Cost-Sensitive Operation Partitioning for Synthesizing Custom Multicluster Datapath Architectures

Michael L. Chu Kevin C. Fan Rajiv A. Ravindran Scott A. Mahlke

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

ABSTRACT

Clustered architectures have the primary benefit of mitigating the bottleneck associated with large, centralized register files. In a conventional multicluster processor, the architecture has fixed, homogeneous clusters; each cluster replicates the same functionality and capabilities. Fixing the architecture allows the compiler's operation partitioning algorithm to focus on minimizing intercluster communication and intelligently grouping operations to improve performance. However, in application-specific processor design, using such a fixed specification unnecessarily narrows the possible design choices. Rather than partitioning in a performance-centric manner, the compiler could potentially partition operations with a hardware-sensitive mindset. Thus, the compiler could derive application-specific heterogeneous clusters and a multicluster datapath from the partitioning of operations across clusters. For example, by simply grouping all operations with large bitwidth or high gate cost into one cluster, the remaining clusters could be scaled down and designed in a low-cost or low-power manner. In this work, we propose a system to create application-specific multicluster architectures which can specify each cluster's design and capabilities. This preliminary study focuses on the compiler method for partitioning operations across a set number of clusters to lower hardware cost with a minimal impact on performance.

1. INTRODUCTION

Modern processors achieve high performance by exploiting instruction-level parallelism (ILP) to issue multiple operations each cycle. In a conventional processor, a centralized register file supplies operands to the function units (FUs) where the operations are executed. This centralized register file quickly becomes a bottleneck as issue width grows, as both cost and access time scale quadratically with the number of register ports. In addition, a larger number of registers need to be available to maintain the larger number of temporary values. An increase in register file access latency arises from both slower access times to larger structures and the increasing distance between FUs and the register file [6].

To combat this bottleneck associated with the register file, one solution 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. Clustered architectures are becoming increasingly popular in many recent processor designs including the Lx/ST200, TI C6x series, and Analog Tigersharc.

In creating application-specific processors, current trends have moved toward automated design to reduce the complex design process. Automated design of a multicluster architecture is non-trivial, because traditional methods of design space exploration (DSE) through a heuristic search either become infeasible as the design space grows to an enormous size or only explore a small subset of the space by assuming a small number of fixed components [10]. From both a cost and performance perspective, all the details of a multicluster datapath must be defined to create a highly customized architecture, including number of clusters, number of FUs per cluster, opcode repertoire, bitwidth and interconnectivity of FUs, number of register files per cluster, width, ports, and number of entries per register file, etc.

To accomplish automated multicluster datapath design, we propose a hierarchical DSE system consisting of a heuristic search across high-level design parameters (number of clusters, parallelism per cluster) and a heuristic algorithm for determining the details of the datapath (FU type, bitwidth, etc.). For the latter step, our approach is to use sophisticated compiler techniques and analyses to determine both cluster configuration and capabilities. In designing a multicluster processor, the key compiler technique is the operation partitioning method, which partitions the dataflow graph (DFG) of the program into distinct regions to be executed on specific clusters. As the job of the partitioning algorithm is to determine where each operation is to execute, the algorithm plays the defining role in the cluster configuration. Given a cluster assignment of operations from the partitioner, along with a performance requirement, the custom datapath architecture can then be synthesized.

In this work, we focus on this partitioning step, by extending a partitioning algorithm to effectively balance cost and performance while placing operations in clusters. Our approach carefully separates operations across a set number of clusters considering the effects of operation bitwidth as well as the cost of adding additional FU capabilities. The result is a set of highly tuned clusters specifically designed to execute a given application at a low cost, while maintaining a high level of performance.

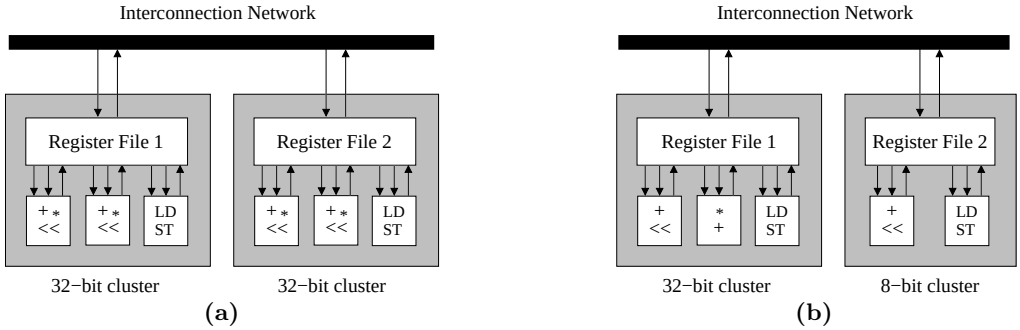

Figure 1: (a) A homogeneous two-cluster machine. (b) A heterogeneous two-cluster machine with separate 32-bit and 8-bit clusters.

2. BACKGROUND

2.1 Multicluster Architectures

In this paper, the architectural model assumed is that of a statically scheduled, clustered VLIW processor. Figure 1(a) shows a typical two-cluster machine, where each cluster contains a register file and several FUs. The two clusters communicate via an interconnection network with explicit move instructions that are inserted by the compiler. Note that the machine in this figure is homogeneous – that is, all clusters have the same types, widths, and numbers of FUs and register files. Homogeneity has the benefit of simplifying the compilation process as the compiler can take advantage of the symmetry in the machine to reduce the complexity of partitioning operations. In addition, designing application-specific homogeneous clusters is a much simpler task, as the design architect need not worry about the performance impact of independently altering each cluster's configuration.

However, allowing multicluster processors to be heterogeneous permits greater design flexibility and less hardware redundancy when customizing for a given set of applications. For example, certain "expensive" FUs such as multipliers or dividers can exist in a subset of clusters in the machine, or a given cluster may support narrow bitwidth operations while other clusters support full width operations. For the purposes of this paper, "expensive" refers to the relative gate cost for constructing an FU to support the operation. Figure 1(b) shows an example of a heterogeneous machine where the second cluster supports operations on 8-bit data and contains a different mix of FUs than the first cluster.

Thus, a heterogeneous multicluster architecture can be far superior to a homogeneous one in terms of cost while still executing an application with a high level of performance. The difficult task for a heterogeneous multicluster datapath architecture is then the design of the clusters themselves. The design must specify exactly how each cluster should differ in order to maximize cost savings while minimizing performance loss.

2.2 Hierarchical Multicluster Synthesis

Designing an application-specific multicluster datapath architecture is a daunting task. Traditional datapath synthesis techniques have always focused on designing the datapath for one single block of code on a single cluster architecture. Typically, these designs go toward hardware accelerators or ASICs. For an application-specific multicluster datapath architecture, the obvious method of design space exploration becomes infeasible, as the design space is far too large.

Rather than focus entirely on design space exploration, our goal is to use a compiler-directed approach to datapath synthesis. The compiler has sophisticated analyses and optimizations at its disposal which can tailor an application to a specific datapath. Thus, the compiler can exploit the structure and available transformations to design alternatives that are cost-effective while still maintaining a high level of performance.

Our proposed system for a multicluster architecture synthesis system is shown in Figure 2. We envision a system where a custom multicluster architecture can be designed and tailored toward a specific application. The spacewalker sits above the entire system, reading in cost and performance measurements from previous architecture designs and code compilations to decide on the next high level machine specification to explore. The high level machine specification includes the number of clusters and the generic FUs within the system (i.e. the number of integer, float, memory and branch units per cluster (IFMB)). By limiting the spacewalker to these decisions, this narrows down the once infeasible design space to a more manageable one.

Given this high-level machine specification, the goal then is to trade off hardware cost and application performance to create a low-cost, high-performance machine for the given number of clusters and FUs. This preliminary work focuses on the cost-sensitive operation partitioner, whose job is to define the functionality of the clusters in both bitwidth and specific opcode repertoire required for each FU. The inputs to the operation partitioner are each basic block from the application itself and the abstract machine specification from the spacewalker.

Since each basic block in an application could potentially create a totally different cluster FU configuration, this causes a problem in deciding which of the machine configurations to actually use for the architecture. Thus, the separate multicluster datapath configurations are all sent to the unionizing step, which combines the features of all the designed datapaths. Using profile information from the original application, the functionality pruning step begins removing parts of the specified configurations based on their importance for the performance of the entire application.

After the pruning completes, what is left is the final multicluster architecture for the application. This can then be used to find the definite cost for the architecture. In addition, a machine description is then extracted from the final architecture and passed to our retargetable compiler which can compile for the final machine and return a performance metric. The cost and performance estimates are then used

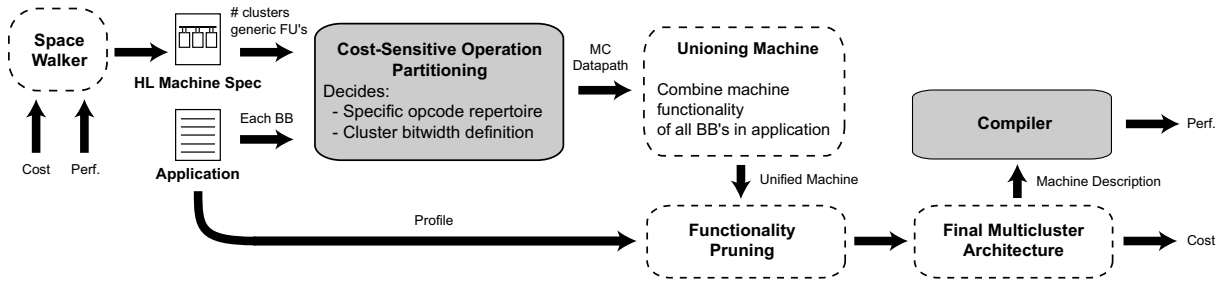

Figure 2: Our proposed system for a multicluster datapath architecture synthesis system

by the spacewalker to decide on how to vary the number of clusters and generic FUs within the system for the next iteration.

Thus, this entire system requires many distinct parts to fully create a multicluster datapath architecture. This paper will focus on the first step: given a high level machine specification and an application, partition operations in a way where the hardware synthesis can take advantage of the defined clusters to improve cost from both reducing bitwidth and FU capabilities.

2.3 Related Work

Datapath synthesis in a cost-effective manner has been widely explored in previous work, but most have focused on single basic blocks and single cluster systems. Automatic datapath synthesis has been studied by Cathedral-III [15], a complete synthesis system developed at IMEC, which designs a dedicated datapath for DSP applications based on their signal flow. Paulin and Knight [17] also proposed a technique for ASIC datapath design. Their force-directed scheduling technique integrated FU resource allocation and scheduling into a cost-minimizing synthesis algorithm. The Sehwa design system [16] automatically designed processing pipelines based on behavioral specifications. Marwedel [13] studied a technique to allow the use of common hardware to treat expressions with related semantics. VLIW synthesis has also been previously investigated by PICO [1], HP Labs' Program-In Chip-Out system for designing custom VLIW processors for specific applications. Design space exploration of clustered VLIW datapath was studied by Lapinskii et al. [10]. They use clock rate and power dissipation as their figures of merit for exploring the design space, and varied maximum cluster capacity, number of cluster and interconnect capacity.

There has also been prior work in bitwidth sensitive datapath synthesis. Valen-C [19] is a method for augmenting the C language in a way to convey bitwidth information for datapath synthesis. Their main focus was creating cost effective ASICs.

Partitioning operations for a multicluster architecture is also related to our work. The most well-known clustering algorithm was the Bottom-Up Greedy (BUG) algorithm [5]. BUG is recurses depth-first along the DFG, critical paths first, greedily assigning operations to clusters based on estimates of when it can schedule the operation the earliest. The most similar clustering work to ours focuses on using graph partitioning methods to decide on cluster assignment. Aletà et al. [2] proposed a similar multilevel graph partitioner, but mainly worked on determining the optimal initiation interval (II) for a modulo-scheduled loop using a pseudo-scheduler. Capitanio et al. [3] introduced a graph partitioning-based method for clustering using iterative operation swapping in order to improve partitions. While there has been a rich history of previous work on clustering algorithms, all have been partitioning operations in a performance-based manner and have clustered toward fixed machine specifications.

3. COST-SENSITIVE PARTITIONING

Cost-sensitive partitioning of operations for a multicluster architecture requires an estimate for the cost of a given clustering and a heuristic for balancing the tradeoff between performance and cost. In this section, we first describe a simple estimate to compute the cost of a cluster. Then, we introduce a code segment which we will use throughout the rest of this section as a running example. Finally, we present the details of our cost-sensitive partitioning algorithm.

3.1 Cluster Cost Model

In order to cluster for minimized cost, we created a cost estimate which tries to gauge the minimal cost required to support the partitioned operations on the cluster. If there were only one FU of each type (IFMB) within the cluster, then calculating the cluster cost would be rather trivial. In this case, if a cluster required a certain opcode to be executed, then that one FU in the system must support it. Calculating the cost when more FUs are inserted into the cluster becomes a much more difficult task. For example, if two FUs of each type are in the cluster, then a given opcode does not need to be supported on both. However, if there is a significant number of those opcodes assigned to that cluster, then it may be preferable to have it supported on both units in order to achieve higher performance.

In order to estimate the minimal cost of the cluster assignments, a greedy FU assignment heuristic was used. For the purposes of this example, we will only consider the integer FUs. Each FU available in a cluster is considered as a bucket which we continually fill with opcodes that can execute on it. The FUs can support a maximum of either total number of ops divided by the number of FUs or the critical path length, whichever is larger. The FUs begin in an empty state: they can support no opcodes and have no cost. Every operation in the cluster is then sorted in order from highest to lowest inherent cost (the cost to implement the opcode on a dedicated FU). One by one, the operations are placed into an FU, adding its opcode to the opcode repertoire of the FU and thus increasing the FU cost. The FU chosen to place the operation on is the one whose placement increases the total cost of the cluster the least.

Figure 3 shows an example cluster cost calculation. As-

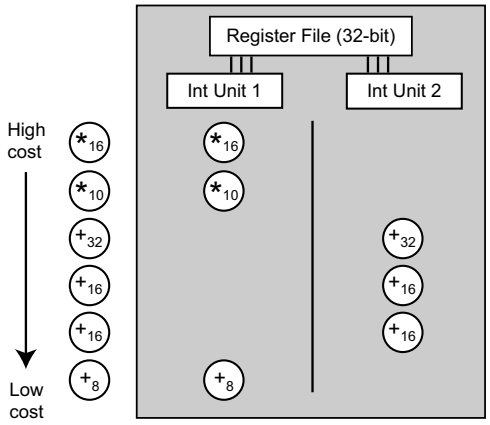

Figure 3: An example of our cost model calculation

sume the operations assigned to cluster 1 are ordered from highest to lowest cost as shown on the left side of the figure. The cost calculation begins by placing the 16-bit multiply. Since both integer units begin with no functionality, it doesn't matter which one is given the multiply; assume the multiply is placed on unit 1. Next, a 10-bit multiply could be placed for no additional cost on integer unit 1, and would require an entire multiplier to be added to unit 2. Therefore, our cost model greedily chooses unit 1.

Continuing the example, when the 32-bit add is added to the system, this requires the integer unit 1 to increase to a 32-bit unit, also increasing the cost of the 16-bit multiplier already there. Thus, it is cheaper to simply place the 32-bit adder on integer unit 2. The next two 16-bit adds are placed on integer unit 2 as they do not increase cost. The final operation, an 8-bit add, could be placed for no cost on integer unit 2, but this surpasses our restriction of maximum allowable operations on a unit. The 8-bit add is then placed on integer unit 1, increasing its cost by including an adder in its opcode repertoire. Since the maximum bitwidth of an operation within this cluster is 32-bit, a 32-bit register file is then included in the cost calculation. The ultimate cost of this cluster is therefore the gate cost of a 32-bit register file, a 16-bit FU that implements multiply and add, and a 32-bit FU that implements add.

Thus, this greedy placement of FU abilities is an estimate for the minimal cost of a cluster. It doesn't fall into traps overestimating required costs of a cluster. If there are significant numbers of a certain operation existing within the application, the cost model will begin balancing workload across the different FUs.

3.2 Example Program

Figure 4 is an example program that will be used throughout the rest of this section. Although rather trivial, it presents a good example of how cost-sensitive partitioning works. In this example, one large loop continually runs through several arrays, multiplying values and storing them in another array. The loop iterates over lines 7 and 8; line 7 only multiplies 8-bit values while line 8 only multiplies 32-bit values.

This program has two main flows of computation, one for the 8-bit values and one for the 32-bit values, as indicated in its data-flow graph (DFG) in Figure 5. In addition, it has a

```
 1:  char a[100], b[100], c[100], d[100];
 2:  int a1[100], b1[100], c1[100], d1[100];
 3:
 4:  main() {
 5:    int i;
 6:    for(i = 0; i < 100; i++) {
 7:      d[i] = a[i] * b[i] * 7 * c[i];
 8:      d1[i] = a1[i] * b1[i] * 7 * c1[i];
 9:    }
10:  }
```

Figure 4: An example program

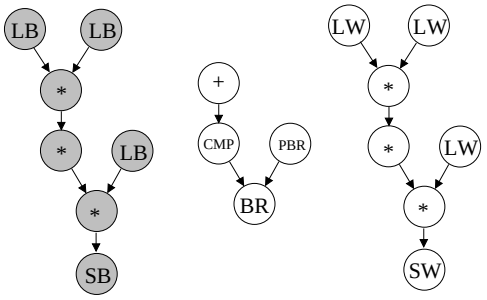

Figure 5: DFG for the example program

loop counter increment and a comparison check to possibly branch out of the loop. Most clustering algorithms would correctly see that the two main computation graphs should be placed on separate clusters, as they are large contiguous segments of dependent operations with no communication between them. Since the loop increment is not on the critical path, many algorithms would simply split this computation for the purposes of balancing workload; for example, they might place the add and compare on one cluster and the branch on the other. Furthermore, algorithms which do keep the loop counter increment code together on the same cluster have no preference for one cluster over the other, as again it saves no communication to place it in either cluster. Obviously, by placing this code with the 32-bit DFG, an 8-bit cluster and a 32-bit cluster could then be formed, rather than having two full 32-bit clusters.

3.3 Cost-Sensitive Partitioning Algorithm

Our cost-sensitive operation partitioning is based off our previous work on a Region-based Hierarchical Operation Partitioning (RHOP) algorithm [4], which is a performance-based operation partitioning algorithm for clustering. In this section, the traditional performance-centric RHOP is first discussed. Then we will detail the extensions for making the algorithm cost-sensitive.

3.3.1 Traditional RHOP

Traditional RHOP is completely performance-centric; it iteratively improves a given partition by placing operations in a manner which either minimizes intercluster communication, balances workload or both. To achieve this goal, RHOP employs a multilevel graph partitioner [8] which has two main phases: **coarsening** and **refinement**. The purpose of coarsening is to group likely related operations together so they can be considered as a unit to be placed in a cluster. Coarsening consists of many stages of pairing

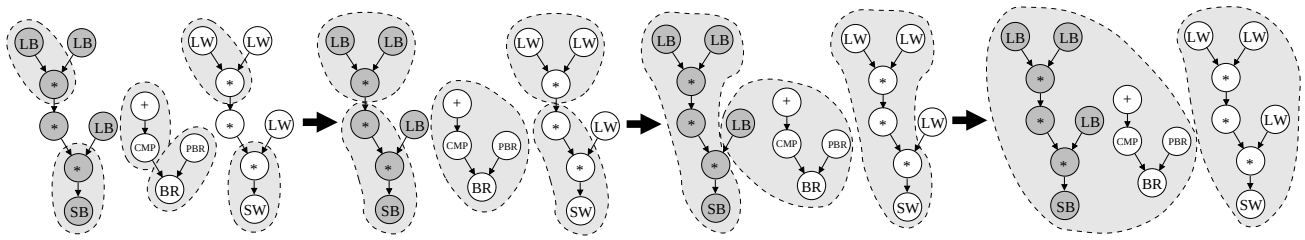

Figure 6: Stages in the coarsening process

operations or coarse operations together into single units. When coarsening completes, the refinement process begins. Refinement moves backward through the coarsening stages, uncoarsening operations and at each point considering the existing coarse operations for moving across clusters. The process of refinement uses a modified Kernighan-Lin algorithm [9] to decide whether or not to make a move.

During coarsening, operations are continually grouped together in pairs forming coarse operations until they reach the point of having the same number of coarse operations as clusters. The purpose of coarsening is by grouping operations together, the partitioner can then be forced to consider them as a single unit, potentially increasing the likelihood that dependent operations end up on the same cluster. The process of coarsening on our running example is shown in Figure 6. Each stage of coarsening pairs up operations or coarse operations at most once, which is why not all operations are coarsened in every step. This resulting set of coarsened operations is considered a snapshot of the coarsened states. Then, the process repeats until the resulting number of coarse operations equals the number of clusters.

Operations are coarsened together in a heuristic manner, based on **edge weights**. RHOP uses a slack-distribution method to assign edge weights to dependence edges between operations. Thus, every edge on the critical path would have the maximal weight, because breaking that edge across clusters would force an intercluster move that increases the critical path length. The remaining edges are given smaller edge weights depending on how likely breaking the edge would increase the resultant schedule length.

The refinement stage then iteratively uncoarsens backward across each of the coarsening snapshots in Figure 6 and tries moving coarsened operations across clusters to improve performance. At each snapshot, each coarsened operation must be considered for movement as a single unit. Thus, the refinement process will begin with very few choices for movable operations, since every operation has been coarsened, and continue until every operation is back in the uncoarsened state. In the final state, every operation is then a candidate for a possible move, and is likely grouped together on a cluster with operations with which it has a high affinity.

Therefore, at each of these coarsened snapshots, a decision needs to be made as to the benefits of a move. First, each operation is given a **node weight**, which is an account of how much each operation is likely to affect the load of the FUs in the cluster. In order to decide whether a move is beneficial, the **system load**, a heuristic to decide how overloaded clusters are, is formed. This heuristic creates a cycle-by-cycle account of where each operation is likely to be placed (formed from its scheduling range, the time between its earliest start time, **estart** and latest start time, **lstart**), and distributes the node weight of the operation across that those cycles. Thus, by summing the loads of the operations on the cluster, an estimate can be created for how overloaded a cluster is. Given this metric and an estimate for the cost of the edges merged or cut, a decision is made whether or not a single coarse operation should be moved. This process continues through the rest of the uncoarsening stages until every operation has been uncoarsened.

3.3.2 Cost-Sensitive Coarsening Stage

The coarsening stage for a cost-sensitive operation partitioner provides three interesting possibilities. First, operations can continue to be coarsened as they currently are in a performance-centric mindset. Second, operations could be coarsened in a cost-centric mindset. Finally, a hybrid scheme which combines the effects of both cost and performance could be formed.

The performance-based coarsening has the benefit of normally grouping operations in a manner which will reduce overall schedule length. Since operations are grouped irregardless of cost, it is possible that high-cost and low-cost operations are initially coarsened and thus not separated and independently considered for refinement until too late in the refinement process. A cost-based coarsening also has its drawbacks. Grouping operations based on how costly an FU is to execute them could potentially ignore the criticality of dependence edges and thus result in poor performance. A hybrid scheme for coarsening both cost and performance requires a new formulation to estimate the benefits of a cost-based coarsening versus a performance-based coarsening.

For this study, we focused on a performance-based method for coarsening. This decision was made because, in general, operations with similar-sized bitwidth are dependent on one another, and thus have a good chance of being coarsened with one-another. In addition, our goal is to create a low-cost multi-cluster datapath which still performs at a high level; a high degree of performance loss was unacceptable.

3.3.3 Cost-Sensitive Refinement Stage

Once the uncoarsening phase begins, refinements are made to the initial partition to improve the system load of the cluster and thus the estimate of performance impact. Currently, the RHOP algorithm already has a good metric for estimating the schedule impact of a proposed move, called the system load (SL). The system load is an estimate for the number of cycles over the critical path length (CPL) that a given cluster is. In addition, an *edge cut weight* is available which is the sum of the weight of the cut edges in the current clustering. Therefore, we have an estimate of the cycle increase because of both SL and intercluster communication (*edge cut weight*). Thus, we define our performance

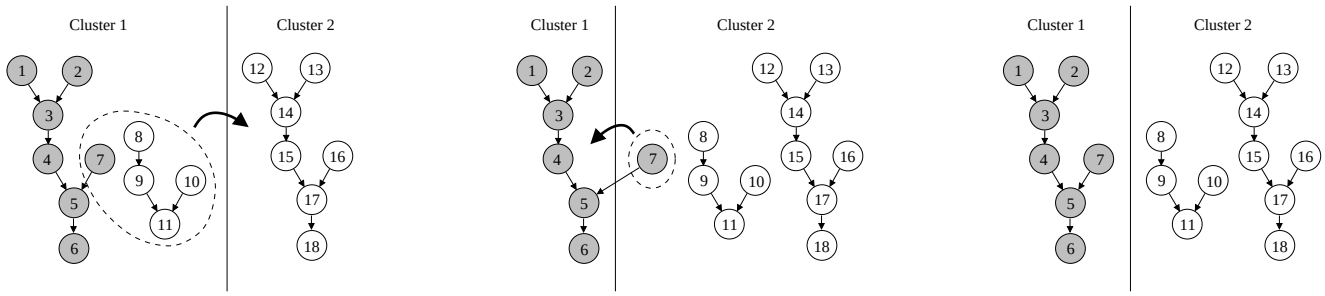

Figure 7: Uncoarsening and cost-sensitive refinement of our example program. Gray shading indicates 8-bit operations while white operations are 32-bit. (a) The original partition and the first move made. (b) A move made back for load balancing. (c) The final partition.

estimate, *est_cycles*, as:

$$\textbf{est\_cycles} = CPL + SL + edge\ cut\ weight$$

From the cost model described in Section 3.1, another estimate can be made for the cost of a specific cluster placement for operations. Thus, by balancing these cost/performance estimates, a low-cost multicluster machine with good performance can be designed.

Therefore, for every proposed move, the original cost of the clustering before the move, $cost_o$, and the original performance, est_cycles_o can be computed. Similarly, estimates can be made for the cost and performance of the clustering after the proposed move is made, $cost_n$ and est_cycles_n, respectively. These values are computed for each proposed move, and are used to determine whether or not a move is considered a good move.

Given the current clustering, the first step is to find all free moves, which are moves that decrease cost at either the same or higher performance, or increase performance and the same or lower cost point. Since these moves are considered positive in both respects, they are immediately moved to the other cluster; they cannot hurt either cost or performance. After these free moves are made, the remaining operations are considered for movement across clusters.

The remaining coarsened operations fall into one of three categories: the move decreases cost but lowers performance; the move raises performance but increases cost; or the move both increases cost and lowers performance. Obviously, potential moves which fall into the third category are bad choices for moves. For all other possible moves, a **move benefit** is calculated as shown in Equation 1.

$$\textbf{benefit} = \frac{1}{cost_n \times est\_cycles_n} - \frac{1}{cost_o \times est\_cycles_o} \quad (1)$$

This benefit metric is used to identify the point of diminishing returns on cost reduction as performance is decreased. Therefore, this estimate for the clustering compares the previous cost/performance of the clustering to the new cost/performance. If this value is positive, then the move is considered a good move to make.

One final check before a move is actually made is to make sure the overall performance remained within a reasonable estimated schedule length from the original performance-based RHOP operation assignment. We varied the allowed performance decrease from 5% to 40%.

3.3.4 *Refinement Example*

Returning to our example from Section 3.2, assume the initial partitioner had coarsened and initially placed the operations as shown in Figure 7(a). Each step in the refinement stage is labeled with the current cost and performance estimate for the clustering. After this first uncoarsening state, there are three coarsened operations: one with operations 1 through 6, one with operations 7 through 11, and one with operations 12 through 18. The refinement stage begins by considering operations for movement. Since cluster 1 is the more heavily loaded cluster, it is first examined for moving operations. Moving the coarse operation containing operations 1 through 6 would significantly impact the performance for the clusters, as there would be too many operations conflicting for resources at the same time. However, this move would somewhat improve costs by removing all the costly multiplies from cluster 1.

On the other hand, by moving the encircled coarse op in Figure 7(a), the increase in performance isn't extremely large, and it allows cluster 1 to decrease to an entirely 8-bit cluster, a large cost savings. Thus, the move of this coarsened operation is made, as it creates a positive gain in using Equation 1. No more moves at this level of coarsening are made, as the performance penalty would be far too high. The resulting assignment is as shown in Figure 7(b). Thus, this move cause a slight increase in the performance estimate, from 7 to 8 cycles, but was able to drop the cost estimate from 28114.4 to 15065.6.

The next stage of uncoarsening then occurs in Figure 7(b), which separates operation 7 from 8 through 11. Just as before, the interesting uncoarsened op is encircled. In this case, moving operation 7 from cluster 2 to cluster 1 increases its performance, by merging the edge between operations 5 and 7, and also helps to balance out the workload of cluster 2, as one less operation is required to execute there. At the same time, this move doesn't increase the cost of this cluster because the load-byte opcode is already supported in this cluster by operations 1 and 2. Thus, the move of operation 7 is considered a free move and is made. After this move, no more cost/performance improving moves are made and the final partition is as shown in Figure 7(c). This partition has created one 32-bit cluster and one 8-bit cluster.

4. EXPERIMENTS

We implemented our cost-sensitive operation partitioner using the Trimaran tool set [18], a retargetable compiler for VLIW processors. Gate cost estimates were computed using the Synopsys design tools and a popular 0.18μ standard cell library. For each opcode supported by the system, a width-parameterized cost formula was created by synthesizing a series of hardware components to implement the opcode and fitting a curve to the reported cost. Bitwidth information was gathered by propagating the required widths for literals and C variables types, as previously discussed in [12].

Since this work is focused primarily on partitioning operations in a cost-sensitive manner, we ran the most frequently executed loop of several kernels, four of the MediaBench [11] benchmarks, blowfish of the MiBench [7] benchmark suite and crc and url from the NetBench benchmark suite[14]. Only a single loop was run because our current system is incomplete; however, when a final multicluster datapath architecture is created for these applications, the loop will be the dominant portion of the application.

We compared the performance and cost of the hardware needed for clusters created by the traditional RHOP algorithm to that of a cost-sensitive model. Table 1 shows the results of these experiments on both a 2-cluster machine and a 4-cluster machine. The high level machine specification for these machines was fixed with the following generic FUs: 2-integer, 1-floating point, 1-memory and 1-branch unit per cluster. In addition, Table 1 contains percentages for the source of the 2-cluster cost savings for each benchmark, either from specialization of the opcode repertoire per FU or by pruning the required bitwidth of the clusters.

Overall, the cost-sensitive operation partitioner was able to reduce the total cost of the 2-cluster machine by an average of 20.4%, while sacrificing an average of 5.4% in performance. For a 4-cluster machine, the cost savings were 28.0% at a performance decrease of 2.5%. Several interesting results appear; for example, in six of the benchmarks for the 2-cluster machine, no performance decrease occurred at all. In these cases, the cost-unaware traditional RHOP could easily have clustered the operations in a more cost-effective manner, but failed to do so. In the 4-cluster machine, many benchmarks were able to have a much higher gate-cost reduction, generally because additional clusters increase the total cost required for the machine, so intelligent opcode repertoire specialization becomes more important. However, in some benchmarks, the partition choices for distributing operations across four clusters was limited by low inherent parallelism. In these cases, the cost of the hardware created by traditional RHOP was already fairly low, so a cost-sensitive approach was not able to lower cost significantly.

Both opcode repertoire and bitwidth cost savings were prevalent in the benchmarks. Several of the benchmarks had 100.0% of their cost savings from improving the FU opcode repertoires. In these cases, the benchmark had most, if not all, of their operations as 32-bit wide, limiting the amount of available bitwidth savings. When benchmarks had a large variation in bitwidths, such as channel, dct and fsed, the cost-sensitive algorithm was able to intelligently use the bitwidth information to save cost.

Figure 8 shows a pareto chart of the possible cluster assignments considered by the cost-sensitive partitioner on the fsed kernel on the 2-cluster machine. Each point on the chart

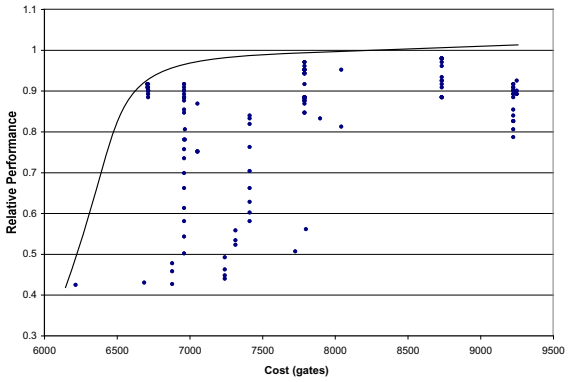

Figure 8: Pareto chart for the fsed kernel

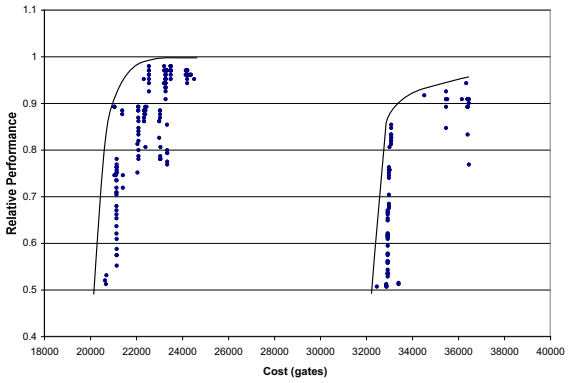

Figure 9: Pareto chart for the LU kernel

indicates the normalized estimated schedule length and cost for a given clustering that was chosen during the run of the partitioner. The line indicates the pareto-optimal designs, which are the best performance possible at a given cost and vice versa. As expected, better performance normally came at the expense of much higher cost. The vertical bands of points in the chart appear when the RHOP partitioner first explores the lowest point of the band, where performance is very low, and begins improving performance by moving operations across clusters.

Figure 9 is a pareto chart of cluster assignments for the LU kernel on the 2-cluster machine. In this example, the algorithm produced two pareto curves while determining the cluster assignment. Such behavior occurred in several of the benchmarks and was generally caused by an expensive unit, such as a multiplier, being required by the application. The cost-sensitive algorithm typically begins at the higher cost pareto curve, when both clusters support the expensive operation. During the design space exploration, the algorithm reaches the point where it shifts all the expensive operations to one cluster, thus forming the lower pareto curve.

5. CONCLUSION AND FUTURE WORK

This paper proposes a preliminary study on the automated application-specific design of a multicluster datapath architecture. Our vision of a complete system for custom multicluster architecture synthesis is introduced. We

	2-cluster 2111(IFMB)		4-cluster 2111(IFMB)		2-C Savings Breakdown	
Benchmark	**Performance Decrease**	**Gate Cost Reduction**	**Performance Decrease**	**Gate Cost Reduction**	**Opcode Repertoire**	**Bitwidth Reduction**
channel	0.0%	17.5%	0.0%	25.9%	66.3%	33.7%
dct	3.4%	4.9%	6.6%	3.0%	54.9%	45.1%
fft	11.8%	38.2%	0.0%	58.4%	100.0%	0.0%
fsed	6.7%	36.0%	6.7%	36.1%	46.1%	53.9%
huffman	2.0%	7.2%	0.0%	15.8%	100.0%	0.0%
LU	0.0%	13.3%	1.5%	57.2%	100.0%	0.0%
rls	13.7%	38.0%	0.0%	3.7%	100.0%	0.0%
rawcaudio	10.3%	0.7%	0.0%	32.9%	91.0%	8.9%
rawdaudio	0.0%	15.2%	0.0%	12.7%	92.6%	7.4%
gsmdecode	17.0%	36.3%	2.1%	58.2%	92.6%	7.4%
gsmencode	0.0%	38.6%	0.0%	59.8%	97.9%	2.1%
blowfish	10.5%	9.1%	9.0%	5.7%	94.3%	5.7%
crc	0.0%	5.8%	0.0%	19.7%	46.0%	54.0%
url	0.0%	25.3%	11.1%	2.9%	100.0%	0.0%
Average	5.4%	20.4%	2.5%	28.0%	84.4%	15.6%

Table 1: Performance and cost savings for kernels and benchmarks from MediaBench, MiBench and NetBench

present the first step toward this goal, a novel technique for partitioning operations into clusters with a cost-sensitive mindset. Our algorithm is presented with a set number of clusters and generic function units per cluster and then progressively tailors the opcode repertoire and bitwidth of each function unit in each cluster to an application.

We compared our results to a similar performance-centric partitioning algorithm and found that for many benchmarks, large wins in cost savings were attainable at very reasonable performance degradation. On average, a cost savings of 20.4% was achieved at with only a schedule length increase of 5.4% for a two-cluster machine.

In the future, we plan to continue to work toward our final goal of a complete multicluster datapath architecture synthesis system. This includes setting up the initial system to create a unified machine from the several different machine designs created for each basic block. Then, using the application profile, an intelligent pruning method will be created to remove unnecessary functionality to improve cost.

6. ACKNOWLEDGMENTS

We thank the anonymous referees for their comments and suggestions. This research was supported in part by the DARPA/MARCO C2S2 Research Center, National Science Foundation grant number CCR-0325898 and equipment donated by Intel Corporation.

7. REFERENCES

[1] S. Aditya and B. R. Rau. Automatic architecture synthesis and compiler retargeting for VLIW and EPIC processors. Technical Report HPL-1999-93, HP Laboratories, 1999.

[2] A. Aletà et al. Exploiting pseudo-schedules to guide data dependence graph partitioning. In *PACT 2002*, Sept. 2002.

[3] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *MICRO-25*, pages 103–114, Dec. 1–4, 1992.

[4] M. Chu, K. Fan, and S. Mahlke. Region-based Hierarchical Operation Partitioning for Multicluster Processors. In *PLDI '03*, pages 300–312, Jun. 2003.

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

[6] K. Farkas et al. The multicluster architecture: Reducing cycle time through partitioning. In *MICRO-30*, Dec. 1997.

[7] M. R. Guthaus et al. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Dec. 2001.

[8] B. Hendrickson and R. Leland. A multi-level algorithm for partitioning graphs. In *Supercomputing*, 1995.

[9] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, Feb. 1970.

[10] V. S. Lapinskii, M. F. Jacome, and G. A. de Vaciana. Application-specific clustered VLIW datapaths: Early exploration on a parameterized design space111. *IEEE TCAD*, 21(8):889–903, 2002.

[11] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *MICRO-30*, Dec. 1997.

[12] S. Mahlke et al. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *IEEE TCAD*.

[13] P. Marwedel. Matching system and component behavior in MIMOLA synthesis tools. In *EuroDAC*, Mar. 1990.

[14] G. Memik, W. H. Mangione-Smith, and W. Hu. NetBench: A benchmarking suite for network processors. In *ICCAD*, pages 39–42, 2001.

[15] S. Note et al. Cathedral-III: Architecture-driven high-level synthesis for high throughput DSP applications. In *DAC-28*.

[16] N. Park and A. Parker. Sehwa: A software package for synthesis of pipelines from behavioral specifications. *IEEE TCAD*, 7:356–370, 1988.

[17] P. Paulin and J. Knight. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE TCAD*, 8:661–679, 1989.

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

[19] H. Yasuura et al. Embedded system design using soft-core processor and Valen-C. *IIS Journal of Information Science and Engineering*, 14:587–603, 1998.