Region-based Hierarchical Operation Partitioning for Multicluster Processors

Michael Chu Kevin Fan Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI 48109
{mchu, fank, mahlke}@umich.edu

ABSTRACT

Clustered architectures are a solution to the bottleneck of centralized register files in superscalar and VLIW processors. The main challenge associated with clustered architectures is compiler support to effectively partition operations across the available resources on each cluster. In this work, we present a novel technique for clustering operations based on graph partitioning methods. Our approach incorporates new methods of assigning weights to nodes and edges within the dataflow graph to guide the partitioner. Nodes are assigned weights to reflect their resource usage within a cluster, while a slack distribution method intelligently assigns weights to edges to reflect the cost of inserting moves across clusters. A multilevel graph partitioning algorithm, which globally divides a dataflow graph into multiple parts in a hierarchical manner, uses these weights to efficiently generate estimates for the quality of partitions. We found that our algorithm was able to achieve an average of 20% improvement in DSP kernels and 5% improvement in SPECint2000 for a four-cluster architecture.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation, retargetable compilers*; C.1.1 [**Processor Architectures**]: Single Data Stream Architectures—*RISC/CISC, VLIW architectures*

General Terms

Algorithms, Experimentation, Performance

Keywords

clustering, instruction-level parallelism, instruction scheduling, multicluster processor, operation partitioning, region-based compilation

1. INTRODUCTION

Superscalar and VLIW processors achieve high performance by exploiting instruction-level parallelism (ILP) to issue multiple operations each cycle. As the number of operations issued each cycle grows, the demands to supply operands to these operations increases in turn. In a conventional processor, a centralized register file is responsible for operand supply. Supplying a larger and larger number of operands each cycle from a centralized register file can quickly become the bottleneck in a processor design. The bottleneck results due to the combined effects of: register file cost and access time growing with the square of the number of register ports; a larger number of registers being necessary as issue width increases to maintain more temporary values; register bypass logic growing quadratically with the number of operations issued per cycle; and the distance separating function units (FUs) from the register file increasing with a larger number of FUs [8] [7].

A natural solution to these problems is to remove the centralized register file and create a decentralized architecture with several smaller register files. Each of the smaller register files supplies operands to a subset of the FUs. These smaller register files can be efficiently designed, thereby alleviating the register file bottleneck while maintaining the desired level of ILP. This strategy is generally referred to as a clustered architecture or a multicluster processor [10]. One of the first clustered architectures was the Multiflow Trace. Clustered architectures are becoming increasingly popular in many recent processor designs including the Alpha 21264, TI C6x series, and Analog Tigersharc. Each of these processors is a two-cluster design.

The central challenge with clustered architectures is compilation support. The compiler must effectively partition operations across the resources available on each cluster to maximize ILP. However, this goal must be achieved while carefully considering the implications of inter-cluster communication. Communication of values between clusters is both slow and bandwidth-limited. Thus, operations must be partitioned to ensure that ILP is not constrained by frequent inter-cluster communication. A common rule of thumb is that breaking a processor into two identical clusters reduces program performance by around 20%. Furthermore, a four cluster processor loses around 30% performance over the equivalent single cluster processor [7]. Generally, these numbers get worse when the clusters are not identical. While clustering makes sense from an architectural perspective, a large amount of performance is left on the table with this choice.

Examining common operation partitioning algorithms in more depth reveals two recurring problems. First, partitioning algorithms are modeled closely after operation scheduling. They make local, greedy decisions to optimize the placement of an operation based on the placement of its neighbors. This strategy makes sense as clustering and scheduling are heavily intertwined. However, locally optimal decisions may actually be poor decisions when the global picture is considered. The second problem is that clustering algorithms are notoriously slow due to the detailed modeling of processor resource constraints. Resource models similar to (or often identical to) those used during operation scheduling are repeatedly evaluated for each candidate operation placement. The final code schedule is indeed very sensitive to the partition chosen, so this seems like the proper strategy. However, detailed modeling of a particular placement can be counterproductive when it limits the number of choices that can be considered. Furthermore, as processors have more resources and their resource usage patterns become more complex, detailed modeling of each placement choice may become infeasible for production compilers.

We use an approach opposite to this scheduler-centric methodology. Operation partitioning is performed at a global scope with the view of all operations in a region (a group of closely related basic blocks is referred to as a region [11]). We adapt two powerful techniques that are commonly used in VLSI design: multilevel graph partitioning and slack distribution. Multilevel graph partitioning divides the dataflow graph into multiple parts in a hierarchical manner. Operations are iteratively partitioned from a coarse level of groups of related operations down to a fine-grained level of individual operations. Slack distribution identifies available scheduling slack within a region and allocates it to specific dataflow edges. In this manner, the cost of cutting specific dataflow edges to create a partition is determined.

Graph partitioning also requires a processor resource model to determine the quality of a partition. Again, we take a non-scheduler-centric approach. We employ a simple estimation strategy that is similar to the RESMII (resource minimum initiation interval) calculation used with modulo scheduling [24]. However, we focus on scalar scheduling as opposed to software pipelining. Resource usage estimates are computed prior to partitioning and used to estimate the resource load for each candidate partition. While this method suffers inaccuracies, it is both more efficient and accurate enough to provide a suitable guide to the operation partitioning algorithm. Our proposed approach is referred to as region-based hierarchical operation partitioning, or RHOP.

The remainder of this paper is organized into five sections. Section 2 provides an overview of architectural model and the basics of operation partitioning. Section 3 presents RHOP algorithm itself. A preliminary experimental evaluation is given in Section 4. We compare and contrast our work with previous work in Section 5. Conclusions and future work are discussed in the last section.

2. OVERVIEW OF CLUSTERING

This section introduces the clustered architectural model that is assumed for this paper and the basic process of partitioning a dataflow graph (DFG) for this architecture.

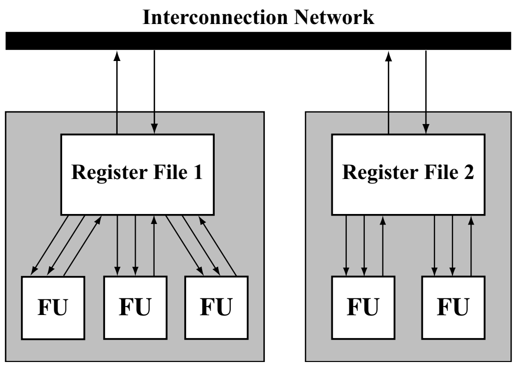


Figure 1: A heterogeneous two-cluster machine.

Next, we present a high-level classification of the common approaches for clustering and break them down by four categories: phase ordering, scope, desirability metric, and operation grouping. Last, we conclude with a discussion of the limitations of scheduler-centric approaches to motivate the work in this paper.

2.1 Basics

The architectural model assumed in this paper is as in Figure 1. Each cluster consists of a tightly connected set of register files (RFs) and function units (FUs). FUs in a cluster may only address those registers within the same cluster. Transfers of values between clusters are accomplished through explicit move operations that go through an interconnection network. The interconnection network is assumed to have a uniform connection to all clusters with a fixed bandwidth. Though this assumption is not necessary, it simplifies the compiler algorithms. Clusters in a machine may be homogeneous, each containing the same types and numbers of RFs and FUs, or heterogeneous, each having a unique mix of resources. The machine in Figure 1 is heterogeneous and has three FUs and one RF in cluster 1, and two FUs and a RF in cluster 2.

The goal of clustering is to obtain a balanced workload that takes advantage of parallelism available within the machine. The notion of balance on a cluster relates to the resources available on that cluster and the operations scheduled on it. For example, given a machine with two heterogeneous clusters such that cluster 1 has twice as many FUs as cluster 2, a balanced workload would tend to have twice as many operations scheduled on cluster 1 as on cluster 2.

Data is transferred from cluster to cluster via explicit inter-cluster move operations. Inter-cluster moves have a non-zero latency (1 cycle is assumed in this paper) and thus can lengthen the schedule. However, if the latency of the move can be overlapped with the execution of other operations, then the inter-cluster moves will not affect performance by much. A good partitioning of operations minimizes overall schedule length by simultaneously maximizing the number of operations executed in parallel while minimizing the number of moves that negatively affect performance.

Figure 2 shows two possible clusterings of an example DFG. For simplicity, the machine that will be used for the example is a homogeneous two-cluster machine, with one RF and one FU per cluster. Each FU is capable of executing any operation. The latencies of all operations are assumed

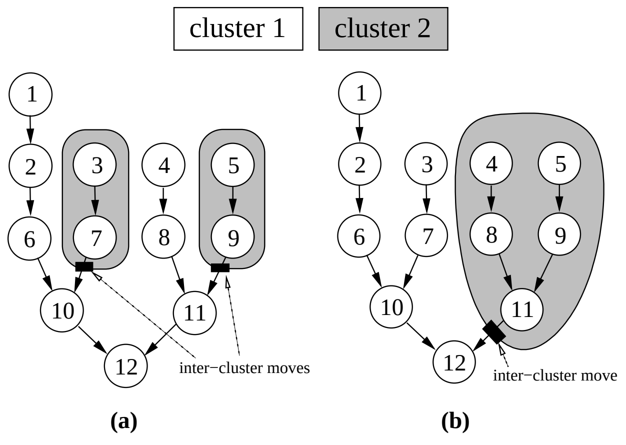


Figure 2: Example dataflow graph with (a) locally greedy and (b) region-aware cluster assignment.

to be one cycle. Furthermore, the interconnection network is capable of sustaining one inter-cluster move per cycle.

For the example in Figure 2, the clustering algorithm must partition the operations into two sets with each set executing on a particular cluster. Any time an edge is broken, an inter-cluster move is required to copy the data from the producing cluster to the consuming cluster. The critical path through this example graph is 5 cycles going through operations 1, 2, 6, 10, and 12. Cutting edges along the critical path increases the critical path length; thus, most clustering algorithms avoid such cuts.

2.2 Approaches to Clustering

A large number of algorithms to perform clustering have been proposed by the research community. To better understand the operation and relative strengths of these algorithms, it is useful to understand the major characteristics that differentiate them. We have identified four primary characteristics of clustering algorithms: phase ordering, scope, desirability metric, and grouping.

Phase Ordering. In the compilation process, cluster assignment can take place as a separate phase before scheduling, where the scheduler is constrained by the decisions made in the clustering phase. This isolates the clustering problem from the scheduling problem. Alternatively, clustering can be integrated with the scheduling phase; in this case, more information is available for making decisions, but the complexity of the problem increases, limiting the number of options that can be considered during the process. As a compromise, clustering and scheduling can be done iteratively, such that decisions made by one phase are used to guide the decisions made in the other, until a suitable result is obtained.

Scope. The scope of the clustering algorithm can be local, region-based, or global. A local algorithm generally examines one operation at a time and decides which cluster it should be assigned to based on its immediate neighbors. As with scheduling, operation priorities may guide the order in which operations are considered. A region-based algorithm, on the other hand, considers all of the operations within a region such as a basic block or set of basic blocks at once. Finally, a global algorithm uses knowledge of the entire program or function to make intelligent decisions. The complexity of the algorithm increases with the scope, but better decisions can be made with higher level knowledge.

Desirability Metric. The cluster assignment algorithm can use one of several ways to measure the quality of a candidate partition. It can perform an actual scheduling of the code, which gives the most accurate measure since the performance is then known. It can generate a pseudo-schedule using an approximate machine model to provide a reasonable estimate of the actual schedule. It can use quantitative resource usage estimates to project the load a set of operations places on a cluster. Finally, it can use a simple count of how many operations are on each cluster and how many moves are required to get an idea of the desirability of a partition.

Grouping. Another property of clustering algorithms is whether they employ a hierarchical or flat partitioning scheme. In the case of region-based or global clustering, a hierarchical approach means that decisions are made on multiple levels, with information available in a finer-grained view of the operation graph being used to refine previous decisions made from a coarser view of the graph.

2.3 Pitfalls of Scheduler-Centric Approaches

Scheduler-centric approaches to clustering employ a natural extension of the scheduling process to perform cluster assignment. This does not mean clustering is done during scheduling. In fact, any phase ordering is possible. The two distinguishing characteristics of scheduler-centric approaches are local scope and flat grouping. These are the primary characteristics of operation scheduling where operations are greedily placed into the schedule one by one considering the placement of those operations with higher priority that have already been scheduled. The desirability metric for scheduler-centric approaches is most often through the use of an actual schedule. But again, this is not a requirement.

The most well known scheduler-centric clustering algorithm is Bottom-Up Greedy, or BUG [6]. BUG occurs before scheduling; other algorithms such as [19], [22], and [23] are similarly scheduler-centric though they take place during or interleaved with scheduling.

BUG proceeds by recursing depth-first along the DFG, critical paths first. It assigns each operation to a cluster based on estimates of when the operation and its predecessors can complete earliest. These estimates are based on resource usage information from the scheduler, and BUG queries this information twice whenever it considers each operation on each cluster—once before and once after its predecessors have been bound.

This works well for simple graphs, but when the graph becomes more complex such that locally good decisions may have negative effects on future decisions, the algorithm can be fooled into making a bad partition.

Figure 2(a) shows a likely partition generated by BUG or other similar local, scheduler-centric algorithms. The critical path (1, 2, 6, 10, 12) is considered first, and nodes 1, 2, and 6 are placed together on one cluster. Nodes 3 and 7 are placed on the other cluster, since this allows node 10 to begin and complete executing soonest. However, the right subtree is now constrained by the decisions that were locally optimal for the left subtree. As a result, in our example machine which executes one operation per cycle per cluster, this code would take 8 cycles to complete.

The optimal partitioning requires only 7 cycles and is shown in Figure 2(b). Operations 4, 5, 8, 9, and 11 should be placed on one cluster, with the remaining operations on the other cluster, as the shading in the figure indicates. With this partition, one inter-cluster move is needed along the edge from 11 to 12. This partitioning minimizes the resultant schedule length by balancing the workload effectively and introducing only one move operation, which is not on the critical path.

Another limitation of BUG is that it keeps track of which resources are busy as it proceeds, and at every step of the algorithm it performs checks to see if a cluster is free at a certain time to perform a certain operation. Therefore, the number of queries to the resource information grows with the number of clusters in the machine and with the number of nodes in the graph.

In order to avoid the potential pitfalls of local decision-making and the compiler overhead of using detailed scheduling information for resources, our approach is to view the graph more globally and to use estimates for determining resource load balance.

3. REGION-BASED HIERARCHICAL OPERATION PARTITIONING

Our region-based hierarchical operation partitioning algorithm consists of two distinct phases: weight calculation and partitioning. Each operation is represented by a node in a DFG, and node weights are created to represent the resource utilization of the operation. The edges connecting the nodes are given edge weights, which represent the cost on the schedule length for adding an inter-cluster move between those operations. Both node and edge weights are used to guide the partitioning phase. The node weights are used to balance the workload among the clusters, while the edge weights are used to minimize the communication required between them. The partitioning phase consists of a coarsening process, where highly related operations are combined together and placed into clusters, and a refinement process, which improves the initial partitioning. The refinement process uses the calculated weights to consider moving operations between clusters, then iteratively improves the partition by weighing the benefits of improving load balance and reducing inter-cluster communication.

The rest of this section includes a more detailed explanation of the clustering process. The example code and DFG shown in Figure 3 will be used throughout this section to demonstrate the process.

3.1 Weight Calculation Phase

3.1.1 Node Weights

Node weights enable the algorithm to calculate an estimate of how many cycles it will take to execute a set of operations on a cluster, ignoring dependencies, when it is determining the quality of a partition under consideration. Thus, the weights reflect the quantity of resources an operation uses in the machine.

Resources can be characterized as being used by an individual operation, such as FUs, or shared between operations, such as buses or RF ports. In general, resource usage in a machine forms a spectrum between these two extremes. We use the two endpoints to compute an individual node weight and a shared node weight for each operation. The worst case of these provides an approximation of the operation's resource usage.

Individual node weight, or op_wgt_c, is calculated per node for each cluster c, and is a measure of the resources used by this particular node. Note that, in the case of heterogeneous clusters, the weight of a node is dependent on which cluster it is being considered on. For example, an ADD operation on a cluster with one adder carries more weight than an ADD operation on a cluster with two adders, because in the second case it only uses up half of the available resources.

To determine the weight of a node on a cluster, the number of times that the resources available on that cluster will support the execution of that operation in a single cycle is counted. The weight is the inverse of this number:

$$op\_wgt_c = \frac{1}{\#ops\ supported\ on\ c\ in\ 1\ cycle}$$

Since the example machine executes one operation per cycle, the individual weight of all of the nodes in the graph is 1.0 for both clusters.

To account for shared resources, a shared node weight value is calculated per cluster for the region being clustered. This shared node weight on a cluster, $shared\_wgt_c$, is determined by placing all of the operations in a region on cluster c, and dividing the resulting resource-limited minimum schedule length by the number of operations. The minimum schedule length is similar to RESMII used in modulo scheduling. Since this is done only to determine resource availability and ignores data dependencies, no actual scheduling is done and the calculation is fast.

$$shared\_wgt_c = \frac{resource\ limited\ sched\ length\ on\ c}{\#ops}$$

In the example, placing the 14 operations on either cluster reveals that a minimum of 14 cycles is required. Thus the shared weight is 1.0 on each cluster. Due to the simplicity of the machine, these numbers are somewhat trivial. Given a real machine with more and varied resources available, as was used in our experiments, the node weights become more interesting.

3.1.2 Edge Weights

The weight of an edge is a measure of its criticalness. If an edge is critical, then placing the nodes on either end of the edge on different clusters and inserting the required move will impact the schedule length. Therefore, edges on the critical path have a higher weight than other edges, and the graph partitioning algorithm attempts to minimize the sum of the weights of the edges that are cut by the partition.

Once critical edges are assigned a high weight, the remaining edges can be assigned a low weight. However, this can be dangerous as there may be a non-critical path that, once a few moves are inserted, becomes critical. Therefore, a more intelligent system for assigning edge weights to non-critical edges is beneficial.

The concept of *slack* is a measure of how critical an edge is. An edge on a path where nodes can be delayed without affecting the overall minimum schedule length has more slack, while a critical edge has no slack. We use a definition of slack similar to that of *global slack* in [9], though our measurement is per-edge rather than per-node.

The slack of a directed flow edge from *src* to *dest* is defined

```
                              (estart, lstart)
1:  ADD  r1  <- r2, 4             (0,0)
2:  SUB  r3  <- r4, 2             (0,0)
3:  SHR  r5  <- r1, r3            (1,1)
4:  ADD  r6  <- r7, 8             (0,1)
5:  ADD  r8  <- r9, r10           (0,1)
6:  SUB  r11 <- r12, 2            (0,1)
7:  SUB  r13 <- r14, r15          (0,1)
8:  ADD  r16 <- r5, 8             (2,2)
9:  MAC  r17 <- r6, r8, r11       (1,2)
10: SHL  r18 <- r19, r20          (0,2)
11: SHR  r21 <- r13, 2            (1,2)
12: SUB  r22 <- r16, r17          (3,3)
13: SUB  r23 <- r18, r21          (2,3)
14: ADD  r24 <- r22, r23          (4,4)
```

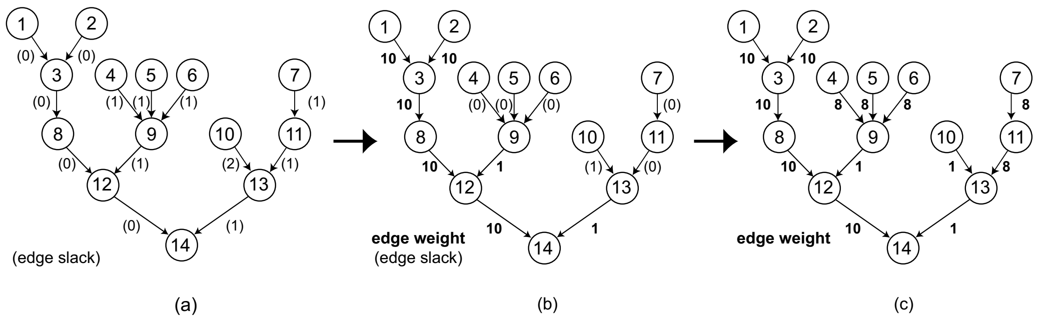


Figure 3: The example code and corresponding DFG with slack distribution defining the edge weights.

as:

$$slack_{edge} = lstart_{dest} - lat_{edge} - estart_{src}$$

Here, $estart$ refers to the earliest cycle an operation can begin executing (i.e. its inputs are available), and $lstart$ refers to the latest cycle it can begin executing without delaying the exit operation(s) of the region. The latency of the edge, lat_{edge}, is defined to be the latency of the src operation.

Thus, in the example DFG of Figure 3(a) with $estart$ and $lstart$ as shown next to the assembly code, the edges on the critical path 1–3–8–12–14 have zero slack; the edge from node 10 to 13 has a slack of $3 - 1 - 0 = 2$; and the remaining edges have a slack of 1.

A method of first-come first-serve slack distribution is used to account for paths that have some slack in them by taking up slack (by increasing latency) starting from edges close to the critical path. This increased latency may lower the $lstart$ of operations higher on the path, thereby decreasing the slack on their incoming edges. The process continues for the next edge on the path until all of the slack has been allocated. The edge weights are assigned depending on whether or not slack was allocated to the edge, based on the following numbers:

$$edge\_wgt = \begin{cases} 10 & \text{if critical} \\ 8 & \text{if no slack after distribution} \\ 1 & \text{if slack allocated} \end{cases}$$

These numbers were chosen because cutting a critical edge will increase the schedule time so it is weighted high; cutting an edge that has slack allocated to it is essentially free, so it is weighted low. Cutting a non-critical edge that has no slack remaining after distribution is not guaranteed to increase the schedule, but it is likely especially if the "free" edges are cut; therefore, it is given a high weight but not as high as that of a critical edge.

Using this slack distribution algorithm, edges closer to the critical path are more likely to be cut. This accomplishes the goal of offloading as much work as possible from the critical path. It also discourages cutting a single non-critical path too many times such that it becomes critical.

As shown in Figure 3(b), the edges which initially had zero slack (indicating that they are critical) are assigned a weight of 10. Now, the non-critical edge from node 13 to 14 can have a unit of slack allocated to it, giving it a weight of 1. This lowers the $lstart$ of node 13 by one cycle, with the result that the slacks on the edges coming into node 13 are decreased. Similarly, the edge from node 9 to 12 receives a weight of 1, and the slacks on the edges coming into node 9 are decreased. Figure 3(c) shows the final edge weights for this DFG, with the edges that had zero slack remaining after step 3(b) receiving a weight of 8, and the edge from node 10 to 13 receiving a weight of 1.

3.2 Partitioning Phase

The partitioning phase of RHOP employs a multilevel graph partitioning algorithm to cluster the DFG of a region into distinct groups. Multilevel graph partitioning, known for its efficiency and good results, is available in many software packages such as Chaco [12] and Metis [15].

A multilevel algorithm coarsens highly related nodes together and places them into partitions. The nodes within the graph are continually coarsened by grouping pairs of nodes together. At each level of coarsening, a snapshot of the currently coarsened nodes is taken. When the number of coarse nodes reaches the number of desired partitions, coarsening stops. The coarse nodes are then assigned to different clusters, and the uncoarsening process begins. During uncoarsening, the algorithm backtracks across the earlier snapshots of coarse nodes, considering moving operations at each stage. A refinement algorithm is used to decide the benefits of moving a node from one cluster to another in order to improve the partition.

3.2.1 Coarsening

Coarsening takes a DFG representing the region to be clustered and produces an initial partition for the graph. Producing a good initial partition has been shown to have a large impact on how well the algorithm produces results [12]. The coarsening algorithm uses edge weights determined earlier during the weight calculation phase to intelligently group operations together. Operations separated by a high weight edge are thus first targeted for coarsening.

Each stage of the coarsening process groups together operations into pairs based on the weights of their edges. All operations are sorted based on the highest weight on any of its edges and considered for coarsening in that order. Operations on the critical path will then most likely be paired together. In order to try and coarsen as many nodes as possible at each stage, operations are coarsened from the outside of the DFG toward the center; thus, operations with a single neighbor have higher priority for coarsening. Ties in preferences for coarsening are broken arbitrarily.

Figure 4 shows how coarsening progresses through the running example. At the first stage, the first priority is to

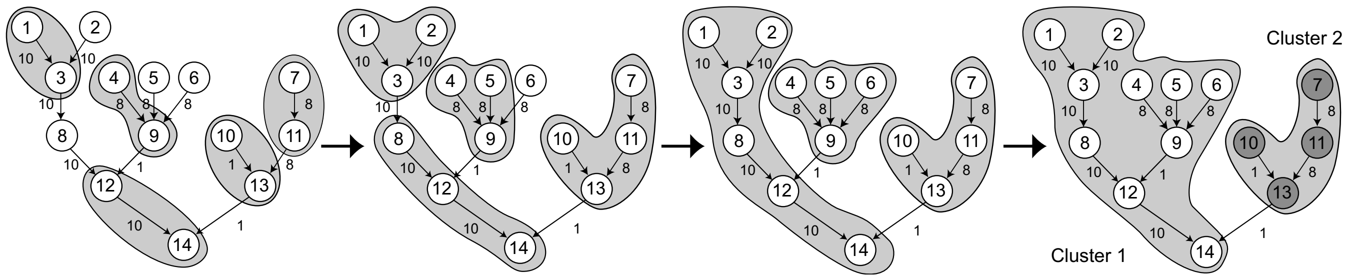


Figure 4: The coarsening process to group together highly related operations and create the initial cluster assignment.

coarsen together critical edge paths, then the lower weight edges are considered. Each stage of the process only pairs up a single operation once, and every operation that has an available neighbor to coarsen with will be paired up. Operations that cannot be paired up, either because they have no neighbors or all of their neighbors have already coarsened with other operations, will be left as is for the current coarsening stage. For example, in the first coarsening stage of Figure 4, operation 8 no longer has any uncoarsened neighbors, so is not coarsened for this particular stage. When no more operations in a stage can be coarsened, the entire coarsening process is repeated with the resulting coarse nodes.

The coarsening phase ends when the number of coarse nodes is equal to the number of desired clusters for the machine. The coarse nodes are then divided up between the clusters to form the initial partition. For the running example, the final partition ends with operations 1, 2, 3, 4, 5, 6, 8, 9, 12 and 14 on cluster 1; with the rest on cluster 2.

3.2.2 Refinement

The refinement process traverses back through the coarsening stages, making improvements to the initial partition. At each uncoarsening stage, the coarsened nodes available at that point are considered for movement to another cluster. In order to properly improve the partition of the operations in the graph, the algorithm must have metrics for deciding which cluster to move from, the desirability of the current partition, and the benefits of an individual move. The refinement process uses the following to judge each of these:

- **Cluster Weight:** The node weights for each operation are used to generate an estimate for the load per cluster; the cluster with the highest weight is denoted the imbalanced cluster.
- **System Load:** Similar to the cluster weight, the system load uses the node weights of all the operations, but estimates the load across all clusters, generating a metric for the current cluster assignment desirability.
- **Gain:** Once the imbalanced cluster has been targeted, the gain of moving each operation to the other clusters is calculated using the change in system load and the change in edge cuts.

Since the process backtracks through the coarsening stages, highly related operations are grouped together at each stage, and the algorithm can then account for a group of operations preferring to move together. This helps to alleviate situations where moving one operation to another cluster is not beneficial, but moving a group of related operations together will show improvements.

The refinement algorithm is a slightly modified Kernighan-Lin partitioner, which is known to be a good algorithm for partitioning graphs. Traditionally, Kernighan-Lin tries to match pairs of operations from different partitions to swap. Each swap incurs some cost upon the system, and swaps are continually made until the overall cost gain is negative. This allows individual negative moves, which may in fact allow future positive moves to occur. By allowing individual negative moves, the algorithm avoids falling into local minima.

A modified version of Kernighan-Lin is used which considers node and edge weights to determine the gain of moving an operation to another cluster. Unlike Kernighan-Lin, which weighs the benefits of swapping nodes between partitions, RHOP considers explicitly moving each operation within the imbalanced cluster, rather than swapping. Like Kernighan-Lin, RHOP allow moves with negative gain as long as the overall gain for the current refinement step is positive.

Cluster weight. To determine the cluster which is most imbalanced, cluster weights, the metric for the load per cluster, is calculated. In order to calculate the weight of a particular cluster, a weight for each execution cycle of the region is computed. To estimate the weight of each operation at each cycle, we use the scheduling range, which is the $estart$ of the operation to its $lstart$. The operation must be placed within this range in order to achieve minimum schedule length.

The two important factors in regards to the load of operations on a cluster are: the individual resource constraints for the operations at each cycle, and the total node weight which is the constraint on the shared resources of a given cluster. The individual resource constraint is the load put on any one specific resource. The shared resource weight is the load put on all the resources within the cluster as a whole. Since these individual resource and shared resource weights are competing with one another, the overall cluster weight is the max between them.

To compute the individual resource constraints, each operation is placed in a single *op group*, which groups similar operations by their resource usage. For each operation in an *op group*, its total impact to a particular cycle is the node weight for the operation, calculated earlier, divided by the slack+1 of the operation. This value, the individual weight, $Iwgt_{c,\tau}$, for cluster c at cycle τ gives a general approximation of the impact of those operations which use a similar resource, currently placed in that cycle on the cluster load.

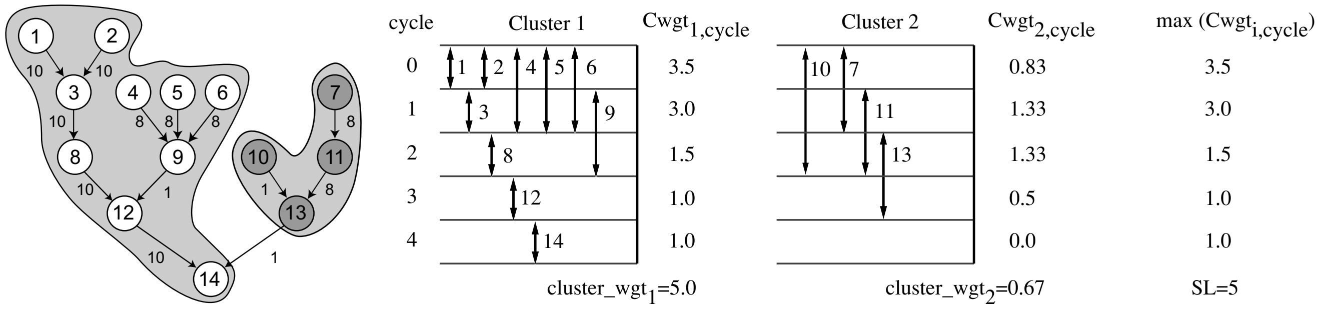


Figure 5: The initial partition after coarsening and the cluster weights.

The total node weight, $Twgt_{c,\tau}$, is then calculated as the total number of operations currently placed within cluster c at cycle τ divided by the average slack for all the operations. This is then multiplied with the shared resource weight from the weight calculation phase to give an approximation of the constraints on the shared resources. This gives an estimate as to how well the operations share the resources at a cycle. The total node weight is effective in situations where there is a lot of parallelism and the assumption of operations finishing within the scheduling range breaks down. Thus, the desired partition focuses more on spreading the work out evenly among the clusters.

The cycle weight $Cwgt_{c,\tau}$ in cluster c at cycle τ is therefore determined by:

$$Iwgt_{c,\tau} = \max_{o \in opgroups} \sum_{op \in o \ at \ \tau} \frac{op\_wgt_c}{op_{slack} + 1}$$

$$Twgt_{c,\tau} = \frac{\#ops \ in \ c \ at \ \tau}{slack_{ave} + 1} * shared\_wgt_c$$

$$Cwgt_{c,\tau} = \max(Iwgt_{c,\tau}, Twgt_{c,\tau}) \qquad (1)$$

For example, at the end of the coarsening process, the graph reaches a partition as shown in Figure 5 with its corresponding cycle-by-cycle representation of all operation scheduling ranges (*estart* to *lstart*). For the simplicity of the example, we will not consider the explicitly consider total node weight, as it has no effect in the result. The $Cwgt_{c,\tau}$ of each cluster is shown in the figure. $Cwgt_{1,1}$, the cycle weight of cluster 1 at cycle 0 is calculated as follows: ops 1 and 2 each have a node weight of 1 and slack+1 of 1. Ops 4, 5, and 6 each have node weight of 1 and a slack+1 of 2. Therefore, ops 1 and 2 each contribute 1 to the cycle weight while ops 4, 5, and 6 each contribute 0.5, which forms a cycle weight, $Cwgt_{1,1}$, of 3.5.

The weight of a cluster, $cluster\_wgt_c$, is simply the sum of all cycle weights from 0 until the max estart, minus one. One cycle is subtracted from each cycle weight to evaluate how overloaded each cycle is. Since every cycle can do one cycle's worth of work, any amount greater than one means the cycle has too much work assigned to it. Therefore, our cluster weight equation is:

$$cluster\_wgt_c = (\sum_{t=0}^{max \ estart} Cwgt_{c,t} - 1) \qquad (2)$$

System Load. While equation 2 gives an estimate to the weight of any one cluster, it doesn't give a general estimate for the overall desirability of the current chosen clustering. This is defined by the system load, SL, which gives a cycle by cycle account for the clustering. At any given cycle, whenever one cluster's weight dominates that of the other cluster, the smaller load is subsumed by the larger. Therefore equation 1, which calculated the load every cycle in a cluster, is maxed it across all clusters and summed for the scheduling range. The system load then results in the maximum any cluster is overloaded over all cycles in the scheduling range.

Since inter-cluster moves have a limited bandwidth, the system load is maxed with consideration of the inter-cluster moves required for this cluster. The inter-cluster move bandwidth, *icm bw*, is used to determine the overhead, icm_o, of making the inter-cluster moves required in the current partition. This inter-cluster move overhead is mostly used as a safeguard to prevent clusters from forming that contain far too many inter-cluster moves. In general, the partitioner tries to minimize edge cuts, so this simple estimate of total inter-cluster moves by the cluster is all that is necessary. The system load is therefore determined by:

$$icm_o = \frac{\#icm}{icm \ bw} - (max \ estart \ + 1)$$

$$SL = \max((\sum_{\tau=0}^{max \ estart} \max_{i \in cluster} Cwgt_{i,\tau} - 1), icm_o) \qquad (3)$$

Gain. At each uncoarsening stage, our algorithm calculates the weight of both clusters using equation 2. The cluster with the higher weight, which we refer to as the imbalanced cluster, is chosen as the one to begin moving operations from. A metric is then needed for determining the benefits of moving an operation to a different cluster. Each time a node is moved to another cluster, there is a shift in both the load balance, as a different set of operations are now on each cluster, and also the cut edges, as there will now be different edges between clusters requiring inter-cluster moves.

Thus, the load gain, *Lgain*, is defined as the difference in the system load before and after the proposed move is made. The edge gain, *Egain*, is the sum of the edge weights of the edges merged minus the sum of the edge weights of the edges cut. The algorithm counts an increase of one on the load gain as equal importance to cutting a critical edge, as it will mean the cycle is so overloaded with work that schedule length must be increased by one. Therefore, the difference in system load is multiplied by the cost of a critical edge, which is currently specified as 10. The overall gain for a

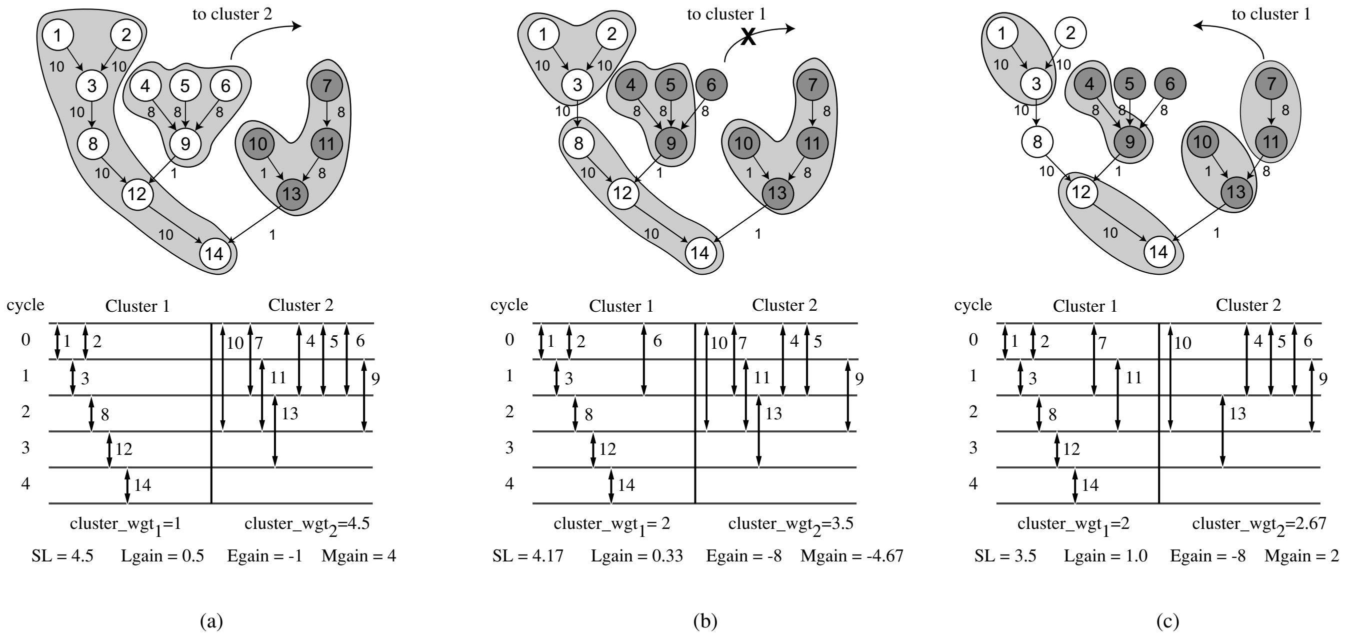


Figure 6: The refinement process traveling back through coarsened states. (a) The beneficial move of the coarsened node containing operations 4, 5, 6 and 9 to cluster 2. (b) A situation where no positive moves exist and the move is not made. (c) Moving the coarsened node containing 7 and 11 to cluster 1 is now beneficial.

particular move, $Mgain$, is then determined by:

$$Egain = \sum_{i \in merged\ edges} edge\_wgt_i - \sum_{j \in cut\ edges} edge\_wgt_j$$

$$Lgain = SL_{(before)} - SL_{(after)}$$

$$Mgain = Egain + (Lgain * CRITICAL\ EDGE\ COST)$$

Figure 6 shows the refinement process and gain calculations on the running example. In 6(a), the proposed move is to change the coarsened node containing operations 4, 5, 6, and 9 from cluster 1 to cluster 2. This decreases the system load from 5.0 to 4.5, a $Lgain$ of 0.5. By moving this operation over, no edges are merged, and a weight 1 edge is cut (between operations 9 and 12). Therefore the $Mgain$ for this operation is $-1 + 5 = 4$. No other moves in this uncoarsening step are beneficial, so the graph is uncoarsened again.

The next uncoarsened state is shown in Figure 6(b), where operation 6 has been uncoarsened from 4, 5, and 9, all of which are now in cluster 2 after step 6(a). Of interest is that even though moving operation 6 from cluster 1 to cluster 2 provides a positive $Lgain$ by dropping the system load from 4.5 to 4.17, This is not enough to counteract the cost of cutting the weight 8 edge from operation 6 to 9, therefore this move is not made. Since no move in this coarsening state is beneficial, uncoarsening continues.

Next the graph reaches the uncoarsening state in Figure 6(c). At this stage of uncoarsening, both the coarsened nodes containing 4 and 9 as well as 7 and 11 decrease the system load from 4.5 to 3.5 if moved to cluster 1, as they both have the same node weights and affect the same cycles. The coarse node with 7 and 11 is chosen for moving, though, because it only cuts one weight 8 edge and thus remains a positive move, while the coarse node 4 and 9 cuts two weight 8 edges and merges one weight 1 edge, making it a negative $Mgain$.

Each uncoarsening stage finishes when it can make no more moves and the same imbalanced cluster is chosen twice in a row. Then, it moves on to the the next uncoarsened stage and the refinement process is repeated. When the uncoarsening process completes its refinement of the original, totally uncoarsened snapshot of the region, one final pass through each of the clusters is run, to ensure that no positive moves out of a cluster were ignored because another cluster was extremely out of balance. In this final phase, each cluster allows only positive moves.

When this final phase completes, the resulting partitions correspond to the desired clusters. The node weights and edge weights ensure that there exists a good load balance between the clusters as well as a minimal cut set for intercluster communication. For this example, the final uncoarsening step yields no change from the partition after the move in Figure 6(c). Thus, the final partition is operations 1, 2, 3, 7, 8, 11, 12 and 14 on cluster 1, with the remaining operations on cluster 2. Even though there are three cuts in this partition, none are critical and this results in the optimal schedule length of 8 cycles.

4. EXPERIMENTAL EVALUATION

We implemented the RHOP algorithm using the Trimaran tool set [25], a retargetable compiler for VLIW processors.

4.1 Methodology

To gauge the performance of our algorithm, we compared our results to the BUG algorithm. We evaluated the performance of both BUG and RHOP on several DSP kernels and the SPECint2000 benchmark suite. DSP kernels

Kernel	**2-1111**	**2-2111**	**4-1111**	**4-2111**	**4-H**
adpcm	-2.09	3.25	12.03	8.95	11.98
atmcell	-0.32	3.34	34.86	32.58	14.04
channel	-3.35	-0.73	11.20	20.44	6.50
dct	-0.64	10.53	31.24	28.86	17.31
fir	4.75	15.74	30.90	12.34	11.62
fsed	4.39	6.52	22.87	27.90	10.65
halftone	1.17	4.91	27.99	34.18	-2.12
heat	-6.24	21.50	31.23	33.32	15.26
huffman	-4.84	-3.87	24.65	24.79	19.76
LU	-2.65	-1.23	-1.42	12.44	4.51
lyapunov	1.83	9.43	13.26	6.63	13.41
rls	-1.90	4.50	6.09	30.51	11.42
sobel	-2.04	1.02	20.67	20.92	22.20
Average	-0.92	5.75	20.43	22.60	12.04

SPEC	**2-1111**	**2-2111**	**4-1111**	**4-2111**	**4-H**
164.gzip	-2.18	5.21	8.67	6.86	4.12
175.vpr	-5.98	2.42	3.26	6.15	3.41
181.mcf	-1.72	-1.49	3.99	-5.99	-3.44
197.parser	-3.45	-2.76	-1.22	-1.40	-1.62
253.perl	-3.16	0.00	6.25	5.13	1.84
254.gap	-5.79	0.34	-0.76	0.47	-1.08
255.vortex	-2.61	2.08	-5.29	7.59	1.84
256.bzip2	-1.02	-0.29	25.45	21.66	9.66
300.twolf	-2.16	1.13	8.24	3.41	4.04
Average	-3.11	0.73	5.40	4.87	2.28

Table 1: Percentage improvement by RHOP on cycle time over the BUG algorithm for several kernels and the SPECint2000 benchmarks on five different machine models.

were investigated because of their characteristically high ILP that make them ideal candidates for wide-issue processors. As a result, they provide a true measure of the clustering algorithm's ability to exploit high levels of ILP. The SPECint2000 benchmarks[1] were also used because of their generally low and irregular ILP. These benchmarks provide the challenge of exploiting ILP when it is available, but not over-partitioning when ILP is limited.

Five different machine configurations were used to compare our performance with BUG. Common to all these machines are 64 registers per cluster, operation latencies similar to those of the Itanium, and perfect caches. Four of the machine configurations have homogeneous clusters (i.e. the resources on each cluster are identical), and the last one is a heterogeneous machine. Each has varying numbers of integer (I), float (F), memory (M) and branch (B) units. The different machine configurations are summarized below:

Name	**Configuration**
2-1111	2 Homogeneous Clusters 1I, 1F, 1M, 1B per cluster
2-2111	2 Homogeneous Clusters 2I, 1F, 1M, 1B per cluster
4-1111	4 Homogeneous Clusters 1I, 1F, 1M, 1B per cluster
4-2111	4 Homogeneous Clusters 2I, 1F, 1M, 1B per cluster
4-H	4 Heterogeneous Clusters IF, IM, IB, and IMF clusters

For each benchmark, the dynamic cycle count was used as the evaluation metric for how well the clustering algorithm was able to partition the code into clusters. After clustering, prepass scheduling, register allocation and postpass scheduling are performed to generate the final assembly code.

4.2 Analysis of Results

Table 1 shows our improvement over BUG for 13 kernels and the SPECint2000 benchmarks for the five different machine models. For each kernel, we present the percentage improvement in dynamic total cycles of RHOP over BUG. Positive results mean RHOP performed better than BUG, while negative results mean BUG performed better.

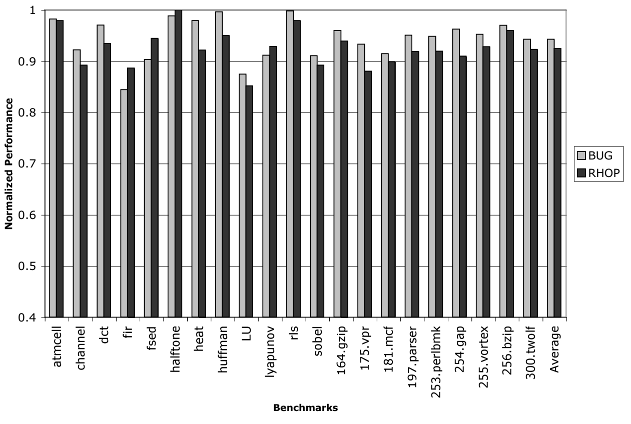
Overall, our results for a two cluster machine with one resource of each type are rather poor, with an average increase in dynamic cycles of 0.92% on the kernels. As the machine configuration becomes more complex, by adding more resources and additional clusters, results dramatically improve in the quality of our operation clustering. On the kernels, there was an average of 20% improvement on the 4-1111 machine, and a 23% improvement on the 4-2111 machine. The results for the four-cluster heterogeneous machine fell between the two and four-cluster homogeneous machines. A similar trend is seen on the SPECint2000 programs in Table 1 except the improvements are more modest. In general, the SPECint2000 benchmarks have less ILP than the kernels, thus there is less opportunity for distributing work across clusters.

The data in the table shows that local, greedy methods for clustering can perform quite well in constrained, resource limited situations. The poor results from the two-cluster machine occur because of the inaccuracies of our resource model. Estimates, in general, can be wrong, and at times we observe one cluster gets more operations than it should. One major factor is that our resource load estimate ignores edges; thus, it also ignores dependencies between instructions, and assumes reordering is possible where in actuality, it may not be. We then have too many operations being placed into a cluster and forced to execute serially.

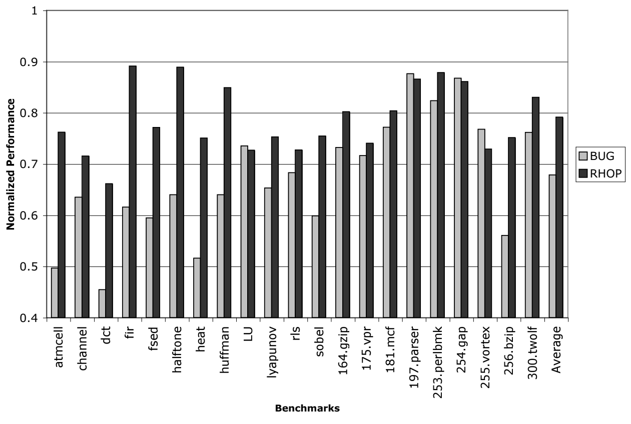
On the other hand, for four-cluster machines, the most important factor is carefully spreading out the workload among all the different clusters. In such a situation, the region-level scope used by RHOP becomes much more effective than a local, operation-centric scope. Thus, we are able to achieve a drastic improvement for four clusters.

Figure 7 compares the performance of the 2-1111 and 4-1111 machines using BUG and RHOP with a single cluster machine containing the sum of the resources of all the clusters. The single cluster machine provides an upper-bound of performance. For the two-cluster results, both BUG and RHOP achieve greater than 92% of the upper-bound. Conversely, for a four-cluster machine, BUG only achieves 68%

[1]176.gcc, 186.crafty, and 252.eon were not run due to limitations of the current Trimaran compiler system.



(a)



(b)

Figure 7: Comparison of BUG and RHOP clustering performance degradations on (a) 2 cluster (2-1111) and (b) 4 cluster (4-1111) machine configurations versus a 1-cluster machine with the sum of the resources of the clusters.

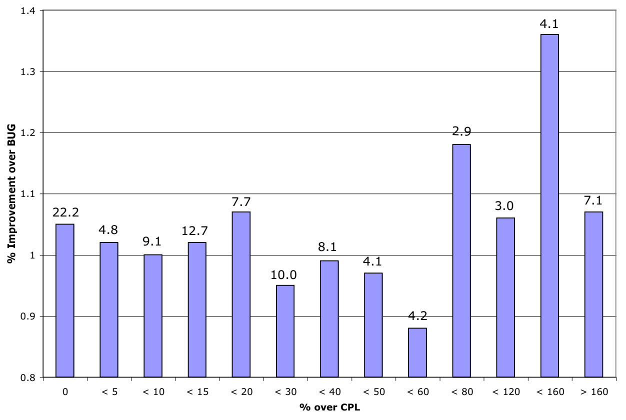


Figure 8: Histogram comparing the performance of RHOP and BUG; each category is the achieved schedule length of the region with respect to the critical path length. The numbers on top are the dynamic execution percentage of the category.

of the upper-bound. Again this is due to the local, greedy heuristics breaking down for wide machines. RHOP increases performance to 79% of the upper bound. Clearly, there is still room for improvements to the RHOP algorithm.

Since our desirability metrics assume that schedules finish within the critical-path length (CPL) number of cycles, we performed a study of RHOP performance as a function of schedule length relative to the CPL. In Figure 8, each bar represents the cumulative ratio of RHOP cycles over BUG cycles for all regions (across all benchmarks) in the range. The bars are annotated on top with the percentage of dynamic cycles that occur within these ranges. For regions very close to the CPL, our algorithm performs modestly well. These regions are critical-path limited, and our system load estimates are quite accurate. For regions much higher than the CPL, where the regions are resource-constrained, our algorithm performs even better. In such regions, the key to a good partition is properly spreading out work across the clusters, and the total node weight heuristic in RHOP intelligently balances the workload. The middle ground, when regions are neither critical-path nor resource constrained, is where RHOP has the most difficulty. Since neither resources nor CPL dominate, our resource estimates lose substantial accuracy and thus bad clustering decisions can be made.

In addition, the runtime of the two algorithms was evaluated. For a research-oriented compiler like Trimaran, simply evaluating raw compute time is a rather inaccurate way to measure the speed of an algorithm. A more realistic measurement is the number of calls to the resource table, which gives an estimate on how often the algorithm is checking and rechecking its resource model. This is the heart of the scheduler, where most of the time is spent. Thus, minimizing entries into this function is a key metric to improving compiler run-time. The results from this experiment are presented in Table 2. Our algorithm shows significant improvement over BUG, taking an average of 1.2 times the runtime of the scheduler alone, versus 3.0 times for BUG. This is a result of the necessity of scheduler-centric algorithms requiring a detailed model of the current resource constraints and repeatedly reevaluating the model for each step of the process.

5. RELATED WORK

There has been a large body of research conducted in the area of clustering. In Table 3, we summarize our general categorization of many of them based on the four characteristics of clustering algorithms presented in Section 2.

The most closely related work to our clustering approach is with algorithms that use graph partitioners, and those that use an estimate-based approach. Capitanio et al. [3] proposed a graph partitioning method to clustering operations, but focused mainly on a Kernighan-Lin like approach to improving partitions. They focus their improvements strictly on a function of the partition cut set, and weighing the benefits of making a cut with the probability that it will increase the schedule length.

Aletà et al. use a similar multilevel graph partitioner, but

Kernel	Sched	BUG	RHOP
adpcm	4550	13777 (3.0)	6676 (1.5)
atmcell	31560	109880 (3.5)	33244 (1.1)
channel	12294	32094 (2.6)	14686 (1.2)
dct	16646	47346 (2.8)	17148 (1.0)
fir	9284	26434 (2.8)	10474 (1.1)
fsed	13300	40244 (3.0)	13910 (1.0)
halftone	14109	29519 (2.1)	15475 (1.1)
heat	5159	13113 (2.5)	5667 (1.1)
huffman	22030	54974 (2.5)	25296 (1.1)
LU	1935	4563 (2.4)	3105 (1.6)
lyapunov	16256	43234 (2.7)	17940 (1.1)
rls	25305	84413 (3.3)	26471 (1.0)
sobel	8138	23145 (2.8)	9414 (1.2)
Average		(2.8)	(1.2)
SPEC	**Sched**	**BUG**	**RHOP**
164.gzip	385173	1303443 (3.4)	455795 (1.2)
175.vpr	1356211	4555987 (3.4)	1528947 (1.1)
181.mcf	220845	700958 (3.2)	245269 (1.1)
197.parser	1238238	4045074 (3.3)	1434704 (1.2)
253.perl	2102449	7066202 (3.4)	2862355 (1.4)
254.gap	2046872	6754402 (3.3)	2813026 (1.4)
255.vortex	2133516	7199868 (3.4)	2635402 (1.2)
256.bzip2	489923	1580643 (3.2)	550493 (1.1)
300.twolf	1405475	4861800 (3.5)	1681433 (1.2)
Average		(3.3)	(1.2)

Table 2: Number of calls to the resource table. For BUG and RHOP, the ratio of total calls over Scheduling-only calls is given in parentheses.

focus on tightly integrating the clustering algorithm with the instruction scheduling and register allocation [1]. Also studied was clustering via a multilevel partitioner to determine the optimal initiation interval (II) for a modulo scheduled loop using a pseudo-scheduler [2]. Their work focuses on scheduling cyclic code in multicluster domains, while ours is targeted toward acyclic code. We also use substantially different models for computing node and edge weights.

While not a heavily researched area, there has been some work on estimate-based approaches for clustering. Lapinskii et al. [17] base their estimate off three major factors: the data transfer penalty, FU serialization penalty and bus serialization penalty. They use a local approach like BUG to minimize the data transfer penalties. The FU serialization is basically the load of the cluster, which is determined in a cycle by cycle approach.

Partitioning can also be approached by considering operands rather than operations [13]. Research on partitioning for multiprocessors has many similarities to clustering for multicluster processors. Yang and Gerasoulis [26] proposed a low-complexity method for clustering and scheduling parallel tasks for multiprocessors. Liou and Palis [20] improved upon the complexity of this algorithm.

6. CONCLUSION

This paper proposes a novel technique to cluster operations for multicluster processors. A slack distribution algorithm is presented, which effectively weights edges based on their preference for being broken across clusters. We introduce a new way to estimate the impact of clustering decisions, which is used to guide our graph partitioner. Our graph partitioner is able to consider an entire region of code and base its decisions off a view of the code as a whole, rather than what the best clustering is for a single operation.

We compared our results to a popular algorithm, BUG, and results show that for larger number of clusters, our algorithm is able to efficiently produce better partitions. Two-cluster machines saw an average performance decrease of 1.8% across all kernels and benchmarks. As we increased the number of clusters, there was a dramatic increase in the performance of our partitioner. A four-cluster machine provided an average improvement of 14% in our experiments.

In the future, we plan to improve our system load estimation heuristic in situations where regions are neither resource nor critical path limited. This corresponds to the middle portion of Figure 8, where RHOP's performance decreased in comparison to BUG. In addition, we plan to investigate the effects of register allocation and register pressure on clustering decisions. Extending RHOP to effectively cluster modulo scheduled loops presents another area of future study.

7. ACKNOWLEDGMENTS

We thank K. V. Manjunath for his help in debugging the compiler infrastructure. We also thank the anonymous referees for their comments and suggestions. This research was supported in part by the DARPA/MARCO C2S2 Research Center and equipment donated by Intel Corporation.

8. REFERENCES

[1] A. Aletà, J. Codina, J. Sánchez, and A. González. Graph-partitioning based instruction scheduling for clustered processors. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, Dec. 2001.

[2] A. Aletà, J. Codina, J. Sánchez, A. González, and D. Kaeli. Exploiting pseudo-schedules to guide data dependence graph partitioning. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2002.

[3] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 103–114, Dec. 1992.

[4] J. Codina, J. Sánchez, and A. González. URACAM: A unified register allocation, cluster assignment and modulo scheduling approach. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, Dec. 2001.

[5] G. Desoli. Instruction assignment for clustered VLIW DSP compilers: A new approach. Technical Report HPL-98-13, Hewlett-Packard Laboratories, Feb. 1998.

[6] J. Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, Cambridge, MA, 1985.

[7] P. Faraboschi, G. Desoli, and J. Fisher. Clustered instruction-level parallel processors. Technical Report HPL-98-204, Hewlett-Packard Laboratories, Dec. 1998.

[8] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time

	When (rel. to sched)			**Scope**		**Desirability Metric**				**Grouping**	
Algorithm	Before	During	Iterative	Local	Region	Sched	Pseudo	Est	Count	Hier	Flat
BUG [6] [21]	√			√		√					√
PCC [5]	√			√	√	√				√	
UAS [23]		√		√		√					√
ABC [16]	√				√				√		√
Eichenberger [22]			√	√		√					√
Leupers [19]			√	√		√					√
Capitanio [3]			√		√				√		√
URACAM [4]		√		√			√				√
GP(A) [1]			√		√	√				√	
GP(B) [2]		√			√		√			√	
B-ITER [17]	√			√				√			√
CARS [14]		√			√			√			√
Convergent [18]		√			√			√			√
RHOP	√				√			√		√	

Table 3: A comparison of several different clustering techniques based on four important characteristics: when the clustering occurs in relation to scheduling, the scope of the algorithm, the metric used in order to determine the quality of the partition, and whether operations are considered individually or in groups.

through partitioning. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, Dec. 1997.

[9] B. Fields, R. Bodík, and M. D. Hill. Slack: Maximizing performance under technological constraints. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[10] J. Fisher. Very long instruction word architectures and the ELI-52. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 140–150, June 13–17, 1983.

[11] R. Hank, W. Hwu, and B. Rau. Region-based compilation: An introduction and motivation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 158–168, Nov. 1995.

[12] B. Hendrickson and R. Leland. *The Chaco User's Guide*. Sandia National Laboratories, July 1995.

[13] J. Hiser, S. Carr, and P. Sweany. Global register partitioning. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, pages 13–23, Oct. 2000.

[14] K. Kailas, K. Ebcioğlu, and A. Agrawala. CARS: A new code generation framework for clustered ILP processors. In *Proceeding of the 2001 International Conference on High Performance Computer Architecture*, pages 133–142, Feb. 2001.

[15] G. Karypis and V. Kumar. *Metis: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparse Matrices*. University of Minnesota, Sept. 1998.

[16] G. Krishnamurthy, E. Granston, and E. Stotzer. Affinity-based cluster assignment for unrolled loops. In *Proceedings of the 2002 International Conference on Supercomputing*, pages 107–116, June 2002.

[17] V. Lapinskii, M. Jacome, and G. de Veciana. High-quality operation binding for clustered VLIW datapaths. In *Proceedings of the 2001 Design Automation Conference*, June 2001.

[18] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe. Convergent scheduling. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, Nov. 2002.

[19] R. Leupers. Instruction scheduling for clustered VLIW DSPs. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2000.

[20] J. Liou and M. Palis. A new heuristic for scheduling parallel programs on multiprocessor. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 358–365, Oct. 1998.

[21] P. Lowney et al. The Multiflow Trace Scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, 1993.

[22] E. Nystrom and A. E. Eichenberger. Effective cluster assignment for modulo scheduling. In *Proceedings of the 31th Annual International Symposium on Microarchitecture*, pages 103–114, Nov. 1998.

[23] E. Özer, S. Banerjia, and T. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proceedings of the 31th Annual International Symposium on Microarchitecture*, pages 308–315, Nov. 1998.

[24] B. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, Nov. 1994.

[25] Trimaran. An infrastructure for research in ILP. http://www.trimaran.org.

[26] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 1994.