Application-Specific Instruction Generation for Configurable Processor Architectures

Jason Cong, Yiping Fan, Guoling Han, Zhiru Zhang
Computer Science Department, University of California, Los Angeles
Los Angeles, CA 90095, USA
{cong, fanyp, leohgl, zhiruz}@cs.ucla.edu

ABSTRACT

Designing an application-specific embedded system in nanometer technologies has become more difficult than ever due to the rapid increase in design complexity and manufacturing cost. Efficiency and flexibility must be carefully balanced to meet different application requirements. The recently emerged configurable and extensible processor architectures offer a favorable tradeoff between efficiency and flexibility, and a promising way to minimize certain important metrics (e.g., execution time, code size, etc.) of the embedded processors. This paper addresses the problem of generating the application-specific instructions to improve the execution speed for configurable processors. A set of algorithms, including pattern generation, pattern selection, and application mapping, are proposed to efficiently utilize the instruction set extensibility of the target configurable processor. Applications of our approach to several real-life benchmarks on the Altera Nios processor show encouraging performance speedup (2.75X on average and up to 3.73X in some cases).

Categories & Subject Descriptors

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

General Terms

Algorithms, Performance, Design, Experimentation

Keywords

ASIP, configurable processor, compilation, technology mapping, binate covering

1. INTRODUCTION

Embedded systems have been widely used in various fields in today’s world. However, designing a modern embedded system in nanometer technologies is more difficult than ever, and the problems continue to worsen with shrinking feature sizes. Due to the complexity and electrical design challenges posed by each new technology generation, the design productivity gap continues to grow larger despite the increasingly expensive CAD tools. This urges a move toward the use of programmable and reconfigurable solutions to allow more flexibility for accommodating specification changes and avoiding potential design errors.

Application-Specific Instruction-set Processors (ASIPs) have gained popularity in production chips as well as in the research community. They offer a viable solution to tradeoff between efficiency and flexibility for the embedded System-on-a-Chip (SoC). Generally, an ASIP has the capability to extend the base instruction set of a general-purpose processor with a set of customized instructions supported by the specific hardware resources provided on the ASIP. The hardware implementing the specific instructions can be either runtime reconfigurable functional units [10][5], or pre-synthesized circuits [27]. As an example, Figure 1 (taken from Altera’s website [24]) shows the instruction logic of a commercially available configurable processor architecture, called Nios. The custom logic can extend the functionality of the Nios ALU by implementing the custom instructions for complex processing tasks as either single-cycle (combinatorial) or multi-cycle (sequential) operations.

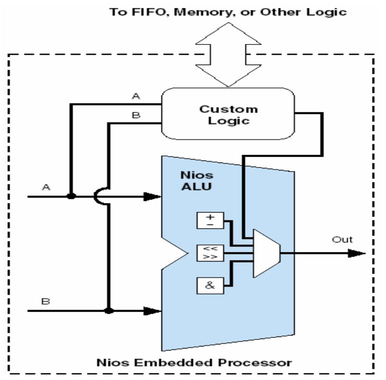

Figure 1. Custom instruction logic of Nios.

Selecting an optimal extended instruction set is crucial to enhancing the performance of the ASIP. However, for the large programs, this is a difficult task to be achieved by manual designs, and is further complicated by various design constraints, such as the format of the extended instructions (e.g., the number of input and output operands), clock period, available chip area, etc. We believe that a fully automated compilation flow is needed to generate application-specific instructions, taking full advantage of the extensisble capability of the ASIP.

Several techniques and tools to aid ASIP design automation have been presented in recent years. [13] proposed a template generation, matching, and covering algorithm. The candidate templates are first generated by a clustering algorithm based on the occurrence frequency. Then the directed acyclic graph (DAG) covering is formulated as the maximum independent set (MIS) problem to maximize the number of covered nodes using a minimum number of templates. Unfortunately, the optimization objective to minimize the number of templates may not lead to faster execution. Moreover, real-life processors always have a limited number of input and output operands, but this work does not address the architecture constraints for the templates.

A more general method for application-specific instruction-set extension is presented in [2]. The authors define the candidate extended instruction to be a convex directed acyclic subgraph (which is defined as a *cut*) with certain input and output constraints. They use a branch and bound method to identify a single cut in a basic block with maximum speedup. The algorithm can be also extended to find the best set of disjoint cuts in multiple basic blocks with maximum sum of speedup. However, the complexity of the branch and bound algorithm grows very fast when the number of instructions becomes large. Also, the objective to maximize the sum of speedup of each individual cut may not result in the minimum execution time. More importantly, cut reuse is not considered in this work.

A complete ASIP compilation flow is proposed in [19]. The flow contains two phases: instruction selection and instruction mapping. The authors use either a greedy algorithm or the method in [2] to solve the instruction selection problem. Symbolic algebra is used to estimate the cost of every specific instruction and to map the application to the generated instruction set. The objective of the mapping is to decompose the polynomial representation of the code into a minimum number of polynomial representations of the instructions. Again, the goal to minimize the instruction number cannot guarantee the minimum execution time, since an extended instruction could have a latency of multiple clock cycles.

Interestingly, instruction overlapping (or operation duplication) may also improve the resulting execution time, while most prior works only generate extended instructions with disjoint node sets, either during instruction selection [2][19], or DAG mapping [13]. Figure 2 shows a data flow graph in which two subgraphs share a multiplication operation. The operation will be duplicated when the subgraphs are implemented as two extended instructions. Clearly, the operation duplication provides more opportunities for speedup, as opposed to the non-overlapping constraint assumed by previous works.

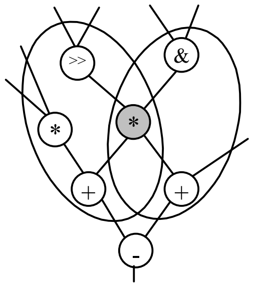

Figure 2. Operation duplication.

In this paper, we propose a new performance-driven approach to the application-specific instruction generation for the configurable processor architectures problem (or *ASIP compilation problem*, for short). The problem is solved in three steps. We first enumerate all candidate *patterns* for the given data flow graph, subject to the given constraints. Instruction set selection is then performed in the second step. A cost function that considers the occurrence, speedup, and area cost of a pattern is calculated to guide the selection. In this step, a graph isomorphism algorithm is used to count the occurrence of a certain pattern. In the final step which is called *application mapping* in this paper, we map the data flow graph into the selected patterns to minimize the total latency by binate covering.

Our contributions in this work are as follows:

- This work transforms the application mapping problem to a library-based technology mapping for the area minimization problem, which has been extensively studied in the logic synthesis domain. Any existing algorithms to solve the minimum-area technology mapping problem, such as binate covering [20] and tree-based decomposition [20][15], can be applied.
- This work allows the operation duplication implicitly during the cut enumeration and the mapping, and thus potentially achieves a higher speedup.
- In contrast to previous works, our optimization goal is the minimum execution time, which is the actual performance metric of the processor.

The rest of the paper is organized as follows. We formulate the ASIP compilation problem in Section 2. Section 3 introduces our algorithms to solve the problems, including pattern enumeration, pattern selection, and application mapping. Experimental setup and results are presented in Section 4, followed by conclusions in Section 5.

2. PROBLEM STATEMENT

Traditionally, applications are specified by programs in high-level languages. Compilation optimization algorithms are usually performed on the control data flow graph (CDFG) derived from the program. A control flow graph consists of a set of basic block nodes and control edges. Each basic block node is a data flow graph in which operation nodes are connected by edges that represent data dependencies. We use $G(V, E)$ to denote a data flow graph, which is essentially a DAG. Without loss of generality, we assume $G(V, E)$ contains only one source node and one sink node. Otherwise, a new source or a new sink could be added into the graph, and edges from the new source to the old ones and those from the old sinks to the new sink could be constructed to meet the assumption. In addition, We assume that $G(V, E)$ is already decomposed according to a given basic instruction set, so that every node (except the source and sink) corresponds to a basic instruction.

We define a pattern p as a cone. For a node v in the DAG, a cone of v, denoted as C_v, is a subgraph consisting of v and its predecessors, such that any path connecting a node in C_v and v lies entirely in C_v. v is the root of C_v. And in our case, a trivial pattern contains only one node and can be implemented as a basic instruction. A non-trivial pattern satisfying given constraints (described below) can be implemented as a special instruction. We will not distinguish the terms *pattern* and *extended instruction* hereafter.

Every pattern p is associated with execution time in software, execution time in hardware, input and output numbers, and occurrence, etc. In addition, every non-trivial pattern is also associated with an area usage when it is implemented in custom logic. For a trivial pattern, we define its execution time in hardware to be equal to that in software.

Since most existing configurable processors only have one write port in register file (or memory) [24][8], we only consider the instruction format with multiple inputs and single output (MISO).

Considering the limited reconfigurable resources, we introduce area constraint for the final ASIP implementation. Let N_{in} be the number of read ports in the register file of the target ASIP architecture, A be the area constraint, and $P = \{p_1, ..., p_N\}$ be the set of selected non-trivial patterns. We have:

(i) $|IN(p_i)| \le N_{in}, \forall i;$

(ii) $|OUT(p_i)| = 1, \forall i;$

(iii) $\sum_{1 \le i \le N} area(p_i) \le A$,

where $IN(p_i)$ and $OUT(p_i)$ are the input set and output set of pattern p_i, and $area(p_i)$ is the area usage when pattern p_i is implemented in custom logic.

The **ASIP Compilation Problem** can be formulated as follows:

NOTATIONS:

- *I:* Basic instruction set
- $\mathcal{S}$: Set of all candidate non-trivial patterns
- *P:* Pattern library (i.e., set of selected non-trivial patterns)
- I^+: Extended instruction set, i.e., $I^+ = I \cup P$

PROBLEM: Given *G(V, E)*, constraints (i), (ii) and (iii) described as above, and a basic instruction set *I*, generate a selected pattern library *P* and map *G* to the extended instruction set I^+, so that every node $v \in V$ is covered and the total execution time is minimized, where the total execution time is the sum of the execution time of every pattern instance used in the mapping.

We believe that the ASIP synthesis should consider the characteristics of the applications and the extensible architecture simultaneously. However, due to the high complexity of the task, we divide the compilation problem into three sub-problems:

SUB-PROBLEM 1. Pattern Enumeration:

Given a *G(V, E)* and constraints (i) and (ii), generate all of the patterns $\mathcal{S}$ satisfying the constraints.

SUB-PROBLEM 2. Instruction Set Selection:

Given a *G(V, E)*, pattern set $\mathcal{S}$, and constraint (iii), select a subset *P* of $\mathcal{S}$ to maximize the potential speedup while satisfying the area constraint. Note that *P* only contains non-trivial patterns of $\mathcal{S}$.

SUB-PROBLEM 3. Application Mapping:

Given a *G(V, E)*, basic instruction set *I*, and a pattern library *P*, generate a mapping from *G* to I^+ so that the total execution time of *G* is minimized.

3. PROPOSED ALGORITHMS FOR ASIP COMPILATION

We have developed the ASIP synthesis flow shown in Figure 3. SUIF [26] is used to transform the C programs into lower-level representation, from which a CDFG is generated. Standard compilation optimizations, such as loop optimization, have been applied within the transformation process. Given the CDFG and ASIP constraints, pattern generation and instruction selection are performed to produce a pattern library. The refined C program using application-specific instructions is then generated by the application mapping.

We will discuss the three sub-problems and their solutions in detail in the following sections.

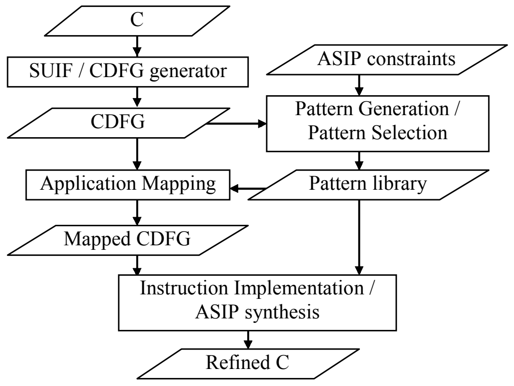

Figure 3. Proposed ASIP compilation flow.

3.1 Pattern Enumeration

For the pattern enumeration problem, all possible application-specific instruction patterns in a DAG should be enumerated. Since the number of input for each cone should not exceed N_{in}, each pattern generated in our algorithm is a *N_{in}-feasible cone* in a DAG. We call a cone (or pattern) *K-feasible* if its input size is less than or equal to *K*.

To identify all patterns with no more than N_{in} inputs, all *N_{in}-feasible* cones for each node in the DAG should be enumerated. A cut of C_v, denoted as $CUT(C_v)$, is defined to be the set of input nodes of C_v. $CUT(C_v)$ is *K-feasible* if C_v is a K-feasible cone. We can see that every cone of a node corresponds to a cut of the node, and vice versa.

A cut can be represented using a product term (*p-term*) of the variables associated with the nodes in the cut. A set of cuts can be represented by a sum-of-product expression using the corresponding p-terms. For a node $u \in CUT(C_v)$, a cut of u is a *subcut* of *v*. It is clear that a cut of *v* can be obtained by merging one subcut from each of its inputs together. We can use an unate Boolean function, called *generating function*, to represent all the cuts based on this representation. For a node *w*, let $f_c(K, w)$ be the generating function for all K-feasible cuts of *w*. For the source node *s* of the DAG, we define $f_c(K, s) = 0$. Then, we can show

$$f_c(K,w) = \otimes^K_{u \in inputs(w)} [u + f_c(K,u)],$$

where operator + is Boolean OR, and $\otimes^K$ is Boolean AND while filtering out all the p-terms with more than *K* variables. For a DAG with a single sink *t*, all *K*-feasible cuts rooted at t are enumerated by $f_c(K, t)$.

The above formulation computes *K*-feasible cuts for node *v* by merging the cuts of the fan-ins of *v* and rejecting those cut combinations that are not *K*-feasible. In theory, the number of *K*-feasible cuts grows exponentially with respect to *K*. However, for $K \le 5$, this computation is very efficient in practice. The same technique is used in FPGA technology mapping [6], where a certain (homogeneous or heterogeneous) LUT library is given for covering a gate-level network. If we regard the LUT size as the input number constraint of the patterns, and regard the gates in the network as operations, these two problems are equivalent.

3.2 Pattern Selection

After the pattern generation, the resource cost and the execution time of every pattern can be obtained using high-level estimation tools. We need to meet the total reconfigurable resource constraint (i.e., constraint (iii)) in the final hardware implementation for the extended instructions. There are two approaches to solving this problem. One is to use all the enumerated patterns during the application mapping. As will be explained later, optimal code can be generated with all the candidate patterns. However, computation for the mapping may become unaffordable due to the extremely large number of enumerated patterns. The other method is to heuristically select a set of patterns satisfying the constraint first, and then use them to cover the application.

We employ the second approach in this work and account for pattern occurrence, speedup, and area simultaneously during the selection decision.

3.2.1 Pattern Gain Calculation

In [13], the authors select the most frequently appearing pattern with highest priority. However, they only consider pattern occurrence. For less frequent patterns with substantial speedup, the substitution of frequent patterns with these patterns may provide more speedup. Therefore we combine speedup and occurrence as the measurement of gain.

The speedup of a pattern is measured by comparing the estimated cycle number of the execution on customized logic with the estimated cycle number of the execution in software. For a trivial pattern, the software execution time T_{sw} equals the hardware execution time T_{hw}, which is the execution time of the corresponding basic instruction on the general-purpose processors. The software execution time T_{sw} and hardware execution time T_{hw} (in terms of cycles) of a non-trivial pattern p is computed in following equations:

$$T_{sw} = \sum_{n \in V(p)} T_{sw}(n) \qquad (1)$$

$$T_{hw}(p) = \textit{Length of the critical path of scheduled } p, \qquad (2)$$

where $V(p)$ is the node set of p.

Equation (1) indicates that all the instructions in a pattern need to be executed sequentially in a basic (single-issue) pipeline processor. Therefore, the number of cycles should be added. With the consideration of data hazards [11] in the pipeline execution, it is not trivial to compute the total latency. In our estimation, we assume an ideal pipeline without any data hazards. Equation (2) uses the critical path of a schedule to compute the latency. If a pattern is implemented with customized logic, the inherent instruction level parallelism (ILP) could be fully exploited.[1] Here, the critical path can be computed with different scheduling algorithms and resource constraints, such as the number of function units, etc. The speedup of p can be calculated as

$$Speedup(p) = T_{sw}(p) / T_{hw}(p) \qquad (3)$$

To count the occurrence of each type of pattern, a graph isomorphism algorithm is needed to identify whether two pattern instances are identical. The graph isomorphism problem is known to be in the set of NP (nondeterministic polynomial), but it is not clear whether it is NP-complete [12] or not. A number of algorithms such as [21], [3], and [17] have been proposed to compute graph isomorphism. In our system, we use the *nauty* package [25] for the isomorphism test. The pattern size is normally small because of architecture constraints, so the graph isomorphism test is fairly fast.

[1] We assume that the custom logic would not degrade the clock period of the processor.

Combining potential speedup and occurrence together, the gain of pattern p is defined as

$$Gain(p) = Speedup(p) \times Occurrence(p) \qquad (4)$$

3.2.2 Selection under Area Constraint

The selection of the most profitable instructions under area constraint can be formulated as a 0-1 knapsack problem.

0-1 Knapsack Problem: Given n items and weight W, and the ith item is associated with value v_i and weight w_i, select a subset of items to maximize the total value, while the total weight does not exceed W.

In our problem, the gain and area cost of a pattern corresponds to the value and the weight of the item, respectively, and the area constraint corresponds to W. A dynamic programming algorithm can be applied to solve this problem optimally, with the complexity of $O(nW)$.

3.3 Application Mapping

After a pattern library is generated, application mapping covers each node with the extended instruction set to minimize the execution time. The execution time of a mapped DAG is defined as the sum of the execution time of the pattern instances covering the DAG. For each non-trivial pattern instance p, the execution time is $T_{hw}(p)$; for trivial pattern instance p, it is $T_{sw}(p)$. Therefore, the execution time T of the mapped DAG is

$$T = \sum_{p:\text{ non-trivial}} T_{hw}(p) + \sum_{p:\text{ trivial}} T_{sw}(p)$$

The optimal mapping refers to the covering with minimum execution time.

THEOREM: The application mapping problem is equivalent to the minimum-area technology mapping problem.

The basic idea of the proof is as follows:

Library-based technology mapping [18] transforms a technology-independent logic network into a bounded network, i.e., into an interconnection of components that are instances of element of a given library. For minimum-area technology mapping, the total area after mapping needs to be minimized.

Given an instance of the minimum-area technology mapping problem with a logic network N and cell library L, we can make the transformations to the application mapping problem with a subject graph G and extended instruction library I^+ as follows: (1) G is constructed by directly making a copy of N; (2) I^+ is constructed by interpreting every component of L as a pattern, and the area value of each component as the execution time of the corresponding pattern. Since the total execution time after application mapping is the sum of the execution time of all the pattern instances covering the subject graph, it is obvious that a minimum execution time solution for the application mapping problem is also a minimum-area solution for the original technology mapping problem.

Vice versa, we can also reduce the application mapping problem to the minimum-area technology mapping problem in a similar way. Therefore, these two problems are equivalent.

COROLLARY: The application mapping problem is NP-hard.

Since library-based technology mapping for the area minimization problem is proven to be NP-hard [14], the application mapping problem is NP-hard as well, according to the above theorem.

Several approaches have been proposed for minimum-area technology mapping. In [20] the DAG is partitioned into a forest of trees for DAG covering. Then a tree pattern matching automation is used to match the individual trees. Dynamic programming, based on the approach of [1] is used to achieve the minimum area mapping. In [20] and [16] binate covering is applied to a DAG covering problem. Although binate covering is an NP-hard problem, much effort has been spent on this because of its wide application. In [4] and [9] an exact solution based on branch and bound algorithm was discussed. The work in [7] has provided better lower-bound computation and two new pruning techniques for an exact solver.

In this work, we use the binate covering approach because it produces the exact solutions with affordable runtimes for the size of our problem instances. We review min-cost binate covering briefly as follows:

Binate Covering Problem: Given a Boolean function $f(x_1,...,x_n)$: $\{0,1\}^n \rightarrow \{0,1\}$ in conjunctive normal form (CNF), and a function C which associates a nonnegative cost with the assignment of variable x_k to a value v (denoted as $x_k(v)$, $v\in\{0,1\}$), find an assignment of $(x_1, ..., x_n)$ to $(v_1, ... ,v_n)$ which evaluates f to 1 and the cost is minimized, where the cost is computed as $\sum_{k=1}^{n} C[x_k(v_k)]$.

Here we use the example DAG in Figure 4 and patterns in Table 1 to illustrate the clause generation. Table 1 lists all the patterns and their functionality, cost and covered nodes. p_0, p_1, p_2, p_3 and p_4 are trivial patterns. p_5, p_6, p_7 and p_8 are patterns of a multiplication followed by an addition. p_9 and p_{10} consist of two parallel multiplications followed by an addition. The cost is directly set to be the execution time. To solve the DAG covering problem, two sets of clauses need to be generated:

- Each node v_i that fans out to the sink node must be covered by at least one pattern. In our example, since n_4 and n_5 are the fan-ins of the sink node, the covering clauses should be created as $(p_3+p_5+p_6+p_9)$ and $(p_4+p_7+p_8+p_{10})$, which means that n_3 can be covered by any of the patterns among p_3, p_5, p_6 and p_9, and n_4 can be covered by any pattern among p_4, p_7, p_8 and p_{10}.
- Any satisfying assignment to the clauses formed so far guarantees that each node fanning out to the sink node is covered by a pattern p_i. In addition, we must ensure that the appropriate inputs are available to each chosen pattern p_i. For example, the inputs of p_3, which are p_5 and p_6, should be covered. Precisely, assume that pattern p_i with m inputs has nodes n_{i1}, n_{i2}, …, n_{im} as inputs. If pattern p_i is chosen, one of the patterns which root at n_{ij} must also be chosen for each input j, $j=1$, …, m. Let s_{ij} be the clause which is a Boolean sum of all the patterns whose root is n_{ij}. Selecting p_i implies that s_{ij} must be satisfied for $j=1$, …, m. This can be written as $p_i \rightarrow s_{ij}$ and be further translated to $\neg p_i+s_{ij}$. For node n_3 in Figure 4, we generate the clauses $(\neg p_3+p_0)$, $(\neg p_3+p_1)$ in the case where we choose p_3. Similarly, clauses $(\neg p_5+p_1)$, $(\neg p_6+p_0)$ are generated in the cases where patterns p_5 and p_6 are chosen, respectively. If node v_i can be covered by p_{i1}, …, p_{ik} with m_1, …, m_k inputs respectively, there are $\sum_{l=1}^{k} m_l$ such clauses.

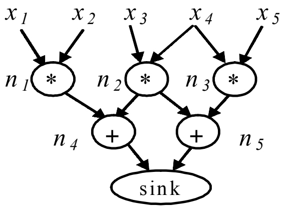

Figure 4. Example graph for DAG-covering.

To cover the DAG, all the clauses should be satisfied. Therefore the Boolean function f is defined as the product of all the clauses. For the example in Figure 4, $f = (p_3+p_5+p_6+p_9)\ (p_4+p_7+p_8+p_{10})\ (\neg p_3+p_0)\ (\neg p_3+p_1)\ (\neg p_4+p_1)\ (\neg p_4+p_2)\ (\neg p_5+p_1)\ (\neg p_6+p_0)\ (\neg p_7+p_2)\ (\neg p_8+p_1)$. Suppose that the cost for every variable in the negative form is 0. It is easy to verity that f is satisfied by selecting p_0, p_1, p_2, p_3 and p_4 with a cost $1+1+2+2+2=8$. f is also satisfied by selecting p_9 and p_{10} with a minimum cost $3+3=6$.

Table 1. A list of patterns.

Pattern	Function	Cost	Covers
p_0	+	1	n_1
p_1	+	1	n_2
p_2	*	2	n_3
p_3	*	2	n_4
p_4	*	2	n_5
p_5	*+	3	n_4, n_1
p_6	*+	3	n_4, n_2
p_7	*+	3	n_5, n_2
p_8	*+	3	n_5, n_3
p_9	(*) + (*)	3	n_4, n_2, n_1
p_{10}	(*) + (*)	3	n_5, n_3, n_2

After binate covering, we have selected patterns for each node in the DAG. In this step the graph is transformed to a new DAG in which each node is a pattern instance. The nodes covered by the same pattern instance will be collapsed to a new node, and the input and output edges of the pattern will be connected to the new node. If a non-root node of a pattern belongs to or fans out to other patterns, the node needs to be duplicated. For the example in Figure 4, if p_9 and p_{10} are selected to cover the DAG, node n_2 will be automatically duplicated, and the mapping solution is shown in Figure 5.

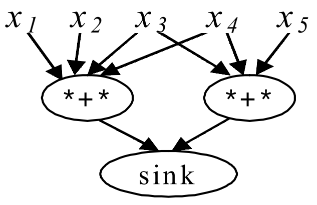

Figure 5. A mapping solution for the example.

4. EXPERIMENTAL RESULTS

We implemented our algorithms in a C++/Unix environment. The C examples used in the experiments are DSP applications from [23] and [22].

Figure 6 shows the relationships between pattern size and occurrence. The trend is quite consistent for these benchmarks. Basically, there are more small patterns than large ones in a DAG. In this experiment, there are few patterns with more than seven operations.

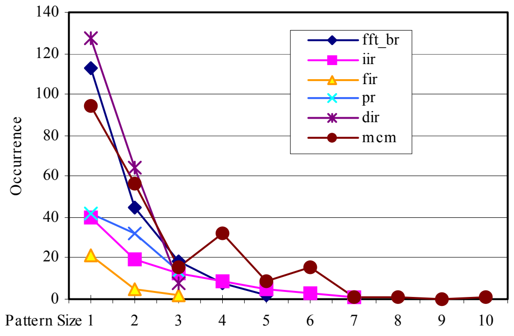

Figure 6. Pattern size vs. number of pattern instances (2-input patterns).

Intuitively, the number of legal patterns becomes larger when the architectural constraints are relaxed. In Figure 7, the number of patterns increases when the input constraint changes from two to four. The bars corresponding to 3- and 4-input constraints indicate the increments of pattern numbers over those with 2-input constraint.

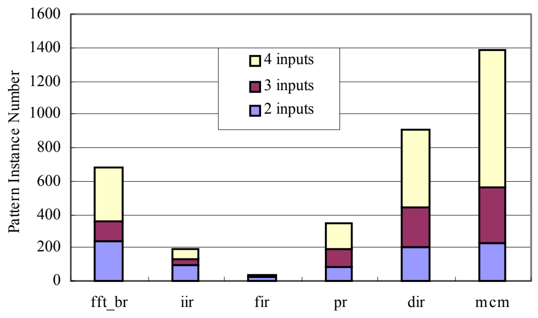

Figure 7. Number of pattern instances under different input size constraints.

4.1 Estimated Speedup

Figure 8 shows the estimated speedups of the compilation results versus the original code executed with the basic instruction set. In theory, in order to obtain an accurate estimation of the execution time for a program running on a processor, we should consider the run-time schedule of the operations and its impact on the processor's pipeline. This could be achieved by simulating the executions on the processor model. However, since we only try to obtain the estimated execution time for comparisons, in our experiment configuration we use an approximate throughput value for every instructions, for example, we assume that a 32-bit multiplication needs two cycles to complete, and an addition needs one cycle. The execution time of the extended instructions is estimated using the method discussed in Section 3.2.1. Since the number of the pattern types is small for these examples, no more than nine, we ignore the area constraint and use all the patterns generated for application mapping in this experiment.

The results shown in Figure 8 indicate that for these examples, a major portion of the speedup is obtained from the 2-input special instructions. Figure 8 also shows that we could achieve more than 4.5 times speedup with the 3-input pattern constraint, and more than 7.5 times with the 4-input constraint.

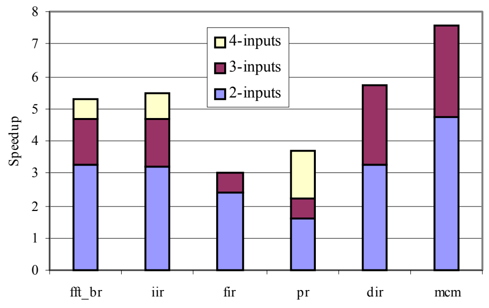

Figure 8. Speedup under different input size constraints.

4.2 Validation on Nios

To validate our estimates, we use a commercial reconfigurable system — Altera's Nios [24] to implement the ASIPs. We used the Stratix version Nios 3.0 system board running at 50 MHz and Altera's Quartus II 3.0 for synthesis and physical design of the custom logic. The Nios processor is able to implement five special instruction formats, each of which could have up to 2048 instructions. Stratix DSP blocks are employed to implement the fast multiplications in custom logic.[2] Table 2 lists several resource numbers of the Stratix EP1S40 device and its usage when a standard Nios (without custom logic) is implemented on it. In our experimentation with the Nios system, we select the MUL option, which configures the processor with the fastest multiplier implementation. We also configure the processor to be fully pipelined and optimized for performance.

Table 2. Resource usage of Nios on Stratix EP1S40.

	Logic Elements	On-chip Memory Bits	DSP Elements
Stratix EP1S40	41,250	3,423,744	112
Standard Nios	6,730 (16.3%)	669,696 (19.6%)	2 (1.8%)

Table 3 shows the results of the speedup and resource overhead when the special instructions are implemented to custom logic, compared to the system running on the basic instruction set. The results show consistent speedups with those we estimated.[3] Specifically, a maximum speedup of 3.73X and an average speedup of 2.75X are achieved for these examples. The average resource overheads are 2.54% in logic element and 1.77% in on-chip memory.

5. CONCLUSIONS

In this paper, novel algorithms addressing application-specific instruction compilation have been discussed. A pattern enumeration algorithm is used to generate instruction candidates, subject to certain port constraints. The pattern library is selected to maximize the potential speedup subject to a total area constraint. We formulate the mapping from the original data flow graph to the extended instruction set as the same problem as the

[2] The optimization during the implementation of extended instructions on the custom logic also influences the final speedup. Currently, the extended instructions are implemented by hand for high quality.

[3] The disparities are due to the ignoring of the impact of the pipeline and memory/cache during the execution time estimation.

area minimization problem for Boolean network mapping. The novelties of this work reside in: (1) transforming the instruction mapping problem into a minimum-area logic covering problem, and allowing many existing algorithms to solve this problem; (2) considering operation duplication implicitly during cut enumeration and mapping; and (3) using the actual performance metric, execution time, as the optimization objective. Experimental results have shown the efficacy of our algorithms.

ACKNOWLEDGMENT

This research is partially funded by MARCO/DARPA Gigascale Silicon Research Center (GSRC), National Science Foundation under award CCR-0096383, and Altera Corporation under the California MICRO program.

REFERENCES

[1] A. Aho and S. C. Johnson, "Optimal Code Generation for Expression Trees," *Journal of the ACM*, vol. 23, pp. 488-501, Jul. 1976.

[2] K. Atasu, L. Pozzi, and P. Ienne, "Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints," *in Proceedings of the 40th DAC Design Automation Conference*, pp. 256-261, Jun. 2003.

[3] L. Babai and E. M. Luks, "Canonical Labeling of Graphs," *in Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pp. 171-183, Dec. 1983.

[4] R. K. Brayton and F. Somenzi, "Boolean Relations and the Incomplete Specification of Logic Networks," *in Proceedings of the 1989 International Conference on Computer-Aided Design*, pp. 316-319, Nov. 1989.

[5] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The Garp Architecture and C Compiler," *IEEE Computer*, vol. 33(4), pp. 62-69, Apr. 2000.

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

[7] O. Coudert, "On Solving Binate Covering Problems," *in Proceedings of the 33rd Design Automation Conference*, pp. 197-202, Jun. 1996.

[8] R. E. Gonzalez, "Xtensa: A Configurable and Extensible Processor," *IEEE Micro*, vol. 20(2), pp. 60-70, Mar. 2000.

[9] A. Grasselli and F. Luccio, "A Method for Minimizing the Number of Internal States in Incompletely Specified Machines," *IEEE Transactions on Electronic Computers*, vol. 14(3), pp. 350-359, Jun. 1965.

[10] S. Hauck, T.W. Fry, M.M. Hosler, and J.P. Kao, "The Chimaera Reconfigurable Functional Unit," *in Proceedings of Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 87-96, Apr. 1997.

[11] J. L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach, Second Edition," *Morgan Kaufmann Publishers*, San Francisco, 1996.

[12] R. M. Karp, "Reducibility Among Combinatorial Problems," *Tech. Rep #3, EECS Department, University of California, Berkeley*, Apr. 1972.

[13] R. Kastner, A. Kaplan, S. Ogrenci Memik, and E. Bozorgzaden, "Instruction Generation for Hybrid Reconfigurable Systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 7, pp. 605-627, Oct. 2002.

[14] K. Keutzer and D. Richards, "Computational Complexity of Logic Synthesis and Optimization," *in Proceedings of International Workshop on Logic Synthesis*, May 1989.

[15] K. Keutzer, "DAGON: Technology Binding and Local Optimization by DAG Matching," *in Proceedings of the 24th Design Automation Conference*, pp. 341-347, Jun. 1987.

[16] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang, "Instruction Selection Using Binate Covering for Code Size Optimization," *in Proceedings of International Conference on Computer Aided Design*, pp. 393-399, Nov. 1995.

[17] B. D. McKay, "Practical Graph Isomorphism," *Congressus Numerantium*, vol 30, pp. 45-87, 1981.

[18] G. D. Micheli, "Synthesis and Optimization of Digital Circuits," *McGraw-Hill,* 1994.

[19] A. Peymandoust, L. Pozzi, P. Ienne, and G. De Micheli, "Automatic Instruction-Set Extension and Utilization for Embedded Processors," *in Proceedings of the 14th International Conference on Application-specific Systems*, Architectures and Processors, Jun. 2003.

[20] R. L. Rudell, "Logic Synthesis for VLSI Design," *Ph.D. Thesis, U. C. Berkeley, ERL Memo 89/49*, 1989.

[21] D. C. Schmidt and L. E. Druffel, "A Fast Backtracking Algorithm to Test Directed Graphs for Isomorphism Using Distance Matrices," *Journal of the ACM*, vol. 23 no. 3, pp. 433-445, Jul. 1976.

[22] M. B. Srivastava and M. Potkonjak, "Optimum and Heuristic Transformation Techniques for Simultaneous Optimization of Latency and Throughput," *IEEE Transactions on VLSI Systems*, vol. 3, no. 1, pp. 2-19, Mar. 1995.

[23] V. Zivojinovic, J. M. Velarde, C. Schlager, and H. Meyr, "DSPStone – A DSP-oriented Benchmarking Methodology," *in Proceedings of International Conference on Signal Processing Application Technology*, pp. 715-720, Oct. 1994.

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

[25] Nauty Package, http://cs.anu.edu.au/people/bdm/nauty.

[26] SUIF Compiler, http://suif.stanford.edu.

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

Table 3. Speedup and resource overhead on Nios implementations.

	Extended Instruction #	Speedup		Resource Overhead				
		Estimation	Nios	LE		Memory		DSP Block
fft_br	9	3.28	2.65	408	6.06%	65,536	9.79%	16
iir	7	3.18	3.73	255	3.79%	4,736	0.71%	40
fir	2	2.40	2.14	51	0.76%	1,024	0.15%	8
pr	2	1.57	1.75	71	1.05%	0	0.00%	14
dir	2	3.28	3.02	54	0.80%	0	0.00%	16
mcm	4	4.75	3.22	186	2.76%	0	0.00%	56
Average		3.08	2.75	-	2.54%	-	1.77%	-