Cost-Sensitive Partitioning in an Architecture Synthesis System for Multicluster Processors

THIS HIERARCHICAL SYSTEM AUTOMATICALLY DESIGNS HIGHLY CUSTOMIZED MULTICLUSTER PROCESSORS. IN THE FIRST OF TWO TIGHTLY COUPLED COMPONENTS, DESIGN SPACE EXPLORATION HEURISTICALLY SEARCHES THE BASIC CAPABILITIES THAT DEFINE THE PROCESSOR'S OVERALL PARALLELISM. IN THE SECOND, A HARDWARE COMPILER DETERMINES THE DETAILED ARCHITECTURE CONFIGURATION THAT REALIZES THE PARALLELISM.

Michael L. Chu
Kevin C. Fan
Rajiv A. Ravindran
Scott A. Mahlke
University of Michigan, Ann Arbor

Many portable devices must perform computationally demanding processing of images, sound, video, or packet streams. Current embedded processors cannot meet these devices' performance requirements under tight power and cost constraints. Therefore, hardware designers must create a new generation of processors that provide far greater performance and flexibility while using less power and space.

Application-specific instruction processors (ASIPs) have great potential to meet the challenging demands of pervasive systems. A custom-designed processor provides highly specialized computation capabilities to meet an application's specific needs. With high degrees of specialization, ASIPs can achieve design wins of an order of magnitude in terms of power, cost, or performance.

Although ASIPs have great potential, building them entails two critical issues. First, like most modern processors, ASIPs are both time-consuming and costly to build. Their complexities require large design teams. As a result, their use is generally limited to the largest market segments, where sales volume can offset the cost of a large design team. The second issue is that application-specific customization has traditionally resulted in a loss of programmability. Compilers are ineffective for ASIPs because of irregular designs and highly specialized data and control paths. Therefore, hand-coded assembly is often necessary, particularly for performance-critical portions of applications.

The key to simultaneously addressing the critical issues of cost, design time, and programmability is automation. Reducing the human involvement necessary in the design of specialized hardware platforms can enable

ASIPs to proliferate across both large and small markets. With stylized architecture templates, automation can also ensure programmability by focusing on more systematic customization strategies.

The traditional approach to customizing an ASIP for a particular application is design space exploration (DSE). DSE is a heuristic-guided search across a parameterized processor architecture.[1-4] Parameters include number and type of function units, register file sizes and ports, cache sizes and associativities, and so on. At each step in the search, the DSE engine uses cost and performance feedback from previous designs to select a new candidate design. Next, the system synthesizes the design to estimate the cost and compiles the application to produce an estimate of its performance. DSE often starts at an extreme of the design space: the most expensive machine (highest performance) or the cheapest machine (lowest performance). DSE's objective is to identify the Pareto-optimal set of designs for the target application while visiting as few suboptimal points as possible.

To control design space size and facilitate a reasonable search, traditional DSE limits the number of architectural parameters. However, this approach is undesirable because it overly constrains the possible designs and thus the possible successful design wins achievable through customization. We believe all aspects of the processor architecture and microarchitecture, including the data path, control path, instruction set, and memory subsystem, must be specialized to push ASIPs to the next level of performance, cost, and power efficiency. Unfortunately, traditional DSE does not generalize. The design space becomes enormous and thus too unwieldy to search heuristically. Furthermore, parameters such as the instruction set, instruction encoding, and data path width are not amenable to search. Designers must determine such parameters more algorithmically, by analyzing the target application.

Our approach combines traditional DSE and compiler-directed architecture synthesis. Hierarchically dividing the design space reduces the search space without artificially constraining the degree of specialization permitted for any aspect. The search covers basic capabilities that define the processor's overall parallelism, and the compiler determines the detailed architecture that realizes the parallelism. This article focuses on the latter topic—compiler-directed architecture synthesis. More specifically, we examine compiler-directed synthesis of an ASIP's data path architecture. Sophisticated dependence, control flow, and dataflow analysis techniques permit the discovery of an application's computation and communication structure. The compiler can exploit the discovered structure and transform the application to create high-performance, cost-effective ASIPs.

Related work

Cost-effective data path synthesis has been widely explored in the literature. Here, we provide just a sample of the most relevant works. Most researchers have focused on single basic blocks and single-cluster systems. Automatic data path synthesis is the focus of Cathedral-III.[5] This complete synthesis system, developed at IMEC Laboratory in Belgium, designs a dedicated data path for digital signal processing (DSP) applications on the basis of their signal flow. Paulin and Knight also propose a technique for application-specific integrated circuit data path design.[6] Their force-directed scheduling technique integrates function unit (FU) resource allocation and scheduling into a cost-minimizing synthesis algorithm. The Sehwa design system automatically designs processing pipelines on the basis of behavioral specifications.[7] Very long instruction word (VLIW) synthesis is the target of PICO, HP Labs' Program-In, Chip-Out system for designing application-specific custom VLIW processors.[3] Lapinskii et al. propose design space exploration of clustered VLIW data paths.[4] They use clock rate and power dissipation as their DSE figures of merit, varying maximum cluster capacity, number of clusters, and interconnect capacity.

Multicluster architectures

An underlying processor model for an application-specific design is essential for achieving a design that provides high performance in a cost-effective manner. A multicluster VLIW processor model is the focus of our work. Such processors are attractive for embedded computing because of their relatively low cost and ability to exploit high levels of instruction-level parallelism (ILP). The critical problem with scaling VLIW processors is the centralized

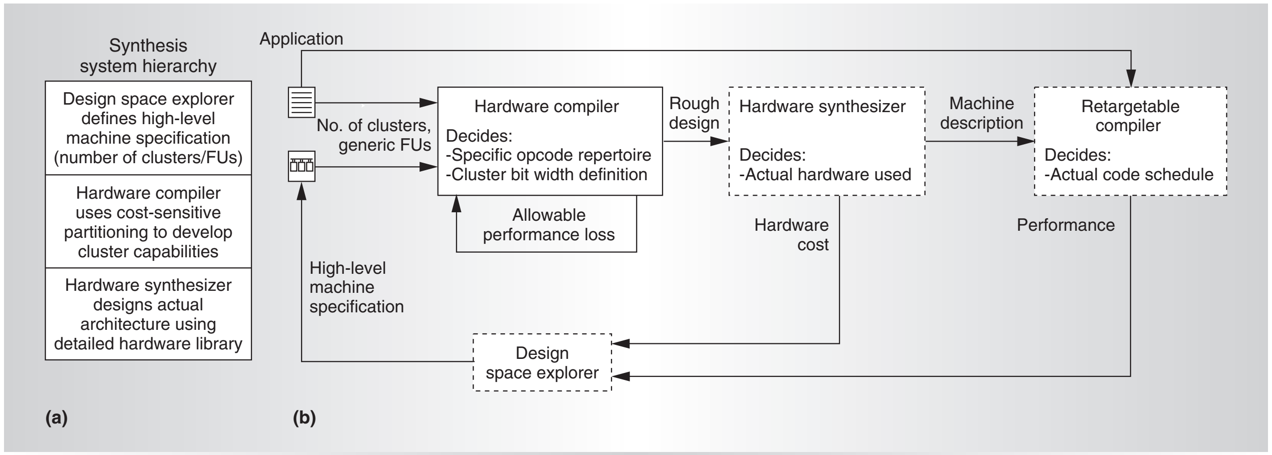

Synthesis system hierarchy
Design space explorer defines high-level machine specification (number of clusters/FUs)
Hardware compiler uses cost-sensitive partitioning to develop cluster capabilities
Hardware synthesizer designs actual architecture using detailed hardware library
(a)
Application
No. of clusters, generic FUs
Hardware compiler
Decides:
-Specific opcode repertoire
-Cluster bit width definition
Allowable performance loss
Rough design
Hardware synthesizer
Decides:
-Actual hardware used
Machine description
Retargetable compiler
Decides:
-Actual code schedule
Hardware cost
Performance
High-level machine specification
Design space explorer
(b)

Figure 1. Hierarchical synthesis system for multicluster architectures: hierarchy (a), synthesis system flow diagram (b).

register file. As the issue width grows, both cost and access time scale quadratically with the number of register ports. The multicluster architecture addresses this growing problem and reduces the access time penalty by splitting the register file into smaller files and assigning FU subsets to each file.[8] These smaller, decentralized register files are efficient to design, and they alleviate the register file bottleneck while maintaining the desired ILP. The idea of decentralized register files can also extend to instruction and data caches to produce a scalable architectural platform.[9]

Compiler support is a major challenge to clustered architectures because the compiler must effectively partition operations across the available resources on each cluster to maximize utilization. Moreover, compilers must achieve this goal while carefully considering the implications of intercluster communication. Pairs of producer and dependent consumer operations placed on different clusters must communicate data across an intercluster communication network. This network is both slow and bandwidth limited, so the compiler must minimize the amount of intercluster communication.

Clustered architectures are either homogeneous or heterogeneous. Clusters in homogeneous designs are configured identically. Homogeneity simplifies compilation because the compiler can take advantage of the machine's symmetry to reduce the complexity of partitioning operations. In addition, application-specific homogeneous clusters are easier to design because the design architect needn't worry about the performance impact of altering each cluster's configuration.

Heterogeneous multicluster processors, however, permit greater design flexibility and less hardware redundancy when customized for a given set of applications. For example, FUs with high gate cost, such as multipliers or dividers, can exist in a subset of clusters in the machine, or a given cluster can support small bitwidth operations while other clusters support full-width operations. Thus, a heterogeneous multicluster architecture can be far superior to a homogeneous one in terms of cost, yet can execute a high-performance application. Therefore, we chose this model as our architecture template for ASIP design.

Hierarchical multicluster data path synthesis system

Figure 1 shows our hierarchical system for multicluster architecture synthesis. The three-level system first makes broad, high-level design decisions and then progressively refines the details. The idea of this process is to consider first-order cost factors at the outermost layer through search and then algorithmically determine the specific hardware realization.

In the first stage, DSE sits above the entire system, reading in cost and performance measurements from previous architecture designs to decide on the next high-level machine specification to explore. In our system, DSE works much like traditional DSE systems, iteratively searching across a wide range of parallelism specifications for a clustered machine. Therefore, it outputs high-level machine information such as the number of clusters, the data memory bandwidth, and the generic FUs within the system—that is, the number of integer, floating-point, memory, and branch (IFMB) units per cluster. We assume the generic FUs support all opcodes at full bitwidth and are fully interconnected with all other data path elements in the same cluster. Thus, these parameters determine the target machine's available parallelism. Limiting the design space explorer to these high-level decisions narrows the once-infeasible design space to a more manageable size.

The second level of the hierarchy is the hardware compiler, whose goal is to structure the resources within the DSE-specified high-level design. Naturally, the hardware compiler could design a multicluster data path such that each cluster supports the union of all possible opcodes in the application. Such a design, although performance friendly, would be extremely expensive in terms of gate cost and inefficient because of redundancy and the lack of enough parallelism in the program to take advantage of the additional resources. A cost-friendly approach that tries to maximize performance while taking into account the repercussions on hardware can produce a far more efficient application-specific architecture. Therefore, this design stage uses a cost-sensitive operation partitioner to divide the operations across the clusters defined in the high-level machine specification. The partitioner uses hardware cost estimates to determine a set of opcodes for assignment to each cluster, as well as the data path bitwidth. These two specifications (cluster capabilities) then pass to the third level of the design system.

The third and final level is the hardware synthesizer, which takes the capabilities and requirements from the previous phase and designs the actual multicluster hardware. The synthesizer selects from a hardware library the specific components that meet the resource and parallelism requirements from the previous levels and the interconnect needed to combine the components. From this design, the synthesizer produces more-precise cost measurements and a machine description for the retargetable compiler.

Thus, the full system takes the target application as input and continually generates new machines with cost and performance measurements. These hardware models evolve in a hierarchical mix of both traditional DSE techniques and heuristic search of possible design choices for a multicluster data path. From both the cost and performance perspectives, creating a highly customized architecture requires that all the details of a multicluster data path be defined, including

- number of clusters, FUs, and register files per cluster;
- opcode repertoire, bitwidth, and interconnectivity of the FUs; and
- width, ports, and number of entries per register file.

This article focuses on the second level of the hierarchy, which determines data path resource functionality (opcode repertoire and FU bitwidth).

Cost-sensitive operation partitioning

The primary goal of our synthesis system's hardware compiler stage is to appropriately balance hardware cost and performance while shaping and structuring the resources of the clustered data path. Intelligently partitioning operations in a multicluster architecture can significantly lower the hardware cost of data path resources. The main technique for specifying resource capabilities is to use an operation partitioner, which divides the operations in a dataflow graph (DFG) into separate groups and assigns them to clusters. The partitioner's per-cluster output is an assignment of the opcodes executable on each FU and each data path element's bitwidth.

There are many operation-partitioning approaches (see the "Other clustering algorithms" sidebar). Our synthesis system views this step as a graph-partitioning problem in which the nodes are the operations and the edges represent communication between them. Like most graph-partitioning algorithms, the system determines a set of possible choices for node partitions, calculates a figure of merit for each partition, and then chooses the most desirable partition on the basis of its value. The main difficulty is determining a figure of merit that can balance performance and hardware cost.

When partitioning operations, our synthesis

Other clustering algorithms

Traditionally, the operation-partitioning phase in a software compiler for multicluster architectures simply focuses on dividing operations efficiently across multiple clusters to maximize performance.[1-3] A significant amount of work exists on different techniques for partitioning operations; most of it uses schedule length as the metric for determining good partitions. That is, given two possible partition choices, the partition with the shorter schedule is selected.

The best-known clustering algorithm is the Bottom-Up Greedy (BUG) algorithm implemented in the Bulldog compiler.[2] BUG recurses depth-first along the dataflow graph, critical paths first, greedily assigning operations to clusters on the basis of estimates of the earliest possible operation scheduling. The clustering work most similar to ours uses graph-partitioning methods to decide on cluster assignment. Aletà et al. proposed a similar multilevel graph partitioner but worked mainly on determining the optimal initiation interval for a modulo-scheduled loop using a pseudoscheduler.[1] Özer et al. introduced a unified approach to consider the interrelated problems of clustering, scheduling, and register allocation.[3] The rich history of previous work on clustering algorithms has mostly focused on partitioning operations in a performance-based manner and has targeted fixed machine specifications.

References

1. A. Aletà et al., "Exploiting Pseudo-Schedules to Guide Data Dependence Graph Partitioning," *Proc. Int'l Conf. Parallel Architectures and Compilation* (PACT 02), IEEE Press, 2002, pp. 281-290.
2. J. Ellis, *Bulldog: A Compiler for VLIW Architectures*, MIT Press, 1985.
3. E. Özer et al., "Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures," *Proc. Int'l Symp. Microarchitecture* (MICRO 31), IEEE CS Press, 1998, pp. 308-315.

system must account for the hardware cost of design decisions. The resource hardware cost is a direct byproduct of the operations partitioned to the clusters. When we know how the operations are grouped, we also know the minimum required set of opcodes that each cluster must execute, as well as a minimum required bitwidth. For example, if a set of operations assigned to a cluster includes a 32-bit multiply operation, that cluster's resources must include a 32-bit multiplier. Acknowledging the cost of adding expensive FUs by trying to group all the expensive opcodes permits hardware cost savings. Similarly, grouping small-bitwidth operations to create an entirely narrow cluster makes bitwidth cost savings possible. Thus, our synthesis system adds an estimate of a candidate partition's hardware cost to the figure of merit.

The multilevel region-based hierarchical operation-partitioning (RHOP) algorithm proposes an initial partition of a DFG to a set of clusters and then continually improves the partition by moving operations between the assigned clusters.[10] Using its figure of merit as the factor determining whether moving an operation or a set of operations is beneficial, the algorithm decides whether the move is actually made. With a cost estimate added to the figure of merit, RHOP can consider cost-performance tradeoffs.

DFG coarsening

The algorithm's multilevel nature means it has two main phases: coarsening and refinement. Coarsening, which groups potentially related operations for consideration as a unit, consists of many stages of pairing operations. When coarsening completes, the algorithm creates an initial DFG partitioning and then refinement begins. Refinement moves backward through the coarsening stages and considers moving any coarsened operations that exist in those stages to different clusters. When the algorithm finds no more beneficial moves, it again uncoarsens the DFG.

Figure 2 demonstrates coarsening on a sample DFG. Grouping operations permits initial coarse-grained partition decisions, leaving fine-grained decisions to the end. Thus, refinement begins with very few choices for movable operations because every operation has been coarsened; refinement continues until every operation is back in the uncoarsened state. In the final state, every operation is a candidate for a possible move, and operations with a high affinity are grouped on the same cluster. Our coarsening stage uses edge weights on the graph to prioritize operation grouping. The edge weights result from estimates of the scheduling impact of splitting edges across clusters.

Refining the partition

After coarsening, the partitioner has the same number of partitions as clusters. Then, RHOP assigns each partition to a cluster and calculates both the initial performance metric and the cost metric. The initial partition's performance is simply the schedule length estimate of the operations assigned to the current clusters with generic FUs. The data path elements (FUs and register files) required to execute those operations constitute the hardware cost.

The refinement phase then iteratively works backward across each of the coarsening stages in Figure 2 and tries to move coarsened oper-

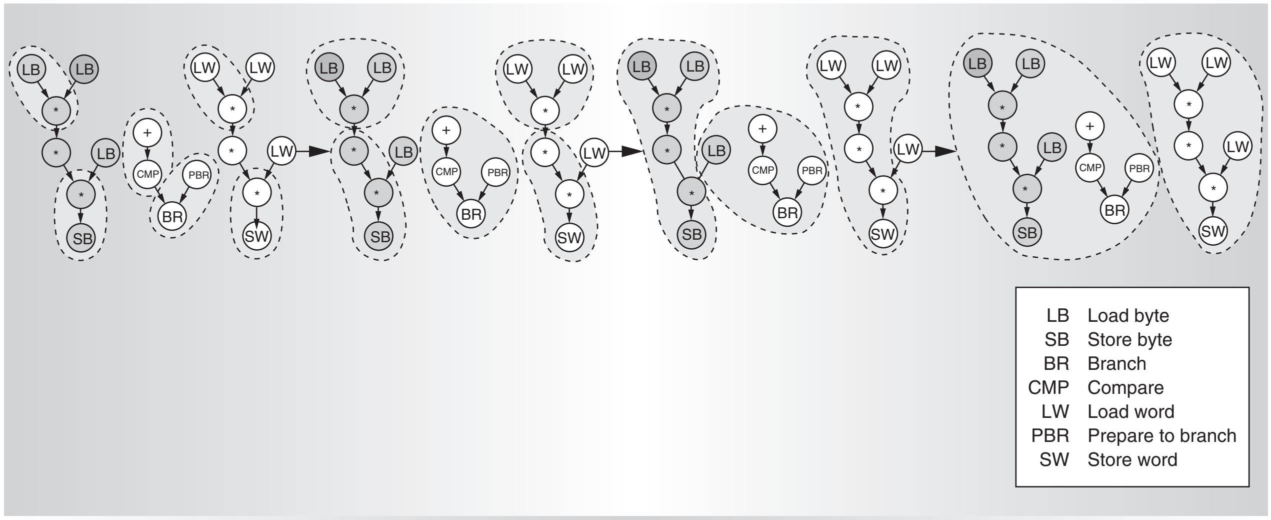


Figure 2. Coarsening on a sample dataflow graph. Shaded operations are 8 bits and nonshaded are 32 bits. Dashed lines represent the grouping of operations defined by each stage of coarsening.

ations across clusters to improve performance or cost. At each stage, the algorithm must consider moving coarsened operations as a single unit. While backtracking across these coarsened stages, the algorithm must decide how the proposed moves affect performance and hardware costs.

Two estimates determine the improvement a move can achieve: *est_cycles* for performance and *est_cost* for hardware cost. The traditional, performance-centered RHOP uses the *est_cycles* metric to determine a partition's schedule length. RHOP creates this metric by calculating node weights, which estimate the operation's resource utilization, and edge weights, which estimate the performance impact of cutting the edge across clusters. The node weights estimate the resource load factor, expressed as the number of cycles required to execute the operations. The edge weights estimate the number of extra cycles that intercluster communication requires to transfer values across the interconnection network. These two estimates combine to generate *est_cycles*, the total estimated cycles required for a given partition.

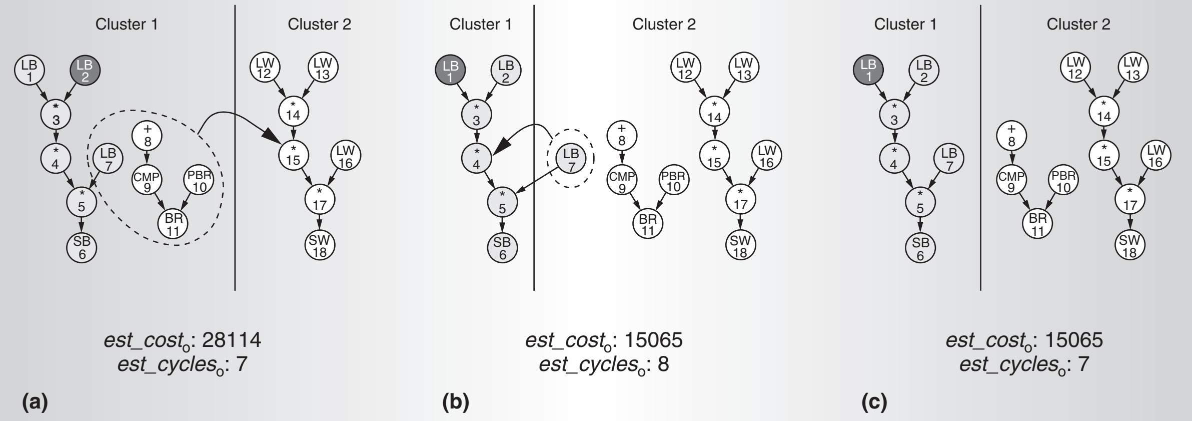
The hardware cost estimate metric, *est_cost*, approximates the number of gates needed to realize a multicluster data path that supports all the operations as currently partitioned. To compute this estimate, cost-sensitive RHOP uses gate cost estimates for the resources at the operations' required bitwidths. For every proposed move, the algorithm can compute the original cost of the clustering before the move, est_cost_o, and the original performance, est_cycles_o. Similarly, it can estimate the clustering's cost and performance after the proposed move, est_cost_n and est_cycles_n.

Given the partition, the first refinement step is to find all free moves—that is, moves that decrease cost at either the same or higher performance, or increase performance at the same or lower cost. These moves—positive in both respects—occur immediately; they cannot hurt either cost or performance. Then the algorithm considers the remaining operations for movement across clusters.

The remaining coarsened operations fall into one of three categories: the move decreases cost but lowers performance, raises performance but increases cost, or both increases cost and lowers performance. Obviously, potential moves in the third category are bad choices. For all other possible moves, the following equation shows the move benefit calculation:

$$benefit = \frac{1}{est\_cost_n \times est\_cycles_n} - \frac{1}{est\_cost_o \times est\_cycles_o}$$

This benefit metric identifies the point of diminishing returns on cost reduction as performance decreases. Therefore, this clustering



LB	Load byte
SB	Store byte
BR	Branch
CMP	Compare
LW	Load word
PBR	Prepare to branch
SW	Store word

Figure 3. Uncoarsening and cost-sensitive refinement: original partition and first move (a); backward move for load balancing (b); final partition (c). Shaded operations are 8 bits and nonshaded are 32 bits.

estimate compares the original cost-performance with the new cost-performance. A positive value indicates a good move.

One final check before making the move ensures that overall performance remains within a reasonable estimated schedule length from the original performance-based RHOP operation assignment.

We varied the allowed performance decrease from 5 percent to 40 percent. This gave the partitioner more freedom to make moves that produce a greater performance decrease for potentially larger hardware cost savings.

Refinement example

Returning to the earlier example (Figure 2), assume that the partitioner had coarsened and initially placed the operations as shown in Figure 3a. Each refinement step carries the clustering's current cost and performance estimate (each loop iteration's estimated schedule length). After this first uncoarsening stage, there are three groups of coarsened operations: one containing operations 1 through 6, one containing operations 7 through 11, and one containing operations 12 through 18. Refinement begins with the partitioner considering operations for movement. Because cluster 1 is more heavily loaded, it's the first candidate for moving operations. Moving the coarse operation group containing operations 1 through 6 would significantly affect cluster performance because too many operations would contend for resources at the same time. However, this move would reduce the cost somewhat by removing all the costly multiplies from cluster 1.

On the other hand, moving the coarse operation in the dashed circle in Figure 3a doesn't increase performance much, and it lets cluster 1 shrink to an entirely 8-bit cluster, producing a large cost savings. Therefore, the partitioner moves this coarsened operation because, according to the benefit calculation equation, the move creates a benefit. The performance penalty would be too high to make any more moves at this level of coarsening. Figure 3b shows the resulting assignment. This move causes a slight increase in the estimated execution time, from seven cycles to eight, but it drops the cost estimate from 28,114 to 15,065 gates.

Figure 3b shows the next uncoarsening stage, which separates operation 7 from operations 8 through 11. Again, the interesting uncoarsened operation appears in a dashed

Table 1. Effectiveness of cost-sensitive partitioning versus performance-centered partitioning on a two-cluster 2111 (IFMB) machine.

	Performance and cost		Savings breakdown	
Benchmark	Relative performance (%)	Gate cost reduction (%)	Opcode repertoire (%)	Bitwidth reduction (%)
channel	100.0	17.5	66.3	33.7
dct	96.6	4.9	54.9	45.1
fft	88.2	38.2	100.0	0.0
fsed	93.3	36.0	46.1	53.9
huffman	98.0	7.2	100.0	0.0
LU	100.0	13.3	100.0	0.0
rls	86.3	38.0	100.0	0.0
rawcaudio	89.7	0.7	91.0	8.9
rawdaudio	100.0	15.2	92.6	7.4
gsmdecode	83.0	36.3	92.6	7.4
gsmencode	100.0	38.6	97.9	2.1
blowfish	89.5	9.1	94.3	5.7
crc	100.0	5.8	46.0	54.0
url	100.0	25.3	100.0	0.0
Average	94.6	20.4	84.4	15.6

circle. In this case, moving operation 7 from cluster 2 to cluster 1 increases the partition's performance by merging the edge between operations 5 and 7. It also helps balance the workload because one less operation must execute on cluster 2. At the same time, this move doesn't increase the cost of cluster 1 because operations 1 and 2 already support the load-byte (LB) opcode in this cluster. Therefore, operation 7 is a free move. After this move, there are no more moves that improve cost-performance, and the final partition appears in Figure 3c. This partition has created one large-bitwidth cluster and one small-bitwidth cluster.

Experimental evaluation

We implemented our hardware compiler using the Trimaran tool set, a retargetable compiler for VLIW processors.[11] Synopsys design tools and a popular 0.18-micron standard cell library were used to estimate the gate cost. For each opcode the system supports, we used a width-parameterized cost formula created by synthesizing a series of hardware components to implement the opcode and fitting a curve to the reported cost. The compiler gathered bitwidth information by propagating the required widths for literals and variable types, as discussed by Mahlke et al.[12] Because we were focusing on partitioning operations in a cost-sensitive manner, we ran the most frequently executed loops from several kernels and selected applications from the Mediabench, Mibench, and Netbench suites.

The baseline hardware cost and performance measurements result from running RHOP without any cost-sensitive metric, simply the original performance-centered partitioning. The algorithm estimates hardware cost on the resulting partition, and the schedule length serves as the performance measurement. To focus on the hierarchical system's second level, we targeted the parallelism specification at the high-level machine under design. This machine specification calls for the following generic FUs in each cluster: two integer units, one floating-point unit, one memory unit, and one branch unit, in both two- and four-cluster variants (abbreviated as 2111 IFMB).

We compare the baseline machine's cost and performance with the same measurements on the final partition produced by the cost-sensitive RHOP. We define the final partition as the point at which the performance loss rate exceeds the cost savings rate. Table 1 shows the results of these experiments on the two-cluster machine. Overall, the cost-sensitive operation partitioner

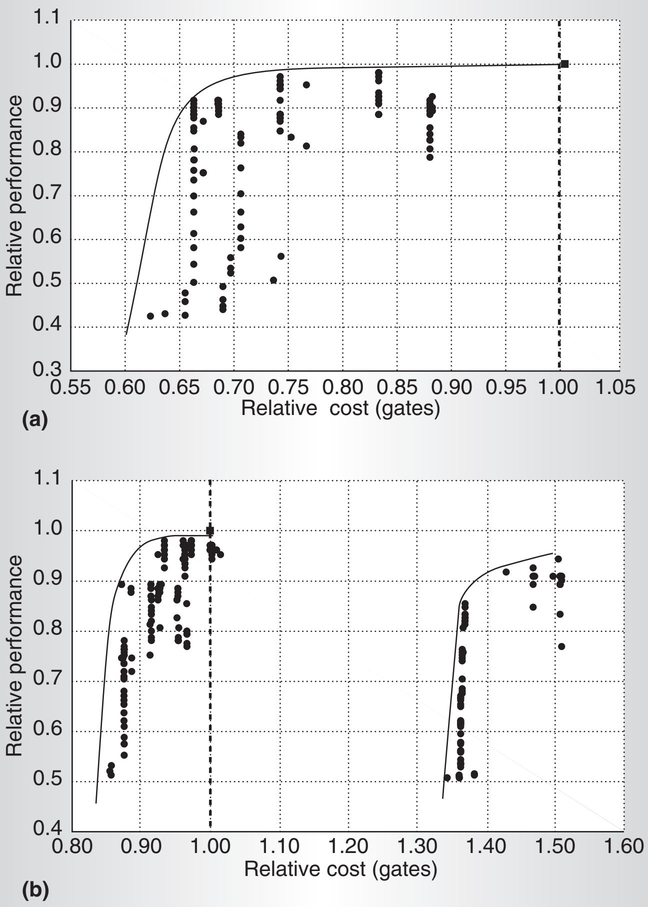


Figure 4. Set of possible cluster assignments the cost-sensitive partitioner considered for the fsed kernel (a) and the LU kernel (b) on a two-cluster configuration.

reduced the two-cluster machine's total cost by an average of 20.4 percent while retaining an average of 94.6 percent of the original performance. For a four-cluster machine, results were even better, with average cost savings of 28 percent at 97.5 percent of the original performance. Note that in six benchmarks, performance did not decrease 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. Additionally, the cost-sensitive partitioner reduced cost by more than 35 percent for five benchmarks. These benchmarks were highly amenable to cost-sensitive partitioning because they contained a few expensive operations that could be grouped in a single cluster.

The last two columns of Table 1 show the cost savings source breakdown. Both opcode repertoire and bitwidth cost savings were prevalent in the benchmarks. Several benchmarks achieved 100 percent of their cost savings from the cost-sensitive partitioner's improvements on the FU opcode repertoires. In these cases, the benchmarks performed most if not all their operations at the full 32-bitwidth, limiting the amount of available bitwidth savings. For benchmarks with a large variation in bitwidths—for example, channel, dct, and fsed—the cost-sensitive algorithm intelligently used the bitwidth information to reduce cost.

Although Table 1 shows a single design point created by the algorithm, the system actually examines many data path architectures during partitioning. We now examine the full design space for two representative applications. Figure 4a shows a Pareto chart of the possible cluster assignments the cost-sensitive partitioner considered on the fsed kernel. Each point on the chart indicates the relative estimated schedule length and relative cost for a given clustering chosen during the partitioner's run. The cost values are relative to the cost of the machine designed by the baseline performance-centered RHOP. The curved line roughly indicates the Pareto-optimal designs, which offer the best performance possible at a given cost or the lowest cost for a given level of performance. The dashed line indicates the machine designed by the baseline performance-centered RHOP algorithm. As we expected, better performance typically came at a much higher cost. The vertical bands of points in the chart appear when the partitioner first explores the band's lowest point, where performance is very low, and the partitioner begins making all the free moves that improve the performance at the same cost.

Figure 4b is a Pareto chart of cluster assignments for the LU kernel. In this example, the algorithm produced two groupings while determining the cluster assignment. This behavior occurred in several benchmarks, generally because the application required an expensive unit, such as a multiplier. When both clusters support the expensive operation, the cost-sensitive algorithm typically begins at the higher-cost grouping. During partitioning, the algorithm reaches a point where it shifts all the expensive operations to one cluster, thus forming the lower-cost grouping. Again, the dashed line indicates the machine designed by the performance-centered algorithm. In this case, the performance-centered RHOP correctly designed toward the lower-cost grouping, but it decided, purely for performance reasons, to group operations requiring expensive resources. Even within this lower-cost grouping, the cost-sensitive RHOP designed cheaper machines.

Overall, substantial hardware cost savings were possible in many applications with minimal performance impact. These cost savings came from specializing the FUs' opcode repertoire as well as paring down the bitwidths of certain clusters. Therefore, application-driven partitioning is an effective method for customizing a high-level data path specification to an application.

This work completes the first step of our hierarchical architecture synthesis system for multicluster processors. Our ongoing work focuses on expanding the hardware compilation techniques beyond the data path to include the memory subsystem, the control path, and the instruction set. Additional areas of our ongoing work include the design of a hierarchical architecture description language to represent customized ASIP architectures, and the hardware synthesis to realize these designs in Verilog. MICRO

Acknowledgments

This research was supported in part by National Science Foundation (NSF) grant CCR-0325898, NSF Career Award grant CCF-0347411, the WIMS Engineering Research Center, and equipment donated by Intel.

References

1. J. Hoogerbrugge and H. Corporaal, "Automatic Synthesis of Transport Triggered Processors," *Proc. 1st Ann. Conf. ASCI*, 1995.
2. J.A. Fisher, P. Faraboschi, and G. Desoli, "Custom-Fit Processors: Letting Applications Define Architectures," *Proc. 29th Ann. Int'l Symp. Microarchitecture* (MICRO 29), IEEE CS Press, 1996, pp. 324-335.
3. S. Aditya and B.R. Rau, *Automatic Architecture Synthesis and Compiler Retargeting for VLIW and EPIC Processors*, tech. report HPL-1999-93, HP Labs, 1999.
4. V.S. Lapinskii, M.F. Jacome, and G.A. de Vaciana, "Application-Specific Clustered VLIW Datapaths: Early Exploration on a Parameterized Design Space," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 8, Aug. 2002, pp. 889-903.
5. S. Note et al., "Cathedral-III: Architecture-Driven High-Level Synthesis for High Throughput DSP Applications," *Proc. 28th Design Automation Conf.* (DAC 28), IEEE CS Press, 1991, pp. 597-602.
6. P. Paulin and J. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 6, June 1989, pp. 661-679.
7. N. Park and A. Parker, "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 7, no. 3, Mar. 1988, pp. 356-370.
8. K. Farkas et al., "The Multicluster Architecture: Reducing Cycle Time through Partitioning," *Proc. 30th Ann. Int'l Symp. Microarchitecture* (MICRO 30), IEEE CS Press, 1997, pp. 149-159.
9. E. Gibert, J. Sánchez, and A. González, "An Interleaved Cache Clustered VLIW Processor," *Proc. 16th Ann. Int'l Conf. Supercomputing* (ICS 16), ACM Press, 2002, pp. 210-219.
10. M. Chu, K. Fan, and S. Mahlke, "Region-Based Hierarchical Operation Partitioning for Multicluster Processors," *Proc. Conf. Programming Language Design and Implementation* (PLDI 03), ACM Press, pp. 300-312.
11. "Trimaran: An Infrastructure for Research in Instruction-Level Parallelism," http://www.trimaran.org.
12. S. Mahlke et al., "Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 11, Nov. 2001, pp. 1355-1371.

Michael L. Chu is a PhD candidate working in the Advanced Computer Architecture Laboratory of the Electrical Engineering and Computer Science Department at the University of Michigan, Ann Arbor. His research interests include multicluster compilation and automatic design and synthesis of application-specific processors. Chu has BS and MS degrees in computer science and engineering from the University of Michigan.

Kevin C. Fan is a PhD student working in the Advanced Computer Architecture Laboratory at the University of Michigan, Ann Arbor. His research interests include automated design of application-specific processors and compilation for irregular and embedded architectures. Fan has a BS from UCLA and an MS from the University of Michigan, both in computer science and engineering.

Rajiv A. Ravindran is a PhD candidate in the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor. His research interests include compilation for low-power embedded DSPs, automatic compiler and architecture synthesis, and compilation for high-performance processors. Ravindran has an MTech in computer science from the Indian Institute of Technology, Kanpur. He is a student member of the IEEE and the ACM.

Scott A. Mahlke is an assistant professor in the Electrical Engineering and Computer Science Department at the University of Michigan, Ann Arbor. His research interests include application-specific processors, high-level synthesis, compiler optimization, and computer architecture. Mahlke has a PhD in computer engineering from the University of Illinois. He is a member of the IEEE and the ACM.

Direct questions and comments about this article to Michael Chu, Advanced Computer Architecture Laboratory, Dept. of Electrical Engineering and Computer Science, University of Michigan, 1301 Beal Ave., Ann Arbor, MI 48109-2122; mchu@umich.edu.