Compile-time Techniques for Efficient Utilization of Parallel Memories*

Rajiv Gupta
Philips Laboratories
North American Philips Corporation
Briarcliff Manor, NY 10510

Mary Lou Soffa
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, Pa. 15260

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, no 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 avoided 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.

Keywords - memory bandwidth, parallel memories, memory access conflicts, long instruction word architectures.

1. 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[3, 15, 14, 1, 4]. The memory modules are shared by the processors which access them through an interconnection network. 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 slowed down as the operands cannot be accessed in parallel. Work has been done to improve access performance of *vectors* by storing them in a skewed fashion in parallel memories[12, 3]. In previous work by Mace and Wagner, techniques to determine how to store data for performing vector operations were developed[17, 16].

In practice there are always parts of a program that operate on scalar data rather than vectors. In order to achieve high overall speed-up, 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. 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. 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)

* This work was done at the Univ. of Pittsburgh and was partially supported by NSF under Grant DCR 811934.

architectures[7,9], 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. The TRACE[4], a LIW machine developed by Multiflow, allows for up to eight memory controllers each of which carries up to eight independent memory modules.

In order to avoid memory access conflicts, the operands required by the operations to be executed in parallel must first be determined. Following this, the values 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 that are scheduled at compile-time. The transfers can result in increased execution time. Thus, an attempt should be made to minimize 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 carried out.

This paper presents compile-time techniques for storage allocation of scalars 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. An instruction is composed of the operations that execute in parallel and the corresponding operands. However, the operands required by an instruction cannot always be determined at compile time. In the presence of arrays, the specific array element required for an operation may only be known at run-time. In this work algorithms for avoiding those memory access conflicts that can be predicted at compile time are described. Complexity analyses has been done[10] and the results are briefly stated. The techniques have been implemented as part of a compiler for a reconfigurable long instruction word (RLIW) architecture[10,11]. 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 do not cause significant deterioration in performance. Finally, other applications of the technique are discussed.

2. 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 accesses performed by the same instruction are known. Thus, when binding the addresses to memory modules, an attempt can be 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 a unmanageable task. One solution to this problem is to perform the memory module assignment for one program region[6] at a time. Another approach is to carry out the module assignment in two phases: **global** module assignment and **local** module assignment. During global module assignment, the binding of data values that are used in more than one region[6] of the program can be carried out. 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 which data values are accessed in the same instruction and hence should be stored in different memory modules.

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

Problem: Given memory $M=<M_1,M_2.....M_k>$ where memory modules $M_1,M_2.....M_k$ can be accessed in parallel and a sequence of instructions each of which requires up to k operands from among data values $V_1,V_2.....V_n$.

Goal: Allocate storage for $V_1,V_2.....V_n$ among memory modules so that the instructions can be executed without encountering any memory access conflicts.

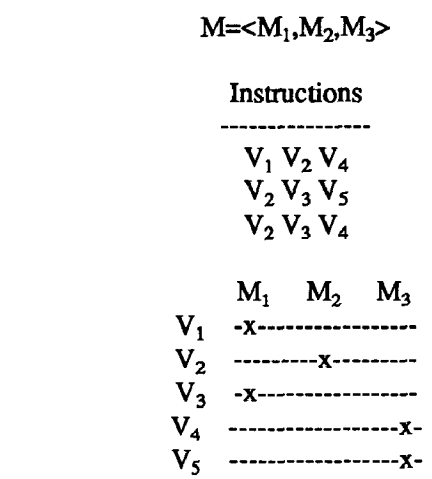

Fig. 1. Avoiding Access Conflicts

The example given in Fig. 1 shows how the storage can be allocated avoiding 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 k copies of a variable may be required with one copy in each memory module to avoid all memory conflicts. In the above example, if the following operand usage 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 can be stated as follows:

Problem: Given memory $M=<M_1,M_2.....M_k>$ where the memory modules $M_1,M_2.....M_k$ can be accessed in parallel and a sequence of instructions each of which requires up to k operands from among data values $V_1,V_2.....V_n$.

Goal: Allocate storage for $V_1,V_2.....V_n$ among 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 no data value is ever updated.

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 are stored 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 them is constructed. Finding whether an allocation exists that avoids all the conflicts or not is the same as determining whether the graph constructed in the above fashion can be colored using k colors, where k is interpreted as the number of memory modules in the system. It is unlikely that the problem can be solved in polynomial time because determining whether an arbitrary graph is k-colorable for a fixed k is an NP-complete problem[8]. Therefore a heuristic is used for coloring that removes nodes from the graph when 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 ($V_{unassigned}$), are avoided by *duplicating* these values and *placing* them in different memory modules.

Memory Module Assignment

{

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

Using *graph coloring* assign values to memory modules

$M_1, M_2..M_k$ such that accesses to the values

$V_{assigned} \subseteq V$, assigned to memory modules, do not conflict.

Let $V_{unassigned} = V - V_{assigned}$, 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 $V_{unassigned}$.

Placement: distributing these copies among memory modules.

}

Fig. 2. Overall Strategy for Memory Module Assignment

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. 2.

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 need to have three copies are used, a sub-optimal solution may be obtained. This is demonstrated by the example in Fig. 3. 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 after placing two copies of values V_4 and V_5, a memory conflict still exists. This conflict is 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 the same number of nodes were removed in both cases the second solution resulted in less copying of data values. This shows that even if a coloring algorithm that removes a minimum number of nodes from the graph is used, the resulting solution may be sub-optimal.

$M=<M_1,M_2,M_3>$

Instructions

$V_1\ V_2\ V_3$

$V_2\ V_3\ V_4$

$V_1\ V_3\ V_4$

$V_1\ V_3\ V_5$

$V_2\ V_3\ V_5$

$V_1\ V_4\ V_5$

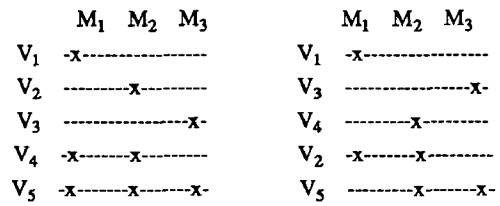

Fig. 3. Choosing Nodes to be Removed

2.1. 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 graph is decomposed into atoms which are subgraphs that do not have clique separators[18]. 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[18]. Thus the coloring algorithm need only concern itself with coloring the atoms rather than the entire graph at the same time. When coloring an atom, a heuristic removes nodes whenever it becomes impossible to continue coloring.

```
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 */
  ∀ (n_i,n_j) ε E,
    if d(n_i)<k then wt(n_j→n_i) = 0 else wt(n_j→n_i) = conf(n_i,n_j)
  ∀ n_i compute S_{n_i} = Σ_{n_j} wt(n_i→n_j), where (n_i,n_j) ε E
  /* Initialize Sets */
  n_first = n_i such that S_{n_i} = max_{n_j} S_{n_j}
  ASSIGN(n_first) = M_1;   V_assigned = {n_first}
  V_rest = V − {n_first};   V_unassigned = φ
  while V_rest ≠ φ
   {
    Choose n_next = n_i st U_{n_i} = max_{n_j} U_{n_j},
                          Σ_{n_k} wt(n_k→n_j)
     Urgency U_{n_j} = ------------------, where
                            K_{n_j}
      n_k ε V_assigned,
      n_j ε V_rest,
      (n_k,n_j) ε E,
      K_{n_j} is the number of modules still assignable to n_j.
     if K_{n_next} = 0 then V_unassigned = V_unassigned ∪ {n_next}
     else {
          ASSIGN(n_next) = one of the available modules
          V_assigned = V_assigned ∪ {n_next}
        }
     V_rest = V_rest − n_next
    }
 }
```

Fig. 4. Heuristic for Graph Coloring

The heuristic developed for coloring an atom G=(V,E) is described in Fig. 4. 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}). In order to choose the node to be colored next, the urgency for each uncolored node 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 till each node has either been colored or cannot be colored.

The example in Fig. 5 demonstrates how the above algorithm is used. In the figure, each edge (n_j,n_i) in the graph (representing conflicts) 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.

Run-time Complexity: The above algorithm has been implemented with the running time of $O((n+e)\log(n+e))$, or $O(n^2 \log n)$ in the worst case, where $n = |V|$ and $e = |E|$.

Worst Case Performance: It can be shown that in the worst case the heuristic may leave (n-k) nodes uncolored while the optimal only leaves two nodes uncolored, where n is the number of nodes in the graph. Therefore the ratio of heuristic to optimal is $\frac{\text{heuristic}}{\text{optimal}} = \frac{(n-k)}{2}$.

2.2. Duplication and Placement Strategies

After determining what values have to be replicated ($V_{unassigned}$) 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 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 to avoid all conflicts. As a result, a copy of a value created can help in

avoiding conflicts in several instructions, leading to a lesser degree of duplication.

$M=<M_1,M_2,M_3>$

Instructions

$V_1\ V_2\ V_3$
$V_2\ V_3\ V_4$
$V_5\ V_1\ V_2$
$V_3\ V_4\ V_5$

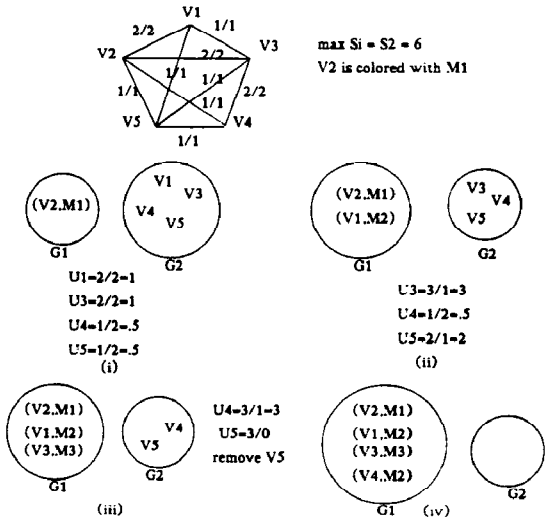

Fig. 5. Applying the Coloring Heuristic

2.2.1. 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 $V_{unassigned}$. 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 algorithm for duplication efficient to use. The instructions that have only one operand in $V_{unassigned}$ are examined first and the instructions in which all k operands are in $V_{unassigned}$ 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 then creates and places new copies in other modules if the conflicts cannot be avoided. The placement that requires the least number of additional copies to be created is used. If there is more than one solution, a random choice is made. The reason for ordering the instructions is as follows. A conflict in an instruction that 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. 6.

Backtrack

{
Divide the instructions into sets $S_1,S_2.....S_k$ such that
$S_i = \{I:$ instruction I has i operands in $V_{unassigned}\}$
for i = 1 .. k loop
{
$\forall\ I \varepsilon S_i$
Let $O_1,O_2..O_i$ denote the operands in I from $V_{unassigned}$
Using *backtracking* determine all module assignments
$P_j = \{(O_1,M_{k_1}),(O_2,M_{k_2})..(O_i,M_{k_i})\}$ 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, and choose the placement P_{min} such that
$CP_{min} = \min_j CP_j$
and create the additional copies of the operands.
}
}

Fig. 6. Approach Based on Backtracking

Run-time Complexity: The run-time complexity of the algorithm is O(k! i) or $O(n^k)$, where i is the number of instructions with access conflicts after graph coloring, and n is the number of values in $V_{unassigned}$.

Worst Case Performance: The drawback of examining the instructions one at a time is that the copies created for an instruction are less likely to be used by other instructions than if the requirements for all instructions were determined before placing copies. It can be shown that (k-1) times more copies than the optimal number of copies may be

created by the above algorithm.

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 $V_{dup} \subseteq V$ that should be duplicated to avoid conflicts in instructions with *op* operands. Duplication requires the use of a heuristic for finding hitting sets. */
{
Let $S_i^n = \{OP_1, OP_2 \cdots OP_n\}$ denote a combination of n operands st $\exists$ j $S_i^n \subseteq I_j^k$
/* place the first copies of operands yet to be assigned modules */
Place($V_{unassigned}$)
Place($V_{unassigned}$) st $\forall$ i S_i^2 is conflict free
for num = 3 .. k loop
{
Duplication Phase:
Using the hitting set heuristic determine $V_{dup} \subseteq V_{unassigned}$ st if an additional copy of each value in V_{dup} is made $\forall$ i S_i^{num} is conflict free
Placement Phase: Place(V_{dup})
}
}

Fig. 7. The Hitting Set Approach

2.2.2. The Hitting Set Approach

As mentioned earlier, in the second approach all instructions are examined before the decision on what values to replicate is taken. The motivation for this approach is as follows. In general it may be possible to avoid a conflict present 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 $V_{unassigned}$ 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 $V_{unassigned}$ are created and placed to avoid all conflicts in combinations of 3 or more operands. In order to determine the number of copies of data values that are needed to avoid memory access conflicts, the following procedure is used. First, all combinations of (k-1),(k-2),(k-3)....3 operands that occur together in any of the instructions are determined. In order to avoid all memory access conflicts the memory access conflicts in combinations of 3....k operands generated from the instructions have to be avoided. The choice of placement of values to avoid conflicts among pairs of operands is important as this will determine what combinations of three operands still conflict and thus require additional copies of values to be made. Now if the combinations of three operands are examined, some of the memory access conflicts may still be unresolved. These conflicts are resolved by making an additional copy of a subset of values that already have more than one copy. 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. 7, 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. The example in Fig. 8 demonstrates the importance of proper placement of values. In this example access conflicts are avoided by creating multiple copies of V_4 and placing them in different memory modules. In solution 1 the first two copies are placed in memory modules M_2 and M_4, and in solution 2 they are placed in modules M_1 and M_4. Next additional copies of V_4 are made to avoid conflicts in all combinations of three and four operands. The choice of modules made in solution 1 results in four copies of V_4 while solution 2 requires only three copies of V_4. This demonstrates that the choices made in the placement of values influences the number of copies created.

In the subsequent sections, the problems of determining the subset of values to be duplicated and the placement of the copies made is considered separately. First heuristics for the duplication of values and the placement of these values are presented. Next the combined run-time complexity of the duplication and placement algorithms is presented. Lastly the overall performance of the hitting set approach, assuming that a fixed strategy of placement is being used, is analyzed.

$M=<M_1,M_2,M_3,M_4>$

Instructions

$V_1\ V_2\ V_3\ V_5$
$V_4\ V_2\ V_3\ V_5$
$V_1\ V_2\ V_3\ V_4$
$V_4\ V_2\ V_1\ V_5$

During coloring V_4 is removed

Solution 1

$M_1\ M_2\ M_3\ M_4$
V_1 -x----------
V_2 ------x----------
V_3 ----------x-----
V_5 ----------x-
V_4 ------x--------x-

$M_1\ M_2\ M_3\ M_4$
V_1 -x----------
V_2 ------x----------
V_3 ----------x-----
V_5 ----------x-
V_4 -x----x--------x-

Solution 2

$M_1\ M_2\ M_3\ M_4$
V_1 -x----------
V_2 ------x----------
V_3 ----------x-----
V_5 ----------x-
V_4 -x----------x-

$M_1\ M_2\ M_3\ M_4$
V_1 -x----------
V_2 ------x----------
V_3 ----------x-----
V_5 ----------x-
V_4 -x--------x---x-

$M_1\ M_2\ M_3\ M_4$
V_1 -x----------
V_2 ------x----------
V_3 ----------x-----
V_5 ----------x-
V_4 -x---x---x---x-

Fig. 8. Placement of Values

2.2.2.1. 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. It is shown how 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 instruction 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 a 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 $\{V_1,V_2,V_3\}$.

In each of the above cases there is at least one data value that has two copies. This is because for a combination of three values, each of which has one copy, it must be true that the values are stored in different memory modules and thus the combination must be free of memory access conflicts.

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

{
$HS = \bigcup_j s_j$, $|s_j| = 1$
for size = 2 .. k loop
{
Let S_{v_iP} = # of sets s such that $v_i \in s \wedge |s| = p$
$\forall\ s_j = \{v_1,v_2 \cdots v_{size}\}$ such that $s_j \cap HS = \phi$
$HS = HS \cup v_n$ where $v_n \in s_j$ and $\forall v_l \in s_j$ st $v_l \neq v_n$
$\nexists\ m \le k$ st $(S_{v_jm} > S_{v_nm}) \wedge (S_{v_ji} = S_{v_ni}, i = size....m-1)$
}
}

Fig. 9. Heuristic for finding the Hitting Set

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 these sets of values, at least one value is chosen from each of the sets 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[8]. Therefore in order to find the hitting set, the heuristic given in Fig. 9 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.

2.2.2.2. 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. Placing the first copies so that maximum number of conflicts are avoided is NP-complete, for solving the placement problem in this case is same as finding the largest bipartite subgraph[8] $G_1=(V,E_1)$ of a graph $G=(V,E)$ where $E_1 \subseteq E$.

Place: a copy each of value $v_i \in HS$

{

Divide I the set of instructions into groups $I_1 \cup I_2 \cup .... I_k = I$
st each instruction in I_i contains i operands in $V_{unassigned}$

$\forall$ 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 $1 \le i \le k$ and $i \neq p$

$\exists$ $z \le k$ st $(C_{M_i I_z}(v) > C_{M_p I_z}(v)) \wedge$

$(C_{M_i I_\alpha}(v) = C_{M_p I_\alpha}(v)$, $\alpha = 1 .. z-1)$

Place a copy of v in M_p

}

}

Fig. 10. Placement Algorithm

In order 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. Consider a k-operand instruction with an access conflict which has (k-1) operands with one copy each. This instruction has only one operand that can be duplicated. There is only one memory module where the copy of the operand can be placed to avoid the conflict. Similarly consider an instruction with (k-2) operands with one copy each and a memory access conflict. In this case, there are two choices for the placement of the values of the two operands allowed to have more than one copy. It is possible that same variable may be involved in conflicts in instructions of both types described above and the choice of memory module in the first case may also avoid the conflict in the second instruction. Thus an attempt is made to avoid conflicts in instructions with (k-1) operands with single copies first. The instructions with (k-2) operands with single copies are considered next and, last of all, the instructions with only one operand with a single copy are examined.

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. If there is more than one choice the next group of instructions are examined and the best choice for this group of instructions is determined. This process continues until either the conflict is avoided or there are no more groups of instructions, in which case a random choice is made. 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 in the first group 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. If the variables cannot be ordered on the basis of the first group of instructions, the subsequent groups are used. The algorithm is summarized in Fig. 10.

Run-time Complexity: The run-time complexity of the duplication and placement algorithms is $O(ki^2)$ or $O(kn^{2k})$, where i is the number of instructions with access conflicts after graph coloring, n is the

number of data values in $V_{unassigned}$. The total time spent in placing the values is $O(kn^{k+1})$. The total time spent on duplication is $O(kn^{2k})$. Thus the total time taken by the algorithm is sum of the time spent on duplication and placement which is $O(kn^{2k})$.

Worst Case Performance: The worst case performance of the hitting set heuristic is expressed in terms of m, the number of different sets in which an element can occur, is as follows:

$$\frac{\text{heuristic}}{\text{optimal}} = H_m = (1+\frac{1}{2}+\frac{1}{3}+.....+\frac{1}{m})$$

The overall worst case performance was analyzed using a fixed strategy for the placement of values. Assume that the first two copies of the values in $V_{unassigned}$ are placed in memory modules M_1 and M_2. A fixed strategy of placement P, is defined as a one to one mapping $P:I \longrightarrow M$ where $M=\{M_3,M_4.....M_k\}$ is the set of memory modules and $I=\{3,4....k\}$ is the number of operands in combinations being considered in the current iteration of the algorithm. For a fixed placement strategy it can be shown that the replicating procedure can create H_m times the copies that need to be created[10].

3. Experimental Results and Conclusions

The techniques presented were implemented as part of a compiler for a reconfigurable long instruction word architecture[10,9]. 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 2.2.1 and 2.2.2, were quite similar and thus, only the results obtained using the second approach are presented. The test cases include 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 graph 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, for large programs, the size of the graph may be too large to be practical. 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.

Table 1. 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

The results of the experiments are presented in Table 1. In Table 1 the first column (=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. >From the results obtained for strategies STOR1 and STOR3, it can concluded that most memory access conflicts can be avoided with very little duplication of data.

The memory access conflicts due to array references cannot be detected at compile time. In order 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 2. 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_{ave}. In computing time t_{ave} 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 for an instruction, t_{ave}, was computed as follows:

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

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.

Table 2. Memory Conflicts due to Array Accesses

	$M = <M_1, M_2, .., M_8>$		$M = <M_1, M_2, .., M_4>$	
	t_{ave}/t_{min}	t_{max}/t_{min}	t_{ave}/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

The results in Table 2 show that in the 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 2 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 speed-up in execution on the reconfigurable long instruction word (RLIW) system varied from 64-300%. Compared to the overall speed-up, 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.

It should be noted that the results would likely be improved by first applying renaming[13,5] techniques to the code to remove storage related dependences. This is because instead of assigning a variable to the same memory module for the entire program, each renamed definition can be assigned to a different memory module. These techniques can also be used in shared cache multiprocessor systems. In systems where the caches are associated with the shared memory[2], the shared data can reside in the shared caches and can be accessed in parallel by the processors at high speed. However, the performance of the system can deteriorate if multiple hits occur on the same cache. Information on access frequency of shared data items can be used to determine a distribution of data items in the memory modules which is likely to avoid multiple hits on the same cache. If the data is read-only, then the techniques described in this paper can be used to create multiple copies of data items which are stored in different main memory modules. The Alliant FX/8 is an example of a machine that supports shared caches.

References

1. K.E. Batcher, "The Multidimensional Access Memory in STARAN," *IEEE Trans. on Computers*, pp. 174-177, Feb., 1977.

2. F.A. Briggs and K. Hwang , *Computer Architecture and Parallel Processing*, McGraw-Hill, Inc. , 1984.

3. P.P. Budnik and D.J. Kuck, "The Organization and Use of Parallel Memories," *IEEE Trans. on Computers*, vol. C-20, pp. 1566-1569, Dec, 1971.

4. 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," *Proc. Second International Conf. on Architectural Support for Programming languages and Operating Systems*, pp. 180-192, 1987.

5. R. Cytron and J. Ferrante, "What's In a Name? -or- The Value of Renaming for Parallelism Detection and Storage Allocation," *Proc. International Conf. on Parallel Processing*, pp. 19-27, August, 1987.

6. J. Ferrante, K. Ottenstein, and J. Warren, "The Program Dependence Graph and its Use in Optimization," *Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319-349, July, 1987.

7. J.A. Fisher, "The VLIW Machine: A Multiprocessor for Compiling Scientific Code," *Computer*, pp. 45-53, 1984.

8. M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, 1979.

9. R. Gupta and M.L. Soffa, "A Reconfigurable LIW Architecture," *Proc. of the International Conf. on Parallel Processing*, pp. 893-900, August, 1987.

10. R. Gupta, "A Reconfigurable LIW Architecture and its Compiler," Dept. of Computer Science; Ph.D. dissertation, Tech. Report 87-3, University of Pittsburgh , August, 1987.

11. R. Gupta and M.L. Soffa, "A Matching Approach to Utilizing Fine-Grained Parallelism," *21st Annual Hawaii International Conference on System Sciences*, vol. I, pp. 148-156, Jan., 1988.

12. D.T. Harper III and J.R. Jump, "Vector Access Performance in Parallel Memories Using a Skewed Storage Scheme," *IEEE Trans. on Computers*, vol. C-36, no. 12, pp. 1440-1449, Dec., 1987.

13. D.J Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *8th Annual ACM Symp. on Principles of Programming Languages*, pp. 207-218, 1981.

14. D.J. Kuck, "ILLIAC IV Software and Application Programming," *IEEE Trans. on Computers*, vol. C-17, no. 8, pp. 758-770, August, 1968.

15. D.J. Kuck and R.A. Stokes, "The Burroughs Scientific Processor (BSP)," *IEEE Trans. on Computers*, vol. C-31, no. 5, pp. 363-376, May, 1982.

16. M.E. Mace and R.E. Wagner, "Globally Optimum Selection of Storage Patterns," *IBM, Research Report RC 10676*, T.J. Watson Research Center, Yorktown Heights, August, 1984.

17. M.E. Mace, *Memory Storage Patterns in Parallel Processing*, Kluwer Academic Publishers, 1987.

18. R.E. Tarjan, "Decomposition by Clique Separators," *Discrete Math.*, vol. 55, pp. 221-231, 1985.