extends the approach of interchanging loops so as to maximize the savings from scalar-replaced loads and stores to other loop-invariant operations on array references[5] as well. The most notable case is when a divisor is a loop-invariant array reference. In this case, an extra savings is obtained by computing the reciprocal as a loop-invariant and reducing the strength of the divide operation inside the loop to a multiplication by the loop-invariant reciprocal. An example of this transformation is shown in **Figure 15**. Though this transformation is algebraically correct, it can change the bitwise floating-point result that is obtained, so it is performed only when permitted by user options.

## 10. Loop unrolling

Loop unrolling is similar to tiling in that it divides the iteration space into small tiles. The iterations in an unrolled "tile" execute copies of the loop body that have been expanded (unrolled) in place, rather than executing inner control loops as in tiling for cache locality. The benefits of loop unrolling can come from enhanced register locality, enhanced instruction-level parallelism,

[5] Loop-invariant operations on scalar variables are invariant in the entire loop nest and hence insensitive to loop ordering.

and reduced loop overhead. However, the unroll factors must be carefully selected to avoid run-time performance degradation from excessive unrolling due to register spills and code size expansion, as well as the inconvenience of excessively long compile times.

The approach taken by the transformer to select unroll factors is similar to the approach taken in Section 7 for selecting tile sizes. The constrained optimization problem to be solved in this case is as follows:

- The objective function to be minimized is the amortized execution time for a single iteration of the original loop nest, when taking into account the savings from unrolling. In contrast to the case of loop tiling, this cost function cannot be effectively approximated as a rational polynomial of unroll factors. Min and max operators are used in this cost function to properly model cross-reference register reuse, as well as hardware characteristics such as the IBM POWER2* processor's quad-load/quad-store instructions [43].
- The constraints to be satisfied by the solution are the following:
  - Each unroll factor must be integer-valued and is bounded above by a compiler-specified constant for the sake of compile-time efficiency.
  - The number of distinct floating-point array references in the unrolled loop body must not exceed the effective number of floating-point registers available.
  - The number of distinct integer array references and array index registers in the unrolled loop body must not exceed the effective number of fixed-point registers available.

Continuing with the matrix multiply–transpose example from the previous sections, we see that both loops `i1` and `i2` carry register locality that can be exploited by unrolling. For this example, loop unrolling is constrained by the effective number of floating-point registers available, which is assumed to be 28 to allow the compiler at least four registers to use for code generation. Using an approach similar to the memory cost analysis in Section 6, the number of distinct floating-point registers required by a $u_1 \times u_2 \times u_3$ unroll configuration is given by $DR(u_1, u_2, u_3) = u_1u_2 + (u_1u_3 + u_2u_3)$, where $u_1$, $u_2$, and $u_3$ are the unroll factors for loops `i1`, `i2`, and `i3`, respectively. The $u_1u_2$ term represents duplicated copies of the scalar replacement temporary for array `a` after unrolling, and the $(u_1u_3 + u_2u_3)$ term represents the number of registers required to hold values of arrays `b` and `c`. The number of distinct load instructions in an unrolled iteration is also $(u_1u_3 + u_2u_3)$. Ignoring possible instruction-level parallelism benefits, the objective function to be minimized for this example is simply the amortized number of loads per iteration,

$$F(u_1, u_2, u_3) = \frac{u_1u_3 + u_2u_3}{u_1u_2u_3} = \frac{1}{u_2} + \frac{1}{u_1}.$$

We see from function $F$ that we can get an improvement by increasing $u_1$ or $u_2$, but increasing $u_3$ leaves the value of function $F$ unchanged.

The unroll factors selected by the transformer for this objective function are $u_1 = 4$, $u_2 = 4$, and $u_3 = 1$. This leads to a total of $DR(4, 4, 1) = 24$ floating-point registers used in the unrolled loop body, which is less than the limit of 28. Increasing either $u_1$ or $u_2$ to 5 makes $DR$ equal 29, which exceeds the limit.

A source-level listing of the transformed code obtained from the transformation report generated by the XL FORTRAN compiler is shown in Appendix A. The transformations performed include loop tiling and scalar replacement from the previous sections, and loop unrolling as described in this section. Notice the "remainder loops" required for correct code generation for the loop-unrolling transformation when the

```
Original program
----------------
      subroutine foo(a,b,n)
      real*8 a(n,n), b(n,n), c(n)

      do j = 1, n
         do i = 1, n
            a(i,j) = b(i,j)/c(j)
         end do
      end do
      end


After divide replacement
------------------------
      subroutine foo(a,b,n)
      real*8 a(n,n), b(n,n), c(n), DivRep_13

      do j = 1, n
         DivRep_13 = 1.0d0 / c(j)
         do i = 1, n
            a(i,j) = b(i,j) * DivRep_13
         end do
      end do
      end
```

**Figure 15**

Example of divide replacement.

compiler does not know whether a loop iteration count is a multiple of its unroll factor.

## 11. Data dependence analysis

The data dependence tester provides data dependence information for pairs of array subscript references; such information is used by the transformer to test the legality of various proposed transformations. The data dependence tester is designed on a *demand-driven* model. It is invoked when, and only when, it is deemed necessary by the transformer, and at the level of detail that is specified by the transformer. One of the main advantages is the simplification that arises from not having to worry about transforming a data dependence graph after loop transformations such as distribution and fusion. An additional advantage of demand-driven data dependence testing is compile-time savings for cases in which the transformer does not have to compute data dependences, e.g., when the transformer's cost estimation shows that the desired configuration is the same as the original configuration (a frequent case in a well-tuned program). Inherent in the demand-driven model is the possibility that the data dependence tester may sometimes be called repeatedly (hence unnecessarily) for the same pair of array subscript references. In practice this is not a problem, since the cost of the extra calls is typically small compared to the rest of the compile time for these cases.

The input to the data dependence tester comprises

1. Subscript tables for a pair of array subscript references.
2. Loop-bound tables for lower/upper bounds of the common loops enclosing both array subscript references.
3. An input set of dependence vectors which specify conditions under which testing should be performed.
4. A flag to indicate the action that should be taken for implausible (lexicographically negative) dependence vectors found.

The output of the data dependence tester comprises

1. A set of data dependence vectors specifying conditions under which a dependence was found.
2. A flag specifying whether the dependence vectors in this set are exact or conservative.
3. Extra information when the dependence is known to have some additional constraints, e.g., dependence holds for only one iteration of the loop, or for only one pair of iterations in the loop, or if there is a split-point.

Before applying any of the data dependence testing algorithms, some preparation/normalization must be done. In general, algorithms in the data dependence tester require that loops with constant steps have already been at least seminormalized to a stride of +1 (but not necessarily a lower bound of 1), and that symbolic expressions appearing in the array subscript references and loop bounds have been decomposed into constant and symbolic coefficients of the loop index variables. The analyzer component of the ASTI optimizer builds a subscript table for each array reference and a loop bounds table for each loop-bound expression, so that this information is available to the data dependence tester when the transformer is invoked.

- *Data dependence vectors*

Following prior work in the area of iteration-reordering loop transformations, we represent the loop-carried data dependence constraints for a loop nest by a set of dependence vectors **D**. A dependence vector for a loop nest of size $n$ is an $n$-tuple, $\vec{d} = (d_1, \cdots, d_n)$, where entry $d_k$ corresponds to the $k$th loop (counting from outermost to innermost). In practice, there are two kinds of values for $d_k$ that are of interest [44]:

1. *Distance*—$d_k$ is an integer value, $d_k = y \in Z$.
2. *Direction*—$d_k$ is one of the six values + (positive), − (negative), $+$ (nonnegative), $-$ (nonpositive), $\pm$ (nonzero), $*$ (any)[6].

Let us use $S(d_k)$ to denote the set of integer values that is represented by $d_k$. In the case of a distance value, $S(d_k)$ is a singleton set, i.e., $S(d_k) = \{y\}$. When $d_k$ is one of the six possible direction values, $S(d_k) = \{x | x \in Z \wedge x$'s sign is contained in $d_k\}$. The set of integer tuples denoted by dependence vector $\vec{d} = \{d_1, \cdots, d_n\}$ is given by tuples($\vec{d}$) $= S(d_1) \times \cdots \times S(d_k)$, i.e., the Cartesian product of the integer sets corresponding to $d_1, \cdots, d_k$. If **D** is the set of all of the dependence vectors for a loop nest, tuples(**D**) is simply the union of the tuple sets for each dependence vector in **D**, that is, tuples(**D**) $= \cup_{\vec{d} \in \mathbf{D}}$ tuples($\vec{d}$). An integer tuple, $\vec{a} = (a_1, \cdots, a_n)$, is lexicographically negative (positive) if and only if its first nonzero element, $a_i$, is negative (positive); i.e., $a_j = 0$, $\forall 1 \leq j < i$ and $a_i < 0$ ($a_i > 0$). Finally, an execution instance of a loop nest of size $n$ is defined as an $n$-tuple, $\vec{a} = (a_1, \cdots, a_n)$, where $a_k$ specifies the iteration number of the $k$th loop.

The set of dependence vectors for a loop nest, **D**, enforces a partial order on the execution instances of the loop nest; for any two distinct execution instances $\vec{a}$ and $\vec{b}$, if their difference $\vec{b} - \vec{a}$ belongs to tuples(**D**), instance $\vec{b}$ must be executed after instance $\vec{a}$. This partial order summarizes all of the iteration-reordering constraints

[6] An alternate notation for the six direction values is to use the relational operators, $<$, $>$, $\leq$, $\geq$, $\neq$, and $*$ respectively [1]. We do not represent an $=$ direction in our framework because it is equivalent to a zero distance.

imposed by data dependences. We assume that the original execution order satisfies this partial order. This implies that tuples(**D**) cannot contain a lexicographically negative integer tuple for the original loop nest (otherwise the original execution order can be shown to violate some data dependence constraint). Further, if **D′** is the set of dependence vectors that result after a loop transformation is applied, and tuples(**D′**) contains a lexicographically negative tuple, the transformed loop nest can be shown to violate some data dependence constraint, and so the transformation must be illegal. This fact forms the basis of the data dependence legality test for iteration-reordering loop transformations.

- *Recurrence recognition*

When a recurrence operation is encoded using a FORTRAN loop, the order in which the values are accumulated becomes fixed. For floating-point data, if the values are accumulated in a different order, the numerical result may be different, even though the result is algebraically equivalent to the original. In many cases, the order in which values are accumulated is not important to the programmer, and the programmer may communicate this to the compiler via appropriate options and directives. In the case of recurrences that arise from the scalarization of FORTRAN 90 intrinsic functions, the language specification allows the language translator to accumulate the result in any order. In all of these cases, recognition/identification of recurrence constructs by the compiler is important, since the transformer can then ignore loop-carried dependences carried by accumulator variables and thus select a transformation that might not otherwise have been legal. For some transformations (notably loop unrolling and loop parallelization), some additional transcription support is required to generate correct and efficient code for recurrences.

The recurrence recognition performed by the ASTI transformer identifies definitions that correspond to accumulator updates in a recurrence, and also determines the set of enclosing loops for which the definition constitutes a reduction. A recurrence may span multiple statements in the loop body.

For example, the variable s is used both as an accumulator and as an intraloop temporary in the following loop nest:

```
do i = 1, n
   do j = 1, n
      s = s + a(i,j)
      s = s + b(i,j)
   end do
end do
```

Only the first use of s and the last definition of s are identified as references to the accumulator. Without recurrence recognition, the transformer has to make the worst-case assumption that the initial set of loop-carried dependence vectors is $\mathbf{D} = \{(+, *), (=, +)\}$ due to the definitions and uses of variable s. After recognizing the recurrence, the transformer will compute $\mathbf{D} = \emptyset$ and then proceed to select a loop interchange transformation for improved cache locality. Appendix B shows the source-level listing of the transformed loop nest for this example, after recurrence recognition enabled loop interchange and loop unrolling.

## 12. Related work

To the best of our knowledge, the ASTI transformer is the first system to perform automatic selection of this wide range of transformations using a cost-based framework. The KAP [13] and VAST [14] preprocessors have made high-order transformations available to users for over a decade now. While they provided a great convenience for users, most of the experience with these preprocessors has been with using specialized options for different applications rather than with automatic cost-based selection of high-order transformations. As mentioned earlier, the transformations implemented in the ASTI transformer have all been proposed in past work, but without any algorithms for applying them collectively. Because of space limitations, we discuss only a representative subset of prior work that is most relevant to our paper.

Wolf and Lam [35] propose an algorithm that improves the locality of a loop nest by transforming the code via interchange, reversal, skewing, and tiling based on a mathematical formulation of reuse and locality, and a loop transformation theory that unifies the various transforms as unimodular transformations. This algorithm has been implemented in the SUIF research compiler [45]. Their goal is to find the best combination of loop interchange, skewing, reversal, and tiling that maximizes the data locality within loop nests, subject to the constraints of direction and distance vectors. Unlike our approach of locality analysis by estimating a count of the number of cache misses, they measure the locality of a transformed code by intersecting the reuse vector space with the localized vector space. We expect our cost estimation to be more accurate because it takes into account more factors than just the number of loops carrying reuse, and gives an estimate of the number of distinct cache lines accessed. Also, the locality transformation in their framework is limited to at most two steps consisting of a unimodular transformation followed by a tiling transformation. In contrast, the ASTI transformer supports a general sequence of iteration-reordering loop transformations combined with loop

distribution, fusion, unrolling, and scalar replacement, as described in this paper.

In [46], Bailey presents a thorough analysis of the behavior of a direct-mapped cache with strided data access, and gives a formula for estimating cache efficiency. Using cache efficiency, the compiler can detect unfavorable strides and automatically adjust array dimensions through padding techniques. However, this work does not address the overall problem of locality analysis (estimating the number of misses) and optimization of array references in a loop nest.

In the area of automatic selection of tile sizes, Schreiber and Dongarra [47] address the problem of deriving an optimized tiled (hyperparallelepiped) iteration space to minimize communication traffic. They assume a restricted loop/array model in which the iteration space and data space are isomorphic. Most of the paper is devoted to solving the problem for the special case when all block sizes are equal. Their technique is very different from ours, and also too time-consuming for use in production-quality compilers.

The inspiration for the loop structure graph originated from the forward control dependence graph (FCDG) used in the PTRAN system to represent interval structure and statement parallelism [23–25]. The FCDG is a variant of the PDG in which the program's loop structure is made evident by control dependences that are derived partly from pseudo-control-flow edges connected to interval pre-header and post-exit nodes. We observed three limitations with the FCDG representation:

1. The FCDG is not well defined for a program with an irreducible control flow graph; at the very least, irreducibility causes the FCDG to be cyclic, whereas all of the algorithms that use the FCDG assume that it is acyclic.
2. Though the FCDG facilitates the identification of statement parallelism, it does not lend itself to performing loop transformations.
3. The creation of pseudo-control-flow edges for the FCDG can lead to less precise data flow information and hence less precise analyzer information (constant propagation, induction variables, etc.).

The loop structure graph remedies the above problems with the FCDG as follows:

1. An irreducible region is merged along with its smallest containing single-entry region into a single loop node, thus isolating it from the other loops in the LST, which remain eligible for all transformations and optimizations.
2. Loop transformations are easily performed by local updates to the LST nodes and the LCFG and LDG graphs. For the common case of iteration-reordering loop transformations, the update is minimal since the loop body is unchanged [29].
3. No pseudo-control-flow edges are created in the LSG, since the program's loop structure is made self-evident by the hierarchical structure of the LSG.

Several papers related to iteration-reordering transformations have been published in the literature. Lamport [44] introduces the hyperplane method and the coordinate method for parallel execution of iterations in a loop nest. Both methods are special forms of iteration-reordering transformations. The framework used in [44] included dependence vectors that contain distance or direction values, and iteration-reordering transformations that can be represented by $Z^n \mapsto Z^n$ linear mappings. Further, the legality test for a linear mapping was based on the existence of a lexicographically negative tuple in the set of transformed dependence vectors. However, the focus of the paper was on the two methods for rewriting a sequential loop nest into a form containing parallel loops, and the framework was developed only to the extent required by these transformations. The framework in the ASTI transformer is much more general—we support linear and nonlinear transformations, allow input and output loop nest sizes to be different, and permit all iteration-reordering transformations to be composed together in a general way.

Wolfe [1] introduces a comprehensive data dependence graph, with edges labeled by direction vectors, as the basis for a loop-transformation framework. Several iteration-reordering transformations were supported by this framework, e.g., loop interchanging, iteration space tiling (blocking), loop skewing, vectorization, and concurrentization. However, each transformation had its own special legality test based on the direction vectors and on the nature of loop-bound expressions. Our framework is more general in that we treat transformations as independent entities, separate from the data dependence graph.

Irigoin and Triolet [48] describe a framework for iteration-reordering transformations based on supernode partitioning, an aggregation technique achieved by hyperplane partitioning, followed by iteration space tiling across hyperplane boundaries. In this framework, data dependences are represented by dependence cones rather than dependence vectors. They also provide a general code-generation algorithm for any linear transformation that corresponds to a unimodular change of basis [49]. Their framework incorporates loop interchange, hyperplane partitioning, and loop tiling (blocking) in a unified way, for loop nests with linear bounds expressions. Our framework takes its inspiration from this kind of unified approach to loop transformations, but distinguishes

itself by efficient cost-based algorithms for automatic selection of high-order transformations.

Finally, McKinley, Carr, and Tseng [10] study improvements in data locality by using the loop permutation, fusion, distribution, and reversal transformations. These transformations form a subset of the transformations implemented in the ASTI transformer (their study does not include scalar replacement, loop tiling, and loop unrolling). Their experimental results show wide applicability of these locality-improving transformations for existing FORTRAN 77 and FORTRAN 90 programs.

## 13. Conclusions

In this paper, we described how the transformer component of the ASTI optimizer automatically selects high-order transformations for a given input program and a target uniprocessor, so as to improve utilization of the memory hierarchy (including cache and registers) and instruction-level parallelism. The ASTI transformer is used daily in production mode in the latest IBM XLF product compilers for RS/6000 and PowerPC uniprocessors and SMPs, and in the IBM XLHPF product compiler for the SP distributed-memory multiprocessor. The experience in building the transformer has established the feasibility of pursuing a quantitive approach in building optimizing compilers that deliver effective and robust optimizations for a wide range of programs and target architectures. It also established the feasibility of building a compiler framework that supports incremental and demand-driven optimization with low compile-time overheads. To the best of our knowledge, the ASTI transformer is the first system built that supports automatic selection of the wide range of transformations described in this paper, using a cost-based framework.

## Appendix A: Transformed matrix multiply–transpose program after loop unrolling

```
1585-103 *** Loop Transformation Report ***

subroutine mmt(a,b,c,n)

. . .

do i2=1,n,1
  do i1=1,n,1
    a(i1,i2)=0d0
  end do
end do

do bb$_12=1,n,50
  do bb$_13=1,n,51
    do bb$_14=1,n,51
do i1=MAX0(1,bb$_12),MIN0(n,49+b$_12)-3,4
do i2=MAX0(1,bb$_13),MIN0(n,50+bb$_13)-3,4
  ScRep_19=a(i1,i2)
  ScRep_20=a(i1+1,i2)
  ScRep_21=a(i1+2,i2)
  ScRep_22=a(i1+3,i2)
  ScRep_23=a(i1,i2+1)
  ScRep_24=a(i1+1,i2+1)
  ScRep_25=a(i1+2,i2+1)
  ScRep_26=a(i1+3,i2+1)
  ScRep_27=a(i1,i2+2)
  ScRep_28=a(i1+1,i2+2)
  ScRep_29=a(i1+2,i2+2)
  ScRep_30=a(i1+3,i2+2)
  ScRep_31=a(i1,i2+3)
  ScRep_32=a(i1+1,i2+3)
  ScRep_33=a(i1+2,i2+3)
  ScRep_34=a(i1+3,i2+3)
  do i3=MAX0(1,bb$_14),MIN0(n,50+bb$_14),1
    ScRep_19=ScRep_19+b(i2,i3)*c(i3,i1)
    ScRep_20=ScRep_20+b(i2,i3)*c(i3,i1+1)
    ScRep_21=ScRep_21+b(i2,i3)*c(i3,i1+2)
    ScRep_22=ScRep_22+b(i2,i3)*c(i3,i1+3)
    ScRep_23=ScRep_23+b(i2+1,i3)*c(i3,i1)
    ScRep_24=ScRep_24+b(i2+1,i3)*c(i3,i1+1)
    ScRep_25=ScRep_25+b(i2+1,i3)*c(i3,i1+2)
    ScRep_26=ScRep_26+b(i2+1,i3)*c(i3,i1+3)
    ScRep_27=ScRep_27+b(i2+2,i3)*c(i3,i1)
    ScRep_28=ScRep_28+b(i2+2,i3)*c(i3,i1+1)
    ScRep_29=ScRep_29+b(i2+2,i3)*c(i3,i1+2)
    ScRep_30=ScRep_30+b(i2+2,i3)*c(i3,i1+3)
    ScRep_31=ScRep_31+b(i2+3,i3)*c(i3,i1)
    ScRep_32=ScRep_32+b(i2+3,i3)*c(i3,i1+1)
    ScRep_33=ScRep_33+b(i2+3,i3)*c(i3,i1+2)
    ScRep_34=ScRep_34+b(i2+3,i3)*c(i3,i1+3)
  end do
  a(i1,i2)=ScRep_19
  a(i1+1,i2)=ScRep_20
  a(i1+2,i2)=ScRep_21
  a(i1+3,i2)=ScRep_22
  a(i1,i2+1)=ScRep_23
  a(i1+1,i2+1)=ScRep_24
  a(i1+2,i2+1)=ScRep_25
  a(i1+3,i2+1)=ScRep_26
  a(i1,i2+2)=ScRep_27
  a(i1+1,i2+2)=ScRep_28
  a(i1+2,i2+2)=ScRep_29
  a(i1+3,i2+2)=ScRep_30
  a(i1,i2+3)=ScRep_31
  a(i1+1,i2+3)=ScRep_32
  a(i1+2,i2+3)=ScRep_33
  a(i1+3,i2+3)=ScRep_34
end do
```

```
      do i2=i2,MIN0(n,50+bb$_13),1
      do i3=MAX0(1,bb$_14),MIN0(n,50+ bb$_14),1
        a(i1,i2)=a(i1,i2)+b(i2,i3)*c(i3,i1)
        a(i1+1,i2)=a(i1+1,i2)+b(i2,i3)*c(i3,i1+1)
        a(i1+2,i2)=a(i1+2,i2)+b(i2,i3)*c(i3,i1+2)
        a(i1+3,i2)=a(i1+3,i2)+b(i2,i3)*c(i3,i1+3)
      end do
    end do
  end do
  do i1=i1,MIN0(n,49+bb$_12),1
    do i2=MAX0(1,bb$_13),MIN0(n,50+bb$_13),1
      do i3=MAX0(1,bb$_14),MIN0(n,50+bb$_14),1
        a(i1,i2)=a(i1,i2)+b(i2,i3)*c(i3,i1)
      end do
     end do
    end do
   end do
  end do
 end do
 return
 end

 return
 end
```

## Appendix B: Example of transforming a loop nest containing a recurrence

*Original program:*

```
subroutine foo(a,b,s,n)
real*8 a(n,n),b(n,n),s

do i=1,n
   do j=1,n
      s=s+a(i,j)
      s=s+b(i,j)
   end do
end do
end
```

*After interchange and unrolling:*

```
      subroutine foo(a,b,s,n)
      real*8 a(n,n),b(n,n),s
      real*8 ScRed_7,ScRed_8,ScRed_9,
        ScRed_10,ScRed_11,ScRed_12,
        ScRed_13,ScRed_14,ScRed_15
      ScRed_7=s
      ScRed_8=0.0d0
      ScRed_9=0.0d0
      ScRed_10=0.0d0
      ScRed_11=0.0d0
      ScRed_12=0.0d0
      ScRed_13=0.0d0
      ScRed_14=0.0d0
      ScRed_15=0.0d0
      do j=1,n-2,3
         do i=1,n-2,3
            ScRed_7=ScRed_7+a(i,j)
            ScRed_7=ScRed_7+b(i,j)
            ScRed_8=ScRed_8+a(i,j+1)
            ScRed_8=ScRed_8+b(i,j+1)
            ScRed_9=ScRed_9+a(i,j+2)
            ScRed_9=ScRed_9+b(i,j+2)
            ScRed_10=ScRed_10+a(i+1,j)
            ScRed_10=ScRed_10+b(i+1,j)
            ScRed_11=ScRed_11+a(i+1,j+1)
            ScRed_11=ScRed_11+b(i+1,j+1)
            ScRed_12=ScRed_12+a(i+1,j+2)
            ScRed_12=ScRed_12+b(i+1,j+2)
            ScRed_13=ScRed_13+a(i+2,j)
            ScRed_13=ScRed_13+b(i+2,j)
            ScRed_14=ScRed_14+a(i+2,j+1)
            ScRed_14=ScRed_14+b(i+2,j+1)
            ScRed_15=ScRed_15+a(i+2,j+2)
            ScRed_15=ScRed_15+b(i+2,j+2)
         end do
         do i=i,n
            ScRed_7=ScRed_7+a(i,j)
            ScRed_7=ScRed_7+b(i,j)
            ScRed_8=ScRed_8+a(i,j+1)
            ScRed_8=ScRed_8+b(i,j+1)
            ScRed_9=ScRed_9+a(i,j+2)
            ScRed_9=ScRed_9+b(i,j+2)
         end do
      end do
s=s+ScRed_7+ScRed_8+ScRed_9+
  ScRed_10+ScRed_11+ScRed_12+ScRed_13
  ScRed_14+ScRed_15

do j=j,n
   do i=1,n
      s=s+a(i,j)
      s=s+b(i,j)
   end do
  end do

 end
```


## Acknowledgments

This work would not have been possible without the encouragement and guidance provided by Frances Allen and Randolph Scarborough. The author would like to thank Ray Ellersick, Roy Ju, Paula Newman, John Ng, Khoa Nguyen, Jin-Fan Shaw, and Radhika Thekkath for their contributions to the design and implementation of the ASTI transformer at IBM Santa Teresa Laboratory during the 1991–1993 time period, and all members of the ASTI team for creating an exciting environment in which

research could flourish on a development schedule. The author would also like to thank Alan Adamson and other members of the Parallel Development group in the IBM Toronto Laboratory for their ongoing work on shipping the ASTI optimizer as part of the IBM XL FORTRAN compiler products.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Standard Performance Evaluation Corporation.

## References


1. Michael J. Wolfe, *Optimizing Supercompilers for Supercomputers*, Pitman, London, and MIT Press, Cambridge, MA, 1989; in the series *Research Monographs in Parallel and Distributed Computing*.
2. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Trans. Programming Languages & Syst.* **13,** 451–490 (October 1991).
3. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Co., Inc., Reading, MA, 1986.
4. Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, "Global Value Numbers and Redundant Computations," *Conference Record, Fifteenth ACM Principles of Programming Languages Symposium*, San Diego, January 1988, pp. 12–27.
5. Mark Wegman and Kenneth Zadeck, "Constant Propagation with Conditional Branches," *Conference Record, Twelfth ACM Symposium on Principles of Programming Languages*, January 1985, pp. 291–299.
6. M. Metcalfe and J. Reid, *FORTRAN 90 Explained*, Oxford Science Publishers, England, 1990.
7. Jyh-Herng Chow, Leonard E. Lyon, and Vivek Sarkar, "Automatic Parallelization for Symmetric Shared-Memory Multiprocessors," presented at the CASCON '96 Conference, November 1996.
8. Ko-Yang Wang, Wei-Min Ching, Manish Gupta, Sam Midkiff, Edith Schonberg, and Dave Shields, "Improving the Performance of HPF Compilers," *Proceedings of the Fifth Workshop on Compilers for Parallel Computers (CPC '95)*, Malaga, Spain, June 1995, pp. 22–39.
9. David F. Bacon, Jyh-Herng Chow, Dz-Ching R. Ju, K. Muthukumar, and Vivek Sarkar, "A Compiler Framework for Restructuring Data Declarations to Enhance Cache and TLB Effectiveness," presented at the CASCON '94 Conference, November 1994.
10. Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng, "Improving Data Locality with Loop Transformations," *ACM Trans. Programming Languages & Syst.* **18,** 423–453 (July 1996).
11. Kevin O'Brien, Kathryn M. O'Brien, Martin Hopkins, Arvin Shepherd, and Ron Unrau, "XIL and YIL: The Intermediate Languages of TOBEY," *Sigplan Notices* **30,** 71–82 (March 1995).
12. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide, Second Edition*, Society for Industrial and Applied Mathematics (SIAM), September 1994.
13. *KAP for IBM FORTRAN, User's Guide Version 3.3*, Technical report (Document No. 9603001), Kuck & Associates, Inc., 1906 Fox Drive, Champaign, IL 61820-7334. *http://www.kai.com.*, 1996.
14. *VAST-2 for XL FORTRAN, User's Guide, Edition 1.2*, Technical report (Document No. VA061), Pacific-Sierra Research Corporation, 2901 28th St., Santa Monica, CA 90405, 1994.
15. *Engineering and Scientific Subroutine Library (ESSL), Guide and Reference*, 1994; available through IBM branch offices.
16. R. C. Agarwal, F. G. Gustavson, and M. Zubair, "Exploiting Functional Parallelism of POWER2 to Design High-Performance Numerical Algorithms," *IBM J. Res. Develop.* **38,** 563–576 (September 1994).
17. Theodore H. Romer, Dennis Lee, Brian N. Bershad, and J. Bradley Chen, "Dynamic Page Mapping Policies for Cache Conflict Resolution on Standard Hardware," *Proceedings of the First Symposium on Operating System Design and Implementation*, November 1994.
18. F. E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure," *Commun. ACM* **19,** 137–147 (March 1976).
19. J. T. Schwartz and M. Sharir, "Tarjan's Fast Interval Finding Algorithm," Technical report (SETL Newsletter Number 204), Courant Institute, New York University, 1978.
20. J. Ferrante, K. Ottenstein, and J. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Trans. Programming Languages & Syst.* **9,** 319–349 (July 1987).
21. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, "An Efficient Method for Computing Static Single Assignment Form," *Conference Record, Sixteenth Annual ACM Symposium on Principles of Programming Languages*, January 1989, pp. 25–35.
22. Jong-Deok Choi, Vivek Sarkar, and Edith Schonberg, "Incremental Computation of Static Single Assignment Form," *Proceedings of the 1996 International Conference on Compiler Construction (CC '96)*, Linkoping, Sweden, April 1996.
23. Ron Cytron, Michael Hind, and Wilson Hsieh, "Automatic Generation of DAG Parallelism," *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989, pp. 54–68.
24. Ron Cytron, Jeanne Ferrante, and Vivek Sarkar, "Experiences Using Control Dependence in PTRAN," *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, August 1989; in *Languages and Compilers for Parallel Computing*, D. Gelernter, A. Nicolau, and D. Padua, Eds., MIT Press, 1990, pp. 186–212.
25. Vivek Sarkar, "The PTRAN Parallel Programming System," *Parallel Functional Programming Languages and Compilers*, B. Szymanski, Ed., ACM Press, New York, 1991, pp. 309–391.
26. Francois Irigoin, Pierre Jouvelot, and Remi Triolet, "Semantical Interprocedural Parallelization: An Overview of the PIPS Project," *Proceedings of the ACM 1991 International Conference on Supercomputing*, June 1991, pp. 244–251.
27. Constantine Polychronopoulos, "The Hierarchical Task Graph and Its Use in Auto-Scheduling," *Proceedings of the ACM 1991 International Conference on Supercomputing*, June 1991, pp. 252–263.
28. John R. Allen, "Dependence Analysis for Subscripted Variables and Its Application to Program Transformation," Ph.D. thesis, Rice University, Houston, TX, 1983.
29. Vivek Sarkar and Radhika Thekkath, "A General Framework for Iteration-Reordering Loop Transformations," *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992, pp. 175–187.

30. Nimrod Megiddo and Vivek Sarkar, "Optimal Weighted Loop Fusion for Parallel Programs," *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architecture*, June 1997, pp. 282–291.
31. Randy Allen and Ken Kennedy, "Automatic Translation of FORTRAN Programs to Vector Form," *ACM Trans. Programming Languages & Syst.* **9,** 491–592 (October 1987).
32. Jeanne Ferrante, Vivek Sarkar, and Wendy Thrash, "On Estimating and Enhancing Cache Effectiveness," *Lecture Notes in Computer Science* **589,** 328–343 (1991); in *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau, D. Padua, Eds., Santa Clara, CA, August 1991.
33. Utpal Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, Norwell, MA, 1988.
34. Vivek Sarkar, Guang R. Gao, and Shaohua Han, "Locality Analysis for Distributed Shared-Memory Multiprocessors," *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1996; *Lecture Notes in Computer Science* **1239,** Springer-Verlag, New York, 1996, pp. 20–40.
35. Michael E. Wolf and Monica S. Lam, "A Data Locality Optimization Algorithm," *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation*, June 1991, pp. 30–44.
36. Kyle Gallivan, William Jalby, and Dennis Gannon, "On the Problem of Optimizing Data Transfers for Complex Memory Systems," *Proceedings of the ACM 1988 International Conference on Supercomputing*, July 1988, pp. 238–253.
37. Michael E. Wolf and Monica S. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Trans. Parallel & Distributed Syst.* **2,** 452–471 (October 1991).
38. Nimrod Megiddo and Vivek Sarkar, "Minimizing Loop Execution Time by Optimizing Block/Tile Sizes," *Invention Disclosure ST9-95-008*, October 1994.
39. Vivek Sarkar, "Determining Average Program Execution Times and Their Variance," *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, July 1989, pp. 298–312.
40. Ken Kennedy and Kathryn S. McKinley, "Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution," *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993; *Lecture Notes in Computer Science* **768,** Springer-Verlag, New York, 1993.
41. G. R. Gao, R. Olsen, V. Sarkar, and R. Thekkath, "Collective Loop Fusion for Array Contraction," *Proceedings of the Fifth International Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992; *Lecture Notes in Computer Science* **757,** Springer-Verlag, New York, 1993, pp. 281–295.
42. David Callahan, Steve Carr, and Ken Kennedy, "Improving Register Allocation for Subscripted Variables," *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990, pp. 53–65.
43. Special issue on POWER2 and PowerPC, *IBM J. Res. Develop.* **38,** 489–648 (September 1994).
44. L. Lamport, "The Parallel Execution of DO Loops," *Commun. ACM* **17,** 83–93 (February 1974).
45. Michael E. Wolf, "Improving Parallelism and Locality in Nested Loops," Ph.D. thesis, Stanford University, August 1992; *Technical Report CSL-TR-92-538.*
46. David H. Bailey, "Unfavorable Strides in Cache Memory Systems," *Scientific Programming* **4,** 53–58 (1995); *RNR Technical Report RNR-92-015*, NASA Ames Research Center, Sunnyvale, CA.
47. Robert Schreiber and Jack Dongarra, "Automatic Blocking of Nested Loops," *Technical Report 90.38*, Research Institute for Applied Computer Science (RIACS), Mountain View, CA, August 1990.
48. Francois Irigoin and Remi Triolet, "Supernode Partitioning," *Conference Record, Fifteenth ACM Symposium on Principles of Programming Languages*, 1988.
49. Francois Irigoin, "Code Generation for the Hyperplane Method and for Loop Interchange," *Technical Report ENSMP-CAI-88-E102/CAI/I*, École Nationale Supérieure des Mines de Paris, October 1988.

*Received August 8, 1996; accepted for publication April 21, 1997*

**Vivek Sarkar** *IBM Software Solutions Division, MIT Laboratory for Computer Science, Cambridge, Massachusetts 02139 (vivek@lcs.mit.edu, http://www.csg.lcs.mit.edu/~vivek).*
Dr. Sarkar is a Senior Technical Staff Member in the IBM Software Solutions Division and a member of the IBM Academy of Technology. He is currently a visiting associate professor at the MIT Laboratory for Computer Science. His past research has focused on the area of optimizing and parallelizing compilers. Dr. Sarkar joined IBM in 1987 after receiving a Ph.D. degree from Stanford University. At IBM, he worked on the PTRAN research project from 1987 to 1990, developing compiler technologies for automatic parallelization. From 1991 to 1993, he led the product development effort to design and implement the transformer component of the IBM ASTI optimizer. From 1994 to 1996, he was manager of the ADTI Department, with the mission of transferring high-priority application development technologies from the research stage into IBM products.