Automated Custom Instruction Generation for Domain-Specific Processor Acceleration

Nathan T. Clark, Hongtao Zhong, and Scott A. Mahlke, *Member*, *IEEE*

Abstract—Application-specific extensions to the computational capabilities of a processor provide an efficient mechanism to meet the growing performance and power demands of embedded applications. Hardware, in the form of new function units (or coprocessors), and the corresponding instructions are added to a baseline processor to meet the critical computational demands of a target application. In this paper, the design of a system to automate the instruction set customization process is presented. A dataflow graph design space exploration engine efficiently identifies computation subgraphs to create custom hardware and a compiler subgraph matching framework seamlessly exploits this hardware. We demonstrate the effectiveness of this system across a range of application domains and study the applicability of the custom hardware across an entire application domain. Generalization techniques are presented which enable the application-specific hardware to be more effectively used across a domain.

Index Terms—Automatic synthesis, instruction set interpretation, special-purpose, instruction set design, special-purpose and application-based systems.

1 Introduction

In recent years, the markets for cellular phones, digital cameras, network routers, and other high performance but special purpose devices have grown explosively. In these systems, application-specific hardware design is used to meet the challenging cost, performance, and power demands. The most popular strategy is to build a system consisting of a number of highly specialized application specific integrated circuits (ASICs) coupled with a low cost core processor, such as an ARM [33]. The ASICs are specially designed hardware accelerators to execute the computationally demanding portions of the application that would run too slowly if implemented on the core processor. While this approach is effective, ASICs are costly to design and offer only a hardwired solution that permits almost no postprogrammability.

An alternative design strategy is to augment the core processor with special-purpose hardware to increase its computational capabilities in a cost-effective manner. The instruction set of the core processor is extended to feature an additional set of operations. Hardware support is added to execute these operations in the form of new function units or coprocessor subsystems. The Tensilica Xtensa is an example commercial effort in this area [15].

There are a number of benefits to augmenting the instruction set of a core processor. First, the system is postprogrammable and can tolerate changes to the application. Though the degree of application change is not arbitrary, the intent is that the customized processor should achieve similar performance levels with modest changes to the application, such as bug fixes or incremental modifications to a standard. Second, the computationally intensive portions of applications from the same domain (e.g., encryption) are often similar in structure. Thus, the customized instructions can often be generalized in small ways to make their use have applicability across a set of applications. Last, some or all of the ASICs become unnecessary if the augmented core can achieve the desired level of performance. This lowers the cost of the system and the overall design time.

One central question with this approach is the degree of human effort required to identify and implement an efficient set of instruction set extensions. In addition, the effort required to modify the software development tool chain to effectively understand the extended instruction set is substantial. This effort can often be more time consuming and expensive than the design of an ASIC.

We believe automation is the key to making instruction set customization successful. To this end, this paper presents the design of a system that combines automatic hardware selection and seamless compiler exploitation of the custom instructions. Hardware design is accomplished via intelligent dataflow graph exploration. The exploration subsystem focuses on efficient discovery and selection of computation subgraphs from which custom hardware is constructed. The major challenge is navigating through an almost limitless design space without artificially constraining the size and shape of the subgraphs.

Once the custom instructions are discovered, several generalization techniques are applied to increase the number of subgraphs that map to each hardware unit. This ensures that custom instructions are useful across an entire domain of applications. These generalization techniques are unique to the field of instruction set customization.

Compiler exploitation of the custom instructions is accomplished through a flexible subgraph matching engine. Applications are analyzed to match computation subgraphs

• *The authors are with the Advanced Computer Architecture Lab, Electrical Engineering and Computer Science Department, University of Michigan, 1301 Beal Ave., Ann Arbor, MI 48109-2122.*
E-mail: {ntclark, hongtaoz, mahlke}@umich.edu.

Manuscript received 19 Apr. 2004; revised 30 Dec. 2004; accepted 6 Apr. 2005; published online 16 Aug. 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-0136-0404.

0018-9340/05/$20.00 © 2005 IEEE Published by the IEEE Computer Society

that can be replaced by custom instructions. This allows the customized hardware to be effectively utilized no matter what application is run on it.

Many other researchers have proposed systems to accomplish the task of automated instruction set generation. The contributions of this paper are four-fold. First, we present a novel technique for efficient dataflow graph exploration and selection. Second, we present the design and demonstrate the implementation of a complete system, including retargetable compiler. Most previous work neglects the problem of compiling to a processor with custom instructions. Third, and most importantly, we use the system to analyze how effectively instruction set extensions designed for one application can be applied to other applications in the same domain. Several techniques to increase the cross-application utility are explored. Last, we provide some analysis on how custom instructions differ when designed with multiple applications in mind.

2 Related Work

A large body of research has gone into instruction set customization. Work in [5], [32], [36], [18], [16], and [37] all showed possible gains from using this technique. While these works show the potential utility of instruction set customization, they do not provide methods to automate the process. Many other systems have been proposed to automate this process, though. These systems can be categorized based on how they solve two subproblems: identification of custom instruction candidates and how to make use of the candidates.

Candidate Discovery. Informally stated, candidate discovery is determining subsets of an application's dataflow graph, or *DFG*, that would be amenable for implementation in hardware. In the most general sense, each node of the DFG can either be included or excluded from a candidate, yielding $O(2^{numberofnodes})$ potential candidates. Several techniques have been proposed to handle the intractability of this problem.

Early work [2] sidestepped the candidate discovery problem altogether by predefining a set of candidates. This strategy requires a designer to enumerate a superset of useful candidates to select from and utilizes design automation in the selection phase. While some advantages of customization are realized, this approach is limited by the large amount of work necessary to define an appropriate superset of candidates and the poor results obtained when an appropriate superset is not available.

Work by Bennett [7] proposes iterative combination of primitives that occur in subsequent lines of code to reduce static code size. This method assumes that a base instruction set is given corresponding to a high level language. Statistics are gathered on the frequency of operations occurring near each other and the highest ranking combination is chosen as a new instruction. This technique is irrespective of the dataflow graph and is primarily used as a code size reduction technique.

Bennett's work is similar to candidate discovery algorithms in [31], [30], [38], [6], [9], and [23] in that all of these papers propose iterative combination of primitives. Iterative solutions typically combine two nodes, replace all such occurrences in the DFG, and repeat until some constraints are met. These solutions have the benefit of very good runtimes (typically $O(N^2)$) when compared to more thorough strategies, but risk being stuck in local maxima. Each edge is combined in a locally optimal manner, reminiscent of greedy heuristics.

Holmer proposed a more global technique [19], which was later extended by [21]. This technique discovered candidates by performing an initial grouping of nodes based on the schedule time in the DFG, then iteratively breaking and recombining these groups. Work by Bose and Davidson [8], is similar to this, except that this work operated on a syntax tree, instead of a DFG, and used many more candidate transformations than breaking and combining. Another major difference is that Holmer guided use of the transformations using simulated annealing, attempting to maximize the worth of the instruction set, where Bose and Davidson performed transformations unguided with the expected goal of improvement. The application of these two algorithms was mainly for designing entire instruction sets, as opposed to just ISA extensions.

Choi et al. [10] generated initial candidates in a similar manner to Holmer. This work advocated combining instructions that could be executed in parallel and then combining those parallel sets to create custom instructions that were both wide and deep. In order to cut down on the number of potential candidates explored, Choi et al. used an artificial limit on how deep the combined instructions can be. The main contribution of [10] is a new formulation of the candidate discovery problem: They discovered candidates using a modification of the subset-sum problem and attempted to find the minimal set of instruction extensions to meet a certain performance requirement (as opposed to simply discovering the optimal instruction extensions for a given cost). The main weaknesses of this work are the artificial limit on custom instruction length and the initial phase of combining parallel instructions performed when it is not clear that parallel combination is best.

Other work proposes dealing with intractability by limiting the size of the problem. The algorithm proposed in [4] and [13] searches a full binary tree, where each step decides whether or not to include a node of the DFG into a candidate. Ways to prune the tree are proposed, helping avoid the worst case $O(2^N)$ runtime, but the size of the DFG must still be relatively constrained in order for the algorithm to complete in a timely manner. This limits its usefulness for very large basic blocks.

Some research has proposed heuristic ways to limit the search space without artificially constraining it. In [3], the least used half of all candidates is removed after each iteration of candidate discovery. While this technique will catch all important candidates in hot portions of the code, it potentially misses useful candidates that are moderately used in many portions of the application. Work by Sun et al. [34] performs a similar pruning, but uses a more complex priority function to rank the candidates, taking into account the number of inputs and outputs, as well as dynamic occurrences. In Sun et al.'s work, candidates that do not meet a certain percentage of the best discovered candidate so far are removed. Work in reconfigurable computing [28] initially partitions the DFG

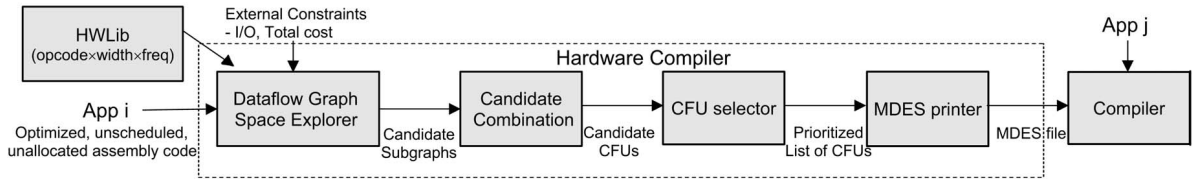

Fig. 1. Organizational structure of the hardware compiler.

into small pieces based on heuristics. Candidates are then selected for these partitions. Heuristic-based methods such as this have the benefit of not artificially constraining the problem by potentially getting stuck in local minima or limiting the types of candidates discovered.

Utilization. The problem of how to make use of the candidates is the other major issue to solve for instruction set customization. The vast majority of automated systems in this field have neglected this problem. Most systems combine the discovery and selection phases so that whenever candidates are selected, they are immediately replaced in the code, e.g., [21]. These systems typically do not provide methods to reuse the new instructions in other applications. As such, it is necessary to look at work in the compiler community.

Automated utilization of custom instructions generally happens during the code generation phase of compilation. Traditional code generation methods use a tree covering approach [1] to map the DFG to an instruction set. The DFG is split into several trees, where each instruction in the ISA covers one or more nodes in the tree. The tree is covered using as few instructions as possible. The purpose behind splitting the DFG into trees is that there are linear time algorithms to optimally cover trees, making the process very quick.

One problem with this method, though, is that DFGs are not trees. It is shown in [25] that tree covering methods can yield suboptimal results, particularly in the presence of irregular subgraphs common in custom instructions. To overcome this, [25] proposes splitting all instructions into "register-transfer" primitives and recombining the primitives in an optimal manner using integer programming. Work by Liao et al. [26] attacked the same problem and developed an optimal solution for DFG covering by augmenting a binate covering formulation. While both of these solutions are optimal, they also have exponential runtime and must be selectively used.

Research in [29] describes a new way to look at the code generation problem. In this work, computationally complex algorithms are used to insert custom instructions and heuristics handle the rest of code generation. An application is initially decomposed into an algebraic polynomial expression which is functionally equivalent to the original application. Next, the polynomial is manipulated symbolically in an attempt to use custom instructions as best as possible. For example, a polynomial could be expanded using function identities (e.g., adding 0 to a value) to better fit an existing custom instruction. Custom instructions are inserted as intrinsic function calls in the polynomial and functionally equivalent high-level language is output once all of this is complete. The high-level language can then be used as input to a standard compiler. The main contribution of this work is the method of algebraically modifying of code to better make use of available instructions.

Our System. The candidate discovery technique proposed in this paper is similar to the work in [4] in that a full exponential search is used where appropriate. The technique in this paper differs in that it incorporates a heuristic function, similar to [34] and [28], to help divide the problem when exponential search is too slow. This is discussed in detail in Section 3. In Section 4, the custom instruction utilization framework is described. This framework ties together several ideas from other work into one system and addresses some runtime issues with previously proposed solutions. The main contribution in this work is presented in Section 5: The custom instructions generated by our system are applied to benchmarks across several domains and the results of these experiments are analyzed. Techniques to improve the effectiveness of cross-domain instructions and the issue of how to design instructions for multiple applications are also tackled. Domain-wide discovery and analysis of custom instructions has not been previously examined.

3 Dataflow Graph Exploration

The purpose of the dataflow graph (DFG) explorer is to determine candidate subgraphs for instruction set extensions. Implementing subsets of the DFG in hardware typically allows for better performance, lower power consumption, and reduced code size than the corresponding implementation as a sequence of primitive operations. Determining which parts of a DFG would make the best custom instructions is a difficult problem, though. The most glaring difficulty is that the number of potential candidates for a given DFG increases exponentially in the number of operations. Exploration heuristics must be developed to overcome this problem.

The overall structure of our DFG exploration engine is shown in Fig. 1. One or more applications are fed into the system as profiled assembly code. The code has not been scheduled and has not passed through register allocation, which is important so that false dependences within the DFG are not created. Initially, the application passes through a DFG space explorer, which determines candidate subgraphs for potential instruction set extensions. The space explorer selects subgraphs subject to some externally defined constraints, such as the maximum die area allowed for any custom function unit (CFU) or the maximum allowable register read and write ports. A hardware library provides timing and area estimates to the DFG explorer so

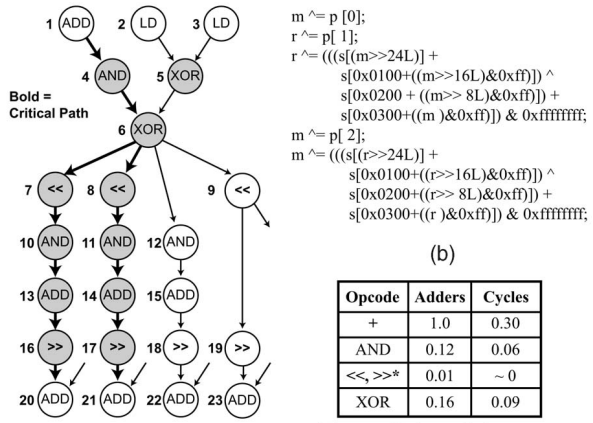

Fig. 2. (a) Sample DFG from blowfish. Shaded nodes delineate a CFU. (b) Preprocessed C code this DFG came from. (c) Excerpt from the hardware library. "Adders" is the die area relative to a 32-bit adder.

that it can accurately gauge the cycle time and area requirements of combined primitive operations.

A list of subgraphs, annotated with area and timing estimates, is passed to a candidate combination stage. This stage groups subgraphs that would be executed on the same piece of hardware. Grouping the subgraphs creates a set of candidate CFUs and enables calculation of an estimated performance gain by using the profile weights of all the set members. The combination stage also performs some generalization steps to enable more subgraphs to map onto the same potential hardware implementations. All of this information is passed to a selection mechanism that determines which CFUs best meet the needs of the application(s). Finally, the prioritized list of CFUs is converted into a machine description (MDES) form that can be fed to the compiler.

Throughout this section, the DFG shown in Fig. 2 from the *blowfish* application [17] is used for illustrative purposes. For simplicity, each operation or node is assumed to take one cycle to execute on the baseline processor.

3.1 Subgraph Discovery

The exploration strategy employed in this work starts by examining each node in the DFG and using it as a seed for a candidate subgraph. This seed is grown downward along dataflow edges to create a new candidate. For example, if the seed was node 6 in Fig. 2, it would be grown to nodes 7, 8, 9, and 12. When candidates overlap with each other (such as the candidates with nodes 6-7 and 6-8 in the example), a new candidate is created with the union of their nodes (6-7-8). During growth, each intermediate candidate is recorded for potential implementation as a CFU. Growing the candidates continues until some external constraints are met, such as the candidate crossing a control flow boundary or exceeding the number of register read ports available.

Initially, this system used a naive implementation that looked at all possible dataflow edges to grow the seed nodes. Using this approach guarantees identification of the best possible set of connected candidates since all possible candidates are generated. However, the number of candidates quickly grows out of control for many applications.

The key observation gained from experimenting with this naive approach is that growing along the majority of dataflow edges examined by exponential growth simply does not make sense. For example, assuming the goal is maximizing performance on the DFG in Fig. 2, growing along the edge between nodes 6 and 9 has little value because node 9 is not on the critical path (i.e., the longest dependence path(s) in the DFG).

Previous work [4], [13] has shown that using an exponential solution, such as growing along all edges, is sufficiently fast for some applications when intelligent pruning is used. There are many applications that have too many nodes for exponential search, though. For these large DFGs, we propose using a *guide function* to determine which edges are directions that do not need to be grown toward. By heuristically removing unimportant edges, the DFG is effectively partitioned into smaller sections, which can then be used by the exponential growth algorithm described above. This strategy allows us to maintain the quality of the resultant candidates without taking exponential time or resources.

Our previous work [12] proposed using a guide function as a method for inclusion, which is to say that only edges which were determined to be important were grown along. This effectively avoided the exponential search associated with subgraph discovery, but, in some instances, overcompensated for the problem. Using the guide function to remove edges takes the opposite approach and only prunes the search space when necessary.

One important characteristic of this technique is that the partitioning step can be tuned to more efficiently explore the design space. Partitioning the DFG into just a few larger sections will ensure better coverage of the design space, while partitioning the DFG into many smaller sections will result in reduced runtimes and memory consumption of the exploration algorithm. An example of exploiting this tradeoff would be using larger partitions in parts of the DFG that have higher profile weight, as they are more likely to yield important candidates. All previously proposed solutions use a single exploration strategy for all parts of the DFG, whereas this technique can modify its strategy to effectively use the computational resources available.

3.2 Guide Function

The purpose of the guide function is to intelligently rank which dataflow edges would most likely be involved in unimportant candidates. The guide function essentially tries to replace the architect by determining these unimportant edges, thus its decisions must reflect the same properties the architect would use. The guide function proposed here uses three categories to rank the desirability of edges: criticality, latency, and area. In the candidate discovery system, each of the guide function categories is allotted 10 points of weight and the sum of these categories determines the total desirability of each edge. The edges with the lowest desirability are more likely to be cut if the DFG needs to be partitioned.

Criticality. This category ranks edges highly when they appear on the longest dependence path(s) of a DFG. CFUs

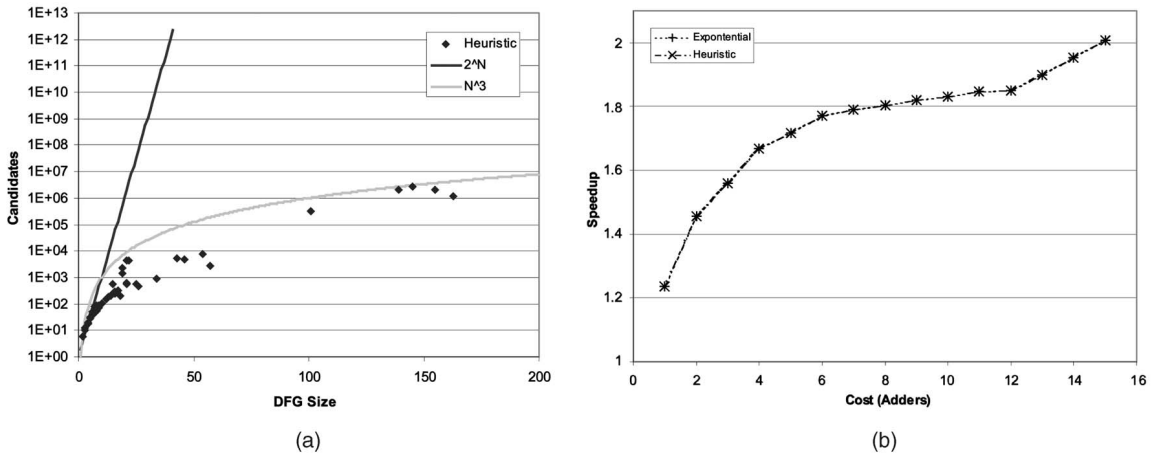

Fig. 3. (a) The number of candidates examined for DFGs in three encryption benchmarks. (b) The average speedup on those benchmarks using full exponential and heuristic search techniques.

that occur on the critical path are likely to give the application performance improvement because they shrink the height of the DFG. Since performance improvement is typically the most desired result of CFUs, cutting edges along these paths is undesirable. An example of this from Fig. 2 would be the edges from node 6. The edges toward nodes 7 or 8 would rank higher in terms of criticality than would the direction toward nodes 9 or 12 because the aforementioned nodes are on the critical path. Points are awarded using the equation $\frac{10}{slack+1}$, where slack is the number of cycles an operation can be delayed without lengthening the critical path. Thus, the edge from node 6 to 7 would get $\frac{10}{0+1} = 10$ points and the edge from node 6 to 9 would get $\frac{10}{2+1} = 3.33$ points. Note that it is important to give candidate edges credit even when they are slightly off the critical path, as the heuristic provides, because auxiliary paths often become critical after several CFUs are formed.

Latency. Combining operations that require fewer cycles to execute in conjunction than they do individually will lead to high-quality CFU candidates. The largest performance gains are possible by combining several low latency operations, such as bitwise logicals, where many can be executed in a single cycle. Conversely, if two nodes on an edge cannot be executed in fewer cycles when combined, then the resultant candidate is less beneficial. The latency category models this trend. Latency points are distributed using the equation $\frac{old\ latency}{new\ latency} * 10$. The latency of a CFU is calculated by summing up the fractional delays of the two atomic operations (see Fig. 2c) connected to the edge. For example, node 10 can be executed in 0.06 cycles as indicated by the "Cycles" entry for the AND opcode in Fig. 2c. Exploring the edge toward node 13, which has a latency of 0.3 cycles, would get $\frac{0.06}{0.06+0.30} * 10 = 1.67$ points. In contrast, growing from node 6 toward node 9 would get nearly all $(\frac{0.09}{0.09+0} * 10 = 10)$ the points allotted for latency.

Area. Since cost is a major constraint in the design of embedded processors, area is an important factor in the choice of CFUs. The guide function considers the area to be the sum of the areas required to implement each primitive operation on an edge (see Fig. 2c). Note that register file ports are a design constraint, thus they do not factor into the area calculation. Further, CFUs do require additional decode logic and interconnect, but we assume that primitive operation area is the dominant term. The area category gives more points to directions that increase the total area of the candidate the least. Area points are calculated the same way as latency, $\frac{old\ area}{new\ area} * 10$. As an example, growing from node 19 to node 23 would yield $\frac{0.01}{1.0+0.01} * 10 = 0.1$ points and growing from 9 to 19 would yield $\frac{0.01}{0.01+0.01} * 10 = 5$ points.

The guide function gives a weight to each edge in the DFG. If the DFG proves to be too large for the exponential exploration algorithm, a recursive bisection is performed on the DFG until the partitions are small enough (typically 50 or fewer nodes). For example, if the DFG in Fig. 2 was too large, then it would have to be split into smaller pieces. To do that, at least one edge would have to be cut. In this figure, the edge from node 18 to node 22 is the first choice to cut because it has the smallest weight according to the guide function. Cutting that edge creates two new partitions and eliminates all candidates that contain that edge from being explored. If the two new partitions are still too large, the process is repeated until they are small enough. The partitioning is performed using hMetis [22], a high-quality multilevel partitioner. Edge weights from the guide function lead hMetis toward cutting edges, which will not lead to good candidates. In practice, partitioning the DFG greatly reduces the design space to the point where most applications can be fully explored in under 5 minutes on a Pentium 4 system.

With the guide function/partitioning heuristic in place, it is important to verify two points: that the heuristic does indeed prune the search space and that good candidates are not missed because the partitioner incorrectly precludes them. Fig. 3a demonstrates the first point. Each dot on this graph represents the number of candidates examined when exploring one basic block from three encryption applications that are characterized by large loop bodies. This figure shows that the partitioner is able to effectively curve the exponential growth associated with the DFG exploration problem. This algorithm can be used on very large DFGs and without artificially constraining the types of candidates

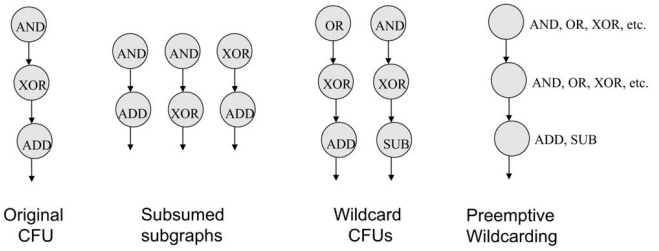

Fig. 4. Examples of generalization techniques.

generated, which are both weaknesses of some previously proposed algorithms.

To ensure that good candidates are not dismissed, the heuristic was compared against a full exponential search using strict external constraints (candidates were only allowed three input and one output port). Fig. 3b shows the speedups[1] attained from using the candidates generated by both algorithms. As shown in the figure, the two curves track identically due to the fact that the partitioning heuristic did not prune any important candidates during its search. DFGs can be constructed where the heuristic will miss important candidates, but this was not observed in any of the test cases.

3.3 Candidate Combination and Generalization

After discovery, it is a straightforward process to group identical candidate subgraphs together into candidate CFUs. A simple test that checks graph equivalence while taking into account commutativity accomplishes this. For example, if subgraphs 7-10-13-16 and 8-11-14-17 were discovered in Fig. 2, the graphs would be checked for equivalence and then combined into the candidate CFU "<<-AND-ADD->>." The profile weights are then used to get an estimate of the number of cycles each CFU improves performance. In the case where CFUs are being designed for multiple applications, the cycle estimates are scaled to ensure that one application does not dominate simply because it has a longer execution profile. Using a compiler to get an exact measurement by scheduling with each instruction is possible, but the complexity makes this solution undesirable. In practice, the profile-based estimates proved reasonably accurate.

After candidate grouping, a generalization process takes place to make the candidates more useful across a domain of applications. Two techniques are employed to accomplish this. The first is *subsumed subgraphs*. Subsumed subgraphs take advantage of the fact that most atomic operations have an associated identity input, allowing values to pass through a node without changing. Using Fig. 4 as an example, if CFU "AND-XOR-ADD" was discovered, CFU "AND-ADD" can be executed on the same hardware because one input of the XOR operation could be set to 0. CFUs "AND-XOR" and "XOR-ADD" could also be subsumed by "AND-XOR-ADD." The cost of implementing these subsumed subgraphs is simply a MUX on one input of every node being bypassed; thus, for very little additional cost, the number of subgraphs that map onto a CFU is increased.

The second generalization technique is called *wildcarding*. Wildcards are subgraphs that are a similar shape as the original CFU, but operations at one node may differ. Combining two CFUs with similar structure allows us to share hardware and map multiple subgraphs onto the same CFU. Two examples of wildcards are given in Fig. 4. If the original CFU implements "AND-XOR-ADD," then both "OR-XOR-ADD" and "AND-XOR-SUB" would be recorded as potential wildcards if they appeared in the input DFG since they only differ by one node from the original CFU.

A stronger version of wildcarding, termed *preemptive wildcarding* in this work, generalizes a CFU to have many potential operations at each node. Unlike regular wildcarding, the preemptive subgraphs do not necessarily have to appear in the input DFG. The idea behind preemptive wildcarding is that many operations have very similar hardware implementations, e.g., ADD and SUB, or can be added to a node for very little cost. Additionally, we have observed that applications within a domain have similarly shaped DFGs, even if the operations at individual nodes do not match. Preemptively adding this functionality allows many more subgraphs to map to a single CFU and makes them much more useful across a domain of applications. Again, an example of preemptive wildcarding is given in Fig. 4. Here, logical operations were added to the AND and XOR nodes and SUB was added to the ADD node.

It is important to note that, in this phase of the exploration framework, no binding decisions are made with regard to subsumed subgraphs and wildcarding. The generalization phase simply creates new, generalized candidates with updated area and cycle estimates to reflect what the potential cost and benefit of generalizing each original candidate. The selection algorithm can then weigh the costs and benefits of generalizing and make an appropriate decision. Since preemptive wildcarding often does not improve the estimated cycle savings for an input DFG, it is always performed if the cost required is less than a predefined threshold.

3.4 Candidate Selection

Selecting CFUs with a given area constraint is similar to the 0/1 knapsack problem. There is a set of resources (the CFUs) that all have a value (the estimated cycle savings) and a cost (die area) and the goal is to maximize the total value for a given cost. It is widely known that the 0/1 knapsack problem is NP-complete, although it is solvable in pseudopolynomial time using dynamic programming. Strategies are needed to avoid intractability in this stage of design automation as well.

It is important to mention that CFU selection has one caveat missing in the 0/1 knapsack problem: The values of all the other CFUs change once a CFU is selected for inclusion. Individual operations can appear in multiple CFU candidates. Once a CFU is selected, it is necessary to update the estimated cycle savings of the other CFUs so that double counting does not occur. Using an example from Fig. 2 again, assume the two highest ranked CFUs were 7-10-13-16 and 7-10-13. If 7-10-13-16 was selected first and did not update the value of 7-10-13 to reflect the fact that it

1. The experimental setup used to obtain this data is explained in detail in Section 5.

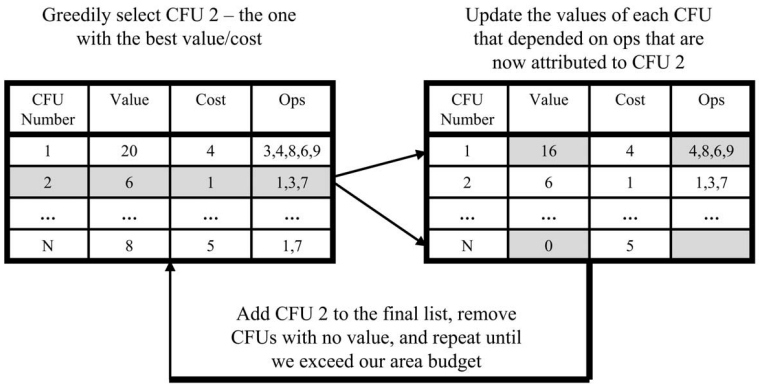

Fig. 5. Greedy approach to CFU selection.

can no longer use any of its operations, then 7-10-13 would be selected also, even though it would provide no gain above what 7-10-13-16 already provided.

One strategy used for CFU selection is a simple greedy method, illustrated in Fig. 5. Given a list of CFU candidates, the one with the best ratio of $\frac{value}{cost}$ is greedily selected. Once CFU 2 is selected, the heuristic iterates through the list of remaining candidates and removes operations that were claimed by it. In Fig. 5, operations 1 and 7 were removed from CFU N and its value was updated to 0 as it had no more operations left. Operation 3 was removed from CFU 1 and its value was likewise updated to 16. Once all CFUs are updated, the selection process is repeated until the area budget is exhausted.

Because the selection heuristic is greedy, it is not guaranteed to give an optimal solution and frequently does not. For example, when the greedy algorithm selects based only on estimated cycle savings, performance does poorly at the low cost budget points compared to when it selects based on $\frac{value}{cost}$. However, the opposite is true at high cost points.

In an attempt to improve the selection heuristic, a version based on dynamic programming was implemented as well. This is a straightforward extension of the algorithms presented in [20]. The problem with the dynamic programming method is that it requires candidates to update their estimated value many more times than the greedy method. This computational overhead is quite significant and, in order to alleviate it, a simplifying assumption is made. Prior to the selection, each operation is assigned to the candidate with the largest estimated speedup. This eliminates the need to frequently update candidate values, but potentially misleads the selector. Despite this, the dynamic programming method typically provides better results than the greedy method.

Dealing with wildcards and subsumed subgraphs adds another challenge to the selection process. The main issue is the possibility that implementing a subsumed subgraph as a separate CFU is more desirable than implementing it on existing, subsuming hardware. As an example consider the large gray CFU from Fig. 2. If "XOR - < <" were to be run on custom hardware, it could be done for a minimal area overhead on the large, gray CFU; however, there would be a latency penalty of going through three more operations (there are no early exits from operations 7 or 8). It may be that creating a special "XOR - <<" unit is the better solution. Subsumed CFUs are not removed from the selection pool so that the option to include both the subsumed and subsuming candidate is available.

Another issue is whether to count all the subsumed subgraphs and wildcards when determining the estimated value of a CFU. If they are counted, then, in addition to updating the estimated value of other CFUs based on the operations in the candidate subgraphs, it is also necessary to update the values based on all the operations in the subsumed or wildcard candidate subgraphs. This creates a large computational overhead for every subgraph selection. Additionally, this means frequently attributing operations to small subsumed portions of a large CFU when more performance could have been gained by attributing them to a separate CFU (like the example in the previous paragraph). The case just described occurs quite frequently, so CFUs are selected as if they had no subsumed subgraphs or wildcards. When a selection is made, the costs of the subsumed subgraphs and wildcards are updated to reflect that they can now be added for very little cost overhead.

3.5 Example CFUs

The types of CFUs generated by this system are quite varied, depending on the input DFGs. Some examples of selected CFUs across six applications are shown in Fig. 6. Often, the system generates instructions that an architect would expect to see, such as the multiply-accumulate selected for djpeg and the saturating-add selected for gsmencode. This provides some empirical evidence that the system is making intelligent decisions. The system also frequently generates custom instructions that appear very unusual, too, such as the ones for 3des, blowfish, crc, and gsmdecode.

4 Compiler Utilization

The purpose of the compiler is to automatically exploit CFUs available for any given application. The basic structure of the retargetable compiler is shown in Fig. 7. Applications are run through a front-end, producing a generic RISC assembly code. The assembly code is unscheduled and uses virtual registers. The compiler uses a machine description, or MDES, to determine what CFUs

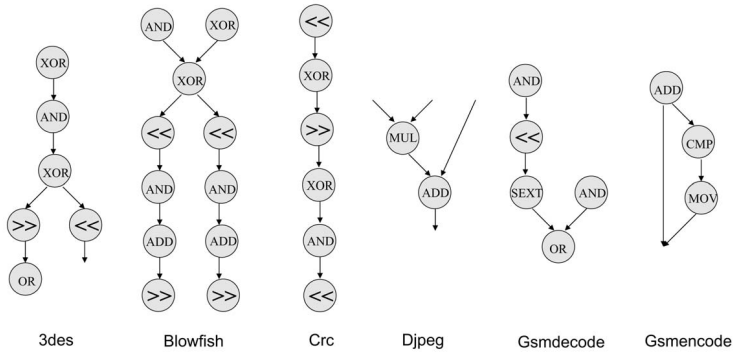

Fig. 6. A selection of CFUs generated by the exploration system.

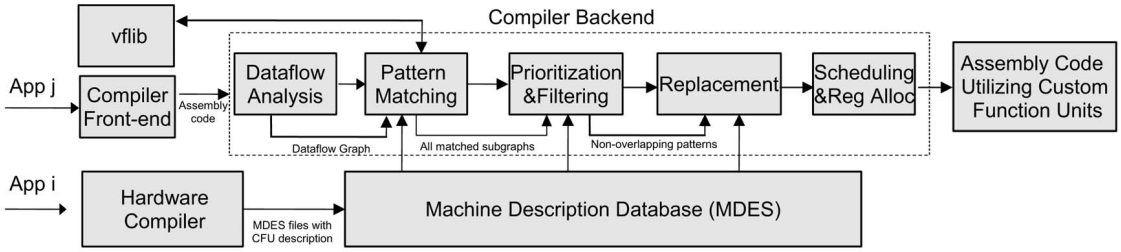

Fig. 7. Organizational structure of a compiler supporting custom instructions.

are available for use. Given the assembly code and MDES, the compiler performs dataflow analysis to generate a DFG, discovers all subgraphs in the DFG that match available CFUs, prioritizes these matches, replaces the matches with custom instructions, and, finally, performs the typical tasks of register allocation and scheduling. The steps that differ from traditional compilation techniques are described in detail below.

4.1 Pattern Matching

Pattern matching is the most critical step in CFU utilization. The first step in this process is determining all available CFUs from the MDES. From a high level, the MDES describes what resources a CFU consumes, the latency of the operation, the number and type of inputs and outputs, and the structure of the subgraph that the CFU implements.

Discovering the subgraphs in the DFG can be viewed as the subgraph isomorphism problem, which is known to be NP-complete. To perform subgraph identification, the vflib graph matching library [14] is employed. While the algorithm used in vflib is still exponential worst case, the best case is only polynomial and the overhead added to the compile time found in practice is minimal.

The vflib algorithm finds matching subgraphs by starting at individual nodes that occur both in the DFG and the CFU. These nodes are termed a *partial match*. The partial matches are then expanded along DFG edges to create new partial matches in a manner that is similar to DFG space exploration.

Fig. 8 shows part of a DFG that is similar to one in the sha benchmark [17]. Given a CFU to implement the operations in subgraph 2-5-6, the pattern matcher would begin by looking at all left shift ($<<$) nodes: 2, 14, and 16. These partial matches would then be grown toward all consumers since node 2 has a consumer in the CFU. This would create partial matches 2-3, 2-6, 14-18, and 16-19. 2-3 and 14-18 no longer match the CFU, so only 2-6 and 16-19 are considered. These two partial matches are then grown toward the producers of 6 and 19 since the original CFU had two producers feeding the OR node. This process continues until all the partial matches either definitively match or do not. Subgraph matching is repeated for all CFUs so that all potential subgraph matches in the DFG are discovered.

At this stage, the same operation may appear in multiple subgraph matches. Deciding which match an operation should be placed in is an NP-hard problem, though. The optimal solution proposed in [26] was prohibitively slow when implemented in our compiler. To overcome this, a partitioning technique was again employed.

The traditional solution mentioned previously uses a binate covering formulation which optimally maps CFUs onto a DFG. Typically, an entire DFG is mapped at one time in this method. However, there are usually very few nodes that appear in multiple matches, meaning that the matches do not frequently overlap. This fact allows the problem to

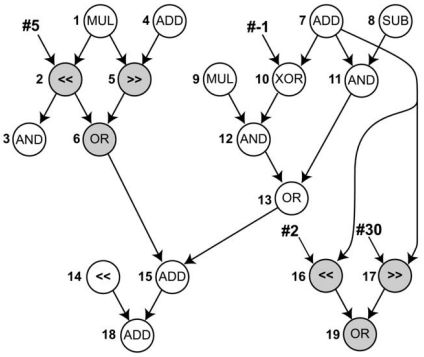

Fig. 8. DFG similar to one from sha.

be separated into several, independent binate coverings on subsets of the DFG. To illustrate this, consider a mapping was being performed on the DFG in Fig. 8. If no matches contained both nodes 13 and 15, then the graph could then be partitioned along the edge that connects them. Binate covering could be done independently on the left nodes and the right partitions without sacrificing optimality. A simple branch and bound algorithm was used to solve the binate covering formulation. Once the subproblems are solved, an optimal solution to the entire DFG can be constructed from the optimal subsolutions, much more quickly than when looking at the entire DFG at once. While cases can be constructed to make this technique prohibitively slow as well, in practice it was very fast, typically taking no more time than the scheduling phase of compilation.

4.2 Custom Instruction Replacement

On the surface, replacing the matched subgraph with a custom instruction is fairly simple. There are some important issues that must be considered in order to guarantee the correctness of the resultant program, however. Using the DFG shown in Fig. 8, subgraph 2-5-6 will be replaced with a custom instruction. The question that arises is, "Where should the custom instruction be placed in relation to other operations in the assembly code?" To ensure correctness of the program, the custom instruction must be placed after all the predecessors of the operations in the subgraph (after nodes 1 and 4 in this example) and also before all the successors (nodes 3 and 15 here). Assuming the node identifiers define the sequential order of the assembly code for this example, there is a potential problem with where to place the custom instruction. Replacing node 2 is incorrect because the custom instruction would be placed before node 4. Similarly, replacing nodes 5 or 6 is incorrect because it would be placed after node 3.

To prevent this from occurring, the assembly code is reorganized prior to subgraph replacement. For subgraph 2-5-6, the last scheduled predecessor is node 4 and the earliest scheduled successor is node 3. As long as the custom instruction is inserted between these operations, program semantics will be maintained. For every subgraph match, if the last predecessor comes after any successors, then those successors and any operations dependent on them are moved after the last predecessor. In this example, we would move node 3 after node 4 and then safely insert the custom instruction after the last predecessor.

Once the subgraphs are replaced and the code is reordered for correctness, scheduling and register allocation take place, leaving us with an application that intelligently utilizes the available CFUs.

5 Experimental Results

The system proposed was constructed as part of the Trimaran research infrastructure [35]. The DFG exploration engine was implemented as a standalone module and the compiler backend was modified to facilitate subgraph matching and replacement. The cycle time and area estimates in the hardware library were calculated using Synopsis design tools and an Artisan 0.18μ standard cell library.

For this evaluation, two simplifying assumptions are made. First, no memory instructions were included in CFUs. Having custom instructions that access memory creates CFUs with nondeterministic latency as well as requires consideration of cache ports during DFG exploration. Memory disambiguation within a custom instruction must also be factored when doing pattern replacement in the compiler. The second assumption was that custom instructions were not allowed to contain branches or cross control flow boundaries (if-conversion of the code is allowed, however). These restrictions were put in place so that custom instructions can remain stateless and atomic. Both assumptions are due to limitations in the DFG explorer and compile, and do not reflect inherent limitations of the approach.

Sixteen full benchmarks were run through the CFU generation system and 15 sets of CFUs for each benchmark were created. Each set corresponds to an area budget allotted to the CFUs (relative to one 32-bit ripple-carry adder, two adders, etc.). The 16 benchmarks can be divided into four domains: encryption, network, audio, and image. The encryption category contains five benchmarks (blowfish, rijndael, and sha) from MiBench [17] and two other encryption applications (3des and Rc4). The network category consists of three benchmarks (crc, ipchains, and url) from NetBench [27] and the audio (gsmdecode, gsmencode, rawcaudio, and rawdaudio) and image (cjpeg, djpeg, epic, and mpeg2dec) domains are from MediaBench [24].

The baseline processor for the experiments is a four-wide VLIW that can issue one integer, one floating-point, one memory, and one branch instruction each cycle. The instruction set and latencies of each instruction are similar to those of the ARM-7 [33]. In all of our studies, the custom instructions require an integer issue slot to execute, thus an integer operation and a custom instruction cannot execute in the same cycle. This was done so that any speedups observed are due to custom instructions and not from adding parallelism to the processor. A 300 MHz system clock was assumed for timing constraints and custom instructions that require more than one clock cycle to execute are pipelined so as not to affect cycle time. A maximum of four input and two output ports was placed as an external limit on all CFUs generated. Generally speaking, approximately 10-20 custom instructions were needed to attain the maximum speedups presented in the following figures. It is important that this number is small in order to keep the impact on instruction set encoding minimal.

Although not presented in this work, a prototype of this system has been built in the ARM OptimoDE framework [11]. This prototype allowed us to measure the actual die area overhead for adding custom instructions to a processor. While the protoype implementation was fairly naive, we found that custom instructions could be added to a processor for roughly 20 percent additional die area. The majority of this overhead was due to additional control bits that resulted from adding an issue slot for custom instructions. Note that, in our simulations, no issue slot is added and, thus, the overhead for the model used in this paper will likely be much less than the 20 percent reported in [11].

Performance versus Area. The four graphs in Fig. 9 compare the performance gain in each of the four benchmark domains as the total cost budget for CFUs is varied. Each line in the graphs represents the speedup of an application with CFUs designed specifically for it compared to the baseline processor. One of the interesting trends in these graphs is that speedups seen in benchmarks vary greatly. Encryption benchmarks tend to benefit quite a bit from CFUs, with

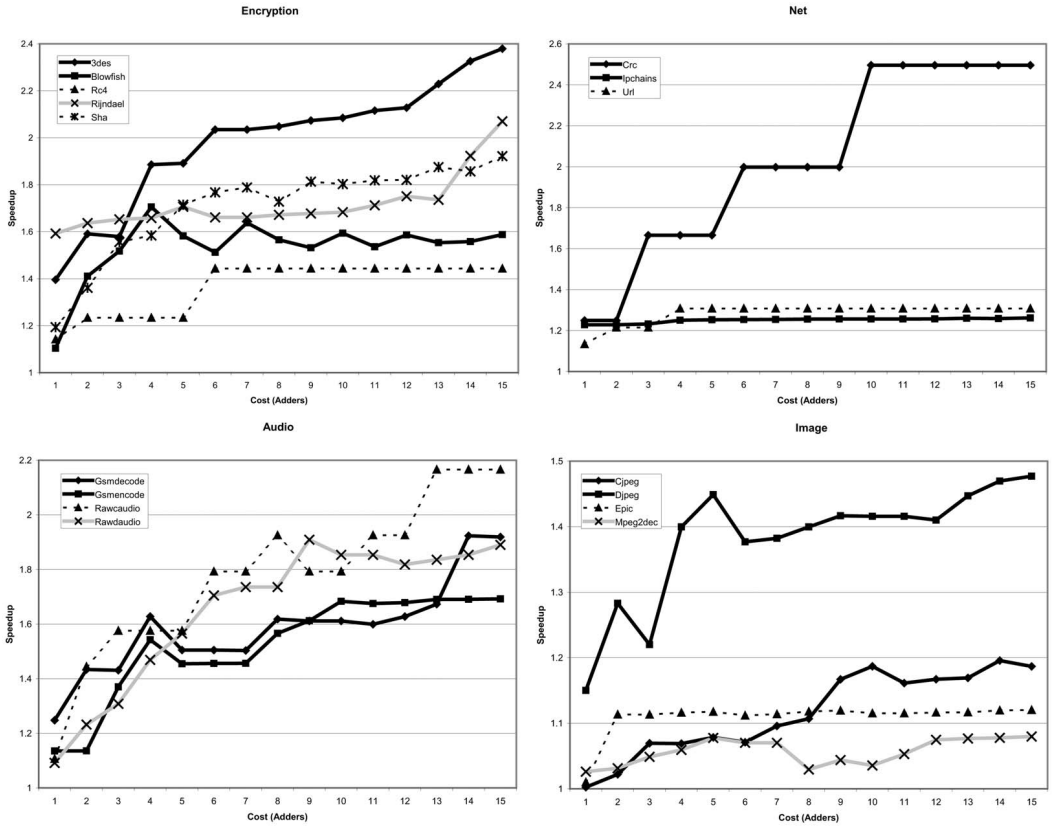

Fig. 9. Performance of four application groups as CFU cost budget is increased from 1 to 15 32-bit adders.

3des, rijndael, and sha showing speedups of 2.39, 2.08, and 1.91, respectively, at the higher cost points. On the contrary, some applications in other domains show very little speedup (e.g., mpeg2dec, epic, and ipchains). Investigation into this revealed that these benchmarks had a significant number of branches and memory operations, which hindered the combinable operations available for the DFG explorer. Conversely, the encryption benchmarks contained large subgraphs dominated by simple arithmetic and logical operations, which are ideally suited for custom hardware.

Another very noticeable trend in Fig. 9 (blowfish and djpeg in particular) is that, at some higher cost points, there is a dip in speedup. This is due directly to the greedy node assignment in the dynamic programming selection heuristic. Recall that, in the proposed selection algorithm (see Section 3.4), when a node appears in multiple candidates, a preselection pass removes that node from all candidates except the one with the largest estimated latency decrease. This assignment saves a great deal of computation during selection, but is just a heuristic and can make bad decisions. For blowfish, a speedup of approximately 1.7 is attained at cost point 4 by assigning several nodes to small and generally useful CFUs. At cost point 5, the heuristic assigned the nodes that used to be in small CFUs to a very large CFU, artificially inflating its value in comparison to the smaller ones. In reality, the compiler was not able to make use of the large CFU as well as the smaller ones and, thus, performance suffered.

Cross Compilation and Generalization. Figs. 10 and 11 show the performance of applications when run with CFUs designed for other applications within the same domain. Two benchmarks are listed for each set of bars; the first one is the application being run and the second one is the application the hardware was designed for. For example, the second set of bars from the left in Fig. 10, 3des-Blowfish, shows the speedup obtained when running 3des on CFUs designed for blowfish. Each bar in these figures uses the CFUs designed at a cost point of 15 adders. The white bars use CFUs that have no generalization, the gray bars utilize wildcarding and subsumed subgraphs, and the black bars have preemptive wildcarding in addition to the subsumed subgraphs. Although only the encryption and audio domains are shown here, the trends in these two figures hold in both the network and image domains.

One interesting pattern in these figures is that when one application does well using another application's CFUs, it does not necessarily mean that the opposite is true. For example, rijndael does well on rc4's CFUs, but rc4 gets almost no speedup from using rijndael's CFUs without generalization.

The most dominant trend in these figures is that, without generalization techniques (i.e., the white bars), most applications do quite poorly when using hardware designed for another application. Rijndael was able to achieve a 1.85 speedup using CFUs designed for rc4, but, apart from that, none of the other performance improvements even begin to approach what was achieved with hardware designed specifically for that application. This result was surprising since applications in the same domain generally have similar DFG structure. The reason this happens is because, while the DFGs are similar, they do not match

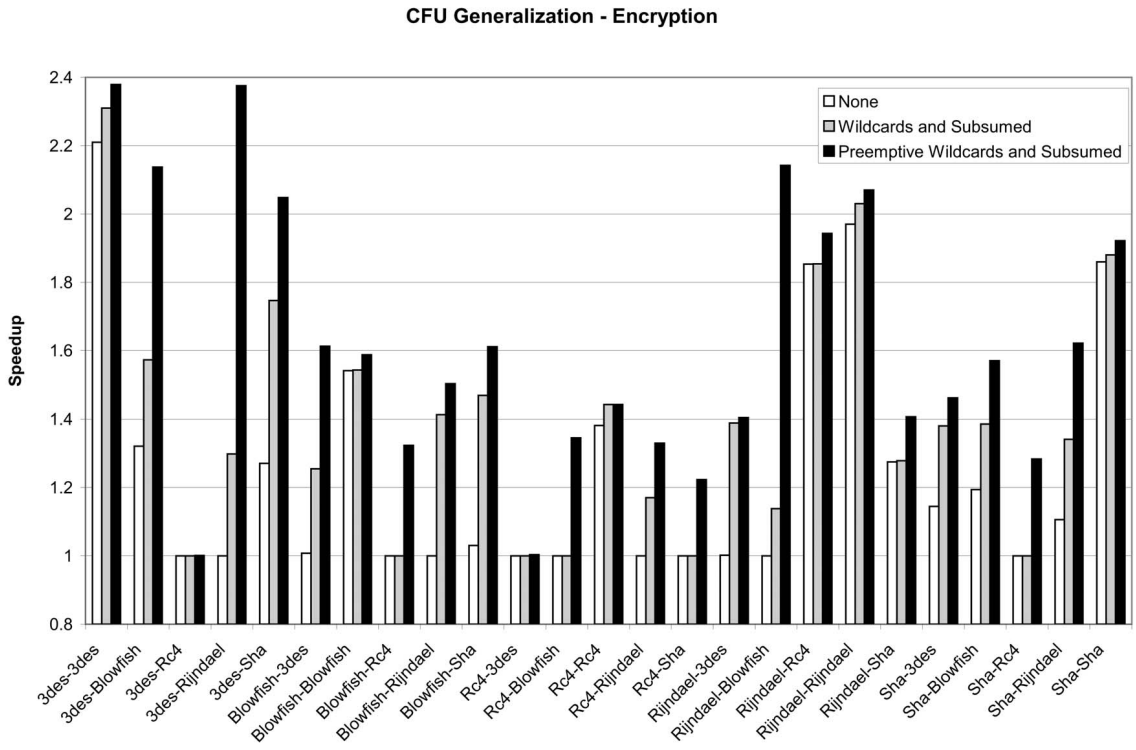

Fig. 10. Effect of subsumed subgraphs and wildcards in Encryption at the 15-adder cost point.

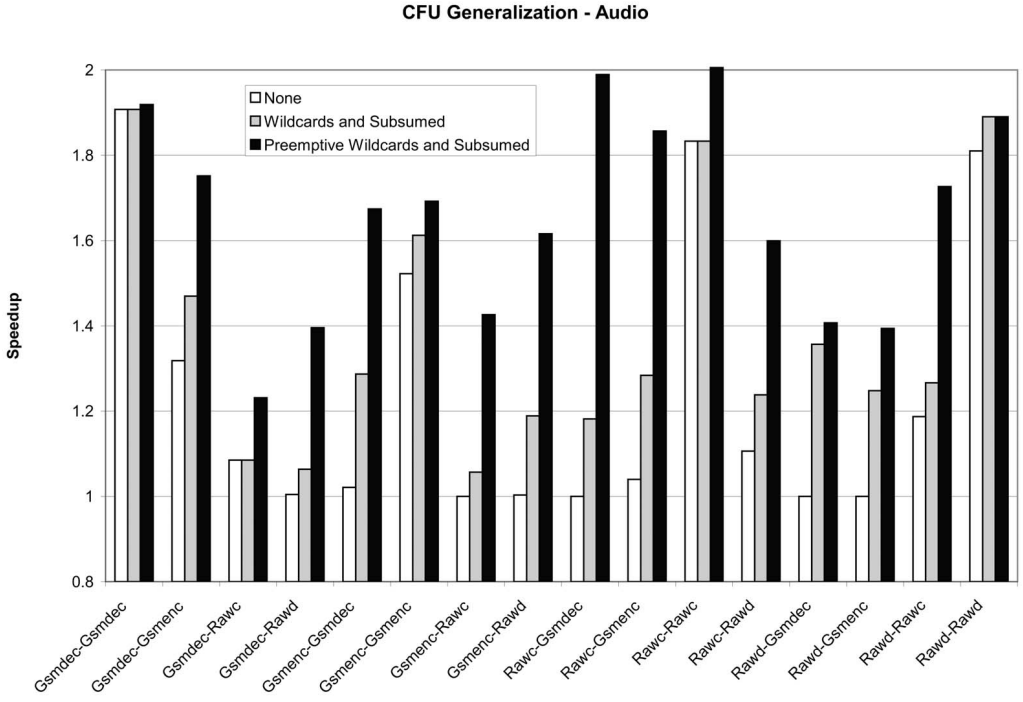

Fig. 11. Effect of subsumed subgraphs and wildcards in Audio at the 15-adder cost point.

exactly. This serves as strong motivation for the use of CFU generalization techniques for domain-specific acceleration.

As CFUs are generalized (moving to the gray and then to the black bars), it becomes obvious that the critical issue to exploiting CFUs across multiple applications is the ability to map multiple subgraphs onto the CFU hardware. Using opcode classes and subsumed subgraphs allows several applications to approach the speedups attained with CFUs designed specifically for them, e.g., rijndael on rc4, 3des on rijndael, and gsmencode on gsmdecode. Most cross compiles show significant speedups when using generalization techniques, which points toward the conclusion that applications within a domain generally have similar DFG structure in the computationally intense portions of their DFGs.

An important point in Figs. 10 and 11 is that the generalization techniques are typically not very useful for

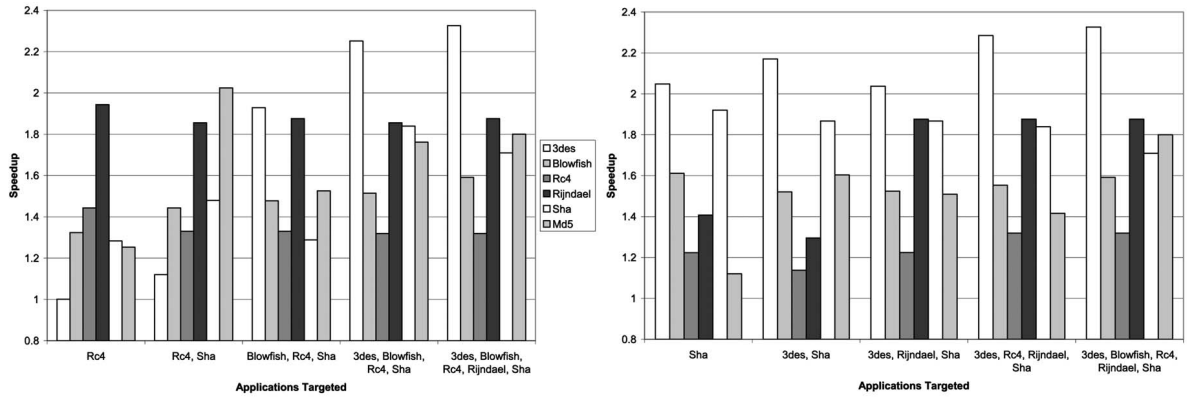

Fig. 12. Effect of targeting multiple applications.

native compiles. For example, gsmdecode shows little improvement from generalization in Fig. 11 on CFUs designed for itself. This is because the CFUs are chosen specifically to handle the most computationally intensive portions of the code, leaving few nodes in important parts of the code available to be utilized by wildcard or subsumed subgraphs. The fact that generalization does not help native compiles provides further evidence that the DFG exploration tool does a good job at finding and selecting appropriate CFUs.

Designing CFUs for multiple applications. An alternative strategy to preemptively generalizing CFUs from one application is to design them with multiple applications in mind. This allows for more certainty that the CFUs designed will work across a domain.

Fig. 12 shows the results of designing CFUs for certain subsets of the encryption domain. The horizontal axis shows which applications the CFUs were designed to target. For example, the middle set of six bars in the left graph shows the speedups of six applications when using CFUs designed for blowfish, rc4, and sha simultaneously, at a cost point of 15 adders. Moving from left to right along the horizontal axis in each graph effectively generalizes the CFUs for the encryption domain since an additional application is used as input at each step. The left and right graphs show two different paths for generalizing the CFUs.

There are three important trends to note in Fig. 12. First, adding applications generally improves average performance. For example, moving from the first set of bars to the second set in the left graph, 3des, blowfish, sha, and md5 all improve performance. Rc4 loses a little performance because, in the first set, the CFUs were designed specifically for that application and, in the second set, part of the area budget is devoted to sha as well. Regardless, the average speedup of the six applications monotonically increases as more applications are taken into account when designing the custom instructions. This is true on the right graph as well.

Second, speedups achieved from the rightmost (domain-wide) set of bars are close to speedups achieved by designing specifically for that application. For example, the speedup for 3des on CFUs designed specifically for it is 2.39 compared to 2.32 on the domain-wide CFUs (rightmost set of bars) and blowfish has a speedup of 1.59 in both the application-specific and domain-wide CFUs used. The reason that these speedups are attainable in the domain-wide CFUs is that the DFGs of these applications are quite similar and the generalization techniques that we have proposed allow for creating hardware that maps to the core computational needs of each application.

The last important trend to note in Fig. 12 is that the md5 application generally improves speedup as the CFUs become more general. Md5 is a program for computing checksums to detect data transmission errors and is similar in structure to encryption algorithms. Since md5 shows good speedups on the domain-wide CFUs and they were not designed with this application in mind, it seems likely that these CFUs will be effective on next generation encryption applications.

6 Conclusion

Application-specific instruction set extensions are an efficient way to meet the growing performance and power demands of embedded applications. Designing these extensions has traditionally been very user intensive as an architect must determine what would make a good extension and manually insert intrinsics into the code to make use of these extensions. In this paper, we have presented a system that automates this process. Using an efficient dataflow graph exploration heuristic, we are able to discover and automatically select custom function units to meet the demands of an application. We have also demonstrated how a compiler can make use of these custom function units in any application and how to increase their utility through simple generalization techniques.

Our system has demonstrated significant speedups for several applications, with as much as 2.39 for 3des and an average of 1.69, while utilizing modest additional die area. We have shown that, typically, exact subgraph matches do not occur across applications in a domain, but, by using simple generalization techniques (wildcards and subsumed subgraphs), cross-application utilization can be substantially improved. Additionally, we have shown that designing custom instructions with several applications in mind at one time is an effective way to achieve the goals of generalization and to design with future algorithms in mind.

Acknowledgments

The authors thank Krisztián Flautner, Koen Van Nieuwenhove, and Michael Chu for their insightful feedback on this work. This research was supported in part by ARM

Limited, US National Science Foundation grants CCR-0325898 and CCF-0347411, and equipment donated by Hewlett-Packard and Intel Corporation.

REFERENCES

[1] A. Aho et al., "Code Generation Using Tree Pattern Matching and Dynamic Programming," *ACM Trans. Programming Languages and Systems,* vol. 11, no. 4, pp. 491-516, Oct. 1989.

[2] A. Alomary et al., "PEAS-I: A Hardware/Software Co-Design System for ASIPs," *Proc. European Design Automation Conf.,* 1993.

[3] M. Arnold, "Instruction Set Extensions for Embedded Processors," PhD thesis, Delft Univ. of Technology, 2001.

[4] K. Atasu et al., "Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints," *Proc. 40th Design Automation Conf.,* June 2003.

[5] P.M. Athanas et al., "Processor Reconfiguration through Instruction Set Metamorphosis," *Computer,* vol. 18, no. 11, Nov. 1993.

[6] M. Baleani et al., "HW/SW Partitioning and Code Generation of Embedded Control Applications on a Reconfigurable Architecture Platform," *Proc. Workshop Hardware/Software Codesign,* pp. 61-66, May 2002.

[7] J.P. Bennett, "A Methodology for Automated Design of Computer Instruction Sets," PhD thesis, Univ. of Cambridge, 1988.

[8] P. Bose and E.S. Davidson, "Design of Instruction Set Architctures for Support of High-Level Languages," *Proc. Int'l Symp. Computer Architecture,* June 1984.

[9] P. Brisk et al., "Instruction Generation and Regularity Extraction for Reconfigurable Processors," *Proc. Int'l Conf. Compilers, Architectures, and Synthesis for Embedded Systems,* pp. 262-269, 2002.

[10] H. Choi et al., "Synthesis of Application Specific Instructions for Embedded DSP Software," *IEEE Trans. Computers,* vol. 48, no. 6, pp. 603-614, June 1999.

[11] N. Clark et al., "OptimoDE: Programmable Accelerator Engines through Retargetable Customization," *Proc. HotChips 16,* 2004.

[12] N. Clark, H. Zhong, and S. Mahlke, "Processor Acceleration through Automated Instruction Set Customization," *Proc. Int'l Symp. Microarchitecture,* pp. 129-140, Dec. 2003.

[13] J. Cong et al., "Application-Specific Instruction Generation for Configurable Processor Architectures," *Proc. Int'l Symp. Field Programmable Gate Arrays,* pp. 183-189, 2004.

[14] L. Cordella et al., "Performance Evaluation of the VF Graph Matching Algorithm," *Proc. Int'l Conf. Image Analysis and Processing,* vol. 2, pp. 1038-1041, 1999.

[15] R.E. Gonzalez, "Xtensa: A Configurable and Extensible Processor," *IEEE Micro,* vol. 20, no. 2, pp. 60-70, Mar. 2000.

[16] M. Gschwind, "Instruction Set Selection for ASIP Design," *Proc. Workshop Hardware/Software Codesign,* May 1999.

[17] M.R. Guthaus et al., "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *Proc. IEEE Fourth Workshop Workload Characterization,* Dec. 2001.

[18] J.R. Hauser and J. Wawrzynek, "GARP: A MIPS Processor with a Reconfigurable Coprocessor," *Proc. Symp. Field-Programmable Custom Computing Machines,* Apr. 1997.

[19] B. Holmer, "Automatic Design of Computer Instruction Sets," PhD thesis, Univ. of California, Berkeley, 1993.

[20] E. Horowitz and S. Sahni, "Exact and Approximate Algorithms for Scheduling Nonidentical Processors," *J. ACM,* vol. 23, no. 2, pp. 317-327, 1976.

[21] I. Huang and A.M. Despain, "Synthesis of Application Specific Instruction Sets," *IEEE Trans. Computer Aided Design,* vol. 14, no. 6, June 1995.

[22] G. Karypis et al., "Multilevel Hypergraph Partitioning: Applications in VLSI Domain," technical report, Univ. of Minnesota, 1997.

[23] R. Kastner et al., "Instruction Generation for Hybrid Reconfigurable Systems," *ACM Trans. Design Automation of Electronic Systems,* vol. 7, no. 4, Apr. 2002.

[24] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *Proc. Int'l Symp. Microarchitecture,* Dec. 1997.

[25] R. Leupers and P. Marwedel, "Instruction Selection for Embedded DSPs with Complex Instructions," *Proc. European Design Automation Conf.,* Sept. 1996.

[26] S. Liao et al., "Instruction Selection Using Binate Covering for Code Size Optimization," *Proc. Int'l Conf. Computer Aided Design,* pp. 393-399, 1995.

[27] G. Memik, W.H. Mangione-Smith, and W. Hu, "NetBench: A Benchmarking Suite for Network Processors," *Proc. Int'l Conf. Computer Aided Design,* pp. 39-43, 2001.

[28] K.V. Palem, S. Talla, and W.-F. Wong, "Compiler Optimizations for Adaptive EPIC Processors," *Proc. ACM Conf. Embedded Software,* pp. 257-273, 2001.

[29] A. Peymandoust et al., "Automatic Instruction Set Extension and Utilization for Embedded Processors," *Proc. Int'l Conf. Application-Specific Systems, Architectures, and Processors,* June 2003.

[30] J.V. Praet et al., "Instruction Set Definition and Instruction Selection for ASIPs," *Proc. Int'l Symp. High Level Synthesis,* 1994.

[31] D.S. Rao et al., "On Clustering for Maximal Regularity Extraction," *IEEE Trans. Computer Aided Design,* vol. 12, no. 8, Aug. 1993.

[32] R. Razdan and M.D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Function Units," *Proc. Int'l Symp. Microarchitecture,* pp. 172-180, Dec. 1994.

[33] D. Seal, *ARM Architecture Reference Manual.* Addison-Wesley, 2000.

[34] F. Sun et al., "Synthesis of Custom Processors Based on Extensible Platforms," *Proc. Int'l Conf. Computer Aided Design,* Nov. 2002.

[35] Trimaran, "An Infrastructure for Research in ILP," http://www.trimaran.org, 2003.

[36] M.J. Wirthlin and B.L. Hutchings, "DISC: The Dynamic Instruction Set Computer," *Proc. IEEE Symp. FPGAs for Custom Computing Machines,* pp. 92-103, 1995.

[37] L. Wu, C. Weaver, and T. Austin, "Cryptomaniac: A Fast Flexible Architecture for Secure Communication," *Proc. Int'l Symp. Computer Architecture,* pp. 110-119, June 2001.

[38] Z.A. Ye et al., "CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit," *Proc. Int'l Symp. Computer Architecture,* pp. 225-235, 2000.

Nathan T. Clark received both the BSE and MSE degrees in computer engineering from the University of Michigan. He is a PhD candidate in the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor. His research interests mainly include designing and compiling for embedded architectures. He is a student member of the ACM.

Hongtao Zhong received the BS and MS degrees from Tsinghua University, China, both in computer science and engineering. He 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 distributed VLIW architectures and compilers.

Scott A. Mahlke received the PhD degree in electrical and computer engineering from the University of Illinois at Urbana-Champaign. He is the Morris Wellman Assistant Professor of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor. He directs the Compilers Creating Custom Processors research group, which focuses on the design of application specific processors and hardware accelerators. His research interests include compilers, computer architecture, and high-level synthesis. He is a member of the IEEE and the ACM.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.