International Journal of Parallel Programming, Vol. 31, No. 6, December 2003 (© 2003)

Automatic Design of Application Specific Instruction Set Extensions through Dataflow Graph Exploration

Nathan Clark,[1,2] Hongtao Zhong,[1] Wilkin Tang,[1] and Scott Mahlke[1]

General-purpose processors are often incapable of achieving the challenging cost, performance, and power demands of high-performance applications. To meet these demands, most systems employ a number of hardware accelerators to off-load the computationally demanding portions of the application. As an alternative to this strategy, we examine customizing the computation capabilities of a processor for a particular application. The processor is extended with hardware in the form of a set of custom function units and instruction set extensions. To effectively identify opportunities for creating custom hardware, a dataflow graph design space exploration engine heuristically identifies candidate computation subgraphs without artificially constraining their size or shape. The engine combines estimates of performance gain, cost, and inherent limitations of the processor to grow candidate graphs in profitable directions while pruning unprofitable paths. This paper describes the dataflow graph exploration engine and evaluates its effectiveness across a set of embedded applications.

KEY WORDS: Application-specific processor; dataflow graph; embedded system; hardware customization; instruction set.

1. INTRODUCTION

In recent years, the markets for PDAs, cellular phones, digital cameras, network routers, and other high performance but special-purpose devices

[1] Advanced Computer Architecture Laboratory, University of Michigan, Ann Arbor, Michigan 48109. E-mail: {ntclark, hongtaoz, tangw, mahlke}@umich.edu
[2] To whom correspondence should be addressed.

0885-7458/03/1200-0429/0 © 2003 Plenum Publishing Corporation

has grown explosively. Many of these devices perform computationally demanding processing of images, sound, video, or packet streams. In these systems, application specific hardware design is used to meet the challenging cost, performance, and power demands. The most popular design strategy is to build a system consisting of a number of special-purpose application-specific integrated circuits (ASICs) coupled with a low cost core processor, such as an ARM processor.[1] 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 no postprogrammibility.

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. There are a couple of benefits to this approach. First, the system is postprogrammable and can tolerate changes to the application. Though the degree of application change is not arbitrary, the intent is 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, 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 key questions with this approach are whether the augmented core can achieve the desired level of performance and how to design an efficient set of extensions for the processor core. For this paper, we focus on the goal of defining a set of instruction set extensions to accelerate a target application in a cost-effective manner. This process can be as time consuming and expensive as designing an ASIC if done manually, thus we believe automation is a key to making this strategy successful. Our approach is to use a dataflow graph exploration engine to identify the critical computation subgraphs in the target application. The subgraphs are analyzed to determine the desirability of using specialized hardware to accelerate them. A number of issues must be considered to determine desirability, including estimated performance gain, estimated cost of the custom hardware, encoding of the new operation in the core processors instruction format, and expected usability of the custom hardware. With this data in place, a set of hardware extensions to processor are selected and a compiler generates code with the selected subgraphs replaced by new instructions.

There are three contributions of this paper. First, we describe and categorize the issues associated with adding custom hardware to a core processor. Next, we propose a novel dataflow graph exploration heuristic to efficiently determine which computation subgraphs are the best candidates for hardware extensions. Our heuristic is applied to several benchmarks in order to determine its effectiveness. Finally, the effect of communication latency to and from the custom hardware is explored.

2. DATAFLOW GRAPH EXPLORATION

The overall structure of the dataflow graph exploration engine is shown in Fig. 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 dataflow graph are not created. Initially, the application passes through a dataflow graph (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. The hardware library was created by synthesizing primitive operations with Synopsis design tools and a popular 0.18μ standard cell library.

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

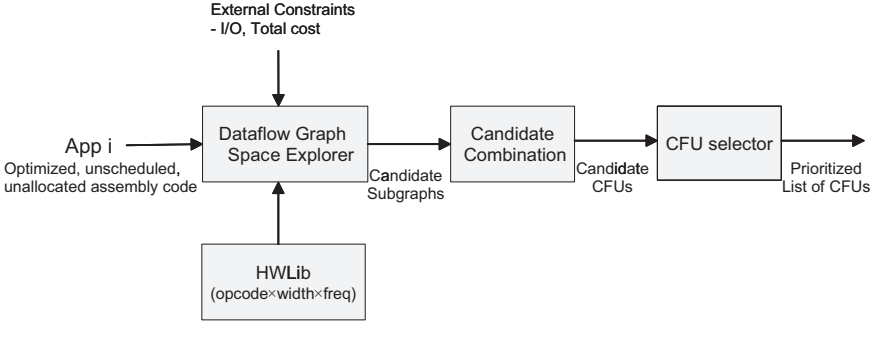
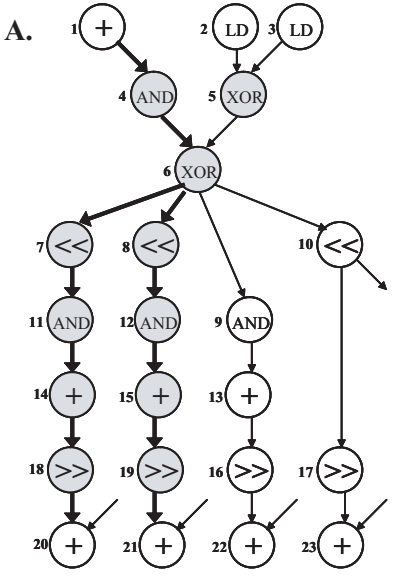


Fig. 1. Organizational structure of the DFG exploration engine.



B.

```
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;
```

C.

Opcode	Area	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

Fig. 2. (A) Sample dataflow graph from the blowfish benchmark. Bold arrows indicate the critical path, and the shaded nodes delineate a CFU discovered by the explorer. An internal node with only one input edge signifies that the other operand is a constant. (B) Part of the preprocessed C code from blowfish that this DFG came from. (C) Excerpt of the data from the hardware library. Areas are relative to a 32 bit carry-lookahead adder and cycle calculations are based on a 300 MHz clock.

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.

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

2.1. Dataflow Graph Exploration

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. In this work we consider only connected subgraphs.

Implementing subsets of the DFG in hardware, 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 clearly a very difficult problem. The most glaring difficulty is that there are an exponential number of potential candidates to select as CFUs for a given DFG. 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. The DFG exploration proposed here is a novel algorithm to effectively curve the exponential growth of this problem.

Exploration starts by examining each node in the DFG and using it as a seed for a candidate subgraph. Initially, the technique used a naïve implementation that looked at all possible directions to grow the seed nodes and then added pairs of nodes (the seed plus a node in one growth direction) to a list of potential candidates. Next, these pairs were used as seeds, and candidates of size three were grown from the dataflow edges entering and leaving the candidates. The algorithm recursed until certain external constraints were met, for example the resultant candidate could not grow without crossing a function call or the die area of the resultant candidate was too large. The number of candidate subgraphs quickly grows out of control with sufficiently loose external constraints, preventing the algorithm from completing on most MediaBench[3] applications. Empirical experiments showed that running the algorithm with tight external constraints produced very poor quality candidates, as it unnecessarily restricted the discovered candidates.

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 Fig. 2, CFU 6-10 has little value, because node 10 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 in. 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. Flexibility is one of the major benefits of this technique. 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.

2.2. Guide Function

The purpose of the guide function is to intelligently rank which growth directions will create the best candidates. The guide function is essentially trying 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 conclusion that, generally speaking, 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 Fig. 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 nodes 5, 9, or 10, because the aforementioned nodes are on the critical path. Points are awarded using the equation $\frac{10}{\text{slack}+1}$, 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 10 would get $\frac{10}{2+1} = 3.33$ points.

Experiments have shown that it is important to give candidate directions credit even when they lie slightly off the critical path. This is because

in several instances replacing some nodes on the critical path can expose an auxiliary critical path in the DFG. For example, if we selected a CFU for nodes 7-11-14-18 and 8-12-15-19 in Fig. 2 then nodes 9, 13, 16, and 22 would become a new critical path. Giving candidates that grow in these directions some credit keeps them available for selection, even if they do not initially appear useful.

Latency. Latency tries to guide exploration towards combining operations which require fewer cycles to execute when combined into a CFU than they do as stand-alone 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 $\frac{\text{old latency}}{\text{new latency}} * 10$. The latency of a CFU is calculated by summing up the fractional delays of each atomic operation (see Fig. 2c) along the critical path of the candidate subgraph. Using candidate 4-6 on Fig. 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 $\frac{0.15}{0.15+0.30} * 10 = 3.33$ points. In contrast, growing towards node 10 would get 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. This metric should factor in the area of the CFU, the additional inter-connect and control logic, and the impact on decode logic to the core processor. Register file ports are considered a design constraint and will never be added to support a custom instruction, so they do not factor into the area. It is difficult to measure the impact on decode and control logic, and so the simplifying assumption is made that CFU area is the dominant term. The guide function considers area to be the sum of the cost of each primitive operation in the CFU (see Fig. 2c).

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, $\frac{\text{old area}}{\text{new area}} * 10$, 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 too small. Consider the case of growing candidate 10-17. If no rounding was done, then growing to node 23 would only yield $\frac{0.02}{1.02} * 10$ points and growing to node 6 would only yield $\frac{0.02}{0.18} * 10$ 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/output operands allowed for a CFU is limited. Register file ports are generally fixed on the

core processor based on cost, power, and cycle time constraints. Further, instructions for a CFU that has too many operands may not be encodable in a conventional instruction set. Thus, the maximum number of input and output operands available is treated as design constraint, meaning any candidates that exceed the prescribed limits are discarded.

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 $\min(\frac{\text{old \# ports}}{\text{new \# ports}} * 10, 10)$. 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 8-12 from Fig. 2 were examined, growing toward node 15 would not increase the number of inputs or outputs, yielding $\frac{2}{2+1} * 10 = 6.67$ points. Growing towards node 6 would actually increase both the number of inputs and outputs, though, yielding $\frac{2}{4+1} * 10 = 4$ points.

One issue not considered by the current guide function is power consumption. As with performance, CFU candidates can be favored that best reduce power by enabling more efficient implementation of the candidate DFG by using custom hardware. The extension of the guide function is relatively straight forward and is the subject of future work.

With the guide function heuristic in place, it was important to verify two points: first that the heuristic does indeed prune the search space, and second 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 many larger benchmarks. Again, the results found using the heuristic were comparable with those of full exponential search.

2.3. Candidate Combination

Once candidate subgraphs are discovered, it is a fairly straight forward process to group identical ones together into candidate CFUs. A simple

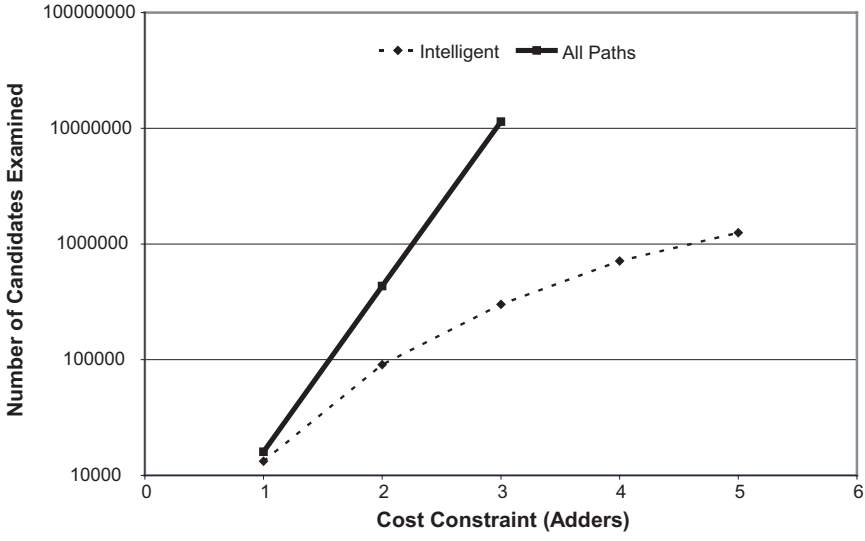


Fig. 3. Comparison of the number of candidates examined by the intelligent heuristic compared with growth in all available directions on blowfish. The X-axis shows the maximum die area the candidates were allowed to grow to. There are no points at costs four and five for "all paths" because we did not have access to a machine with enough memory to run the experiment.

test checking graph equivalence, while taking into account commutativity, accomplishes this. For example, if subgraphs 7-11-14-18 and 8-12-15-19 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 estimation 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. Investigation has shown that performance estimates based solely on the dynamic number of occurrences of each subgraph prove 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. Using Fig. 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 zero. CFUs "AND-ADD" and "ADD->>" would also be recorded as being subsumed by "AND-ADD->>." Recording which CFUs are subsumed

by others proves useful for value and cost estimation done by the selection heuristic described later.

The second pass records a single wildcard option for each CFU. Wildcards are defined as 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. Many CFUs could potentially be wildcard matches for each other, but for simplicity only the wildcard match with the best estimated cycle savings is recorded for each CFU.

2.4. Candidate Selection

Contrary to combining the candidate subgraphs, CFU selection is not straight forward. Selection is very similar to the 0/1 knapsack problem. There is a set of resources (the candidate CFUs) which all have values (the estimated cycle savings) and weights (the cost in die area), and the goal is to maximize the total value for a given weight threshold. 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 subgraphs and thus 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 7-11-14-18, and 7-11-14. If 7-11-14-18 was selected first and did not update the value of 7-11-14 to reflect the fact that it can no longer use any of its operations, then 7-11-14 would be selected also, even though it would provide no gain above what 7-11-14-18 already provided.

The strategy used for CFU selection is a simple greedy method, illustrated in Fig. 4. Given a list of CFU candidates, the one with the best ratio of $\frac{\text{value}}{\text{cost}}$ is greedily selected (note best value alone is also a valid heuristic). Once a CFU is selected (number 2 in this example), the heuristic iterates through the list of remaining CFU candidates and removes operations that were already claimed by CFU 2. The estimated performance gain attributed to each operation is kept track of to make updating statistics easy. In Fig. 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, taking into

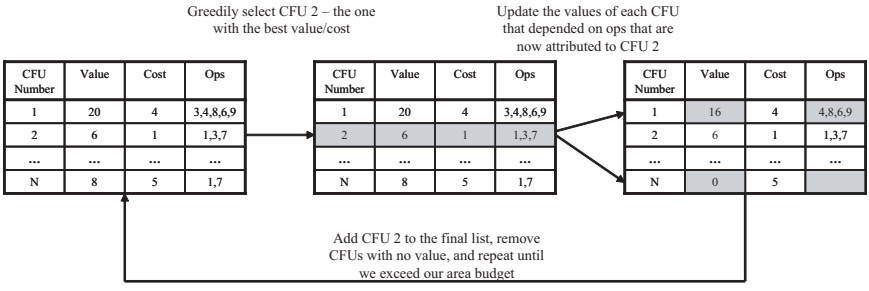


Fig. 4. Basic greedy approach to CFU selection.

account that it can no longer use operation 3. 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 it does not. For example, when the greedy algorithm selects based only on cycle savings, performance does poorly at the low cost budget points compared to when it selects based on $\frac{\text{cycle savings}}{\text{cost}}$. The opposite is true at the high cost points, however. In an attempt to improve the selection heuristic, a version based on the optimal dynamic programming solution to the 0/1 knapsack problem was implemented. The dynamic programming heuristic generally does better (roughly 5–10% on average) than both greedy solutions, however it suffers from a much slower runtime, and thus it was not used in any of the studies in this paper. Though the dynamic programming method provides better results, it is still not necessarily optimal as its decisions are based on cycle gain estimations and it has to make local decisions without knowing how one selection will affect later selections.

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 their subsumed or wildcard candidate subgraphs. This creates a huge computational overhead for every CFU selection. In the case of subsumed subgraphs, this often 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 Fig. 2 was selected then it would be possible to execute 13-16 together on the gray CFU, as 13-16 is subsumed. However if a CFU for 13-16-22 could have been chosen later, that would have helped performance more, but this option has been precluded because operations 13-16 have already been claimed by the large gray CFU. The case just described occurs quite frequently, so subsumed subgraphs and wildcard candidates are not included in our performance estimations. Instead CFUs are selected as if they had no subsumed subgraphs or wildcards, and then the costs of the subsumed subgraphs and wildcards are updated to reflect the fact that they can now be selected for very little cost overhead.

Another issue to consider 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 is available.

3. EXPERIMENTAL RESULTS

The system proposed was constructed as part of the Trimaran research infrastructure.[4] The DFG exploration engine was implemented as a stand-alone module, and the compiler backend was modified to facilitate subgraph matching and replacement. As mentioned previously, the cycle time and area estimates in the hardware library were calculated using Synopsis design tools and a 0.18μ standard cell library.

For this evaluation, two simplifying assumptions were 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 both 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.

Fourteen 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 (one adder, two adders, etc.). The Fourteen benchmarks can be divided into four categories: encryption, network, audio, and image. The encryption category contains three benchmarks from the MiBench[2] benchmark suite, the network category consists of three benchmarks from NetBench,[5] and the audio and image categories are from MediaBench.[3]

The baseline machine for the experiments is a four-wide VLIW machine 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.[1] 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 verses Area. Figure 5 compares the performance gain in each of the four benchmark categories as the total cost of CFUs is varied from one to sixteen adders. Each line in the graphs represents the speedup of an application with CFUs as compared to the baseline machine with no CFUs. 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 the audio and image sections show very little speedup (e.g., mpeg2dec and g721encode). 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 Fig. 5, our system realizes speedups

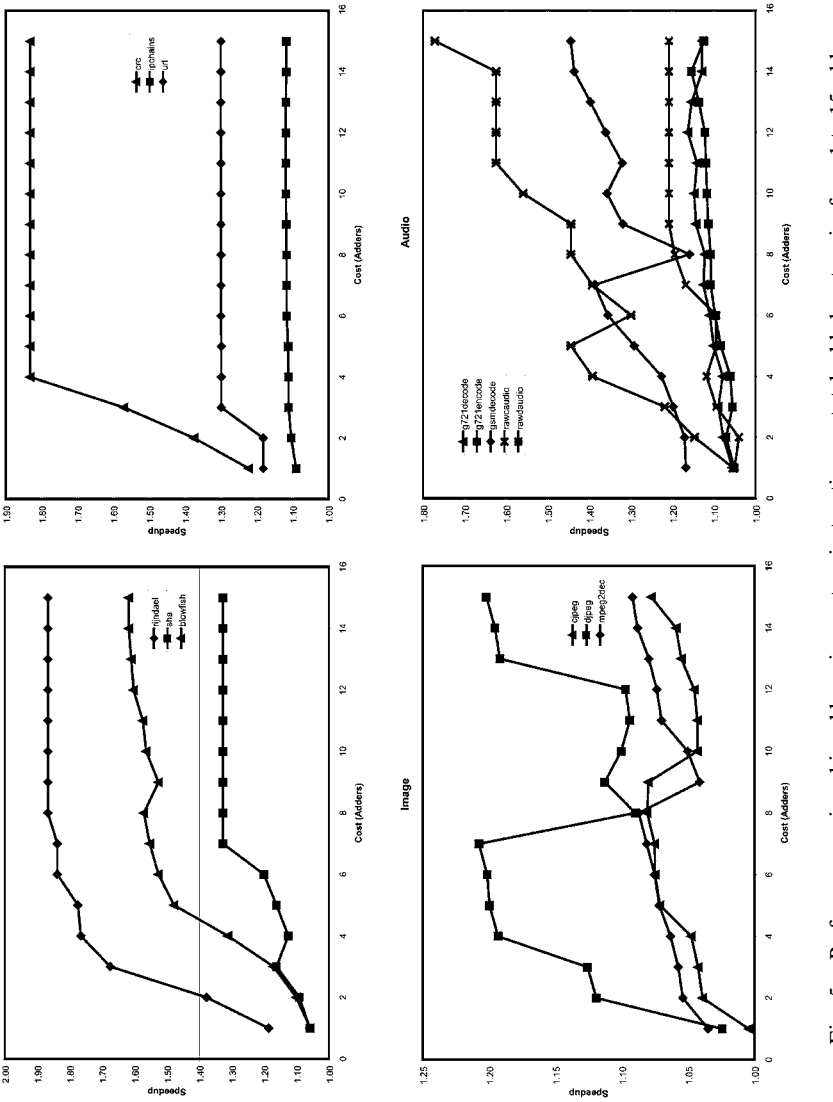


Fig. 5. Performance gain achieved by using custom instructions as total added cost varies from 1 to 15 adders.

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. All 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. 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 some graphs in Fig. 5 (gsmdecode and 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 the gsmdecode benchmark, a speedup of approximately 1.4 is attained at cost point 7, by using several small and generally useful CFUs. At cost point 8 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 for gsmdecode, the greedy selection begins to include the small CFUs used at point 7 along with the large one used at point 8, thus the speedups improve at points 13 and above beyond what was possible at point 7.

Effect of Communication Cost. Another experiment was performed to determine how speedups would be affected if there was communication overhead between the CFUs and the rest of the core. This information is useful because it is typically easier to implement custom instructions in a design flow using a generic coprocessor interface, as opposed to implementing them directly in the core. If they are implemented as a coprocessor, there will likely be some overhead in transferring values from the register file to the coprocessor and from the coprocessor back to the register file. If sufficient instruction level parallelism exists in the applications then this delay is not a problem because the compiler can schedule around the communication latency. Likewise, if the custom instruction is very large, then the delay is spread across many cycles, which will not hurt speedup very much.

The results of this study are in the Table I. The columns show speedups when varying communication latency using the set of CFUs developed

Table I. Speedups Given Various Communication Delays and an Area Budget of 15 Adders

Benchmark	No Delay	1 Cycle	2 Cycle	3 Cycle
bfish	1.62	1.39	1.11	1.05
cjpeg	1.07	1.00	1.00	1.00
crc	1.83	1.57	1.38	1.00
djpeg	1.20	1.08	1.01	1.00
g721dec	1.13	1.09	1.07	1.06
g721enc	1.12	1.10	1.08	1.07
gsmdec	1.44	1.41	1.33	1.27
ipchains	1.12	1.01	1.00	1.00
mpeg2dec	1.09	1.02	1.01	1.00
rawc	1.21	1.15	1.00	1.00
rawd	1.77	1.34	1.15	1.00
rijndael	1.87	1.86	1.18	1.00
sha	1.33	1.23	1.20	1.00
url	1.30	1.00	1.00	1.00

for cost point 15. A communication latency of 1 cycle means that it takes 1 cycle to transfer values to the coprocessor and 1 cycle to transfer values back to the core, totaling a 2 cycle delay. Note how moving from 0 cycles of delay to just 1 cycle causes a major drop in speedup for most applications. This is not due solely to a lack of instruction level parallelism, but also because many frequently used CFUs are only two or three primitives long. Once a communication delay is introduced, it often no longer makes sense to use these small CFUs. Like the limit study, this data points to the necessity to ease system restrictions in order to create larger CFUs for a coprocessor. Larger CFUs are able to amortize the cost of the communication, which is essential as applications have limited instruction level parallelism available to hide delay.

Guide Function Parameters. A final set of experiments we present deals with the various categories of the guide function. In Fig. 6, CFUs are generated for one randomly selected benchmark from each of the four categories. Four of the lines on each chart show the results when only one of the four proposed guide function categories is used. The fifth line shows the results presented earlier using all four categories and are displayed for comparison.

The are two main points to take away from Fig. 6. The first point is that the categories which individually generate good candidates vary

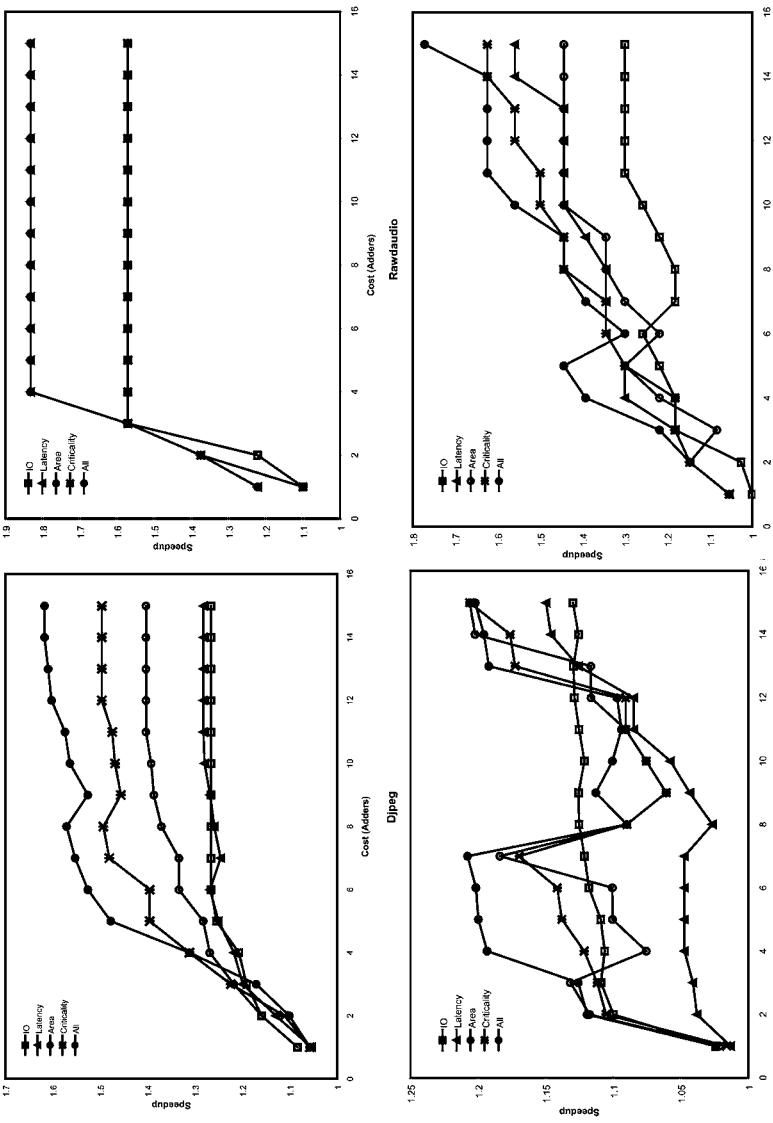


Fig. 6. Results for benchmarks using only one factor in the guide function.

greatly by application. For example, using only latency as the guide function creates very poor candidates for blowfish and djpeg, but it does well for crc and rawdaudio. Using only I/O in the guide function produces the best candidates at cost points 8-12 for djpeg, but does not do an effective job for the other benchmarks. This leads to the second point to take from this figure: using all of the categories in conjunction generally produces better results than any one of them individually. At the majority of cost points on each of these graphs, using all of the categories simultaneously results in higher speedups. Since different categories prove more effective in different applications, using all of them together provides a better guide than any of them in isolation.

4. RELATED WORK

A large body of research has gone into the design of application specific instruction set extensions. Work in Refs. 6–12 all showed some of the gains possible with instruction set customization. While these works show the potential utility of instruction set customization, they do not provide methods to automate the process of instruction candidate discovery. Many other systems have been proposed to tackle this problem and are discussed in more depth below.

Early work[13] in this area 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. Work by Bennett[14] proposes iterative combination of primitives which occur in subsequent lines of code to reduce static code size. 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 Refs. 15–19, in that all of these approaches 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 they risk being stuck in a local maxima. Each edge is combined in a locally optimal manner, reminiscent of greedy heuristics.

Holmer proposed a more global technique[20] that was extended by Ref. 21. 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 using a simulated annealing

algorithm. Work by Bose,[22] is similar to this, except that it 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 with the expected goal of improvement. The application of these two algorithms was mainly targeted toward design entire instruction sets as opposed to just ISA extensions, and to reduce static code size.

Choi[23] 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 this work is a new formulation of the candidate discovery problem: they discovered candidates using a modification of the subset-sum problem, and they 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 Ref. 24 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 to 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.

Some researches have proposed heuristic ways to limit the search space without artificially constraining it. In Ref. 25, 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[26] 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.

The use of a guide function to restrict growth as proposed in this paper is most similar to the work by Sun.[26] 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 in this paper 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 guide function used in this work also takes more factors into account when pruning.

5. 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 the process of instruction discovery. 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. Our system has demonstrated significant speedups for several applications (as much as 1.87 for rijndael) while utilizing very little die area.

ACKNOWLEDGMENTS

We thank Krisztián Flautner, Mike Chu, and Kevin Fan for their comments and suggestions. This research was supported in part by the DARPA/MARCO C2S2 Research Center, ARM Limited, and equipment donated by Intel Corporation.

REFERENCES

1. D. Seal, *ARM Architecture Reference Manual*, Addison–Wesley (2000).
2. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, MiBench: A Free, Commercially Representative Embedded Benchmark Suite, *IEEE 4th Annual Workshop on Workload Characterization* (Dec. 2001).
3. C. Lee, M. Potkonjak, and W. Mangione-Smith, MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems, *MICRO* (Dec. 1997).
4. Trimaran, An Infrastructure for Research in ILP, http://www.trimaran.org.
5. G. Memik, W. H. Mangione-Smith, and W. Hu, Netbench: A Benchmarking Suite for Network Processors, *ICCAD*, pp. 39–43 (2001).
6. P. M. Athanas and H. S. Silverman, Processor Reconfiguration Through Instruction Set Metamorphosis, *IEEE Computer*, Vol. 11, No. 18 (1993).
7. R. Razdan and M. D. Smith, A High-Performance Microarchitecture with Hardware-Programmable Function Units, *MICRO*, pp. 172–180 (Nov. 30–Dec. 2 1994).

8. M. J. Wirthlin and B. L. Hutchings, DISC: The Dynamic Instruction Set Computer, *Field Programmable Gate Arrays for Fast Board Development and Reconfigurable Computing*, pp. 92–103 (1995).
9. J. R. Hauser and J. Wawrzynek, GARP: A MIPS Processor with a Reconfigurable Coprocessor, *Symposium on FPGAs for Custom Computing Machines* (Apr. 1997).
10. K. V. Palem, S. Talla, and P. W. Devaney, Adaptive Explicitly Parallel Instruction Computing, *Proc. Australasian Computer Architecture Conference*, pp. 61–74 (1999).
11. M. Gschwind, Instruction Set Selection for ASIP Design, *ACM Seventh International Workshop on Hardware/Software Co-Design* (May 1999).
12. L. Wu, C. Weaver, and T. Austin, Cryptomaniac: A Fast Flexible Architecture for Secure Communication, *ISCA*, pp. 110–119 (June 2001).
13. A. Alomary, T. Nakata, Y. Honma, and J. Sato, PEAS-I: A Hardware/Software Co-Design System for ASIPs, *EDAC* (1993).
14. J. P. Bennett, A Methodology for Automated Design of Computer Instruction Sets, Ph.D. thesis, University of Cambridge (1988).
15. D. S. Rao and F. J. Kurdahi, On Clustering for Maximal Regularity Extraction, *IEEE Transactions on Computer Aided Design*, Vol. 12 (Aug. 1993).
16. J. V. Praet, G. Goosens, D. Lanner, H. D. Man, and H. Synthesis, Instruction Set Definition and Instruction Selection for ASIP (1994).
17. M. Baleani *et al.*, HW/SW Partitioning and Code Generation of Embedded Control Applications on a Reconfigurable Architecture Platform, *9th Intl. Workshop on Hardware/Software Codesign*, pp. 61–66 (May 2002).
18. P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh, Instruction Generation and Regularity Extraction for Reconfigurable Processors, *CASES*, pp. 262–269 (2002).
19. R. Kastner *et al.*, Instruction Generation for Hybrid Reconfigurable Systems, *ACM TODAES*, Vol. 7 (Apr. 2002).
20. B. Holmer, Automatic Design of Computer Instruction Sets, Ph.D. thesis, University of California, Berkeley (1993).
21. I. Huang and A. M. Despain, Synthesis of Application Specific Instruction Sets, *IEEE Transactions on Computer-Aided Design*, Vol. 14 (June 1995).
22. P. Bose and E. S. Davidson, Design of Instruction Set Architctures for Support of High-Level Languages, *ISCA* (June 5–7, 1984).
23. H. Choi *et al.*, Synthesis of Application Specific Instructions for Embedded DSP Software, *IEEE Transactions on Computers*, Vol. 48, pp. 603–614 (June 1999).
24. K. Atasu, L. Pozzi, and P. Ienne, Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints, *40th DAC* (June 2003).
25. M. Arnold, Instruction Set Extensions for Embedded Processors, Ph.D. thesis, Delft University of Technology (2001).
26. F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, Synthesis of Custom Processors Based on Extensible Platforms, *ICCAD* (Nov. 2002).