Efficient ASIP Design for Configurable Processors with Fine-Grained Resource Sharing

Quang Dinh, Deming Chen, Martin D. F. Wong
Department of Electrical and Computer Engineering
University of Illinois at Urbana Champaign
{qdinh2, dchen, mdfwong}@uiuc.edu

ABSTRACT

Application-Specific Instruction-set Processors (ASIP) can improve execution speed by using custom instructions. Several ASIP design automation flows have been proposed recently. In this paper, we investigate two techniques to improve these flows, so that ASIP can be efficiently applied to simple computer architectures in embedded applications. Firstly, we efficiently generate custom instructions with multi-cycle IO (which allows multi-outputs), thus removing the constraint imposed by the ports of the register file. Secondly, we allow identical portions of different custom instructions to be shared, thus allowing more custom instructions under the same area constraint. To handle the greatly increased exploration space, we propose several heuristics to keep the problem tractable. Experimental results show that we can achieve 3x speedup in some cases.

Categories and Subject Descriptors

B.7.2 [**Hardware**]: INTEGRATED CIRCUITS – *Design Aids.*

General Terms

Algorithms, Performance, Design, Experimentation.

1. INTRODUCTION

Application-Specific Instruction-set Processors (ASIP) are one approach to bring the speed and power advantages of hardware implementations to software applications with little design efforts. While the majority of the code is executed normally by the traditional processor, some specific operations (mostly inside loops) are handled as custom instructions by special hardware units. These groups of operations are subgraphs of the application's dataflow graph (*DFG*), and we call them *patterns*. When proper patterns are implemented in custom logic, execution time, as well as power consumption, is reduced. For example, Figure 1 (from Altera's website [13]) shows the instruction logic implemented in the FPGA-based Nios II embedded processor. It is a combination of a RISC core and some custom logic as extension to the ALU that can execute single-cycle (combinatorial) as well as multi-cycle custom instructions. Other commercially available embedded processors supporting ASIP include Xilinx's MicroBlaze [14] and Tensilica's Xtensa LX2 [15].

For ASIP to become practical, it is essential to have automatic generation of custom instructions that provide high quality results. Several recent papers have looked into various aspects of this interesting design automation problem [1][2][3][4][5][7].

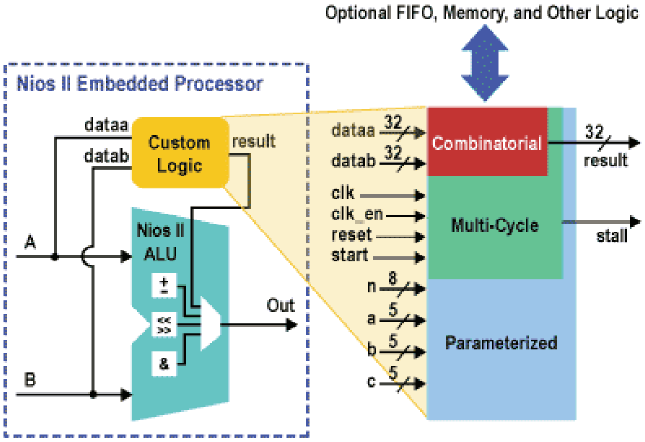

Figure 1. Altera Nios II Custom Instruction Logic

One important factor affecting the performance of ASIP is the bandwidth between the CPU and the custom logic, i.e. the numbers of read ports and write ports on the register file that the custom instructions can access [2][4]. Many existing works consider these architectural parameters as a hard limit, i.e. they impose limits on the numbers of inputs/outputs of custom instructions based on the numbers of read/write ports [1][3][5]. This would severely limit the potential of ASIP, because many architectures for embedded applications have only a small number of ports, e.g. two read ports and one write port.

Some works, such as [4], allow custom instructions that have more inputs and/or outputs than the architectural constraints. This is realized by having extra clock cycles to move data between the custom functional unit and the register files. However, the effect of these extra cycles is considered only after patterns have been generated. In this paper, the extra cycles are taken into account when we generate patterns as well. Under this approach, the numbers of read/write ports are treated as a performance cost, not as hard limits on the numbers of inputs/outputs. With more inputs and outputs available, we can generate more and larger candidate instructions that can provide performance boost.

FPGA'08, February 24–26, 2008, Monterey, California, USA.
Copyright 2008 ACM 978-1-59593-934-0/08/02...$5.00.

When custom instructions become larger (carry out more operations), it is less likely that they occur at multiple places in the code [1]. This reduces the reusability of the custom hardware, its ability to execute the same operations at different points in the program. We propose a scheme to increase reusability, by allowing isomorphic portions of different custom instructions to be shared. In this way, we could potentially pack more custom instructions under the same area constraint.

Overall, our contributions can be summarized as follows:

- A heuristic two-step pattern generation scheme that can handle both complex patterns and resource sharing.
- Several efficient templates for generating complex patterns with large numbers of inputs and outputs.
- A mapping algorithm that optimizes execution time under area constraint.

In the following section, we introduce some related works on ASIP design automation. Section 3 describes the problem formulation, followed by section 4 where we present our detailed algorithm. The results are shown in section 5, and we conclude this paper in section 6.

2. RELATED WORKS AND MOTIVATION

2.1 Related Works

Several ASIP design automation flows have been presented recently [1][3][5][7]. Most of these break the problem into several smaller tasks. The first step is to identify subsets of the DFG, or patterns, which can be implemented as custom instructions. For small patterns (with 1 output and less than 6 inputs), this can be done with a full enumeration of patterns in the DFG, then followed by a selection phase to reduced the number of candidates, as in [1]. Another approach by Pozzi et al. [3] treats the problem as a branch-and-bound binary tree exploration, generating one custom instruction in each iteration. This approach has the advantage of allowing more complex patterns (multiple outputs or disconnected patterns). After all the candidate patterns have been identified, isomorphism test is employed to identify identical patterns, i.e. they are of the same pattern type and belong to one custom instruction [1][5].

When dealing with large DFGs and custom instructions with more inputs/outputs, heuristics are needed to keep the problem tractable. For examples, Clark et al. in [5] used a guide function to rank dataflow edges. This is used to help partition large DFGs into smaller, more manageable sections. In this paper, we explore another approach, combining small, simple patterns called *building-blocks* into larger, more complex patterns according to some heuristic templates.

Various ways to extend the custom instructions beyond the limit of the read/write ports of the register file have been studied. In [2], the authors propose shadow registers located inside the custom logic that can be written to with zero overhead. This approach requires modifications to the instruction format encoding, and does not allow more outputs than the number of write ports. In [4], a scheme similar to our multi-cycle read/write approach was studied. However, it mainly focused on the problem of inputs/outputs scheduling within each instruction. Custom instructions are still generated in the way described in [3], where it imposes limits on the numbers of inputs/outputs of custom instructions based on the numbers of read/write ports. In this paper, we present an efficient method to generate custom instructions without direct constraints from the number of read/write ports.

After the candidate instructions are generated, the second step is to decide how to map those instructions in the application DFG. The goal is to minimize certain cost functions, usually total latency or power consumption. One basic operation may appear in several candidate instructions, and we have to decide to use which instruction to implement this particular operation. This can be formulated as a covering problem [1][5].

Practical applications of ASIP are often constrained by area requirements; this problem is considered in [1] and [5]. The generated instructions are first selected to satisfy the area constraint, which is similar to the 0/1 knapsack problem. Then only the selected instructions are used to map the DFG. This, however, may lead to under-usage if not all custom instructions are used for mapping. Also, when dealing with resource sharing, the area for each instruction changes depending on other selected instructions. In this paper, we present a mapping algorithm that can handle both area constraint and resource sharing.

Under area constraint, resource sharing can be employed to enable more custom instructions. A resource sharing approach was considered in [5], with the idea of generalization. Operations such as ADD/SUB, or AND/OR, are considered identical (for isomorphism purpose) because they can be implemented by the same logic with little overhead. In this way, one generalized custom functional unit can execute several related patterns. This method compares whole subgraph to other subgraphs. In this paper, we study fine-grain sharing possibilities, i.e., sharing between portions of the subgraphs of different custom instructions.

2.2 Motivation of this Work

Large-pattern generation: When generating large and complex custom instructions, simply imposing a limit on the numbers of inputs and outputs, as in [1] and [3], may not be an efficient way. For example, Figure 2 shows two patterns, both having five inputs. The pattern on the right, however, exhibits more parallelism and thus may provide better speedup when implemented in hardware. If we generate all patterns with a certain maximum number of inputs, we will end up with many less useful patterns like the one on the left in Figure 2.

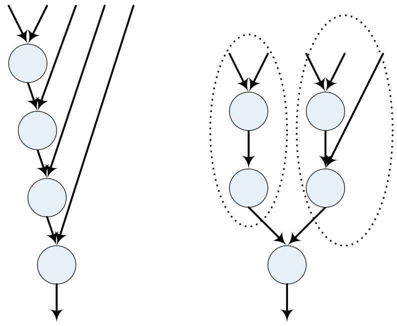

Figure 2. Two examples of five-input patterns

In this paper, we look at how to efficiently generate large instructions from the dataflow graphs, without a hard constraint on the numbers of inputs and outputs. Simply enumerating all possible patterns is impractical, because the worst-case complexity of the enumeration is $O(n^k)$, where n is the number of nodes in the DFG and k is the number of inputs of the patterns. Instead, we selectively combine small patterns into large ones. First, we enumerate all small patterns with limited number of inputs and outputs. Because of the strict constraint, this can be done efficiently. Then we can combine these small patterns into larger ones according to selected templates that provide parallelism. In Figure 2, for example, the pattern on the right can be generated from two small dotted patterns (that can run concurrently). Our pattern generation method should be several orders of magnitude faster than pure enumeration. Experimental results show that our large patterns work well, by offering 50% more speedup over patterns of up to 4 inputs.

Fine-grain resource sharing: With large patterns, there is little chance that they occur at multiple places in the DFG. The smaller portions of these patterns, however, may occur more repeatedly. Thus, we can reduce resource usage, or pack more custom instructions under the same area constraint, by sharing sub-blocks between different instructions. This resource sharing at fine-grain level is illustrated in Figure 3, where two different custom instructions (shown on the left and right) are wired to use one shared multiplier (shown in the middle), reducing the total area requirement.

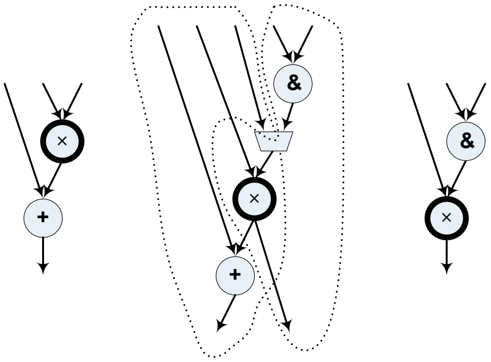

Figure 3. Two custom instructions sharing common hardware

3. DEFINITIONS AND PROBLEM FORMULATION

We use $G(V,E)$ to denote the directed acyclic graphs (*DAGs*) representing the dataflow of each basic block; the nodes V represent basic operations and the edges E represent data dependencies. G is extended to G^*, which contains additional nodes representing input and output variables of the basic block, as well as their dependency edges. Associated with each node (basic instruction) are its software execution time (in term of clock cycles), its estimated hardware latency, and its estimated hardware implementation area.

A *cut* (or pattern) S is an induced subgraph of G: $S \subseteq G$. The *inputs* of S are the predecessor nodes of those edges that enter S from the rest of G^*. These are the input values used by the operations in S. The *outputs* of S are the predecessor nodes of those edges that exit from S to the rest of G^*. These are the output values produced by S and used by other operations, either in S or in other basic blocks. A cut S is called a convex cut if there exists no path that leaves S, then enters S again.

The ASIP compilation problem, divided into two sub-problems, can be formulated as follows:

Notations:

- I: Basic instruction set
- F: Set of forbidden nodes which cannot run in hardware
- C: Set of candidate patterns
- M: Set of patterns used in final mapping ($M \subset C$)
- I': Extended instruction set ($I' = I \cup M$)
- A: Area constraint

Sub-problem 1. Pattern Generation: Given a $G^*(V,E)$ and F, generate C such that each pattern $P \in C$ is convex and $F \cap P = \varnothing$.

Because we do not treat the numbers of read ports N_R and write ports N_W as hard limits, there are no direct constraints on the numbers of inputs and outputs of P. Therefore, we have to come up with good heuristics to keep the size of C manageable while still providing good speedup.

Sub-problem 2. Application Mapping: Given a $G^*(V,E)$, basic instruction set I, and candidate patterns C, generate a mapping from G to I' so that the total execution time of G is minimized, while keeping the total area for implementing M (taking into account resource sharing) satisfying constraint A.

Our algorithm also uses the concept of cones. A *cone* C_V is a convex cut consisting of node v and some of its predecessors, such that any path connecting a node in C_V and v lies entirely in C_V. v is the root of cone C_V, which is also called a single-root pattern. A cone is *K-feasible* if the number of its inputs does not excess K. Note that a cone here is not necessarily single output. We can have outputs at non-root nodes, not just at the root node, as shown in Figure 4.

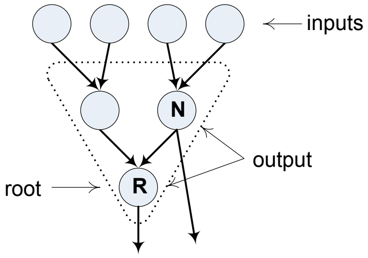

Figure 4. Root (R) and non-root (N) outputs of a cone

4. ALGORITHM DESCRIPTION

The key of our approach is that we generate custom instructions in two steps, handling resource sharing and pattern generation separately. With this heuristic approach, we can keep the complexity under control, while still obtaining good results.

4.1 Overview

In the first step, we enumerate the *building-blocks*: small and simple patterns that contain a small number of nodes and can be efficiently enumerated. These patterns may not have significant amount of speedup, but they can have many occurrences in the application DFG. These building-blocks are our basis for resource sharing. By handling resource sharing at this building-block level, we can have good sharing (building-blocks can have more occurrences comparing to larger patterns) while avoiding excessive isomorphism test (as compared to sharing at individual node level).

In the second step, we combine the building-blocks into larger and more complex patterns. These patterns are more likely to have higher speedup, and are candidates for the final custom instruction set. Resource sharing is implicitly enabled whenever isomorphic building-blocks appear in multiple instructions. This is demonstrated in Figure 5. In this figure, the gray triangles represent the building-blocks. As the two thick-edge blocks in the custom instructions on the left are isomorphic, their hardware implementation shares the same logic resource, as shown in the right diagram. When calculating total area, we count the area of the thick-edge block only once. Meanwhile, we count the extra multiplexer area that is used to enable such a sharing.

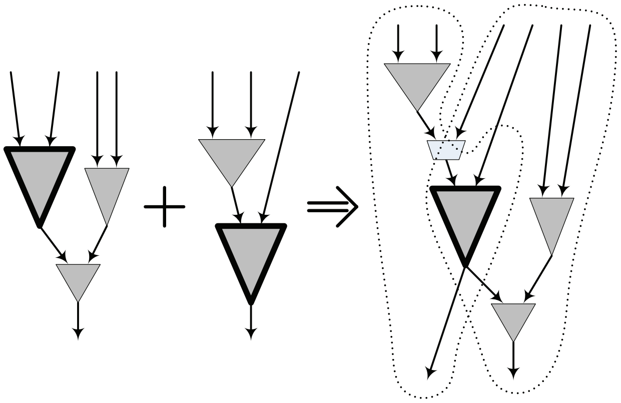

Figure 5. Resource sharing with building-blocks. The isomorphic thick-edge blocks (left) are shared (right).

Because we have no hard constraint on the numbers of inputs and outputs, the number of generated patterns will grow rapidly. To keep the number of patterns under control, we use a simple greedy selection scheme during the generation process. Each pattern is assigned a rank, and we keep only a certain number of top-ranked patterns. Unlike [1] and [5], here we do not try to satisfy the area constraint, as this is handled in the next phase.

After generating the candidate custom instructions, we use a branch-and-bound algorithm to find a mapping that minimizes total execution time, while satisfying the area constraint.

4.2 Building-Block Enumeration

In this step, we want small building-blocks that can be efficiently enumerated. As pointed out in [1] and [8], *K-feasible* cut enumeration is very efficient for $K \leq 5$ (linear runtime with respect to the DAG size). This computation enumerates all single-root patterns (cones) that have a maximum of K inputs in a given DAG. Based on this observation, we select our building-blocks to be single-root patterns with a maximum of $K = 4$ inputs.

A cut rooted on a node v can be represented using a product term (*p-term*) of the variables associated with the inputs of the cut. A set of cuts rooted on node v can be represented by a sum-of-product expression using the corresponding p-terms. Cut enumeration is guided by the following formula [9]:

$$f(K,v) = \otimes_{u \in input(v)}^{K} \left[u + f(K,u) \right]$$

where $f(K,v)$ represents all the K-feasible cuts rooted at node v, operator + is Boolean OR, and $\otimes^{K}$ is Boolean AND while filtering out all the p-terms with more that K variables. The cut-enumeration process merges the cuts of the inputs of v and discards those cuts having more inputs than K. Cut enumeration generates K-feasible cones (refer to Figure 4).

4.3 Pattern Generation

In this step, we generate the candidate patterns for custom instructions, by combining the building-blocks. The goal here is to generate large patterns that have good speedup. While combining building-blocks into more complex patterns can be considered as another enumeration problem, in this work we decide to heuristically generate only certain types of combinations, or *combination templates*, due to complexity reasons. Beside the trivial single building-blocks, four other templates are considered as they have the potential to yield useful patterns. These templates are presented in Figure 6.

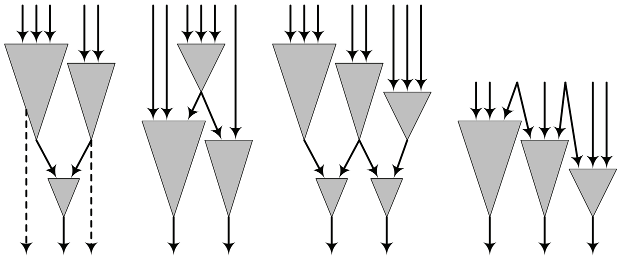

Figure 6. Combination templates A, B, C, and D (left to right)

Template A generates single-root patterns of up to 8 inputs. Template B and C generate 2-root patterns with a shared computation block (or, in special cases, just some shared inputs). Template D generates 3-root patterns with some shared inputs. All of these simple templates exhibit concurrent execution in different paths, which potentially produces good speedup when implemented in hardware. They also can have any number of outputs, not limited to just the root nodes (an example is shown in template A). To generate patterns of template A and C, we start from building-blocks that have two inputs. To generate patterns of template B and D, we start from nodes that have two or more successors.

Greedy Pruning: As our patterns can be large, there will be many patterns. To keep the problem tractable, we use a greedy pruning technique: we keep only a fixed number of top-ranked patterns during the generation process. We rank patterns based on their area efficiency: how much speedup we get for the area taken.

We also take into account occurrences: if a building-block occurs (as isomorphic instances) many time, then it should be ranked as more efficient. The tree isomorphism algorithm is used to determine occurrences.

First, we calculate the adjusted area of a pattern *P*:

$$adjusted\ area(P) = \sum_{B_i \text{ inside } P} \frac{area(B_i)}{occurrence(B_i)}$$

where each B_i is a building-block of *P*, and *occurrence* counts the number of occurrences of B_i in the DFG. Here we try to encourage building-blocks that occur more frequently, as they are more likely to be shared between different patterns.

Then the rank of *P* is:

$$rank(P) = \frac{speedup(P)}{adjusted\ area(P)}$$

Dominance Pruning: As we are trying to minimize execution time (maximize total speedup), we have the following observation: if pattern P_A is contained within pattern P_B, and $Speedup(P_A) \geq Speedup(P_B)$ (P_A has fewer inputs, for example), then we can safely remove P_B from consideration (because area of P_B is larger than that of P_A).

4.4 Area and Speedup Estimations

Area Estimation: First, we estimate the area of each building-block. This is calculated as the sum of the hardware areas of all the nodes in the block. This is done at the end of the building-block enumeration phase. Next, we do not calculate the area for our combined patterns, because they can share building-blocks. Instead, we estimate the total area for each mapping solution during the application mapping phase. The total area is estimated as the sum of areas of all building-blocks used in the mapping solution, not the sum of the areas of the patterns used. This calculation is done in the application mapping phase, which is described in 4.5.

Speedup Estimation: In our model, the speedup is the difference, in term of clock cycles, between executing the pattern in software and executing it as a custom instruction in hardware:

$$Speedup = T_{SW} - \left(\lceil T_{HW} \rceil + T_{IO}\right)$$

T_{SW} is the software execution time. It is estimated as the total sum of the software execution time of each node. Here we assume an ideal pipeline (no data or execution hazards).

T_{HW} is the hardware execution time. It is the estimated length of the critical path of the pattern. As this can be non-integer, the ceiling operator is used to round it up to the nearest integer.

T_{IO} is the extra cycles (if any) required to transfer data (both inputs and outputs):

$$T_{IO} = \left(\left\lceil \frac{|inputs|}{N_R} \right\rceil - 1\right) + \left(\left\lceil \frac{|outputs|}{N_W} \right\rceil - 1\right)$$

where |*inputs*| and |*outputs*| are the numbers of inputs and outputs of the pattern, respectively, and N_R and N_W are the numbers of read and write ports of the register file, respectively. This is because in one cycle, we can, at best, input N_R values and output N_W values. Note that we have one default cycle for inputs and for outputs, which are not counted as extra cycles.

4.5 Application Mapping

In this section, we are to cover the DAG with the candidate patterns. The goal is to have minimum execution time (maximum speedup), while keeping the total area under constraint. We decide to use a branch-and-bound algorithm to efficiently walk through the binary search tree of the searching space. Branch-and-bound approach is also presented in [6] to solve a different covering problem, which tries to minimize the number of custom instructions used without any constraints. The major differences between our algorithm and theirs are that our goal of maximizing speedup requires a different LBC (Lower-bound cost) calculation, and we also handle area constraint with basic-block sharing.

Figure 7 shows an example search tree with four candidate patterns (not all nodes are shown). The root node, "*xxxx*", corresponds to the mapping where all patterns are undecided (*x*). An internal node, "01*xx*" for example, corresponds to the partial mapping where the first pattern is not used (0) and the second pattern is used (1). The leaf nodes at the bottom of the tree correspond to a complete mapping solution.

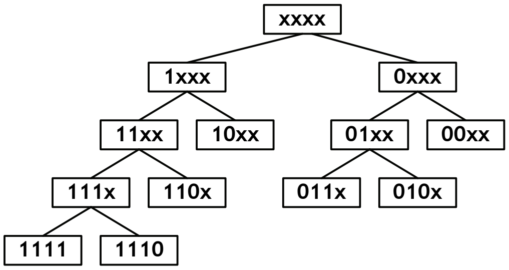

Figure 7. An example search tree

To explain the algorithm, the following terms are defined:

Cost: the cost of a node is the negative of the total speedup of the corresponding mapping, with all undecided nodes considered as not used. Thus, minimum cost means maximum speedup. The cost of a node cannot be lower than the cost of any of its descendant nodes.

Area: the area of a node is the estimated area of the corresponding mapping, with all undecided nodes considered as not used. As mentioned in 4.4, this estimation considers building-block sharing. A search node is called *feasible* if its estimated area satisfies the area constraint. It follows that if a node is not feasible (violates the area constraint), then all its descendant nodes are also not feasible.

Lower-bound cost (LBC): The LBC of a node is the lowest cost that its descendant leaf nodes can possibly get. If a node has a LBC of *x*, then all its descendant leaf nodes have at least a cost of *x*.

We explore the binary search tree depth-first, and always trying to use the patterns in the mapping first (value of '1', the greedy branch to the left in Figure 7). If this node is feasible and not bounded, we continue exploring further down this branch. Otherwise, we can trim away this branch and try the other possibility of not using the pattern in the mapping (value of ‘0’). This way, we can use the cost of the greedy solution to limit the search. The algorithm is presented in Figure 8. The algorithm searches the optimal (least cost) solution by alternating between two steps.

First, in *branch* step, we go from the current search node (at level *pId*) down to its two direct descendants (level *pId*+1), trying the greedy “pattern used” branch first. At this point, all patterns *i* up to *pId* are already decided, either used (*M*[*i*] = 1) or not used (*M*[*i*] = 0). All patterns beyond *pId* are still undecided (*M*[*i*] = *x*).

Then, in the *bound* step, we check to see if we can skip exploring any further down the tree. We can trim the search tree if the LBC is higher than *MinCost*, the minimum cost that has been found during the searching. We also trim away the searching if the area constraint is violated.

```
Input:   S: List of candidate patterns, sorted by rank
         MaxArea: Area constraint
Output:  BestSolution (minimum cost and satisfy area constraint)

M = {x,x,…,x}
MinCost = 0
Calculate initial LBC (with M = {x,x,…,x})
Cover(1, true, M, 0, LBC)
Cover(1, false, M, 0, LBC)
return

Cover(pId, Used, M, Speedup, LBC) {
    if (pId > sizeof(S))
        return     // no more patterns
    if (Used) {
        Calculate TotalArea
        if (TotalArea > MaxArea)
            return     // violate area constraint
        M[pId] = 1
        Speedup += S[pId].Speedup
        if (MinCost > –Speedup) {
            MinCost = –Speedup
            BestSolution = M
        }
    }
    else   M[pId] = 0

    // LBC Bound
    Update LBC
    if (LBC > MinCost)
        return     // this solution cannot possibly be the best

    // Recursive depth-first Branch
    Cover(pId+1, true, M, Speedup, LBC)
    Cover(pId+1, false, M, Speedup, LBC)
}
```

Figure 8. Branch-and-Bound Covering Algorithm

LBC Calculation: A tight LBC can trim away more early on during the search, but may demand more computational time. Here we describe our LBC calculation, which offers a satisfactory trade-off. We ignore the area constraint, and calculate an upper bound of the speedup when all undecided patterns are used for mapping. Because a DFG node may be covered by many patterns, but can contribute to the speedup only once, we try to find the maximum possible speedup for each node.

We distribute the speedup of each pattern *P* over all its nodes, with the partial-speedup *PS*:

$$PS(P) = \frac{1}{|P|} Speedup(P)$$

For a node N_C already covered by pattern P_C, its partial-speedup is fixed:

$$PS(N_C) = PS(P_C)$$

For an uncovered node N_U, its upper-bound partial-speedup is:

$$PS(N_U) = \max_{P_i : N_U \in P_i} PS(P_i)$$

where each P_i is an undecided pattern that covers N_U.

Then the *upper-bound speedup* (*UBS*) is:

$$UBS = \sum_{N_i \in DFG} PS(N_i)$$

And finally the *lower-bound cost* (*LBC*) is simply:

$$LBC = -UBS$$

Note that when an undecided pattern becomes decided (as either mapped or not-mapped), only a small number of nodes have their *PS* changed. Therefore, we can quickly calculate *LBC* from previous value of *LBC*. Also, the list of patterns covering each node is pre-sorted according to their *PS* to help with the MAX operation. The initial LBC is calculated for the root node of the search tree, where all patterns are undecided.

5. EXPERIMENTAL RESULTS

5.1 Experiment Setup

We implemented the algorithm in a C++ environment. The initial DFGs were generated by a custom pass within the MachSUIF framework [10]. Some internal C benchmarks and some C benchmarks from MediaBench [11] were compiled into DFG representations by the MachSUIF flow, preprocessed with an if-conversion pass from [12]. Because our ASIP architecture does not support memory access in the custom logic, our algorithm does not consider memory instructions for implementation in custom instructions. We only consider custom instructions from the computational kernels of these benchmarks. Table 1 shows the sizes of the largest basic blocks in the computational kernels.

Table 1. Benchmark Size

Benchmark	Max. DFG Size (nodes)
adpcm_dec	46
adpcm_enc	68
chendct8	180
dir	131
mcm	97
lee	62
iir	43

The ASIP architecture for our experiment is an Altera's Nios II platform on a Cyclone II FPGA system board (EP2C35 device) [13]. This CPU supports extending the functionality of the ALU through custom logic, with 2 read ports and 1 write port on the register file. The CPU is clocked at 100 MHz. We estimate the hardware latency of different primitive operations from the timing reports of the synthesis results using Quartus II 6.0 flow (Table 2).

Table 2. Estimated hardware latency

Operation	Latency (ns)
Add/Subtract	2.5
Logical	1.0
Shift	0.5
Multiply	12.0

5.2 Estimated Speedup

The speedup estimation is described in section 4.4. The latency of a custom instruction is derived from its critical path delay. The total runtime is estimated as the sum of the execution cycles of all the instructions. Estimated speedups (runtime after ASIP compilation versus software-only runtime), without area constraint, are shown in Figure 9. We present two set of data: one for a 2 read port architecture and another for a more capable 4 read port architecture. Because we already consider patterns with large number of inputs/outputs, the difference between these two is only the extra cycles for inputs.

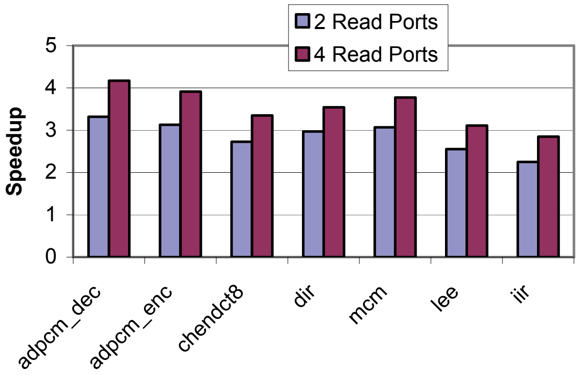

Figure 9. Speedup for different benchmarks

This result should not be compared directly to the result presented in [1], because of different environments. The platform in [1] is a Stratix FPGA, while we use a low-end Cyclone II platform [13]. We provide our estimated logic latencies in Table 2, but no hardware latency information was mentioned in [1]. Another difference is that [1] implemented a 50 MHz Nios processor, while the processor in our experiment is a more advanced Nios II running faster at 100 MHz. It would be harder to achieve hardware speedup on a faster CPU.

To have some form of comparison, we run our algorithm but skipping the patterns combination step in 4.3. This means only patterns of up to four inputs are allowed, which is similar to what presented in [1]. The result in Table 3 shows that we can have about 50% more speedup when larger patterns are allowed.

Table 3. Speedup under different input sizes

Benchmark	Up to 4	More than 4	Increase (%)
adpcm_dec	2.13	3.32	55
adpcm_enc	2.07	3.13	51
chendct8	1.53	2.73	78
dir	1.57	2.97	89
mcm	2.33	3.07	32
lee	1.61	2.55	58
iir	1.81	2.25	24

For an example, Figure 10 shows a pattern generated in the *dir* benchmark. This pattern has 6 inputs (requiring 2 extra input cycles) and 2 outputs (requiring 1 extra output cycle). It is the combination of four building-blocks, shown as the dotted triangles, according to template C in Figure 6. In this case, the shared building-block of template C reduces to just one common input (the input in the middle of Figure 10). We can see that all 8 multipliers run concurrently, much faster than executing 8 sequential multiply instructions if implemented in software.

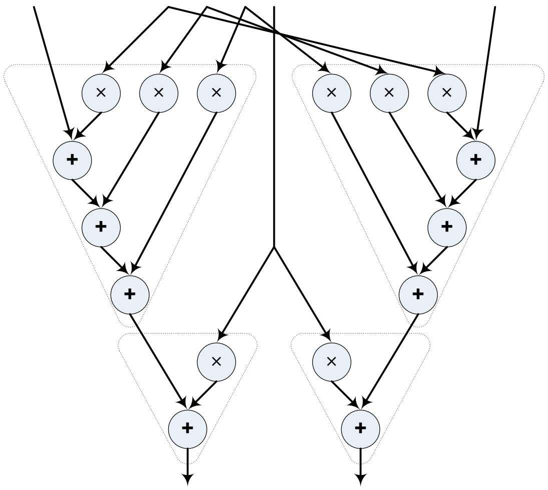

Figure 10. A 6-input 2-output pattern used in the *dir* benchmark

Speedup for different area constraints: by varying the area constraint, we obtain different area/speedup tradeoff configurations. We also compare results between enabling and disabling resource sharing. Figure 11 shows these results for the *chendct8* benchmark. With the help of building-block sharing, we can have up to 25% more speedup under the same area constraint.

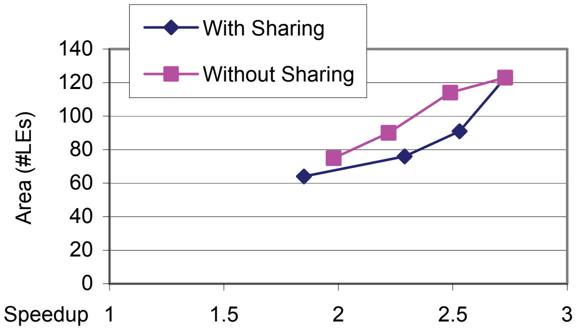

Figure 11. Different area/speedup configurations

5.3 Nios II Implementation

To validate our estimated speedup, we implement the Nios II ASIP on a Cyclone II system board. The Nios II was configured to be fully pipelined and optimized for performance, running on a 100 MHz clock. The custom instructions are written manually in Verilog. Quartus II 6.0 flow was used for synthesis and physical design of the custom logic and the whole system.

The speedups when implemented on the Nios II platform are shown in Table 4. These results agree with our estimated speedups. The small differences are due to pipeline and memory/cache operations in the Nios II and compilation with custom instructions by the Altera's compiler.

Table 4. Speedup on Nios II Platform

Benchmark	Estimated	Speedup
adpcm_dec	3.32	3.15
adpcm_enc	3.13	2.90
chendct8	2.73	2.69
dir	2.97	2.77
mcm	3.07	2.87
lee	2.55	2.41
iir	2.25	2.32

6. CONCLUSIONS

In this paper, we presented two techniques to allow efficient ASIP application to simple embedded computer architectures. We relax the IO constraint on the custom instructions by allowing extra data transfer cycles. We then improved resource usage by enabling the sharing of portions of different custom instructions. We propose several heuristic approaches to keep the runtime of the automated design flow under control. Experiments show that our proposed algorithm can produce quality results efficiently.

ACKNOWLEDGEMENT

The authors would like to thank Altera Corporation for supporting this research. We wish also to thank Nam Duong for his initial discussion.

REFERENCES

[1] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-Specific Instruction Generation for Configurable Processor Architectures," *Proc. of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 183-189, Feb. 2004.

[2] J. Cong, Y. Fan, G. Han, A. Jagannathan, G. Reinman, and Z. Zhang, "Instruction Set Extension with Shadow Registers for Configurable Processors," *Proc. of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, CA, pp. 99-106, Feb. 2005.

[3] L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1209-1229, Jul. 2006.

[4] L. Pozzi and P. Ienne, "Exploiting pipelining to relax register-file port constraints of instruction-set extensions," *Proc. of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pp. 2-10, Sep. 2005.

[5] N. Clark, H. Zhong, and S. Mahlke, "Automated Custom Instruction Generation for Domain-Specific Processor Acceleration," *IEEE Trans. on Computers*, vol. 54, no. 10, pp. 1258-1270, Oct. 2005.

[6] N. Clark, A. Hormati, and S. Mahlke, "Scalable Subgraph Mapping for Acyclic Computation Accelerators," *Proc. 2006 Intl. Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 147-157, Oct. 2006.

[7] F. Sun, S. Ravi, A. Raghunathan, and N. Jha, "Custom-Instruction Synthesis for Extensible-Processor Platforms," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 2, pp 216-228, Feb. 2004.

[8] D. Chen and J. Cong, "DAOmap: A Depth-Optimal Area Optimization Mapping Algorithm for FPGA Designs," *IEEE International Conference on Computer-Aided Design*, pp. 752-759, Nov. 2004.

[9] J. Cong, C.Wu, and Y. Ding, "Cut Ranking and Pruning: Enabling a General and Efficient FPGA Mapping Solution," *Proc. of the 7th ACM/SIGDA International Symposium of FPGAs*, pp. 29-35, Feb. 1999.

[10] M. D. Smith and G. Holloway. (2000). *An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization*. Cambridge, MA: Harvard Univ., http://www.eecs.harvard.edu/hube/software/

[11] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," *Proc. of the 30th Annual International Symposium on Microarchitecture*, Dec. 1997, pp. 330–335.

[12] L. Rolaz, *An Implementation of If-conversion using select instructions for Machine SUIF*, EPFL, 2003, http://lap.epfl.ch/dev/machsuif/opt_passes/

[13] Altera Corp., http://www.altera.com

[14] Xilinx Inc., http://www.xilinx.com

[15] Tensilica Inc., http://www.tensilica.com