# Effective Cluster Assignment for Modulo Scheduling

Erik Nystrom and Alexandre E. Eichenberger
ECE Department, North Carolina State University
Box 7914, Raleigh NC 27695-7915
emnystro, alexe @ eos.ncsu.edu

## Abstract

*Clustering is one solution to the demand for wide-issue machines and fast clock cycles because it allows for smaller, less ported register files and simpler bypass logic while remaining scaleable. Much of the previous work on scheduling for clustered architectures has focused on acyclic code. While minimizing schedule length of acyclic code is paramount, the primary objective when scheduling cyclic code is to maximize the throughput or steady state performance. This paper investigates a pre-modulo scheduling pass that performs cluster assignment in a way that minimizes performance degradation do to explicit communication required as the loops are split over clusters. The proposed cluster assignment algorithm annotates and adjusts the graph for use by the scheduler so that any traditional modulo scheduling algorithm, having no knowledge of clustering, can produce a valid and efficient schedule for a clustered machine.*

KEYWORDS: Cluster assignment, Modulo scheduling, Cluster architecture, ILP.

## 1. Introduction

### 1.1. Clustering

In order to take advantage of existing instruction level parallelism (ILP), the number of function units in processors have been increasing. With more function units executing instructions in parallel, the number of read and write ports required on the register file increases as well. Additionally, the register file must be larger to accommodate the increased demand for registers by operands [1]. Increasing both the size of a register file and the number of ports to the register file leads to a register file that limits the clock cycle of the processor or is impractical to build with current technology. This is due to fact that the IC area of a register file increases linearly with the number and size of registers and quadratically with the number of ports [2]. The cycle time of the register file increases as a logarithmic function of the number of registers and read ports [3].

Since functional units typically need two read ports but only one write port, one approach is to replicate the register file and spread the read ports among the replicated register files while writing all of the results to each replicated register file. The function units split into separate *groups*, each of which is supplied input data from a single register file. This scheme is referred to as *multiple coherent register files*, as a given register always has the same value in each of the replicated register files. For example, the IBM Power2 emulated a register file with 8 read ports and 4 write ports using two coherent register files with 4 read ports and 4 write ports [4]. This scheme has three major limitations. It consumes a large amount of die area for the replicated register files without adding extra register space, requires a lot of routing to provide the write ports to every replicated file, and does not reduce the number of write ports per replicated file.

Llosa et al. proposed a scheme, referred to as *dual non-consistent register files*, where each register can either be globally coherent or local to its function unit group [5]. Dual non-consistent register files reduce the register pressure because a given register may contain two distinct values in two distinct files if the value is never used by function units within a different group. However, this scheme still requires the same amount of write ports and routing and thus does not scale well for very wide machines.

Colwell et al. and Capitanio et al. propose a scheme where each of the replicated files is exclusively local to a group of function units [6] [3]. In this scheme, referred to as *clustering*, the register file is divided into multiple, disjoint register files and the function units are divided into multiple groups severing bypass logic between groups. Each disjoint register file is paired with a corresponding function unit group, which together is called a *cluster*. In order to communicate values between clusters, some form of explicit communication is required, referred to in this paper as a *copy* operation. A bus or direct point-to-point connections may be used to transmit values between clusters. In a clustered architecture, a register file needs only the read and write ports for the local function units, plus a small number to supply the bus or point-to-point network. Thus clustering allows for smaller, less ported register files and simpler bypass logic. The resulting system is scaleable and still allows the processor to take advantage of more ILP. For reference, we will refer to traditional, non-clustered machines as *unified machines*.

This paper targets various clustered architectures, including those that use buses to communicate values and those that use point-to-point connections. For bused configurations, both the number of buses and the number of read and write ports to the bus is varied. Details of the architectures can be found in Section 2.1.

## 1.2. Modulo scheduling

While clustering provides a way to implement scaleable VLIW/Superscalar processors, the compiler must expose enough parallelism so that the processor can take advantage of the available function units. Since there is generally not enough parallelism within a basic block, techniques such as trace scheduling, superblock scheduling, treegion scheduling, and software pipelining [7][8][9][10][11], exploit parallelism across basic block boundaries by moving operations beyond their original basic blocks thereby extracting more instruction level parallelism (ILP).

Software pipelining is a technique for exploiting ILP by overlapping successive iterations of a loop. Traditionally, each loop iteration is started only after the previous iteration completes. In software pipelining, however, the second iteration may be started before the first iteration completes, the third before the second completes, and so forth. Modulo scheduling [11] is a software pipelining technique that use the same schedule for each iteration of the loop and initiates each loop iteration after a fixed interval of cycles, referred to as *Initiation Interval* (II).

Algorithms for implementing modulo scheduling have matured for unified VLIW/Superscalar processors. For example, an iterative modulo scheduler [13] combined with a stage scheduler [14] can attain an optimal II for about 98% of the loops scheduled [15] and with no more than 3 registers above the minimal number for 95% of the loops of a large benchmark for a machine with complex resource use. Similar levels of performance can be can also be achieved though an iterative version of the swing modulo scheduler [16].

Various hardware and compiler techniques have expanded the types of loops for which modulo scheduling is applicable. For example, Predicated execution, All Paths Pipelining, and Superblock Modulo Scheduling allow modulo scheduling to be applied to loops with conditional code and early exits [17][18][19]. Predicated execution can also help reduce code expansion of a modulo scheduled loop [20]. A rotating register file or a technique called modulo variable expansion allows the schedules to avoid register conflicts due to the overlapping iterations [12][21].

While clustering the functional units and register file helps achieve a higher clock rate for the processor, overall modulo schedule performance will be improved only if the performance degradation due to explicit communication is small. Performance degradation may occur because of two factors: additional resources required by the copy operation and additional latencies due to the copy operation.

In this paper, we propose a technique to generate high throughput modulo schedule code for clustered machines with explicit, non-zero latency communication. Specifically, this paper investigates a pre-modulo scheduling cluster assignment phase that is followed by a traditional modulo scheduling phase. The assignment phase assigns operations to clusters and generates any required copy operations when the result of one operation is used by an operation on a different cluster. The assignment algorithm: (1) focuses on critical recurrence cycles to minimize the impact of communication latency, (2) predictively reserves space for future copy operations to minimize copy resource contention, and (3) uses an iterative algorithm to revert sub-optimal assignment decisions. Additionally, the algorithm is shown to be effective on a wide range of cluster configurations.

## 1.3. Result highlights and organization

Modulo scheduling results for each clustered machine are compared to an equally wide non-clustered machine in order to show the impact of clustering on modulo scheduling. The results show that the assignment algorithm allows for a modulo schedule on a clustered machine equal to that of a unified machine for 99% of the loops for two and four clusters of general purpose function units. For two and four clustered fully specialized function units, 95% of the loops match the II of a unified machine. The two and four cluster configurations above each had one bus per cluster.

A four clustered machine setup in a grid arrangement with only two point-to-point connections to each of two neighbors was modeled as well. The results show that the assignment algorithm allows for a modulo schedule on this machine equal to that of a unified machine for 92% of the loops.

The following sections elaborate on the cluster assignment algorithm. Section 2 provides an overview of the assignment process and target architectures. Section 3 presents an example which displays some major issues which must be considered during assignment. Section 4 details the specifics of the assignment process. Section 5 describes the experimental setup. Actual results on 1327 loops from the Perfect Club, SPEC-89, and the Livermore FORTRAN Kernels [22][23][24] follow in section 6. Section 7 presents our conclusions.

## 1.4. Related work

Along with the architectural specification for non-consistent dual register files, Llosa et al. proposed an algorithm for allocating registers on the two register files [5]. The algorithm consists of two phases. The first phase schedules operations onto a given function unit group and classifies register values as right-only, left-only, and global values. The second phase swaps operations between groups to further reduce the number of values classified as global.

The algorithm works well for register file configurations where communication between groups does not impose a transmission delay or resource use penalty because a result is simply labeled as global when its value is needed by function units attached to another register file. However, this algorithm does not apply to a clustered architecture where explicit copy operations consumer resources and have non-zero latencies.

Ellis proposed an algorithm for scheduling of operations onto a clustered architecture referred to as BUG [25][7]. It too consisted of two phases: an assignment phase using a bottom up node priority and a scheduling phase. Both phases take the machines resources into account and attempt to minimize the schedule length of the code. Desoli extended acyclic graph partitioning by using an initial clustering heuristic to exploit DAG symmetries thus simplifying the assignment process and speeding up the convergence of their decent algorithm [26]. Both of these approaches can be extended to loops by performing loop unrolling.

In modulo scheduling, the ILP is further exploited along consecutive (unrolled) loop iterations. In this context, BUG and [26] do not apply as well since the primary objective of modulo scheduling is to maximize the throughput of the loop instead of the schedule length of one iteration. In the presence of recurrences, the throughput is maximized (II minimized) by minimizing the latency along each of the critical cycles formed by the recurrences. An assignment algorithm may significantly affect II because adding a copy to a critical recurrence automatically increases the II of the final schedule. Recurrences correspond to strongly connected components (SCC) in the data flow graph.

Furthermore, the algorithm presented here is iterative which helps prevent poor assignment choices early on from negatively impacting the final assignment. The benefits of an iterative scheduling algorithm have been well documented [13], and this paper will show its benefits also apply to an assignment algorithm.

Capitanio et al. present a scheduling algorithm along with their analysis of clustered architectures [3]. The central portion of the algorithm starts with pre-scheduled VLIW code and, after converting the code to a data dependence graph (DDG), two processing phases are used: a code partitioning phase and a code compaction phase. The partitioning phase divides the code into subgraphs that meet the available resources of each cluster and minimizes schedule length. The compaction phase inserts any required copy operation and compacts the code.

The Capitanio algorithm was used for estimating the degradation in performance caused by using a clustered architecture and therefore performs cluster assignment as a post-scheduling process. Since partitioning is performed after scheduling, loops are effectively treated as straight line code. This process could have severe impacts on modulo scheduled code because the impact of breaking critical recurrences across clusters is not considered. Performing cluster assignment after scheduling may lead to a poorer schedule than assigning before scheduling because a totally new modulo schedule may be required after insertion of copies if the same II is to be maintained. The algorithm presented here performs cluster assignment before scheduling and takes into consideration potential performance issues before modulo scheduling the loop.

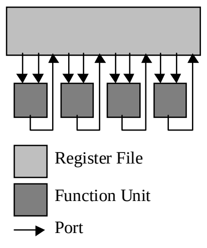

Figure 1. Individual Cluster

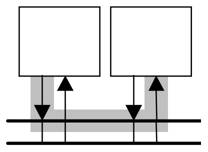

Figure 2. Two cluster setup

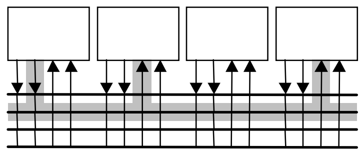

Figure 3. Four cluster setup

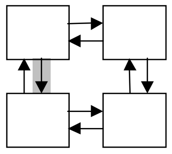

Figure 4. Four cluster grid setup

## 2. Overview

### 2.1. Architectures investigated

Three major cluster organizations are targeted in this paper. These organizations are a two-clustered bused setup, a four-clustered bused setup, and a four-clustered grid point-to-point setup.

For the bused setups, both *general purpose* (GP) and *fully specified* (FS) function unit types are investigated. Each cluster contains four function units, either four GP units or four FS units: one memory unit, two integer units, and one floating point unit .

The four-cluster grid based setup represents a different clustering scheme and serves to show the flexibility of the assignment algorithm. It consists of four clusters, each with three FS units: one memory, one integer, and one floating point. The clusters are connected in a square setup so that each cluster can only communicate with its horizontal and vertical neighbor. Instead of a bus, a dedicated connect between neighbors is modeled.

Figure 1 shows how each individual cluster is arranged, except that the grid based configuration has only three function units. Figures 2-4 show the three major cluster organizations. For these figures, each white block is a cluster as shown in Figure 1 with the addition of any ports shown in the respective figure.

Copies are used to move data between clusters. A copy is modeled as a unit cycle operation that consumes one read port from the source and one write port on the target. For a bused setup, the copy also reserves a bus for one cycle and the data that is being copied is broadcasted across the bus and may be written to any cluster with an available write port. For a point-to-point setup, a copy consumes a read and write port and the entire point-to-point connection. The data passed on a point-to-point connection can go only to the one connected cluster. Finally, a copy is not explicitly modeled as requiring an issue slot, only port and connection resources. In the worst case, one issue slot per read and write port will be required. However, since copies are narrow instructions more novel approaches to issuing them should be possible.

The highlighted resources in Figures 2, 3, and 4 show the resources used by a single copy. Figure 3 shows a copy on a bused setup whose result goes to two target clusters. Figure 4 shows a copy on a point-to-point setup. Note that the techniques presented produce assignments for machines with arbitrary numbers of clusters which can be homogeneous or heterogeneous in the types of function units they contain. Also, any numbers of buses or point-to-point connections and numbers of ports can be modeled.

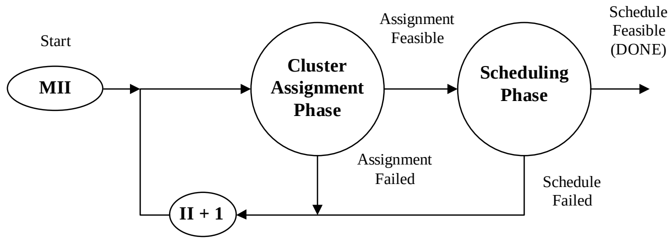

Figure 5. Assignment and Scheduling Process.

### 2.2. Process synopsis

The entire scheduling process consists of two major phases, a cluster assignment phase and a scheduling phase. The cluster assignment phase maps operations to clusters and adds any required copies to the data flow graph. The scheduling phase maps operations to a given cycle in an execution stream. Figure 2 shows the interaction between cluster assignment phase and the modulo scheduling phase.

First, the minimum II (MII) for an equivalent unified machine is calculated. Then, the assignment process starts by computing a suitable order of the operations. A modulo reservation table (MRT) is used to tract the resources used by the modulo scheduled loop kernel. Details of its structure are explained by Rau [13]. Notice that an MRT is used for both assignment and scheduling as both phases precisely model machine resources. For each cluster, a MRT of length II is created and initialized to empty. Local resources for a cluster are in only that cluster's MRT. Global resources, such as buses, are found in every MRT. Each operation from the ordered list and

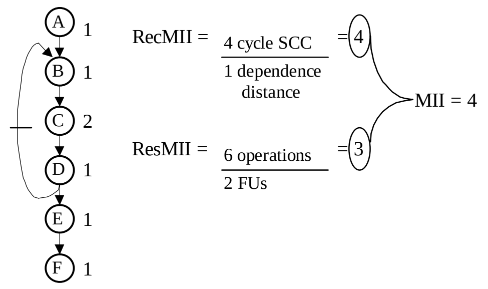
A 1
B 1
C 2
D 1
E 1
F 1
RecMII = 4 cycle SCC / 1 dependence distance = 4
ResMII = 6 operations / 2 FUs = 3
MII = 4

Figure 6. Dependence graph for introductory example.

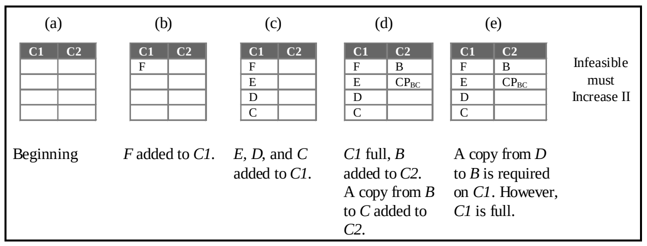
(a)
(b)
(c)
(d)
(e)
C1
C2
F
E
D
C
B
CP_BC
Infeasible must Increase II
Beginning
F added to C1.
E, D, and C added to C1.
C1 full, B added to C2. A copy from B to C added to C2.
A copy from D to B is required on C1. However, C1 is full.

Figure 7. Cluster assignment using a simple bottom up priority.

any required copies are assigned to a cluster and reserved in that cluster's MRT. If a required resource is unavailable in the MRT of length II, and the iterative back-tracking budget is spent, a new assignment phase is initiated with a larger II. The process restarts and continues until a valid assignment is found. Finally, the modulo scheduling phase attempts to schedule each operation and any required copies using the given cluster assignments. If the scheduler fails, II is increased and the entire process starts over at the assignment phase until a valid schedule is obtained. While the previous assignment could be preserved, it is more advantageous to search for a new one since a larger II allows more operations to be assigned to the same cluster thus generally results in an assignment with fewer copy operations.

The paper focuses on only the first phase of the scheduling process described above. Any traditional modulo scheduling algorithm may be used for the second phase because any copies needed due to clustering have been added. Thus, highly effective algorithms such as the Iterative Modulo Scheduler combined with a Stage Scheduler or a Swing Modulo Scheduler may be used directly for a clustered architecture [13][16][14].

## 3. Assignment issue

This section illustrates issues that arise in phase one of the process described in Section 2.2 when assigning loops onto a clustered architecture. The two major issues are: the impact that loop-carried dependences should have on the ordering of operations and the negative impact of aggressively filling clusters.

A simple loop will be assigned onto a hypothetical two clustered machine in which each cluster consists of one GP unit. We assume two buses between the clusters and one read/write port per cluster.

For simplicity neither the buses nor the ports will be shown. Also for simplicity for the example only, either one copy or one operation can be issued on a given function unit per cycle. This contrasts the experimental models that effectively have a separate issue slot for copies.

Figure 6 shows the data flow graph for the introductory example in which the nodes correspond to operations and the edges correspond to the data flow between operations. Each operation has a unit latency with the exception of operation *C*, which takes two cycles. Each dependence is within the same loop iteration with the exception of dependence *(D,B)* in which the result of *D* is used by operation *B* of the next iteration (i.e. a recurrence with a dependence distance of one). Additionally, nodes *B*, *C*, and *D* in Figure 6 are part of a SCC.

The first step, before assignment can begin, is to compute MII which accounts for both the latencies of the operations along recurrence edges in the data flow graph (RecMII) and the resource limitations of the machine (ResMII). RecMII corresponds to the sum of the latencies divided by the sum of the dependence distances along the most critical cycle in the dependence graph. In Figure 1, **RecMII = (1+2+ 1) / 1= 4**. ResMII corresponds, for this target machine, to the total number of operations divided by the issue width of the machine. For the data flow graph in Figure 1, **ResMII = 6/2 = 3**. MII is simply the maximum of the RecMII and the ResMII, which is 4 for this example.

The assignment can now begin. For this example two assignment approaches will be shown:

- The first will take the nodes in a bottom up fashion and assign them to the first available cluster.
- The second will consider nodes within the SCC in the example graph as high priority nodes. It will also predict copy usage in order to spread out the assignments.

### 3.1. Approach 1 - bottom up + first available

Based on the dependence graph of Figure 6, a bottom up traversal yields the node ordering *F, E, D, C, B, A*. Figure 7 shows the results of assigning each node, in order, to a cluster. The algorithm fails after assigning *B*

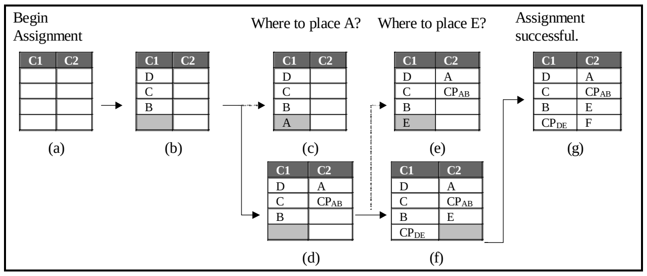

**Figure 8. SCC first and predicted copy usage.**

to cluster *C2* (Figure 7d) because it requires a copy from *D* to *B* on cluster *C1,* and *C1* is already full (Figure 7e). This points out the first of two observations**.**

**Observation One:** The assignment algorithm tried to fill the clusters too aggressively and did not consider that copy operations may be required due to future assignment decisions. Since no room was reserved on *C1* for a future copy, the algorithm was forced to fail.

**Observation Two:** Even if the copy from *D* to *B* could have been assigned on *C1*, two copies would have been added within the strongly connected component of the graph: the first copy between nodes *B* and *C*, and the second copy between the nodes *D* and *B*. Since copy operations have a latency of 1 cycle, the cumulative latency along the critical cycle would have been increased from 4 to 6, thereby increasing RecMII and MII from 4 to 6 as well.

## 3.2. Approach 2 - SCC first + predict copy use

This approach takes into consideration the two above observations to achieve an assignment and schedule with an II of 4 cycles. Specifically, Observation 1 is satisfied by tentatively reserving resources for the copy operations that may be required in the future, depending on where successor/predecessor operation will be assigned. Observation 2 is satisfied by giving higher priority to the nodes within the SCC. One such ordering is *D, C, B, A, E, F*. This node ordering will prevent splitting the SCC across clusters, thus solving the problem mentioned in observation two.

In Figure 8, shaded MRT slots represent slots whose future use by a copy has been predicted to be needed. When given the choice, an operation will be placed in a non-shaded slot over placement in a shaded slot. Copies can be placed into shaded slots whenever necessary since the shaded slots are essentially reserved for copies. The exact details this prediction is presented in Section 4.2.

Figure 8 below shows the assignment process. Nodes *D*, *C*, and *B* are first assigned to *C1* since they are all a part of an SCC (Figure 8b). On C1, node *D* has one successor that has not yet been assigned, therefore one slot on cluster one is shaded. *B* and *C* have all of their successors assigned. *C2* is unchanged.

*A* must now be assigned to a cluster. If *A* is placed on *C1* it will have to go into the shaded slot because *D* still has one unscheduled successor (Figure 8c). If *A* is placed on *C2*, *C1* remains unchanged and a copy is required for *C2* from *A* to *B* (Figure 8d). The assignment of *A* to *C2* is chosen to avoid placing *A* into the shaded slot on *C1*. In fact it can be seen that if *A* were placed on *C1* there would be no room for the future copy from *D* to *E*.

*E* must now be assigned. *E* is placed on *C2* (Figure 8f), again to avoid placing it into the shaded slot on *C1* (Figure 8e). A copy from *D* on *C1* to *E* on *C2* is placed into the shaded slot on *C1*. Finally F is assigned to *C2* because that is the only place it can go (Figure 8g). The assignment process has completed successfully.

## 4. Cluster assignment

The cluster assignment phase takes a data flow graph and a machine description and outputs a new data flow graph that has been annotated to indicate cluster assignments and includes any required copies. This graph is then used by the modulo scheduler in phase two.

There are three integral concepts in the cluster assignment algorithm: (1) Node grouping and ordering, (2) Tentative assignment and selection, and (3) Iteration and maintaining forward progress. First, the assignment algorithm partitions the operations into suitable sets and then orders the nodes within each set so that the impact of the copy operations is minimized. The specific grouping and ordering algorithms are presented in more detail in Section 4.1.

Second, taking each operation one at a time in the order of decreasing priority, the algorithm tentatively assigns an operation to each cluster. The assignment selector then chooses the assignment which appears to be the best. The selected cluster assignment is finalized while the others are discarded. The process continues until all of the nodes have been permanently assigned (i.e. finalized) on a cluster. Section 4.2 explains the details of assignment and selection.

Third, Section 4.3 explains the details of the iterative portion of the algorithm that allows it to continue past seeming assignment failures by selectively removing and reassigning operations. It also explains how to keep the iterative algorithm from getting locked in repetitive sequences of assignments and removals.

```
        Select( LIST, criteria )
(1)             NewLIST = all clusters in LIST that satisfy [criteria]
(2)             LIST = NewLIST if NewLIST is not empty
```

**Figure 9. Definition of a selection.**

```
SelectBestCluster( LIST, N )
(1)     LIST = all feasible clusters
(2)     N = node to be assigned
(3)     if ( N is a member of  S, an SCC ) {
(4)             Select( LIST, "clusters on which another node from S is already assigned" )
(5)     }
(6)     Select( LIST, "predicted copy requests <= maximum reservable copies")
(7)     Select( LIST, "minimize the number of required copies generated")
(8)     Select( LIST, "maximize free resources on the cluster")
(9)     return the first cluster in LIST
```

**Figure 10. Cluster assignment selection algorithm.**

## 4.1. Node ordering for cluster assignment

The main goal of the node ordering is to make sure that nodes in SCCs are assigned first to avoid splitting them across clusters with copies. The performance degradation from splitting SCCs across clusters is reduced by assigning nodes in the SCCs which result in the highest RecMII value to clusters first and then assigning nodes from successively less constraining SCCs. Since the most critical SCCs are handled first, they are less likely to be split across clusters due to unavailable resources within the II. The remaining operations of a loop are then spread across clusters to take advantage of the available resources.

To this end, an ordered list of sets is formed, with the highest priority set containing the nodes of the most constraining SCC. The next highest priority set contains the nodes of the next most constraining SCC. The final and lowest priority set contains any nodes not in any SCC.

The second goal of the node ordering is to assign data dependent nodes on the same cluster whenever possible. This affects both nodes inside and outside SCCs. While adding copies between nodes outside SCCs will not directly impact II, these copies still consume shared resources and may contribute to an increase in ResMII. The number of copies added can be minimized by assigning nodes with predecessors and successors whenever possible.

To achieve this second goal, we leverage on a node ordering heuristic developed in the context of register sensitive modulo scheduling [16]. This heuristic orders nodes such that a given node is listed, when possible, after all its successors (or all its predecessors) have been listed. This order, clearly beneficial when minimizing register lifetime, is also beneficial here as it reduces the likelihood of assigning both the successors and predecessors of an operation to distinct clusters prior to assigning the operation itself. Thus the Swing Modulo Scheduler's node ordering heuristic is applied to each set.

## 4.2. Tentative assignment and selection

The assignment algorithm assigns one node at a time. The next node chosen for assignment is always the highest priority unassigned node. The node is then tentatively assigned to each cluster, and information about the resulting resource use is recorded. For example, after placement of a node information such as the number of copies generated, copies requested, maximum number of assignable copies, and total free resources on the cluster are recorded. A heuristic then uses the information gathered to pick the best assignment from among the feasible cluster assignments.

The heuristic chooses the best cluster assignment though a series of selections. A selection is defined in Figure 9. The list of clusters under consideration is initialized to all clusters onto which the assignment of the current node is feasible. An assignment is considered feasible if enough resources exist for both the operation and any *required copies (RC)*, i.e. copies that must be inserted in order for the current cluster assignments to be valid. After the series of selections, the node's assignment is finalized on the first cluster in LIST and the other tentative assignments are discarded. The series of selections performed are shown in Figure 10.

The selection at line 4 keeps SCCs together as much as possible. Section 4.1 amply explains why SCCs must be kept together when possible. Line 6 attempts to spread out the nodes by predicting future copy usage. When a node *X* is assigned to a cluster, e.g. *C1*, and one of its successors has already been assigned to a different cluster, e.g. *C2*, it is obvious that the result of *X* must be copied to *C2*. However, when some of the successors of *X* have not yet been assigned to a cluster it is not known whether any copies will be required. If the decision to reserve space for a copy is delayed until the successors are assigned, as in Figure 6 in the introductory example, there may no longer be any resources for these extra copies, which would in turn force an increase in II. Two numbers are calculated in order to perform the selection in line 6. These are the predicted copy requests (PCR) and the maximum reservable copies (MRC). For a given cluster C, $MRC_C$ and $PCR_C$, are defined as follows:

$$MRC_C = Maxmimum\ Number\,Of\ Additional\ Copies$$
$$For\,Which\ There\ Is\ Room\ On\ Cluster\,C$$

$$PCR_C = \sum_{N_i \in C} Min(Upperbound(N_i), UnscheduledSuccessors(N_i))$$

where the min term is summed for all operations $N_I$ which have already been assigned to cluster C.

```
ForceSelectBestCluster( N )
(1)     LIST = all clusters
(2)     N = node to be assigned
(3)     Select( LIST, "in which N (excluding copies) can be assigned without resource conflicts")
(4)     Select( LIST, "which minimizes the number of conflicting  predecessors and successors")
(5)     return the first cluster in LIST
```

**Figure 11. Selection of the Best Cluster  when no clusters have enough resources.**

*UnassignedSuccessor($N_i$)* corresponds to the number of successor operations from $N_i$ that are not currently assigned to any clusters. Upperbound() provides an upper-bound on the number of additional copies an operation could require do to the worst-case placement of its unassigned successors. *UpperBound($N_i$)* is defined as follows:

$$UpperBound(N_i) = Max[0, 1 - RC(N_i)]$$

for configurations with broadcast buses

$$UpperBound(N_i) = Max[0, ClusterCount - RC(N_i) - 1]$$

otherwise.

*RC($N_i$)* corresponds to the number of required copies generated by operation $N_i$ and *ClusterCount* is the total number of clusters in the system. The above definition takes into account that on a broadcast machine, the result of an operation needs to be communicated at most once; otherwise, the result of an operation needs to be communicated at most to the other clusters (i.e. *ClusterCount -1*).

In Figure 10, Line 7 simply picks the assignment that generates the fewest required copies. Finally, line 8 picks the cluster with the most free space.

## 4.3. Iteration and forward progress

Since heuristics are intrinsically imperfect they sometimes chose poorly. For the cluster selection heuristic presented in Section 4.2, poor choices tend to occur because too little information is available at the time when the node was assigned. A small number of poorly placed nodes can result in a situation in which no more nodes can be assigned to any cluster, thus forcing a premature failure and an increase in II. Allowing the algorithm to remove conflicting nodes from their current assignments and therefore continuing beyond a failure can result in a successful assignment.

For an iterative algorithm, there are two major considerations:

- Choosing which nodes to remove, thus allowing forward progress,
- Preventing repetitively assigning and removing the same series of nodes to the same clusters.

**4.3.1 Node removal.** When a node can no longer be assigned to any cluster (i.e. LIST in Figure 10 is empty in Line 1), the algorithm picks a cluster on which to force the node's assignment using the selection algorithm in Figure 11. There are two reasons why a node is unable to be assigned to a particular cluster:

- Resources for the node itself are unavailable
- Resources for a required copy from a predecessor or to a successor are unavailable

The series of selections are shown in Figure 11. Once the cluster on which to place N is chosen, any and all nodes conflicting with the resources needed by N are removed as well as any conflicting predecessors and successors.

**4.3.2 Preventing repetition.** To prevent a node from being repetitively assigned and removed from the same cluster, a list of clusters on which a node was previously assigned is kept. When possible, the algorithm will avoid re-assigning a node onto a cluster in which the node has been previously assigned. Once the node has been assigned to all possible clusters, the list is cleared.

This is implemented by adding the following selection *A* between lines 2 and 3 in the cluster selection algorithm in Figure 10 as well as between lines 2 and 3 in the algorithm for cluster selection after a failure as described in Figure 11.

*(A) Select( LIST, "clusters onto which N has not been previously assigned")*

| Statistic | Min | Avg | Max |
|---|---|---|---|
| Nodes | 2 | 17.5 | 161 |
| SCCs per loop | 0 | 0.4 | 6 |
| Nodes in non-trivial SCCs | 2 | 9.0 | 48 |
| Edges | 1 | 22.5 | 232 |

**Table 1. Loop Statistics**

| Operation | Latency |
|---|---|
| ALU, Shift, Branch, Store, FP-Add, Copy | 1 Cycle |
| Load | 2 Cycles |
| FP-Mult | 3 Cycles |
| FP-Div, FP-SQRT | 9 Cycles |

**Table 2. Operation Latencies**

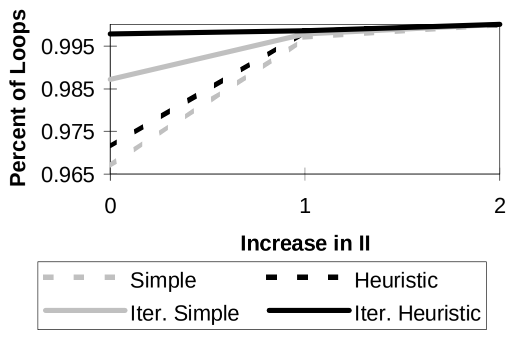

**Figure 12. Comparing heuristics for a two cluster setup (2 buses, 4 GP units per cluster).**

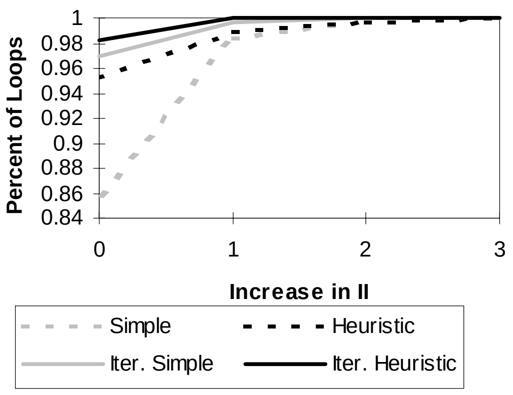

**Figure 13. Comparing heuristics for a four cluster setup (4 buses, 4 GP units per cluster).**

## 5. Experimental setup

The test loops consisted of 1327 loops, 301 containing SCCs, from the Perfect Club[22], SPEC-89[23], and the Livermore Fortran Kernels[24]. The input loops consist exclusively of innermost loops with no early exits, no procedure calls, and fewer than 30 basic blocks, as compiled by the Cydra 5 Fortran77 compiler. Load-store elimination, recurrence back-substitution, and IF-conversion have already been applied to the input loops. Table 1 shows some statistics on the loops in the suite.

A wide range of machine and cluster configurations were tried in order to show the algorithms effectiveness on different architectures. Two cluster machines using buses, four cluster machines using buses, and four cluster grid array were modeled. Section 2.1 explains the details of these three architectures. Table 2 shows the latency used for each operation regardless of machine.

The assignment phase proceeds as described in the previous sections. An iterative version of a swing modulo scheduler was used in the scheduling phase.

## 6. Results

The following results show the performance of the assignment algorithm for the variety of machines mentioned in Section 2.1 and loops as described in Section 5. In each case, the performance of the algorithm for a clustered machine will judged by comparing the final IIs for each scheduled loop on the clustered machine to the II of the same loop scheduled on an equally wide unified machine. While this does not show how far the final II for the clustered machine deviates from optimal, it gives a good estimate of how well communication was hidden. For example, if a loop can only be scheduled within an II of 10 for a unified architecture having a width of 8 and the algorithm partitions the graph across four clusters, with a combined total width of 8, in such a way that a modulo schedule is still able to be found with the same II of 10 then the algorithm has succeeded in its goal. Any copies that had to be added were placed in a way that kept them from exceeding resource limits or breaking critical SCCs.

For the graphs, the x-axis represents how far the II of a loop scheduled on a clustered machine deviated from that of the same loop on a non-clustered machine. The y-axis shows what percentage of loops fell into the x-axis's category. For example, an x=0, y=98% means that the algorithm was able produce schedules for 98% of the loops where the II on the clustered machine matched that of the unified machine. An x=0 is the ideal x value, and means that all required communication was hidden.

The first set of results will show the performance of the iterative assignment with the full cluster selection algorithm, *Heuristic Iterative*, compared to (1) an iterative assignment with a simple cluster selection algorithm (Figure 10, without lines 3 to 8): *Simple Iterative*; (2) a non-iterative assignment with full cluster selection algorithm: *Heuristic*; and (3) a non-iterative assignment with the simple cluster selection algorithm: *Simple*.

Figure 12 compares the algorithm for a machine with two clusters of 4 GP units and two buses. Figure 13 compares the algorithm for a machine with four clusters of 4 GP units and four buses. The graphs show that the Iterative algorithm using the heuristic described in Section 5 is the best algorithm overall. Making the algorithm non-iterative severely impacts the performance of the algorithm causing a 2% to 11% drop in the number of loops that match the unified machine's II. Not using the full selection heuristic also impacts performance, causing a 1% to 9% drop in matching IIs.

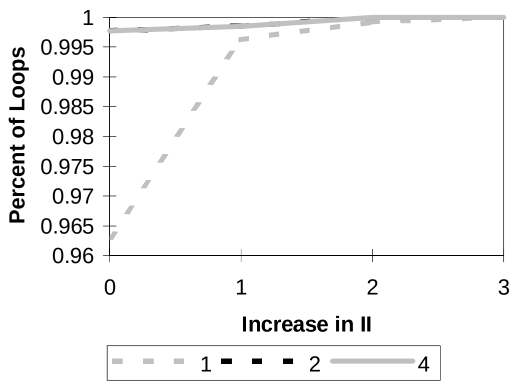

**Figure 14. Varying the number of buses for a two cluster machine (4 GP units per cluster).**

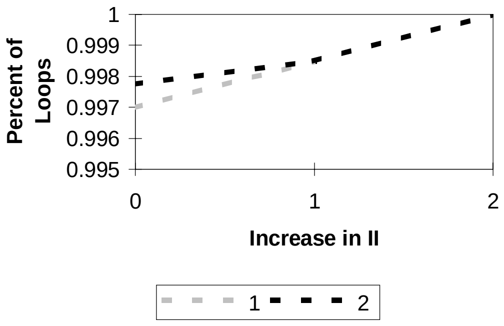

**Figure 15. Varying the number of ports for a two cluster machine (2 buses, 4 GP units per cluster).**

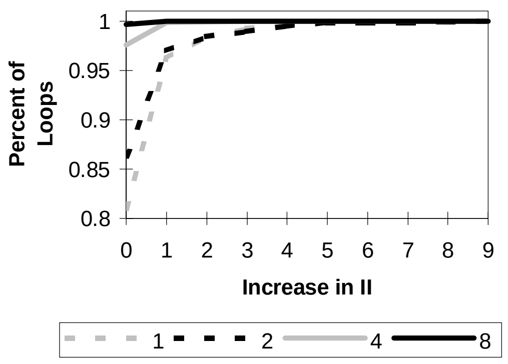

**Figure 16. Varying the number of buses for a four cluster setup (4 GP units per cluster).**

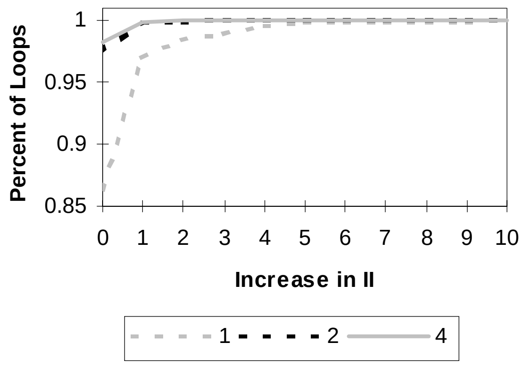

**Figure 17. Varying the number of ports for a four clusters setup (4 buses, 4 GP units per cluster).**

The data shows that removing any part of the algorithm will reduce its effectiveness. The graphs above also show that the algorithm generates schedules virtually matching those of a unified machine.

Figures 14 and 15 show the ability of the algorithm to hide communication on a two- clustered machine with a varying number of buses and number of read and write ports, respectively. For the two-clustered machine it appears that reducing the number of buses to 1 impacts 4% of the IIs. Increasing to 4 buses provides no benefit at all, so the best number of buses appears to be 2. However, it also appears that only one read and write port is need. Adding a second port only improves 0.1% of the loops and, due to the expense of ports, the additional performance is not worth the extra cost. For the two-clustered machine above, two buses with one read and write port per cluster for the buses allow for performance nearly matching that of a unified machine.

Figures 16 and 17 show the ability of the algorithm to hide communication on a four-clustered machine with a varying number of buses and numbers of read and write ports.

For the four-clustered machine, the results show that going from four buses to two buses negatively impacts over 10% of the loops. Due to the cost of buses, the additional increase of 3% from four to eight buses is probably not worth the additional four buses. Additionally, the read and write port results show that two ports is a good number. Going to four is of marginal value and dropping to 1 port decreases the performance of 12% of the loops. While four buses are needed only two read and write ports are needed per cluster for the four clustered machine for performance nearly matching that of a unified machine.

Figures 18 and 19 show a comparison of various bus counts for both a two and four clustered machine with fully specified FUs. The results for the models with FS units are very similar to those for the GP machines. The IIs for about 95% of the loops on the two clustered match those on the non-clustered machine given 2 buses. Given four buses, the IIs of 94% loops on the four-clustered

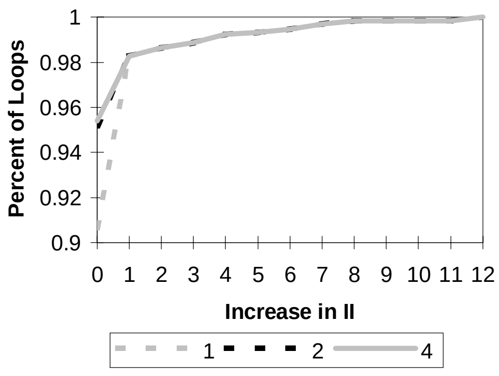

**Figure 18. Varying the number of buses for a 2 cluster setup (4 FS units per cluster).**

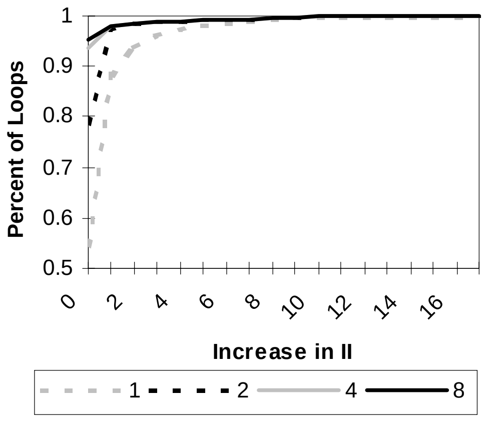

**Figure 19. Varying the number of buses for a 4 cluster setup (4 FS units per cluster).**

match those on the non-clustered machine. The required ports to get the above results is one read and write port for the two cluster machine, and two read and write ports for the four clustered machine.

Simulation runs were also performed for six and eight clustered GP machines. The bus and port counts listed are for the point of diminishing returns for which addition of another port or bus did not result in a significant performance improvement. Table 3 shows an overview of the results for the runs with two, four, six, and eight clusters. There appears to be at worst a linear relationship between the number of clusters and the number of buses/ports needed. However, it should be noted that the relationship between the number of clusters and the number of buses/ports is very dependent on the amount of ILP that is effectively used. Thus a higher number of ports and buses may be needed for benchmarks with higher ILP, and vice versa.

The results were also promising for the four-cluster grid setup as described in section 2.1. The algorithm assigned 92% of the loops on the clustered machine equally as well as it could have on unified machine. 98% of the loops deviated by no more than one cycle from the II of the unified machine. Though this cluster setup has limited communication, no buses for broadcasting, and one less FU per cluster, the algorithm still performed well.

| Clusters | Buses | Ports | Percent of Unified |
|---|---|---|---|
| 2 | 2 | 1 | 99.7 |
| 4 | 4 | 2 | 97.5 |
| 6 | 6 | 3 | 96.5 |
| 8 | 7 | 3 | 99.5 |

**Table 3. Bus/Port Resource Comparisons**

## 7. Conclusions

Clustering is one solution to the demand for wide-issue machines and a fast clock cycle. Clustering allows for smaller, less ported register files and simpler bypass logic. The resulting system is scaleable and still allows the processor to take advantage of more ILP. The resulting throughput of the machine will only be improved, however, if the latency of any inter-cluster communication, i.e. copies, can be hidden.

This paper proposed an algorithm for hiding the latency of copies for modulo scheduled loops. The results show that the cluster assignment algorithm can allow a modulo scheduler to attain nearly identical IIs for clustered architectures as for an equivalently wide non-clustered architecture. For the machines shown above between 94% and 98% of the loops scheduled for the clustered machine matched the non-clustered II given a reasonable number of buses and ports. The algorithm is flexible enough to generate good assignments for a very wide range of machine configurations. This algorithm allows for modulo scheduled loops to take advantage of significantly wider machines without incurring the high penalty of highly ported register files.

## Acknowledgments

The authors would like to thank Michael Schlansker and B. Ramakrishna Rau for providing the input data set. This research was supported in part by a grant from Intel.

## References

[1] W. Mangione-Smith, S. G. Abraham, and E. S. Davidson, 'Register requirements of pipelined processors,' *Proceedings of the International Conference on Supercomputing*, pp. 260-271, July 1992.

[2] C. G. Lee, 'Code Optimizers and Register Organizations for Vector Architectures,' *Ph.D. Thesis, University of California Berkley*, 1984.

[3] A. Capitanio, N. Dutt, and A. Nicolau, 'Partitioned register files for VLIWs: A preliminary analysis of tradeoffs,' *Proceedings of the 25th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 292-300, 1992.

[4] S. W. White and S. Dhawan, 'Power2: Next Generation of the RISC System/6000 family,' *In IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993.

[5] J. Llosa, M. Valero, and E. Ayguade, 'Non-consistent Dual Register Files to Reduce Register Pressure,' *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, pp. 22-31, Raleigh (North Carolina), January 1995.

[6] R. Colwell, et al., 'Architecture and implementation of a VLIW supercomputer,' *Proceedings in Supercomputing*, pp. 910-919, November 1990.

[7] P. Lowney, et al., 'The Multiflow Trace Scheduling Compiler,' *The journal of Supercomputing*, pp. 51-142, 1993.

[8] J. A. Fisher, 'Trace Scheduling: A technique for global microcode compaction,' *IEEE Transactions on Computer*, Volume C-30 No. 7, pp. 478-490, July 1981.

[9] W. W. Hwu, et al., 'The superblock: An effective technique for VLIW and superscalar compilation,' *The Journal of Supercomputing*, Volume 7, pp. 229-248, January 1993.

[10] S. Banerjia , W. A. Havanki, and T. M. Conte, 'Treegion scheduling for highly parallel processors,' *Proceedings of Euro-Par 97*, Passau, Germany, August 1997.

[11] B. R. Rau and C. D. Glaeser, 'Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing', *Proceedings of the Fourteenth Annual Workshop on Microprogramming*, pp. 183-198, October 1981.

[12] M. Lam, 'Software Pipelining: An effective scheduling technique for VLIW machines,' *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 318-328, June 1988.

[13] B. R. Rau, 'Iterative Modulo Scheduling: An algorithm for software pipelined loops,' *Proceedings of the 27th Annual IEEE/ACM International Symposium on Microarchitecture,* pp. 63-74, November 1994.

[14] A. E. Eichenberger and E. S. Davidson, 'Stage Scheduling: A Technique to Reduce Register Requirements of a Modulo Schedule,' *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 338-349, November 1995.

[15] A. E. Eichenberger and E. S. Davidson, 'Efficient Formulation for Optimal Modulo Schedulers,' *Proceedings of the Conference on Programming Language Design and Implementation*, June 1997,

[16] J. Llosa, et al., 'Swing Modulo Scheduling: A lifetime-Sensitive Approach,' *IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, pp. 80-86, Boston (USA), October 1996.

[17] J. C. H. Park, and M. Schlansker, 'On Predicated Execution.' *Technical Report HPL-91-58*, *Software and Systems Laboratory, Hewlett-Packard*, 1991.

[18] D. M. Lavery and W. W. Hwu, 'Modulo Scheduling of Loops in Control-Intensive Non-Numeric Programs,' *Proceedings of the 29th Annual IEEE/ACM International Symposium On Microarchitecture*, 1996.

[19] M. G. Stoodley, and C. G. Lee,' Software Pipelining Loops with Conditional Branches,' *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microachitecture*, pp. 262-273, Paris (France), December 1996.

[20] P. P. Tirumalai, M. Lee, and M. S. Schlansker, 'Parellization of loops with exits on pipelined architectures,' *Supercomputing*, pp. 200-212, November 1990.

[21] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schansker, 'Register allocation for software pipelined loops,' *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 283-299, June 1992.

[22] M. Berry, et al., 'The Perfect Club Benchmarks: Effective performance evaluation of supercomputers,' *International Journal of Supercomputer Applications*, 3(3):5-40, 1989.

[23] J. Uniejewski, 'SPEC Benchmark Suite: Designed for today's advanced system,' *SPEC Newsletter*, 1989.

[24] F. H. McMaho, 'The Livermore Fortran Kernels: A computer test of the numerical performance range,' *TR UCRL-53745, Lawernce Livermore Nat. Lab., Livermore, California*, 1986.

[25] J. R. Ellis, 'Bulldog: A compiler for VLIW Architectures,' *MIT Press*, pp. 180-184, 1986.

[26] G. Desoli, 'Instruction assignment for clustered VLIW DSP compilers: a new approach,' *TR HPL-98-13, HP Labs, Palo Alto, California*, 1998.