Processor Acceleration Through Automated Instruction Set Customization

Nathan Clark Hongtao Zhong Scott Mahlke

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

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 co-processors), and the corresponding instructions, are added to a baseline processor to meet the critical computational demands of a target application. The central challenge with this approach is the large degree of human effort required to identify and create the custom hardware units, as well as porting the application to the extended processor. In this paper, we present the design of a system to automate the instruction set customization process. A dataflow graph design space exploration engine efficiently identifies profitable computation subgraphs from which to create custom hardware, without artificially constraining their size or shape. The system also contains a compiler subgraph matching framework that identifies opportunities to exploit and generalize the hardware to support more computation graphs. We demonstrate the effectiveness of this system across a range of application domains and study the applicability of the custom hardware across the domain.

1. INTRODUCTION

In recent years, the markets for PDAs, cellular phones, digital cameras, network routers and other high performance but special purpose devices has 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 special-purpose ASICs coupled with a low cost core processor, such as an ARM [29]. 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 post-programmability.

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 co-processor subsystems. The Tensilica Xtensa is an example commercial effort in this area [12].

There are a number of benefits to augmenting the instruction set of a core processor. First, the system is post-programmable 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 computation 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.

The 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. The current Xtensa system places much of this burden on the user to define, implement, and exploit the customized processor.

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. Compiler exploitation of the custom instructions is accomplished through a flexible subgraph matching engine. Applications are analyzed to identify the maximal computation subgraphs that can be replaced by custom instructions. Techniques are also used to allow for maximal mapping of subgraphs to each hardware unit.

Many other researchers have proposed systems to accomplish the task of automated instruction set generation. The contributions of this paper are threefold. First, we present a

Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36 2003)
0-7695-2043-X/03 $17.00 © 2003 IEEE

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 machine with custom instructions. Finally, 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.

2. RELATED WORK

A large body of research has gone into instruction set customization. Work in [5], [28], [32], [16], [24], [14], and [33] all showed possible gains using this technique. While these works show the potential utility of this 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 sub-problems: 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 $O(2^{\#\ ops})$ potential candidates. Several techniques have been proposed to handle the intractability of this problem.

Early work [2] side-stepped 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 [27], [26], [6], [9], and [19], 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 run times (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 [17], which was later extended by [18]. This technique discovered candidates by performing an initial grouping of nodes based on the schedule time in the DFG, and then iteratively breaking and recombining these groups. Work by Bose [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 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 [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 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] 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 or in the presence of optimizations that create large basic blocks, such as loop unrolling.

More recent work by Goodwin [13] searches all possibilities connected in a dataflow graph when generating "fused instructions". In order to avoid intractability, the search is subject to restrictions that each opcode can have at most two inputs, one output, and the resultant opcode take only one cycle to execute. We have observed in our experiments that using very strict restrictions, such as these, generally produces poor results. This technique also explores the option of adding vector operations exposed through loop unrolling.

Some researches have proposed heuristic ways to limit the search space without artificially constraining it. In [3], the least used half of all candidates are 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 [30] 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's work, candidates that do not meet a certain percentage of the best discovered candidate so far are removed. Methods such as this have the benefit of not artificially constraining the problem by potentially getting

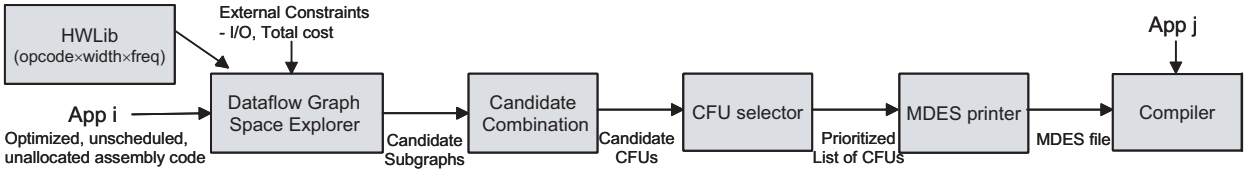


Figure 1: Organizational structure of the hardware compiler.

stuck in local minima or limiting the types of candidates discovered.

Utilization - How to make use of the candidates is another issue to deal with in 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. [18]. 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 [21] that tree covering methods can yield suboptimal results, particularly in the presence of irregular instructions common in custom instructions. To overcome this, [21] proposes splitting all instructions into "register-transfer" primitives and recombining the primitives in an optimal manner using integer programming. Work by Liao [22] 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 worst case exponential runtime, so they must be selectively used.

Research in [25] 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 [30], but is extended to avoid potential shortcomings discussed in Section 3. In Section 4, the custom instruction utilization framework implemented in this paper is described. This framework ties together several ideas from other work into one system, and addresses some runtime issues with previously proposed solutions. In Section 5 the custom instructions generated by our system are applied to applications across several domains, and the results of these experiments are carefully analyzed. Analysis like this has never been done before, to our knowledge.

3. DATAFLOW GRAPH EXPLORATION

The overall structure of our DFG exploration engine is shown in Figure 1. An application is 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 numbers to the space explorer so 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 allows us to calculate an estimate of performance gain by using the profile weights of all the set members. The combination stage also performs some checks to determine which CFUs can potentially be subsumed by other CFUs. All of this information is passed to a selection mechanism that determines which CFUs best meet the needs of this application. Finally, the prioritized list of CFUs is converted in a machine description (MDES) form that can be fed to the compiler.

Throughout this section, the DFG shown in Figure 2 from the *blowfish* application [15] is used for illustrative purposes. For simplicity, each operation or node is assumed to take 1 cycle to execute in the baseline processor.

3.1 Subgraph Selection

A CFU is loosely defined as the hardware implementation of a subset of primitive operations from an application's

DFG. Primitive operations are atomic assembly operations for a generic RISC architecture, such as Add, Or, and Load. These operations correspond to nodes in the DFG. No assumptions are made regarding the connectivity of the nodes in a CFU, so linear, tree shaped, or even cyclic graphs can be implemented as CFUs. However, we consider only connected subgraphs.

Implementing subsets of the DFG as CFUs 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 CFUs is a difficult problem. The most glaring difficulty is that there are an exponential number of potential candidates to select as CFUs for a given DFG. Exploration starts by examining each node in the DFG and using it as a seed for a candidate subgraph. Initially, the system used a naïve implementation that looked at all possible directions to grow the seed nodes. The number of candidate subgraphs quickly grows out of control with sufficiently loose external constraints using this approach.

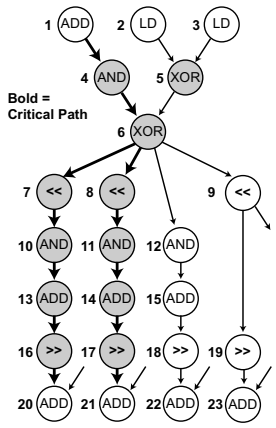
The key observation gained from experimenting with this naïve approach is that the majority of candidates examined by exponential growth simply do not make sense. For example, assuming the goal is maximizing performance on the DFG in Figure 2, CFU 6-9 (i.e. the CFU containing nodes 6 and 9) has little value, because node 9 is not on the critical path. To avoid searching these useless candidates, we propose using a *guide function* to rank which nodes are the best directions to grow. The guide function allows heuristic determination of the merit of each growth direction, and arbitrary control on the fanout from seeds. Allowing a larger number of candidates from each seed, or large fanout, will ensure better coverage of the design space, while allowing smaller fanout will result in reduced run times and memory consumption.

One important part of our technique is that restricting fanout enables more efficient design space exploration. For example, higher fanout could be used in blocks that have higher profile weight, as they are more likely to yield important candidates; alternately, higher fanout could be used at the initial levels of the search and then more tightly constrain the number of growth directions as the candidates increase in size. All previously proposed solutions use a single exploration strategy for all parts of the application, where as this technique can modify its strategy to effectively avoid searching likely useless portions of the design space.

Using a guide function to restrict growth is most similar to the work by Sun [30]. They used a priority function to prune candidates which do not reach a certain percentage of the best priority discovered so far. The candidates that are not pruned are grown in every direction. The guide function proposed here prunes directions of search, not candidates. If the guide function finds no directions worthy of growing a candidate, it is equivalent to that candidate being pruned. The benefit of pruning directions as opposed to pruning candidates is that there is always the possibility that a low ranking candidate will grow into a useful one.

3.2 Guide Function

The purpose of the guide function is to intelligently rank




```
m ^= p [0];
r ^= p[ 1];
r ^= (((s[(m>>24L)] +
        s[0x0100+((m>>16L)&0xff)]) ^
        s[0x0200 + ((m>> 8L)&0xff)]) +
        s[0x0300+((m )&0xff)]) & 0xffffffff;
m ^= p[ 2];
m ^= (((s[(r>>24L)] +
         s[0x0100+((r>>16L)&0xff)]) ^
         s[0x0200+((r>> 8L)&0xff)]) +
         s[0x0300+((r )&0xff)]) & 0xffffffff;
```

(b)

Opcode	Adders	Cycles
+	1.0	0.30
AND	0.12	0.06
<<, >>*	0.01	~ 0
XOR	0.16	0.09

* These are shifts by compile time constants

(c)

Figure 2: A) Sample DFG from blowfish. Shaded nodes delineate a CFU. B) Preprocessed C code. C) Excerpt from the hardware library.

which growth directions will create the best candidates. The guide function essentially tries to replace the architect by making design decisions, thus its decisions must reflect the same desirable properties the architect would strive for. The guide function proposed here uses four categories to rank the desirability of growth directions: criticality, latency, area, and input/output. Giving each of these categories different weights toward the overall score of the guide function will greatly affect the types of candidates that are generated. Many experiments have been performed varying the weights of each of these factors and they point to the general conclusion that evenly balancing the factors yields the best candidates.

In the DFG space explorer, each of the guide function categories is allotted 10 points of weight, and the sum of these categories determines the total desirability of each candidate direction. If a direction receives fewer than half of the total desirability points, then it is considered a bad direction and it will not be explored. This is not to say that half of the directions will be ignored, merely that directions with less than half of the points are not worth investigating.

Criticality - This category rewards candidate directions when they appear on the critical path (longest dependence path(s)) of a DFG. CFUs that occur on the critical path are likely to give the application performance improvement, which is typically the most desired result of CFUs. An example of this from Figure 2 would be investigating ways to grow candidate 4-6. The direction including nodes 1, 7, or 8 would rank higher in terms of criticality than would the direction of node 5, 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, node 1 would get $\frac{10}{0+1} = 10$ points and node 9 would get $\frac{10}{2+1} = 3.33$ points. Note that it is important to give candidate directions 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.

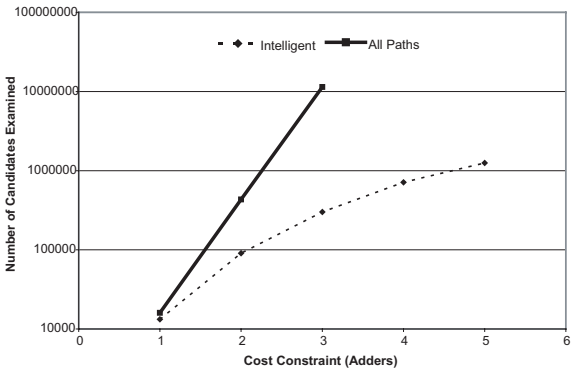


Figure 3: Candidates examined for blowfish.

Latency - Latency tries to guide exploration toward combining operations which require fewer cycles to execute when combined into a CFU than they do as standalone operations. The largest performance gains are possible by combining low latency operations, such as logicals, where many can be executed in a single cycle. 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 each atomic operation (see Figure 2c) along the critical path of the candidate subgraph. Using candidate 4-6 on Figure 2 as an example again, note that currently these operations can be executed back to back in 0.15 cycles. Exploring the direction of node 1, which has a latency of 0.3 cycles, would get $\frac{0.15}{0.15+0.30} * 10 = 3.3$ points. In contrast, growing toward node 10, we would get nearly all ($\frac{0.15}{0.15+0} * 10 = 10$) 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 in the CFU (see Figure 2c). Note that register file ports are a design constraint, thus they do not factor into the area. Further, CFUs do require additional decode logic and interconnect, but CFU area is generally the dominant term. The area category gives more points to directions that least increase the total area of the candidate. Area points are calculated the same way as latency, $\frac{old\ area}{new\ area} * 10$, except that the old and new areas are rounded up to the nearest half adder (that is a cost of 0.49 or 0.01 adders becomes 0.5). Rounding is done so as not to penalize operations unfairly when the seed is very small. Consider the case of growing candidate 9-19. If no rounding was done, then growing to node 23 would only yield $\frac{0.02}{1.02} * 10 = 0.2$ points and growing to 6 would only yield $\frac{0.02}{0.18} * 10 = 1.1$ points. This does not mean that growing toward node 6 is bad decision from an area standpoint, however.

Input/Output - The maximum number of input and output operands for a CFU is limited. Register file ports are generally fixed on the core processor based on cost, power, cycle time, and encoding constraints. Thus, the maximum number of input and output operands available is treated as design constraint. The purpose of the I/O category is to guide the search in directions that reduce or keep constant the number of inputs and outputs to the candidate. Giving preference to directions that do not increase I/O facilitates discovering larger subgraphs that still meet I/O constraints. Points are awarded based on the number of input and output ports using the equation $MIN(\frac{old\ \#\ ports}{new\ \#\ ports} * 10, 10)$. Taking the minimum of the two terms in the equation is done because the number of ports may be reduced by growing certain directions due to reconvergence points in the DFG. As an example of this calculation, if directions from candidate 8-11 from Figure 2 were examined, growing toward node 14 would not increase the number of inputs or outputs, yielding $\frac{2}{2+1} * 10 = 6.7$ points. Growing toward node 6 would increase both the number of inputs and outputs, yielding $\frac{2}{4+1} * 10 = 4$ points.

With the guide function heuristic in place, it was important to verify two points: that the heuristic does indeed prune the search space, and that good candidates are not missed because the guide function incorrectly dismisses them. Figure 3 demonstrates the first point. The intelligent heuristic is able to effectively curve the exponential growth associated with the DFG exploration problem. This algorithm can be used on very large code segments and without artificially constraining the types of candidates generated, which are both weaknesses of previously proposed algorithms. To ensure that good candidates are not dismissed, the heuristic was compared against a full exponential search for several small benchmarks. The results showed that both approaches selected identical sets of candidates. The heuristic was also compared against full exponential search using restricted constraints (3 input, 2 output ports and a five adder maximum cost) on larger benchmarks. Again, the results found using the heuristic were comparable with those of full exponential search.

3.3 Candidate Combination

After discovery, it is a straightforward process to group identical candidate subgraphs together into candidate CFUs. A simple test which 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 Figure 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. Using a compiler instruction scheduler to get an exact measurement is possible, but the complexity makes this solution undesirable and the estimate has proved reasonably accurate.

After candidate grouping, there are two passes over the list of CFUs. The first pass records which CFUs can be subsumed by others. 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. This is similar to the use of symbolic algebra described in [25]. Using Figure 2 as an example, if CFU "AND-ADD->>" was discovered, CFU "AND->>" can be executed on the same hardware because the subsumed hardware could set one input of the ADD operation to 0. CFUs "AND-ADD" and "ADD->>" would also be recorded as

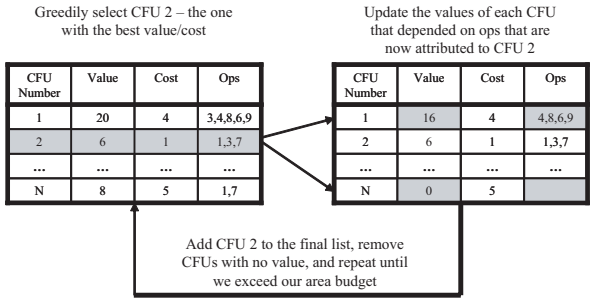


Figure 4: Greedy approach to CFU selection.

being subsumed by "AND-ADD->>".

The second pass records wildcard options for each CFU. Wildcards are CFUs with identical subgraphs except for different operations at one node. Combining two CFUs with similar structure like this allows us to cheaply add another CFU without greatly increasing the associated cost, as much of the hardware can be shared between the two CFUs.

3.4 Candidate Selection

Contrary to combining the candidate subgraphs, CFU selection is not straightforward. Selection is similar to the 0/1 knapsack problem. There is a set of resources (the CFUs) that all have values (the estimated cycle savings) and weights (die area), and the goal is to maximize the total value for a given weight. It is widely known that the 0/1 knapsack problem is NP-complete, although it is solvable in pseudo-polynomial 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 Figure 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 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.

The strategy used for CFU selection is a simple greedy method, illustrated in Figure 4. 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 Figure 4, 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. Custom instruction replacement in the compiler happens in the same order that CFUs are selected, so iteratively updating the values by invalidating selected operations maintains the relative accuracy of the cycle gain estimations.

Because the selection heuristic is greedy, it is not guaranteed to give an optimal solution, and quite 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. The dynamic programming heuristic generally does better (roughly 5 - 10% on average) than greedy solutions, however it suffers from a much slower runtime, and thus was not used in any of the studies in this paper.

Dealing with wildcards and subsumed subgraphs adds another challenge to the selection process. The main issue is whether to count all the subsumed subgraphs and wildcards when determining the estimated value of a CFU. If so, 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 them based on all the operations in the subsumed or wildcard candidate subgraphs. This creates a large computational overhead for every selection. In the case of subsumed subgraphs, this means frequently attributing operations to small subsumed portions of a large CFU, when much more performance could have been gained by attributing them to a separate CFU. For example, if the gray shaded CFU in Figure 2 was selected then it would be possible to execute 12-15 on it. However, a CFU for 12-15-18 could have been chosen later, which would have helped performance more. The case just described occurs quite frequently, so CFUs are selected as if they had no subsumed subgraphs or wildcards. When a selections is made, the costs of the subsumed subgraphs and wildcards are updated to reflect that they can now be added for very little overhead.

4. COMPILER UTILIZATION

The purpose of the compiler is to automatically exploit CFUs in a given application for maximal gain. The basic structure of our retargetable compiler is shown in Figure 5. Applications are run through a front-end, producing a generic RISC assembly code. The assembly code is unscheduled and uses virtual registers. The compiler also uses a machine description, or MDES, to determine what CFUs 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,

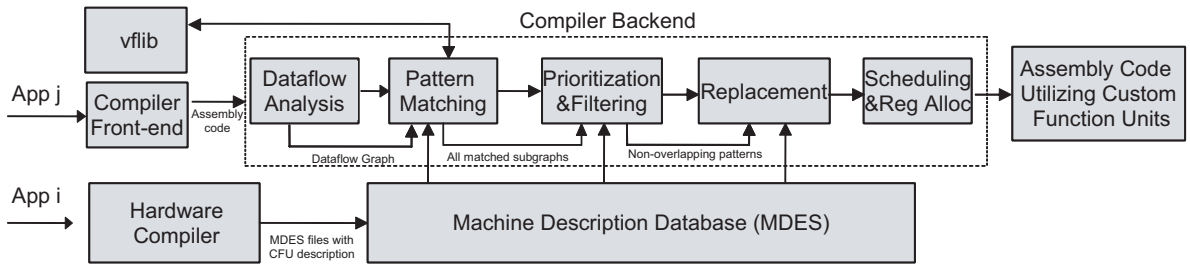


Figure 5: Organizational structure of the compiler supporting custom instructions.

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 [11] 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 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 our DFG space exploration.

Figure 6 shows part of a DFG that is similar to one in the sha benchmark [15]. Given a CFU to implement the operations in subgraph 2-5-6, the pattern matcher would begin by looking at all << nodes: 2, 14, and 16. These partial matches would then be grown to 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. The vflib continues this process 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, it is all right for the same operation to appear in multiple subgraph matches. The hardware compiler provides a desirability ordering on the CFUs so that each operation is only assigned to the CFU that the hardware compiler thinks can make the best use of it. The assignment of operations to CFUs is done in the prioritization and filtering phase. This static desirability ordering can potentially lead to a suboptimal generated code. Solutions proposed in [21] and [22] have worst case exponential run times, however, and were prohibitively slow when implemented in our compiler.

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 Figure 6, subgraph 2-5-6 will be replaced with a custom instruction. The question that arises is, "Where should the 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

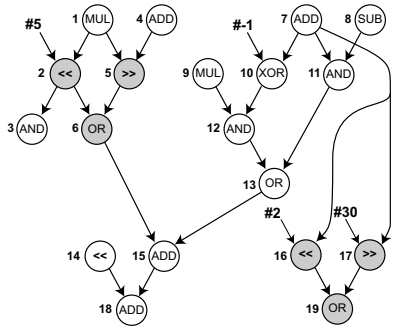


Figure 6: DFG similar to one from sha.

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 operation. 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 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 whatever CFUs are available.

5. EXPERIMENTAL RESULTS

The system proposed was constructed as part of the Trimaran research infrastructure [31]. 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 a popular 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 non-deterministic 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. 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 compiler, and do not reflect inherent limitations of the approach.

Thirteen benchmarks were run through the CFU generation system and fifteen sets of CFUs for each benchmark were created. Each set corresponds to an area budget allotted to the CFUs (relative to one adder, two adders, etc.). The thirteen benchmarks can be divided into four categories: encryption, network, audio, and image. The encryption category contains three benchmarks (blowfish, rijndael, and sha) from MiBench [15], the network category consists of three benchmarks (crc, ipchains, and url) from NetBench [23], and the audio (gsmdecode, gsmencode, rawcaudio, and rawdaudio) and image (cjpeg, djpeg, and mpeg2dec) categories are from MediaBench [20].

The baseline machine 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 [29]. In all of our studies, the CFUs require an integer issue slot to execute, thus an integer operation and a CFU 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 machine. A 300 MHz system clock was assumed for timing constraints, and CFUs that require more than one clock cycle to execute are pipelined so as not to affect cycle time. Cost of the CFUs is measured in die area with respect to a 32 bit ripple-carry adder. A maximum of five input and three output ports was placed as an external limit on all CFUs.

Performance Versus Area: The left four graphs in Figure 7 compare the performance gain in each of the four benchmark categories as the total cost of CFUs is varied. Each line in the graphs represents the speedup of an application with CFUs as compared to the baseline machine. 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 rijndael, blowfish, and sha showing speedups of 1.87, 1.62, and 1.33, respectively, at the higher cost points. On the contrary, several applications in other domains show very little speedup (e.g. mpeg2dec 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.

To further illustrate this point, a limit study was performed to determine the performance improvement attainable for each benchmark given infinite register file ports, an infinite area budget, and the simplifying assumptions mentioned at the beginning of this section. When compared against the cost point of 15 adders in Figure 7, our system realizes speedups very close to the ideal case. This is particularly true with the audio and image benchmarks which show very little speedup.

The exceptions to this are djpeg and cjpeg, where very large CFUs are necessary to achieve the speedup limit. For example, given infinite resources, the system created a CFU for djpeg requiring 24 register file read ports and having an area greater than 8 multipliers. Discrepancies between the theoretical and realized speedups in other applications can be explained similarly. With no limits, the system would create a CFU for blowfish using 80+ register read ports, 40+ register write ports, and containing almost 200 primitive operations. All of this data supports the conclusion that the DFG exploration heuristic makes reasonable decisions when selecting candidates and the compiler effectively uses them. At the same time, the data provides strong motivation to loosen the current restrictions on our system in order to achieve better speedups in future work.

Another very noticeable trend in Figure 7 (djpeg in particular) is that at some cost points there is a large dip in speedup. This is due directly to performance estimations and the greedy selection heuristic. For rawdaudio, a speedup of approximately 1.7 is attained at cost point 5, by using several small and generally useful CFUs. At cost point 6 the heuristic chose one very large CFU, which was estimated to be more useful than the small ones, and the compiler was not able to make use of the large one as well as the smaller ones. Once the cost budget rises, the greedy selection begins to include the small CFUs used at point 5 along with the large one used at point 6, thus the speedups improve beyond what was possible at point 5.

Cross Compilation: To examine the generalizability of custom hardware across multiple applications from the same domain, a set of cross compilation experiments were performed. The four graphs on the right side of the Figure 7 show the speedups for each application compiled on the CFUs for the other applications in the same domain. For example, in the top right graph, rijndael-blowfish is the curve created by compiling rijndael using the CFUs generated for blowfish.

The most noticeable trend is that no application does quite as well on hardware designed for another application as it does for its own. There are several cases where good speedups were attained using another application's CFUs, however. Rijndael shows a 1.52 speedup on blowfish's CFUs, and rawdaudio shows 1.63 speedup on rawcaudio's CFUs. Note that when one application does well using another application's CFUs, it does not necessarily mean that the opposite is true. For example, djpeg does well on cjpeg's CFUs, but cjpeg gets almost no speedup from using djpeg's CFUs. The results show that some of the gains can be preserved across the domains, but that the CFUs lack enough generality to support their widespread use in other applications.

CFU Extensions: The critical issue to exploiting CFUs across multiple applications is the ability to generalize both

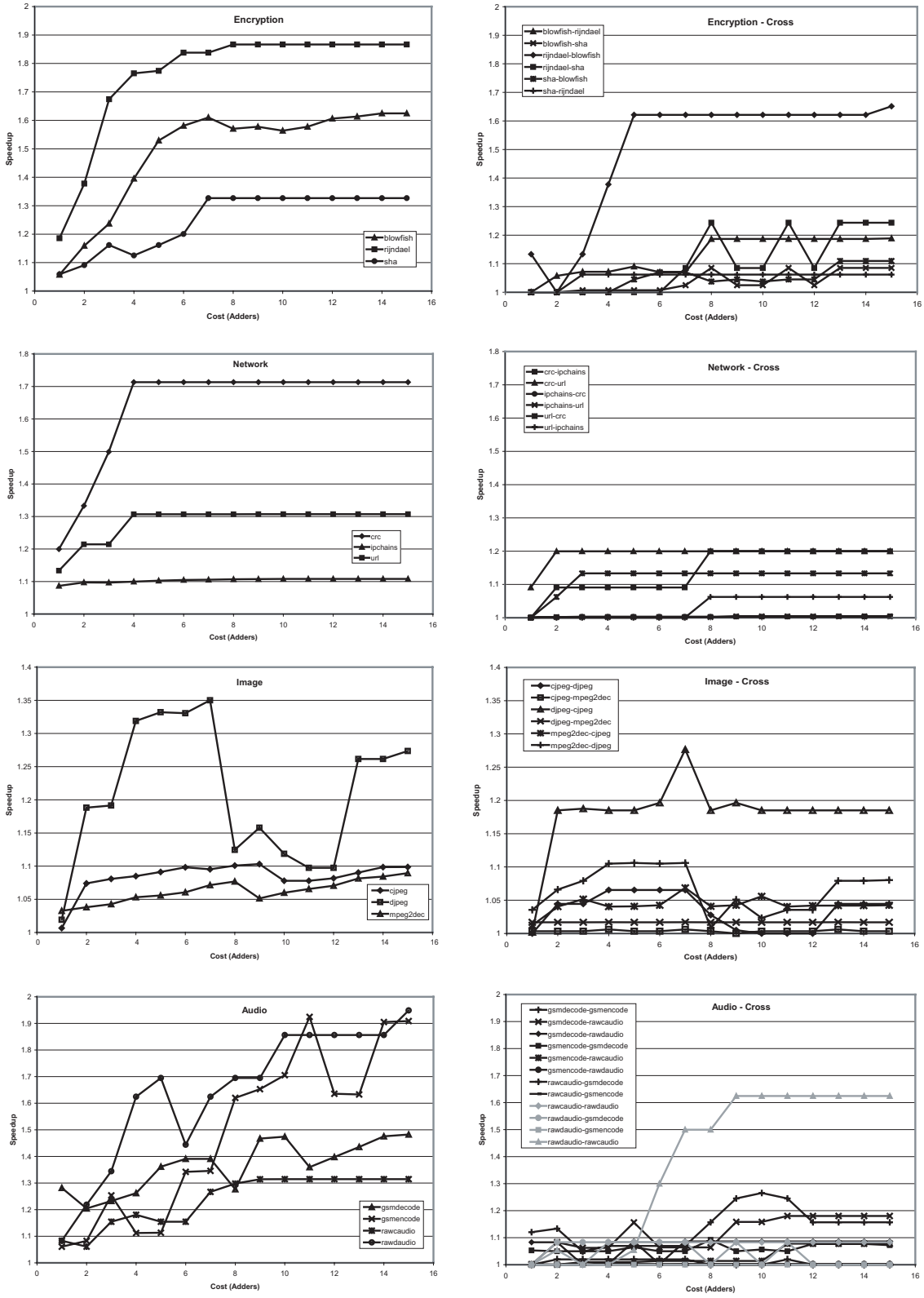


Figure 7: Speedups for various applications on different sets of CFUs.

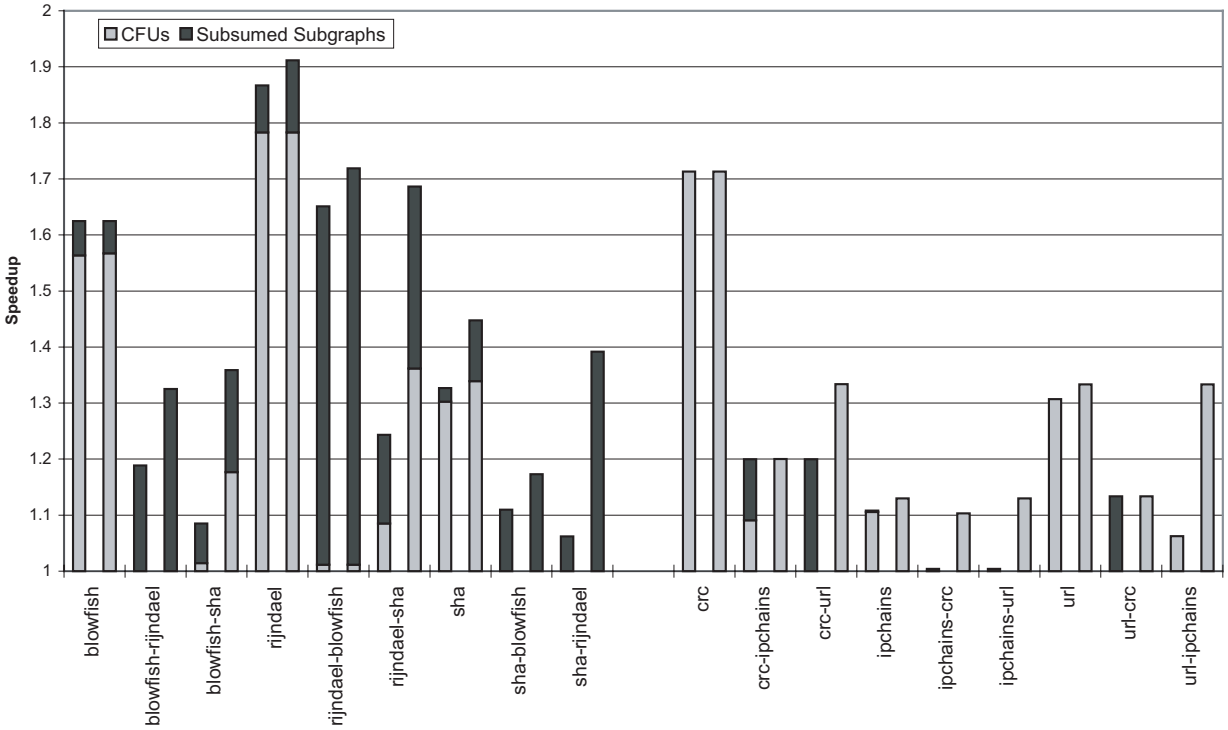


Figure 8: Effect of subsumed subgraphs and wildcards in Encryption and Network at the 15-adder cost point.

the CFU hardware and the compiler matching algorithm. The hardware can be generalized by making the CFUs more multi-functional (wildcards from Section 3). The compiler matching algorithm can be generalized by allowing subgraphs subsumed by the CFU to map onto that CFU (subsumed subgraphs from Section 3). Figures 8 and 9 examine the effects of these issues. The figures show the speedups attained at cost point 15 (adders) for each combination of application and CFUs.

The effect of compiler generalization is shown by comparing the grey and black portions of each bar. The grey portion of the bars is the speedup attributable to exact subgraph matches, and the black portion is the additional gain resulting when subsumed subgraphs are allowed to map onto a CFU. The effect of generalizing the hardware is shown by comparing the two bars. The left bar shows the speedups attained when the hardware supports no wildcarding, i.e. it is defined precisely by the application that created it. This bar corresponds to cost point 15 in Figure 7. The right bar shows speedups when each CFU is generalized to support opcode classes. Opcode classes are groups of opcodes that can match each node of a CFU graph (e.g. ADD and SUB form a class, logical operations form another class). The intuition behind this model is that opcodes in the same class are similar in their hardware implementation or they can be added with little cost overhead, so each node of the CFU can potentially be generalized with a modest cost overhead. Note however, the cost of using opcode classes was not accounted for in Figures 8 and 9, so this data simply serves to estimate the potential gain of multi-function CFUs.

The most important point to take from these figures is that the subsumed subgraphs are generally very useful on the cross compiles, but not so much for native compiles. For example, the subsumed subgraphs associated with blowfish only accounted for a small percentage of the overall speedup of that application, but almost all of the speedup rijndael attained from using blowfish's CFUs is from the subgraphs. Subsumed subgraphs are not very useful on native compiles, because large CFUs are normally chosen for the most computationally intensive portions of the code. This leaves few nodes in important parts of the code available to be utilized by subsumed subgraphs. In cross compiles, however, the DFGs differ between applications and so the smaller subsumed subgraphs take on more importance, as it is easier to find matches.

Using opcode classes for wildcards also enables effective reuse of CFUs. Several benchmarks show very large speedups when wildcards are used, such as sha on rijndael, url on ipchains, djpeg on mpeg2dec, and rawdaudio on rawcaudio. When applications show little speedup improvement due to wildcards in the cross compiles, that means there is very little commonality in the computationally intensive parts of their DFGs not already captured by other CFUs. This is why applications using their own CFUs show little speedup when moving to wildcards. Most cross compiles show significant speedups when moving to wildcards, though, which motivates incorporating this into future work. It also shows that applications within a domain generally have similar DFG structure in the computationally intense portions of their DFGs.

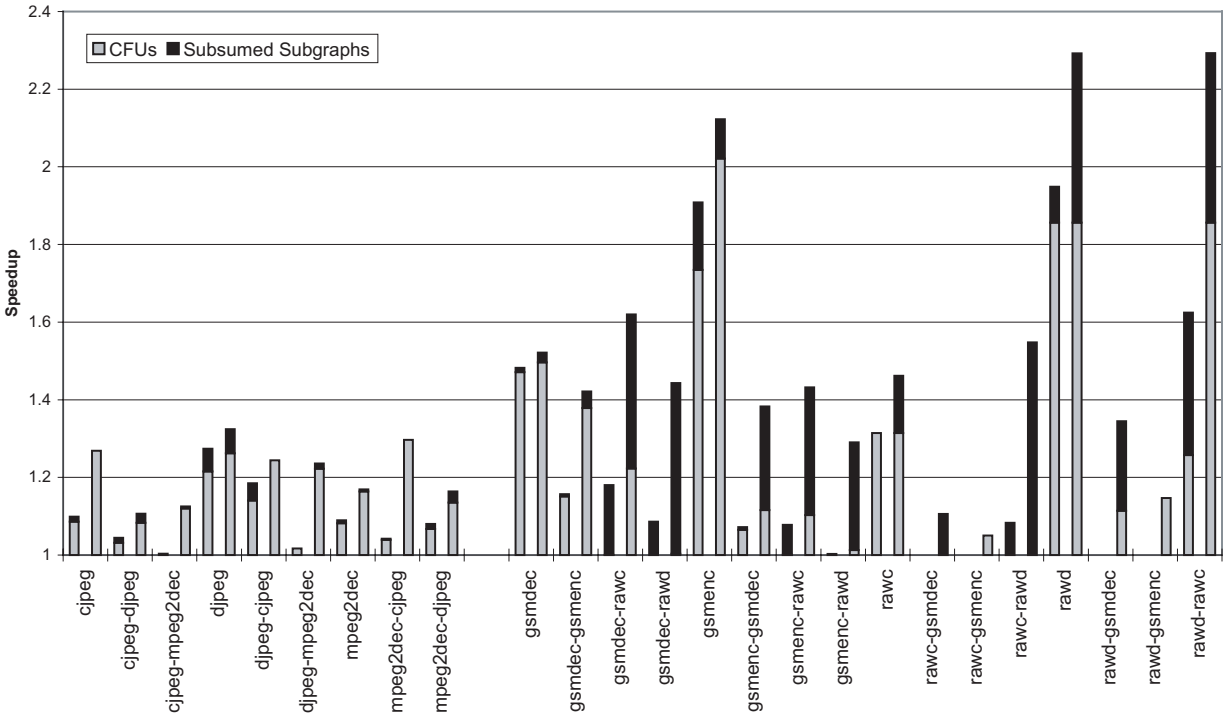


Figure 9: Effect of subsumed subgraphs and wildcards in Image and Audio at the 15-adder cost point.

In summary, Figures 8 and 9 show that the use of wildcards and subsumed subgraphs allows for more effective cross compilation across the domains.

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 these extensions into the code. 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 across a range of applications and shown ways to increase their utility.

Our system has demonstrated significant speedups for several applications, with as much as 1.94 for rawdaudio and an average of 1.47, while utilizing very little die area. We have demonstrated that it is difficult to find exact subgraph matches in order to reuse custom instruction set extensions across applications in a domain. Simple CFU generalization techniques (wildcards and subsumed subgraphs) can substantially improve the number of matches, however. In the future, we plan to relax the memory and control flow restrictions in the present system, and to incorporate multi-function CFUs into the selection process.

7. ACKNOWLEDGMENTS

We thank Michael Chu and the the anonymous referees for their helpful comments and suggestions. This research was supported in part by ARM Limited, the DARPA/MARCO C2S2 Research Center and equipment donated by Intel Corporation.

8. REFERENCES

[1] A. Aho, M. Ganapathi, and S. Tijang. Code generation using tree pattern matching and dynamic programming. *ACM TOPLAS*, 11(4):491–516, Oct. 1989.

[2] A. Alomary et al. PEAS-I: A hardware/software co-design system for ASIPs. In *EDAC*, 1993.

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

[4] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *40th DAC*, June 2003.

[5] P. M. Athanas and H. S. Silverman. Processor reconfiguration through instruction set metamorphosis. *IEEE Computer*, 11(18), 1993.

[6] M. Baleani et al. HW/SW partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *ACM CODES Workshop*, pages 61–66, May 2002.

[7] J. P. Bennett. *A Methodology for Automated Design of Computer Instruction Sets*. PhD thesis, University of

Cambridge, 1988.

[8] P. Bose and E. S. Davidson. Design of instruction set architctures for support of high-level languages. In *ISCA*, June 5–7, 1984.

[9] P. Brisk et al. Instruction generation and regularity extraction for reconfigurable processors. In *CASES*, pages 262–269, 2002.

[10] H. Choi et al. Synthesis of application specific instructions for embedded DSP software. *IEEE Transactions on Computers*, 48(6):603–614, June 1999.

[11] L. Cordella et al. Performance evaluation of the VF graph matching algorithm. In *Proc. of the 10th ICIAP*, volume 2, pages 1038–1041. IEEE Computer Society Press, 1999.

[12] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE MICRO*, 20(2):60–70, Mar. 2000.

[13] D. Goodwin and D. Petkov. Automatic generation of application specific processors. In *CASES*, 2003.

[14] M. Gschwind. Instruction set selection for ASIP design. In *ACM CODES Workshop*, May 1999.

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

[16] J. R. Hauser and J. Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In *Symposium on FPGAs for Custom Computing Machines*, Apr. 1997.

[17] B. Holmer. *Automatic Design of Computer Instruction Sets*. PhD thesis, University of California, Berkeley, 1993.

[18] I. Huang and A. M. Despain. Synthesis of application specific instruction sets. *IEEE TCAD*, 14(6), June 1995.

[19] R. Kastner et al. Instruction generation for hybrid reconfigurable systems. *ACM TODAES*, 7(4), Apr. 2002.

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

[21] R. Leupers and P. Marwedel. Instruction selection for embedded DSPs with complex instructions. In *EDAC*, Sept. 1996.

[22] S. Liao et al. Instruction selection using binate covering for code size optimization. In *ICCAD*, pages 393–399, 1995.

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

[24] K. V. Palem, S. Talla, and P. W. Devaney. Adaptive explicitly parallel instruction computing. In *Proc. Australasian Computer Architecture Conference*, pages 61–74, 1999.

[25] A. Peymandoust et al. Automatic instruction set extension and utilization for embedded processors. In *14th ASAP*, June 2003.

[26] J. V. Praet et al. Instruction set definition and instruction selection for ASIP, 1994.

[27] D. S. Rao and F. J. Kurdahi. On clustering for maximal regularity extraction. *IEEE Transactions on Computer Aided Design*, 12(8), Aug. 1993.

[28] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable function units. In *MICRO*, pages 172–180, Dec. 1994.

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

[30] F. Sun et al. Synthesis of custom processors based on extensible platforms. In *ICCAD*, Nov. 2002.

[31] Trimaran. An Infrastructure for Research in ILP. http://www.trimaran.org.

[32] M. J. Wirthlin and B. L. Hutchings. DISC: The dynamic instruction set computer. In *Field Programmable Gate Arrays for Fast Board Development and Reconfigurable Computing*, pages 92–103, 1995.

[33] L. Wu, C. Weaver, and T. Austin. Cryptomaniac: A fast flexible architecture for secure communication. In *ISCA*, pages 110–119, June 2001.