Compile-Time Techniques for Improving Scalar Access Performance in Parallel Memories

Rajiv Gupta and Mary Lou Soffa

Abstract—**The partitioning of shared memory into a number of memory modules is an approach to achieve high memory bandwidth for parallel processors. Memory access conflicts can occur when several processors simultaneously request data from the same memory module. Although work has been done to improve access performance for vectors, little work has been reported to improve the access performance of scalars. For systems in which the processors operate in a lock-step mode, a large percentage of memory access conflicts can be predicted at compile-time. These conflicts can be avoided by appropriate distribution of data among the memory modules at compile-time. A long instruction word machine is an example of a system in which the functional units operate in a lock-step mode, performing operations on data fetched in parallel from multiple memory modules. In this paper, compile-time techniques for distribution of scalars to avoid memory access conflicts are presented. Furthermore, algorithms to schedule data transfers among memory modules to avoid conflicts that cannot be eliminated by the distribution of values alone are developed. The techniques have been implemented as part of a compiler for a reconfigurable long instruction word architecture. Results of experiments are presented demonstrating that a very high percentage of memory access conflicts can be avoided by scheduling a very low number of data transfers.**

Index Terms—**Long instruction word architectures, memory access conflicts, memory bandwidth, parallel memories, renaming.**

I. Introduction

HIGH memory bandwidth is essential for effective utilization of systems with large numbers of processors. An approach for achieving high memory bandwidth is through partitioning of global data memory into a number of memory modules that can operate in parallel [2], [3], [6], [8], [19], [20]. The organization of a system with multiple processors and multiple memory modules is shown in Fig. 1. The memory modules, accessed through an interconnection network, are shared by the processors. In such a system, degradation in performance can result due to memory access conflicts that occur when a number of processors simultaneously request data from the same memory module. In the presence of these conflicts, the operation of the processors is impeded as the operands cannot be accessed in parallel.

Although techniques have been developed to improve access performance of vectors in parallel memories [3], [16], [21], [22], there are always parts of a program that operate on scalar data rather than vectors. To achieve high overall speedup, it is essential to execute these parts of the program in parallel. Avoiding memory access conflicts in this situation requires that the scalars used in parallel operations be assigned to different memory modules. Although it might seem that the number of scalars in a program is small, this is not the case. In addition to the programmer declared scalar variables, a large number of scalar temporaries is created by the compiler. The techniques developed for storing vectors in parallel memories are inadequate for allocating storage for scalar data, because scalar data accesses do not have a regular pattern, unlike vector accesses.

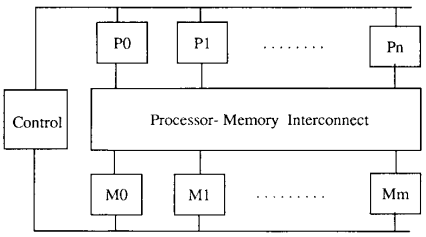

Fig. 1. Processor memory organization.

If the processors in a system operate in a lock-step mode, it is possible to predict at compile time a high percentage of operands required simultaneously by the processors and hence perform their allocation to different memory modules to avoid access conflicts. Long instruction word (LIW) architectures [10], [12], a family of fine-grained architectures, are examples of systems that fall in the above category. LIW machines have multiple functional units that operate in lock-step and perform operations on data fetched in parallel from memory organized in the form of multiple memory modules.

To avoid memory access conflicts, operands required by the operations that execute in parallel must first be determined. Following this, the operand values (not the variables) must be assigned to memory modules to allow conflict free access. Such an assignment may not always exist. However, access conflicts can always be avoided by creating multiple copies of data values and distributing them among the memory modules. Multiple copies can be created by data transfers among memory modules and scheduled at compile-time. The transfers can result in increased execution time. Thus, an attempt should be made to minimize the duplication of values. Creating multiple copies involves determining which values should be replicated and to which memory module they should be assigned, as both have an impact on the degree of duplication performed.

This paper presents compile-time techniques for storage allocation of scalar values into memory modules with the goal of limiting run-time memory access conflicts. The approach presented for allocation is applicable to those operands in instructions that can be predicted at compile-time, where an instruction is composed of the multiple operations and corresponding operands that execute in parallel. In this work, algorithms for avoiding those memory access conflicts that can be predicted at compile time are described. The techniques have been implemented as part of a compiler for a reconfigurable long instruction

Manuscript received May 3, 1989; revised October 2, 1990. This work was supported in part by the NSF under Grant CCR-88001104 to the University of Pittsburgh.

The authors are with the Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260.

IEEE Log Number 9042570.

word (RLIW) architecture [13], [14]. Results of experiments demonstrate that a very high percentage of memory conflicts are avoided without replication of scalar values. Furthermore, the access conflicts caused by the operands that could not be predicted at compile-time (e.g., some array accesses) do not cause significant deterioration in performance. The impact of the renaming optimization for removing storage related dependences is discussed relative to achieving a memory access conflict free assignment for a program. Finally, other applications of the technique are discussed.

II. Background

Techniques have been developed to improve access performance of vectors by storing them in a skewed fashion in parallel memories [3], [16]. In work by Mace and Wagner, techniques to determine the distribution of data used in vector operations were developed [21], [22]. A graph is first constructed where the nodes represent vector operations and the edges represent the data structures required by the operations. Costs are associated with nodes, and these costs are functions of storage patterns assigned to the data structures indicated by the edges of the node. The overall cost of the graph is the sum of the costs of all of the nodes. A globally optimal assignment of storage patterns to the data structures is an assignment that minimizes the cost of the graph. The general problem of assigning globally optimal storage patterns at compile time is shown to be NP-complete. Unlike these techniques for allocation of storage for aggregate data structures used in vector operations, the problem that we address in this paper deals with allocation of storage for scalar values used in fine-grained operations. The allocation techniques for scalar and vector data differ primarily due to the way that the data are accessed. The data access pattern in vector operations is regular while this is not true for scalar data used in fine-grained operations.

The problem of avoiding memory access conflicts is also addressed in the development of the Bulldog compiler for VLIW architectures [8]. The Bulldog compiler, based upon trace scheduling, repeatedly traces out a path of basic blocks in the intermediate-code flow graph and generates instruction schedules for each path. During the generation of these instruction schedules, the compiler must assign memory locations to values in a manner that avoids access conflicts. A value used in different traces may be assigned to different memory modules during code generation. Therefore, copy code to transfer values from one bank to another has to be generated. Consider the example shown in Fig. 2. Let us assume that the order in which traces are processed is T_1, T_2, and T_3. During the generation of code for T_1 and T_2 the value of X may be assigned to different memory modules. Therefore, code to transfer the value of X from one module to another must be introduced along the path from T_1 to T_2. Furthermore, the values Y and Z may be assigned to the same memory module when T_1 and T_2 are processed. If Y and Z are operands in the same instruction in T_3 then one of the values must be moved to a different memory module. From this example, it is clear that data transfers are introduced by the Bulldog compiler because it carries out memory module assignment for the traces locally.

The approach described in this paper avoids the data transfers that may be required by the Bulldog compiler. Using our approach, the compiler would first generate the schedules for all traces to exploit parallelism, without assigning memory. The compiler then examines the possibility of access conflicts in the traces prior to making memory module assignment. The values used in the schedule are assigned to memory modules to avoid access conflicts during execution. For the example shown above, the value of X would not be assigned to different memory modules and the values of Y and Z would not be assigned to the same memory module. In contrast to the Bulldog compiler, which assigns memory modules locally for each trace, we perform global memory module assignment to reduce data transfers.

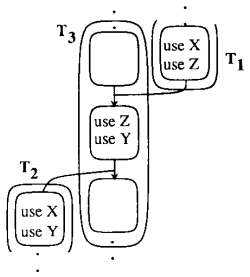

Fig. 2. Conflict avoidance in Bulldog.

III. Memory Module Assignment

The approach for allocation of storage used in this work involves generating all of the instructions without assigning physical memory modules for the operand values. Symbolic addresses are assigned to data values during scheduling of operations in instructions, and then all of the instructions are examined to determine which memory module should be used to store the value of a data item. The advantage of using symbolic addresses is that during memory module assignment, the operands performed by the same instruction are known. Thus, when binding the addresses to memory modules, an attempt is made to avoid multiple accesses to a memory module during the execution of an instruction. Since a program can have a large number of data values (variables/temporaries), examining all of the requirements for the data items at the same time and assigning modules to avoid the memory access conflicts predictable at compile time can be an unmanageable task. One solution to this problem is to perform the memory module assignment for one program region [9] at a time. Another approach is to assign memory in two phases: global module assignment and local module assignment. During global module assignment, data values that are used in more than one region of the program can be performed. After this has been done, the regions can be examined one at a time and module assignment for the data values that are created and used only within that region can be performed. The algorithms used for global and local module assignment would be similar, as in either case, instructions are examined to determine where the data values should be stored.

The problem of memory module assignment can be stated as follows.

Problem: Given memory $\boldsymbol{M}= \langle M_1, M_2 \cdots M_k \rangle$, where memory modules $M_1, M_2 \cdots M_k$ can be accessed in parallel, and a sequence of instructions each of which requires up to k operands from among data values $V_1, V_2 \cdots V_n$, allocate storage for $V_1, V_2 \cdots V_n$ among the memory modules so that the instructions can be executed without encountering any memory access conflicts.

In this work we assume that in a system with k memory

modules, at most k data values can be simultaneously provided to the functional units. The techniques presented in this paper are applicable if the memory system contains multiple banks or interleaved banks, for it is essential to only specify the number of values that can be obtained simultaneously.

The example given in Fig. 3 shows how the storage can be allocated to avoid memory access conflicts for a particular use of the data values. The instructions are denoted by the operands they use, as the operations are of no importance here. In this example, all memory access conflicts are avoided by assigning the modules properly. However, this is not always possible. If an instruction with the following operands is added to the list of instructions

$$V_2\, V_4\, V_5$$

then it is not possible to assign modules and avoid all memory access conflicts. If multiple copies of data items are made and stored in different memory modules in a certain way, memory conflicts can be avoided. In the above example, if a copy of value V_5 is stored in M_1, in addition to M_3, then all memory conflicts are avoided. It is possible that a copy of a variable may be required in each memory module to avoid all memory conflicts. In the above example, if the following operand usage of an instruction is also included

$$V_1\, V_4\, V_5$$

then all memory conflicts can be avoided by adding a copy of V_5 to module M_2. In this situation all three modules contain a copy of value V_5. Thus, depending upon the instructions, varying number of copies of values may have to be created and stored in different memory modules. The module assignment algorithms developed are aimed at finding an allocation that requires the least amount of copying, for copying of values can increase execution time. Thus, the modified memory module assignment problem can be stated as follows:

Problem: Given memory $\boldsymbol{M}= \langle M_1, M_2 \cdots M_k \rangle$, where the memory modules $M_1, M_2 \cdots M_k$ can be accessed in parallel, and a sequence of instructions each of which requires up to k operands from among data values $V_1, V_2 \cdots V_n$, allocate storage for $V_1, V_2 \cdots V_n$ among the memory modules so that the instructions can be executed without encountering any memory access conflicts, and a minimum number of multiple copies of data values are created in the process.

When creating multiple copies of values, the problem of consistency of the multiple copies never arises, for these values do not correspond to variables in the program. Corresponding to each definition of a variable, a distinct data value is created and when memory modules are assigned, the different data values of a variable are treated independently. Thus, it is not the variables but specific data values that are being replicated.

The approach taken for memory allocation is to consider any two operands of an instruction. These operands will not cause a memory access conflict in the instruction if their values are stored in two different memory modules. A given set of instructions will be free of memory access conflicts if each pair of data values that is used in the same instruction has its data values in separate memory modules. To allocate storage for a sequence of instructions, pairwise conflicts among the data values are considered. A graph, in which the nodes represent the data values and the edges represent the conflicts among values, is constructed. Finding whether an allocation exists that avoids all the conflicts is the same as determining whether the access conflict graph is k-colorable, where k is interpreted as

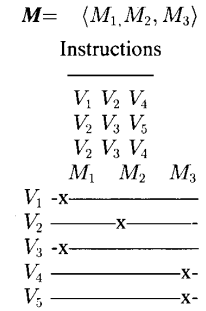

Fig. 3. Avoiding access conflicts.

the number of memory modules in the system. Finding a k-coloring for an arbitrary graph for a fixed k is an NP-complete problem [11]. Therefore, a heuristic is used that removes nodes from the graph if coloring, using k colors, is not possible. The memory access conflicts among the data values, represented by the nodes not removed from the graph (V_{assigned}), can be avoided by placing single copies of these values in the memory modules assigned through coloring. The remaining conflicts, involving the data values represented by the nodes removed from the graph ($V_{\text{unassigned}}$), are avoided by duplicating a subset of values and placing them in different memory modules.

After the set of nodes is removed from the graph, the number and placement of the copies for each data value in this set is determined. The values of the nodes in this set will have at least two copies stored in different memory modules. The overall strategy for avoiding access conflicts is summarized in Fig. 4.

If the number of operands in the instructions is three, determining the smallest subset of values, from among the ones with two copies, that should have three copies to avoid all memory access conflicts is NP-complete. This will be shown later. Even if algorithms for removing a minimum number of nodes from the graph while coloring and determining the smallest subset of values that requires three copies are used, a suboptimal solution may be obtained. This is demonstrated by the example in Fig. 5. In this example, two storage allocations for a given set of instructions are presented. In either case, two nodes are removed from the graph to make it colorable. In the first case, nodes V_4 and V_5 are removed from the graph and in the second case nodes V_2 and V_5 are removed from the graph. In the first solution, conflicts are avoided by making an additional copy of V_5. In the second case, all memory conflicts are avoided after two copies of V_2 and V_5 have been placed. Thus, although same number of nodes were removed in both cases the second solution resulted in less copying of data values.

In an actual implementation, the memory system is so designed that the successive memory locations belong to different memory modules. Thus, a contiguous piece of memory allocated for an activation record will contain memory locations from different memory modules. This organization allows the scalar data local to a program module to be distributed among the memory modules. The same size storage is assigned from all memory modules containing any variables in a program module. The size is the largest number of variables allocated to any one memory module.

A. Heuristic for Removing Nodes

A heuristic is presented for determining the subset of nodes which, if removed, make the graph k-colorable. First of all, the

Memory Module Assignment

{

Construct *Access Conflict Graph* $G = (V, E)$

Using graph coloring assign values to memory modules $M_1, M_2, \cdots M_k$
such that accesses to the values assigned to memory modules, $V_{\text{assigned}} \subseteq V$, do not conflict

Let $V_{\text{unassigned}} = V - V_{\text{assigned}}$, be the values that could not be assigned to memory modules in a conflict free manner. Avoid remaining access conflicts by

Duplication: Creating multiple copies of values in $V_{\text{unassigned}}$

Placement: Distributing these copies among memory modules

}

Fig. 4. Overall strategy for memory module assignment.

graph is decomposed into atoms which are subgraphs that do not have clique separators [23]. A clique separator is a complete graph whose removal disconnects the graph. If each of the atoms in a graph is colored using k colors then the entire graph can be colored using k colors. A n-vertex, e-edge graph can be decomposed into atoms in $O(ne)$ time [23]. The coloring algorithm need only concern itself with coloring an atom rather than the entire graph at the same time. When coloring an atom, a heuristic removes nodes whenever it becomes impossible to continue coloring.

The heuristic developed for coloring an atom G=(V, E) is described in Fig. 6. The edges in the graph are assigned weights to guide the coloring algorithm. If a node has a degree less than the number of memory modules, the edges leaving that node are assigned a weight of zero, as any algorithm will be successful in coloring such a node. Each edge from one of the remaining nodes is assigned a weight equal to the number of conflicts in which the vertices connected by the edge are involved. The node involved in the maximum number of conflicts is first colored.

The graph is viewed as consisting of two subgraphs: G_1 containing the nodes that have been colored (V_{assigned}) and G_2 containing the nodes yet to be colored (V_{rest}). To choose the next node to be colored, the urgency (U_{n_i}) for each uncolored node (n_i) is computed and the node with the highest urgency is chosen. The urgency of a node is proportional to the number of conflicts between the node and all other colored nodes. A high number of conflicts implies that a failure to color the node to avoid the conflicts is likely to leave a high number of conflicts unresolved. This may result in a high degree of duplication of data values. The urgency of a node is also inversely proportional to the number of colors that can still be used to color it, for a small number of colors available implies that a delay in coloring the node might result in an inability to color the node at all. The above process is continued until each node has either been colored or cannot be colored.

The example in Fig. 7 demonstrates how the above algorithm is used. In Fig. 7, each edge (n_j, n_i) in the graph is labeled with weights $wt(n_j \rightarrow n_i)$ and $wt(n_i \rightarrow n_j)$. In this example, four data values are allocated space in the memory modules (k=3) and the node corresponding to V_5 is removed from the graph. Multiple copies of V_5 are created to avoid all memory access conflicts.

The coloring heuristic can be implemented in $O((n+e)\log(n+e))$ time. In the implementation, a graph is represented using adjacency lists. The nodes in G_2, directly connected by an edge to a node in G_1, are represented by set T_1 and the remaining nodes in set G_2 are represented by T_2. Path compression trees are used to represent T_1 and T_2 so that n FIND, INSERT, and DELETE operations can be done on these sets in $O(nG(n))$ time [1]. The function $G(n)$ increases very slowly with n, and thus $O(nG(n))$ is almost linear. A mergeable heap [1] (M) is employed to store the urgencies of nodes. Operations INSERT, DELETE, and MAX can be performed in $O(n\log n)$ time, where n is the number of operations, using a mergeable heap. The operations available for the above data structures allows efficient and convenient implementation of the coloring heuristic.

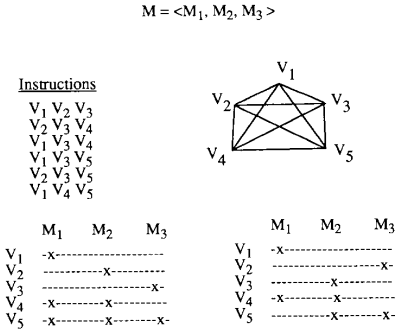

Fig. 5. Choosing nodes to be removed.

The structures T_1, T_2 and M should contain unique values. No node can be in both T_1 and T_2 at the same time. However, in order to ensure that M has unique values, a check is made before inserting a new value to determine whether it is already present.

Claim: The running time of the implementation of the coloring algorithm described in Fig. 6 is $O((n+e)\log(n+e))$ or $O(n^2\log n)$ in the worst case.

Proof: Updating T_1 and T_2 requires $O((2n+e)G(2n+e))$ time each as e FIND operations, n INSERT operations, and n DELETE operations are performed on each tree. Time spent on manipulating the heap is $O((n+e)\log(n+e))$ as n MAX operations, e DELETE operations, and $(n+e)$ INSERT operations are performed. Thus, the overall running time of the algorithm is $O((n+e)\log(n+e))$ or $O(n^2\log n)$ in the worst case. □

The aim of the above heuristic is to color as many nodes as possible using k colors. The graph given in Fig. 8 is one for which the ratio of number of nodes that may be colored by the heuristic and the number of nodes colored by an optimal algorithm is the minimum. The graph has a completely connected subgraph which consists of k nodes and each of these k nodes are connected to each of the remaining $(n-k)$ nodes in the graph. The $(n-k)$ nodes are connected among themselves in the form of a ring. This subgraph does not have a clique separator and thus is an atom. An optimal algorithm will remove two nodes from among the completely connected subgraph and make it colorable in k colors. However, the heuristic described may color the k nodes in the completely connected subgraph first in which case the remaining $(n-k)$ nodes would have to be removed from the graph.

An approach based upon graph coloring is also used by compilers for register allocation. For register allocation, an interference

Color(G=(V,E))

/*$d(n_i)$ is the degree of node n_i */
/* conf(n_i, n_j) is the number of instructions in which both n_i and n_j are used as operands */
{
/* Compute Weights */
$\forall\ (n_i, n_j) \in E$, if $d(n_i) < k$ then wt($n_j \rightarrow n_i$) = 0 else wt($n_j \rightarrow n_i$) = conf(n_i, n_j)
$\forall\ n_i$ compute $S_{n_i} = \sum_{n_j} \text{wt}(n_i \rightarrow n_j)$ where $(n_i, n_j) \in E$
/* Initialize Sets */
$n_{\text{first}} = n_i$ such that $S_{n_i} = \max_{n_j} S_{n_j}$
ASSIGN(n_{first}) = M_1; $V_{\text{assigned}} = \{n_{\text{first}}\}$
$V_{\text{rest}} = V - \{n_{\text{first}}\}$; $V_{\text{unassigned}} = \phi$
while $V_{\text{rest}} \neq \phi$
{
Urgency $U_{n_j} = \frac{\sum_{n_k} \text{wt}(n_i \rightarrow n_j)}{K_{n_j}}$ where
$n_k \in V_{\text{assigned}}$,
$n_j \in V_{\text{rest}}$,
$(n_k, n_j) \in E$,
K_{n_j} is the number of modules that can still be assigned to n_j.
Choose $n_{\text{next}} = n_i$ st $U_{n_i} = \max_{n_j} U_{n_j}$,
if $K_{n_{\text{next}}} = 0$ then $V_{\text{unassigned}} = V_{\text{unassigned}} \cup \{n_{\text{next}}\}$
else {
ASSIGN(n_{next}) = one of the available modules
$V_{\text{assigned}} = V_{\text{assigned}} \cup \{n_{\text{next}}\}$
}
$V_{\text{rest}} = V_{\text{rest}} - n_{\text{next}}$
}
}

Fig. 6. Heuristic for graph coloring.

graph is constructed in which the nodes correspond to variable values that are candidates for registers and edges connect nodes that must be assigned different registers. A coloring of the graph is equivalent to an assignment of registers and various heuristics have been developed to perform this coloring [4], [5]. However, there is an important difference between register allocation and memory module assignment, which led us to develop a new coloring heuristic for memory module assignment. During register allocation, priorities are associated with nodes and preference is given to nodes representing variables that are referenced more often. During memory module assignment, weights are assigned to edges in the graph and these weights indicate the number of access conflicts that would be left unresolved if the nodes connected to an edge are not colored. The selection of a node for coloring is based on the sum of the weights of those edges that represent conflicts that would be resolved if the node is colored. Only those uncolored nodes that are directly connected to previously colored nodes are considered during the selection of the next node to be colored, for only the coloring of such a node will result in the resolution of conflicts. However, during register allocation all uncolored nodes are considered to be valid choices for the next node. Thus, the coloring heuristics developed for register allocation are inappropriate for memory module assignment.

B. Duplication and Placement Strategies

After determining what values have to be replicated ($V_{\text{unassigned}}$) by running the coloring heuristic, the number of copies and placement for these copies have to be determined. Two different approaches are considered for this problem. The first approach is simple and involves examining one instruction at a time and creating copies of values and placing them so that the instruction is conflict free. This is achieved by backtracking across the

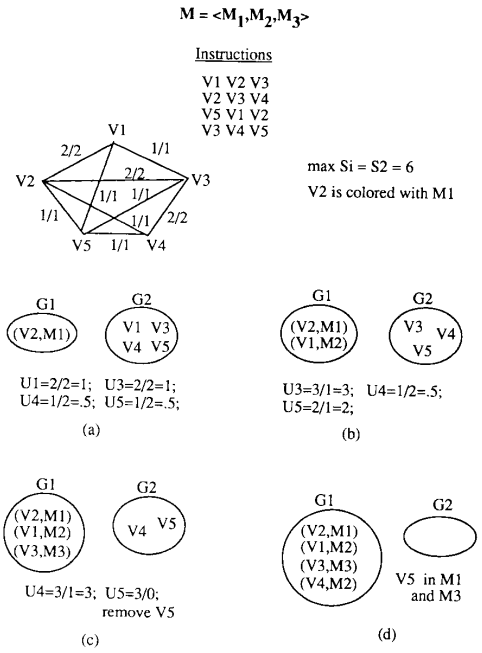

Fig. 7. Applying the coloring heuristic.

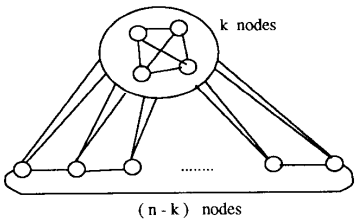

Fig. 8. Performance of the coloring heuristic.

available memory modules. The drawback of examining one instruction at a time is that copies created for a particular instruction are less likely to be used in subsequent instructions. The second approach is more complex and involves examining all instructions before deciding what subset of values should be replicated. As a result, a copy of a value can help in avoiding conflicts in several instructions.

1) Straightforward Approach Based on Requirements of Individual Instructions: In this approach the instructions are examined one at a time and copies of values are created as needed. First the instructions are ordered according to the number of operands targeted for multiple copies, i.e., members of set $V_{\text{unassigned}}$. Although access conflicts can be avoided by duplicating data values that were not removed from the graph during coloring, only values that were removed are considered because their small number makes the duplication algorithm efficient. The instructions that have only one operand in $V_{\text{unassigned}}$ are examined first and the instructions in which all k operands are in $V_{\text{unassigned}}$ are examined last. After examining an instruction, the minimum number of additional copies that have to be created and the placement for these copies are found. The procedure employed for this purpose generates all possible placements for the operands of an instruction by backtracking across different memory modules that can be used to store a data value. The backtracking procedure first tries to use as many existing copies of data values as possible and only creates new copies if needed. The placement that requires the least number of additional copies is used. The reason for ordering the instructions is as follows. A conflict in an instruction which has only one operand that can be duplicated can be avoided only by making a copy of that operand and placing it in a specific memory module. However, if the instruction has multiple operands that can be duplicated then there is likely to be more than one solution. The above approach is summarized in Fig. 9.

The problem of examining instructions one at a time is that the copies created for a particular instruction are less likely to be used by other instructions than if the requirements for all instructions were determined before placing copies. However, no heuristic will create more than k copies because a data value will not cause any conflicts if all k modules contain its value.

Claim: The run-time complexity of the algorithm is $O(n^k)$, where n is the number of variables in $V_{\text{unassigned}}$ and k the number of memory modules.

Proof: The backtracking procedure used to determine the set of variables to be duplicated to avoid access conflicts in a given instruction takes $O(k!)$ time in the worst case, as there are $k!$ possible permutations of the memory modules. Thus, the total time taken by the storage allocation procedure is $O(k!i)$ where i is the number of instructions with access conflicts after graph coloring. The maximum value of i is ${}^nC_k$. Therefore, the run-time complexity of the algorithm is $O(k!{}^nC_k)$ or $O(n^k)$. □

2) The Hitting Set Approach: As mentioned earlier, in the second approach all instructions are examined before the decision on what values to replicate is made. In general it may be possible to avoid a conflict in an instruction by replicating one of many data values. If, instead of making a choice at random, other instructions are examined before choosing the value to be duplicated, less copies are likely to be made, for the requirements of other instructions are determined before making a choice.

In this approach, after assigning the single copies of nodes in V_{assigned} and two copies of each node in $V_{\text{unassigned}}$ to memory modules, conflicts between pairs of operands that occur together in any instruction are eliminated. Next, additional copies of a subset of operands that are in $V_{\text{unassigned}}$ are created and placed to avoid all conflicts in combinations of three or more operands. After determining an appropriate placement of these values, combinations of four operands are examined. This process of duplication and placement is repeated until all the conflicts present in the k operand instructions are resolved.

The procedure discussed, summarized in Fig. 10, involves repeatedly finding a set of values to duplicate and placing them in memory modules. As will be shown later, the problem of finding the smallest subset of values that should be duplicated to avoid a set of conflicts is NP-complete. The problem of finding the best placement is also NP-complete. In the subsequent sections, heuristics for the duplication and placement of values are presented.

a) Duplication: In this section the problem of determining the subset of values to be duplicated so that all combinations of $(i+1)$ operands are free of conflicts, given that all i combinations of operands that occur together in any of the instructions are conflict free, is considered. The algorithm for the above problem can be applied repeatedly to eliminate all conflicts in the k operand instructions.

After running the coloring heuristic and placing the single copies of colored nodes and two copies of each node removed during coloring, conflicts between pairs of operands that occur together in any instuction are eliminated. Next combinations of three operands that occur together in any of the instructions are considered. If a combination of values V_1, V_2, and V_3 has memory conflict then existing copies of the values must be stored in one of the following three configurations.

	M_i	M_j
V_1	x	
V_2		x
V_3	x	x

(i)

	M_i	M_j
V_1	x	
V_2	x	x
V_3	x	x

(ii)

	M_i	M_j
V_1	x	x
V_2	x	x
V_3	x	x

(iii)

The conflict can be avoided by making an additional copy of a value and placing it in a memory module other than modules M_i and M_j.

The conflict in (i) can be avoided by making one more copy of $\{V_3\}$.

The conflict in (ii) can be avoided by making one more copy of one of $\{V_2,V_3\}$.

The conflict in (iii) can be avoided by making one more copy of one of $\{V_1,V_2,V_3\}$.

Backtrack

{
Divide the instructions into sets $S_1, S_2 \cdots S_k$
such that $S_i = \{I$: instruction I has i operands in $V_{\text{unassigned}}\}$
for $i = 1 \cdots k$ loop
{
$\forall\ I \in S_i$
Let $O_1, O_2 \cdots O_i$ denote the operands in I from $V_{\text{unassigned}}$
Using backtracking determine all module assignments $P_j = \{(O_1, M_{k_1}), (O_2, M_{k_2}) \cdots (O_i, M_{k_i})\}$
such that P_j avoids all memory access conflicts in I
$\forall\ P_j$ compute CP_j, the number of copies in P_j that need to be created
Choose the placement $P_{\min}$ such that $CP_{\min} = \min_j CP_j$ and create the additional copies of operands
}
}

Fig. 9. Approach based on backtracking.

Hitting Set Approach

/* Let $I_1^k, I_2^k \cdots I_N^k$ be the k-operand instructions */
/* Procedure Place(V)—places a copy of each value $v_i \in V$ */
/* Procedure Duplicate(V,op)—determines $V_{\text{dup}} \subseteq V$ that should be duplicated to avoid conflicts in */
/* instructions with op operands. Duplication requires the use of a heuristic for finding hitting sets. */
{
Let $S_i^n = \{\text{OP}_1, \text{OP}_2 \cdots \text{OP}_n\}$ denote a combination of n operands such that $\exists\ j$ such that $S_i^n \subseteq I_j^k$
Place($V_{\text{unassigned}}$), place the first copies of operands yet to be assigned modules
Place($V_{\text{unassigned}}$) so that $\forall\ i\ S_i^2$ is conflict free
for num $= 3 \cdots k$ loop
{
Duplication Phase:
Using the hitting set heuristic determine $V_{\text{dup}} \subseteq V_{\text{unassigned}}$ so that
if an additional copy of each value in V_{dup} is made $\forall\ i\ S_i^{\text{num}}$ is conflict free
Placement Phase: Place(V_{dup})
}
}

Fig. 10. The hitting set approach.

Hitting Set: HS for $s_1, s_2 \cdots s_N$ such that $1 \leq |s_j| \leq k$

{
/* all conflicts that can only be avoided by replicating a specific data value must be included in the hitting set */
HS $= \bigcup_j s_j, |s_j| = 1$
for size $= 2 \cdots k$ loop
{
Let $S_{v_i p}$ = # of sets s such that $v_i \in s \wedge \mid s \mid = p$
/* choose candidates for inclusion in the hitting set */
/* give preference to elements that appear in a greater number of sets and hence avoid more conflicts */
$\forall s_j = \{v_1, v_2 \cdots v_{\text{size}}\}$ such that $s_j \cap HS = \phi$
$HS = HS \cup v_n$ where $v_n \in s_j$ and $\forall v_l \in s_j$ such that $v_l \neq v_n$
$\nexists m \leq k$ st $(S_{v_l m} > S_{v_n m}) \wedge (S_{v_l i} = S_{v_n i}, i = \text{size} \cdots m-1)$
}
}

Fig. 11. Heuristic for finding the hitting set.

In each of the above cases there is at least one data value that has two copies, for in a combination of three values, each of which has one copy, the values must be stored in different memory modules, and thus the combination must be free of memory access conflicts. The cardinality of the set of values from among which a value should be chosen for replication varies from one to i as the number of operands with multiple copies in the i operand combination varies from one to i.

After constructing the sets as described above, at least one value from each of these sets is chosen to have an additional copy to avoid memory access conflicts in all combinations of three operands. Ideally the smallest subset of values should be duplicated to avoid the conflicts. However, this subset is the minimum cardinality hitting set of the group of sets, and the problem of finding the minimum cardinality hitting set is NP-complete [11]. Therefore, in order to find the hitting set the

Place: a copy each of value $v_i \in$ HS
{
Divide I the set of instructions into groups
$I_1 \cup I_2 \cup \cdots I_k = I$ such that each instruction in I_i contains i operands in $V_{\text{unassigned}}$
$\forall\ v \in$ HS
{
Compute $C_{M_x I_y}(v)$ = number of instructions in I_y that become conflict free if a copy of v is placed in M_x
Let M_p be the memory module st $\forall M_i\ 1 \leq i \leq k$ and $i \neq p$
$\nexists\ z \leq k$ st $(C_{M_i I_z}(v) > C_{M_p I_z}(v)) \wedge (C_{M_i I_\alpha}(v) = C_{M_p I_\alpha}(v), \alpha = 1 \ldots z-1)$
Place a copy of v in M_p
}
}

Fig. 12. Placement algorithm.

heuristic given in Fig. 11 is used.

In the above procedure, after creating new copies, a placement for these copies has to be found. The placement of the copies made to avoid i operand conflicts has an effect on the number of additional copies made to avoid the conflicts in $(i+1)$ operand combinations. This implies that proper placement of copies is important.

b) Placement: After having determined the set of values to be duplicated, the values are assigned to particular memory modules. A placement that avoids the maximum number of memory access conflicts is desirable, for this leads to fewer values being duplicated to avoid the remaining conflicts.

Consider the situation where the instructions have three operands each, and after running the coloring heuristic, the set of values that should have at least two copies has been determined. The placement algorithm has to be used to place the first copy of each of the values in the set.

Claim: Placing the first copies so that maximum number of conflicts is avoided is NP-complete, for solving the placement problem in this case is the same as finding the largest bipartite subgraph G_1=(V, E_1) of a graph $G = (V, E)$ where $E_1 \subseteq E$ [11].

Proof: To show this, an instance of the placement problem from an instance of bipartite subgraph problem is first constructed in polynomial time. Then it is shown that given a solution for one, the solution to the other can be found in polynomial time.

For each edge e=(v_1,v_2)$\in E$, construct a three-operand instruction consisting of M_1, v_1, and v_2. Here M_1 represents a value that has only a single copy that has been placed in memory module M_1. As a result, in order to make the instructions so constructed conflict free, the values corresponding to the vertices in the graph have to be placed either in memory module M_2 or M_3. From the construction it is obvious that the placement that avoids the maximum number of conflicts also converts the graph into a bipartite graph if the edges, corresponding to instructions that still have an access conflict, are removed from the graph. The values placed in M_2 form one half of the bipartite subgraph and the values placed in M_3 the other half. □

To decide where to place the values, all of the instructions that have conflicts are examined and an attempt is made to find a placement that avoids as many conflicts as possible. First of all, the instructions are divided into groups according to the number of operands with single copies present. The conflicts in instructions with $(k-1)$ operands with single copies are avoided first. The instructions with only one operand with a single copy are examined last, for the higher the number of single copy operands, the fewer is the number of candidates for duplication.

After grouping the instructions in the manner described, one variable at a time is taken and the memory module where its value should be placed is determined. For each instruction in the first group the set of memory modules where the value may be placed to avoid the conflict is determined. The value is placed in the memory module which helps avoid the maximum number of conflicts. The order in which the variables are processed when placing their values determines the placement. The order is determined by counting the number of instructions that involve each of the variables whose values are to be placed. The variable that occurs in the maximum number of instructions with memory access conflicts is processed first. The placement heuristic is summarized in Fig. 12. Additional details regarding the performance of the hitting set heuristic can be found in [15].

Claim: The run-time complexity of the duplication and placement algorithms is $O(kn^{2k})$ or $O(ki^2)$, where n is the number of data values in $V_{\text{unassigned}}$, i is the number instructions with access conflicts after graph coloring and k the number of memory modules.

Proof: The placement algorithm involves construction of sets of memory modules that are candidates for the placement of the value of a variable. If i is the number of instructions with access conflicts after graph coloring, the construction of sets takes $O(k!i)$ time as $k!$ is the maximum number of ways the memory modules can be assigned to the instruction operands. Making a choice of a memory module for the placement of a value takes $O(i)$ time. Thus, the total time spent on the placement of a single value of a data value is $O(k!i)$. This process may have to be repeated nk times, k times for each data value. The maximum number of instructions i is ${}^nC_k$. Thus, the total time spent in placing the values is $O(nkk!i)$ or $O(kn^{k+1})$. The duplication algorithm involves constructing sets of variables and then finding the hitting set for these sets. Constructing the sets when considering combinations of j operands takes $O(j!{}^nC_j)$ time as ${}^nC_j$ is the maximum possible number of j operand combinations. Finding the hitting set takes $O({}^nC_j^2)$· time. The total time spent on duplication is $\sum_{j=2}^{k}(j!{}^nC_j + {}^nC_j^2)$ which equals $O(k^nC_k^2)$ or $O(ki^2)$ if $k \leq n/2$. The total time taken by the algorithm is the sum of the time spent on duplication and placement which is $O(kn^{2k})$ or $O(ki^2)$. □

IV. Experimental Results

The techniques presented were implemented as part of a compiler for a reconfigurable long instruction word architecture [12], [13]. Experiments were conducted to determine the degree of duplication for a set of programs. The results obtained for the backtracking approach and the hitting set approach, given in Sections III-B1 and III-B2, were quite similar for the test

TABLE I
DUPLICATION OF DATA

	STOR1		STOR2		STOR3	
	=1	>1	=1	>1	=1	>1
TAYLOR1	79	1	62	18	74	6
TAYLOR2	53	0	51	2	51	2
EXACT	66	0	57	9	60	6
FFT	20	0	19	1	20	0
SORT	7	0	4	3	7	0
COLOR	21	0	21	0	19	2

programs considered and thus, only the results obtained using the second approach are presented. The test cases included programs to compute Taylor coefficients for complex (TAYLOR1) and real (TAYLOR2) analytic functions, solve a set of linear equations using residue arithmetic (EXACT), fast Fourier transform (FFT), sorting using quicksort (SORT), and the coloring algorithm (COLOR) presented in this paper.

The coloring algorithm used to assign memory modules to data values required the construction of a graph representing conflicts. An implementation of this algorithm is likely to impose a restriction on the size of this graph. Different memory module assignment strategies were used to study the effect of restricting the size of the graph. In the first strategy, STOR1, conflicts among all the variables and temporaries in the program were considered simultaneously, i.e., no restriction on the size of the graph was imposed. In practice, however, the size of the graph may be too large to use the strategy. The next two strategies limit the size of the graphs. In the second strategy, STOR2, the memory module assignment for the data values was carried out in two stages. In the first stage the variables live across regions were assigned memory modules. In the second stage, variables and temporaries local to a region were assigned memory modules. Thus, at a given time only a subset of variables, and hence conflicts, are considered in this strategy which limits the size of the graph constructed. In the third strategy, STOR3, the size of the graph was restricted by limiting the number of instructions processed at a time. In the experiment conducted, the instructions were split into two groups.

The results of the experiments are presented in Table I, where the first column of each strategy (=1) indicates the number of scalars that had single copies and the second column (>1) gives the number of scalars that had multiple copies. In these experiments the system had eight memory modules. Almost no duplication had to be done to avoid memory access conflicts when strategy STOR1 was used. An increase in the amount of duplication was caused when STOR2 and STOR3 were used. However, the duplication caused by STOR3 was significantly lower than the duplication caused by STOR2. This indicates that the allocation is better if the size of the graph is restricted by limiting the number of instructions processed at a time. The performance of STOR2 was poor compared to STOR3 because during the allocation of storage for global variables, very few conflicts are considered, for the majority of operands for an instruction are data values local to a region and very few operands represent global data values. The results obtained for strategies STOR1 and STOR3 indicate that most memory access conflicts can be avoided with very little duplication of data.

In the preceding sections, the worst case run-time complexities of the coloring algorithm and the duplication and placement strategies were analyzed. These results indicated that the coloring heuristic was relatively inexpensive when compared to the duplication and placement algorithms. The experimental results in Table I indicate that in strategy STOR1 almost all scalars have a single copy. In other words, almost all conflicts can be avoided by using the coloring algorithm alone. Therefore, one approach that may be used is to resolve as many conflicts as possible using the coloring algorithm and leave any remaining conflicts unresolved. This implementation is practical for a production quality compiler as it will result in acceptable compilation times. Register allocation techniques that construct interference graphs and perform register allocation by coloring the graph are commonly used in production quality compilers [4], [5], [17]. If the access conflicts are not resolved, the execution time of a program can increase significantly. For the above programs the number of scalars accessed simultaneously was usually greater than one. In program EXACT, the execution of about 80% of the instructions required access to two or more scalar operands. About 30% of the instructions required more than two operands and the maximum number of operands required by any instruction was five. In case of program COLOR over 75% of the instructions required two or more operands. Thus, if the memory conflicts are not resolved the amount of time spent during memory accesses will at least double since most instructions require more than one operand.

The memory access conflicts due to array references cannot be detected at compile time. To measure the deterioration in performance due to these conflicts, an experiment to measure the increase in time spent on memory transfers due to memory access conflicts caused by array references was conducted. The results of the experiment are presented in Table II. In these results, time $t_{\min}$ is the time spent on performing the memory transfers if no memory conflicts occur due to array references. Time $t_{\max}$ is the time spent on performing memory transfers assuming every array access causes a memory access conflict. This can only occur if the storage required for all of the arrays used by a program is allocated from the same memory module. In practice, the elements of the same array will be distributed uniformly among the memory modules. A more realistic estimate of the time spent on performing memory transfers is t_{average}. In computing time t_{average} it was assumed that the probability of the required array element being in any of the memory modules is the same. The average time spent on performing memory transfers

TABLE II
MEMORY CONFLICTS DUE TO ARRAY ACCESSES

	$M=\langle M_1, M_2, \cdots, M_8\rangle$		$M=\langle M_1, M_2, \cdots, M_4\rangle$	
	$t_{\text{average}}/t_{\min}$	$t_{\max}/t_{\min}$	$t_{\text{average}}/t_{\min}$	$t_{\max}/t_{\min}$
TAYLOR1	1.08	1.25	1.10	1.18
TAYLOR2	1.04	1.19	1.08	1.18
EXACT	1.08	1.17	1.09	1.14
FFT	1.02	1.10	1.04	1.09
SORT	1.11	1.31	1.12	1.19
COLOR	1.15	1.38	1.20	1.30

for an instruction, t_{average}, was computed as follows:

$$t_{\text{average}} = \Delta\, p(1) + 2\Delta\, p(2) + \cdots + n_m\, \Delta\, p(n_m) = \sum_{i=1}^{n_m} i\, \Delta p(i)$$

where the time required to supply the operands required for an instruction in absence of memory access conflicts is Δ and $p(i)$ is the probability of the instruction requiring i operands from the same memory module. Thus, in the above computation, it is assumed that for every data transfer that a memory module performs, time Δ is needed.

The results in Table II show that in worst case up to 38% increase in the time spent on memory transfers is observed. However, this is highly unlikely to occur in practice as the array elements will be distributed among the memory modules and not stored in the same memory module. The results in Table II also show that on an average 2–20% increase in the time spent on memory transfers was observed due to memory access conflicts caused by array references. Since the overall execution time of a program includes the time spent on performing the operations also, the percentage increase in the overall execution time is even less. Thus, the expected reduction in the speed of execution due to memory access conflicts caused by array references is less than 20%. The results obtained for the overall speedup in execution on the reconfigurable long instruction word (RLIW) system varied from 64–300%. Compared to the overall speedup, the reduction in speed due to the memory access conflicts caused by array references is small. Thus, it can be concluded that the memory access conflicts, predictable or unpredictable at compile time, do not cause any appreciable deterioration in the performance of the system.

V. CONCLUSION

In this paper, compile-time techniques for distribution of scalars to avoid memory access conflicts were presented. The techniques have been implemented as part of a compiler for a reconfigurable long instruction word architecture. Results of experiments demonstrate that a very high percentage of memory access conflicts can be avoided by scheduling a very low number of data transfers.

It should be pointed out that renaming transformations, [7], [18], useful to increase potential parallelism in programs, can also benefit this memory allocation problem. Work has been done by Ferrante and Cytron [7] to transform Fortran-like programs into programs whose only constraint on the order of execution is the direct flow of values through renaming. Thus, renaming removes all storage-related dependences that might impose constraints on the order of execution of operations. The idea behind renaming is to replace a global variable having several definitions and uses by a number of variables, each with a lesser number of definitions and uses. In the process, dependences among computations that existed due only to the use of the same variable name are eliminated. This not only increases the parallelism in the code but also makes the memory module assignment without memory access conflicts easier, for instead of assigning the variable to the same memory module for the entire program, each renamed definition can be assigned to a different memory module.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.

[2] K. E. Batcher, "The multidimensional access memory in STARAN," *IEEE Trans. Comput.*, pp. 174–177, Feb. 1977.

[3] P. P. Budnik and D. J. Kuck, "The organization and use of parallel memories," *IEEE Trans. Comput.*, vol. C-20, pp. 1566–1569, Dec. 1971.

[4] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Comput. Languages*, vol. 6, pp. 47–57, 1981.

[5] F. Chow and J. Hennessy, "Register allocation by priority-based coloring," in *Proc. SIGPLAN'84 Symp. Compiler Construction, SIGPLAN Notices*, vol. 19, no. 6, pp. 222–232, June 1984.

[6] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," *IEEE Trans. Comput.*, vol. 37, pp. 967–979, Aug. 1988.

[7] R. Cytron and J. Ferrante, "What's in a name? or the value of renaming for parallelism detection and storage allocation," in *Proc. Int. Conf. Parallel Processing*, Aug. 1987, pp. 19–27.

[8] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*. Cambridge, MA: MIT Press.

[9] J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Programming Languages Syst.*, vol. 9, no. 3, pp. 319–349, July 1987.

[10] J. A. Fisher, "The VLIW machine: A multiprocessor for compiling scientific code," *IEEE Comput. Mag.*, pp. 45–53, 1984.

[11] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.

[12] R. Gupta and M. L. Soffa, "A reconfigurable LIW architecture," in *Proc. Int. Conf. Parallel Processing*, Aug. 1987, pp. 893–900.

[13] R. Gupta, "A reconfigurabhle LIW architecture and its compiler," Ph.D. dissertation, Tech. Rep. 87-3, Dep. Comput. Sci., Univ. of Pittsburgh, Aug. 1987.

[14] R. Gupta and M. L. Soffa, "Compilation techniques for a reconfigurable LIW architecture," *J. Supercomput.*, vol. 3, pp. 271–304, 1989.

[15] ——, "Compile-time techniques for efficient utilization of parallel memories," Tech. Rep. TR-89-23, Univ. of Pittsburgh, Oct. 1989.

[16] D. T. Harper III and J. Jump, "Vector access performance in parallel memories using a skewed storage scheme," *IEEE Trans. Comput.*, vol. C-36, no. 12, pp. 1440–1449, Dec. 1987.

[17] S. Jain and C. Thompson, "An efficient approach to data flow analysis in a multiple data pass global optimizer," in *Proc. SIGPLAN'88 Conf. Programming Language Design and Implementation*, June, 1988, pp. 154–163.

[18] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations, "in *Proc. 8th Annu. ACM Symp. Principles Programming Languages*, 1981, pp. 207–218.

[19] D. J. Kuck, "ILLIAC IV software and application programming," *IEEE Trans. Comput.*, vol. C-17, no. 8, pp. 758–770, Aug. 1968.

[20] D. J. Kuck and R. A. Stokes, "The Burroughs Scientific Processor (BSP)," *IEEE Trans. Comput.*, vol. C-31, no. 5, pp. 363–376, May 1982.

[21] M. E. Mace and R. E. Wagner, "Globally optimum selection of storage patterns," IBM Res. Rep. RC 10676, IBM T. J. Watson Research Center, Yorktown Heights, Aug. 1984.

[22] M. E. Mace, *Memory Storage Patterns in Parallel Processing*. New York: Kluwer Academic, 1987.
[23] R. E. Tarjan, "Decomposition by clique separators," *Discrete Math.* vol. 55, pp. 221–231, 1985.

Rajiv Gupta received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, New Delhi, in 1982, and the Ph.D. degree in computer science from the University of Pittsburgh, Pittsburgh, PA, in 1987.

After completing the Ph.D. degree, he joined Philips Laboratories as a senior member of the Research Staff in the Computer Architecture and Programming Systems group. Currently, he is an Assistant Professor in the Department of Computer Science at the University of Pittsburgh. His primary research interests include compilation techniques for parallel systems, parallel architectures, and implementation of programming languages.

Dr. Gupta is a member of the Association for Computing Machinery, SIGPLAN, and the IEEE Computer Society.

Mary Lou Soffa received the Ph.D. degree in computer science from the University of Pittsburgh, Pittsburgh, PA, in 1977.

She has been on the faculty of the University of Pittsburgh since 1977 and is currently a Professor in the Department of Computer Science. She was awarded an NSF Visiting Professorship for Women in 1987 to spend a year at Berkeley. Her main areas of research interest are compiling techniques for parallel computers, incremental compilation, programming languages, and software tools.

Dr. Soffa serves on the Editorial Advisory Board for *Computer Languages* and is a member of the Association for Computing Machinery, SIGPLAN, SIGSOFT, and the IEEE Computer Society.