Pattern-Based Behavior Synthesis for FPGA Resource Reduction

Jason Cong
Computer Science Department
University of California, Los Angeles
CA 90095, USA
cong@cs.ucla.edu

Wei Jiang
Computer Science Department
University of California, Los Angeles
CA 90095, USA
wjiang@cs.ucla.edu

ABSTRACT

Pattern-based synthesis has drawn wide interest from researchers who tried to utilize the regularity in applications for design optimizations. In this paper we present a general pattern-based behavior synthesis framework which can efficiently extract similar structures in programs. Our approach is very scalable in benefit of advanced pruning techniques that include locality sensitive hashing and characteristic vectors. The similarity of structures is captured by a mismatch-tolerant metric: graph edit distance. The edit distance between two graphs is the minimum number of vertex/edge insertion, deletion, substitution operations to transform one graph into the other. Graph edit distance can naturally handle various program variations such as bit-width, structure, and port variations. In addition, we apply our pattern-based synthesis system to FPGA resource optimization with the observation that multiplexors are particularly expensive on FPGA platforms. Considering knowledge of discovered patterns, the resource binding step can intelligently generate the data-path to reduce interconnect costs. Experiments show our approach can, on average, reduce the total area by about 20% with 7% latency overhead on the Xilinx Virtex-4 FPGAs, compared to the traditional behavior synthesis flow.

Categories and Subject Descriptors

B.5.2 [**Hardware**]: Design Aids—*automatic synthesis*

General Terms

Algorithm, Design, Experimentation

Keywords

Behavior Synthesis, pattern, FPGA

1. INTRODUCTION

Pattern-based synthesis has drawn wide interest from researchers who tried to extract and utilize the regularity in applications for design optimizations. The common tasks of pattern-based synthesis consist of pattern matching and pattern recognition. Pattern matching is a technique for checking the presence of a given pattern. Representative works in pattern matching include graph-parsing in cognitive studies [32, 26], symbolic equivalence checking [3], AST-based matching [20, 28] and graph isomorphism algorithms [2]. Pattern recognition [30] was initially studied in the machine learning domain for taking action based on the category of the data, i.e., extracting patterns from the raw data. Of all the pattern recognition techniques, structural pattern recognition is a methodology which attempts to describe objects in terms of their parts and connections. Structural pattern recognition is mainly based on graph matching, where each object is represented by a labeled graph. Recently, graph matching has found many applications in data mining, biochemistry and VLSI CAD domains.

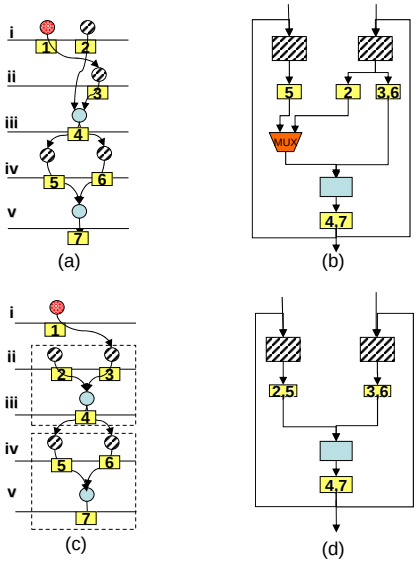

Figure 1: A sample design with different scheduling and binding solutions (different shades represent different function units).

In circuit designs, the intelligent use of regularity usually produces high quality results. Actually, this is one key reason why the deliberate manual design can excel the design synthesized by automated tools. In this paper we attempt to optimize the resource usage of FPGA designs using pattern-based synthesis techniques. Multiplexors, the data routing logics, are particularly expensive for FPGA platforms, e.g., the area, delay and power data of a 32-to-1 multiplexor are almost equivalent to an 18-bit multiplier in modern

FPGA'08, February 24–26, 2008, Monterey,California, USA.

FPGA designs, as shown in [8]. Figure 1 shows how to effectively reduce the number of multiplexers by taking advantage of the regularity in the original specifications. Figure 1(a) is a typical list scheduling result for the given data flow graph with resource constraint $|FU_i| \leq 2$. With the scheduling solution in (a), a possible resource binding solution is shown in (b) where nodes 2 and 3 are bound to the same function unit. For this small example, it is easy to see that nodes {2,3,4} and nodes {5,6,7} have the same structure, which leads to optimized scheduling and binding solutions in (c) and (d). For large designs, the complex interaction among operations make it almost impossible for scheduling and resource binding processes to generate regular data-path without the knowledge of patterns in the original design.

It is not surprising that pattern recognition has been exploited in every level of the large circuit design from layout designs to high-level synthesis [21, 29, 7, 13, 6, 25, 24]. Previous works on pattern matching in behavior synthesis have different limitations, such as pattern size, pattern representation (tree or string), handling program variations, scalability and mismatch-tolerance. In this paper we propose a general and efficient pattern recognition and synthesis framework which benefits from the advanced subgraph enumeration/pruning/matching techniques, targeting computation-intensive applications. Each pattern is represented by a labeled directed acyclic graph (DAG) to capture the data flows inside a basic block in the original program. However a pattern can have instances in multiple basic blocks. In particular, the contributions of our approach include:

(i) Efficient subgraph enumeration and pruning techniques for pattern recognition. A systematic subgraph enumeration methodology is proposed based on the property of DAG to avoid redundant subgraph generation. Characteristic vectors (signatures of patterns) and locality-sensitive hashing are used to prune subgraphs before they are matched with patterns. A hybrid search strategy combines the advantages of both breath-first search and depth-first search to find large and useful pattern instances. Experiments show that our approach can find thousands of patterns (with the largest one containing 40 nodes), all in one minute.

(ii) A graph similarity metric called edit distance [27] to handle variations. The edit distance between two graphs is the minimum number of vertex/edge insertion, deletion, substitution operations to transform one graph into the other. Graph edit distance can naturally handle various program variations such as bit-width variations, structure variations and ports variations. In addition, several interesting theoretic results are presented to bound the edit distance calculation for pruning.

(iii) A pattern-based behavior synthesis flow targeting FPGA for resource reduction. With the knowledge of patterns, our approach minimizes the resource cost through pattern selection, pattern-adaptive scheduling and binding on the basis of the target FPGA platform.

The remainder of the paper is organized as follows. Section 2 discusses related work; Section 3 gives the preliminaries and problem formulation of our pattern-based synthesis problem; Section 4 presents our pattern recognition algorithm; Section 5 presents our overall pattern-based behavior synthesis flow; Section 6 reports experimental results and is followed by conclusions in Section 7.

2. RELATED WORK

Past literature on pattern recognition can be roughly divided into several categories: cognitive approach [32, 26], symbolic equivalence checking [3], and graph-matching based approaches [20, 28, 2]. Cognitive approaches [32, 26] present the pattern as grammar rules, and the parsing process is similar to the automated theorem proof techniques in AI field. The symbolic approach in [3] deploys formal verification techniques to discover subprograms which are semantically equivalent, therefore it can handle more syntactic variations than other approaches. However, both cognitive approaches and symbolic approaches suffer from the scalability problem and are generally not applicable in practice. Graph-matching based approaches, including tree-based and graph-based approaches [20, 28, 2] are usually efficient and scalable, but require careful considerations for handling program variations.

Graph-matching based pattern recognition has been widely applied to data mining, and a number of efficient and scalable algorithms have been developed to find frequent patterns in graphs [17, 23, 33, 19]. AGM [19] uses a level-wise scheme to enumerate the recurring subgraphs. The pattern candidates with size $k+1$ are constructed by joining two frequent graphs with size k, and a frequency count of the current pattern candidate is done by subgraph isomorphism checking. FSG [23] extended the AGM algorithm to handle connected subgraphs, and they both belong to the first category using a breath-first search strategy. Algorithms in the second category use depth-first search to find frequent subgraphs like gSpan [33] and FFSM [17]. Both approaches calculate the canonical label of a graph to avoid redundant subgraph enumeration and graph isomorphism test: gSpan uses a canonical representation of a depth-first traversal of a graph and FFSM uses the adjacent matrix of a graph. The canonical labeling problem is NP-hard in general, and different heuristics have been proposed to incrementally construct the canonical label. All these works can guarantee the completeness of finding frequent subgraphs in terms of graph isomorphism. Instead of these approaches, our work focuses on DAGs. Furthermore, we restrict the graphs to be convex and connected for FPGA resource reduction, and propose efficient subgraph enumeration techniques specifically for DAGs. Our approach can also handle mismatches in the pattern matching step using the edit distance metric.

Regularity extraction has been exploited in the behavior synthesis domain in works [29, 7, 10, 35, 4, 5, 13, 6, 25]. In [29] a string-matching based approach is proposed to cluster similar structures and replace the instances of a pattern with a common implementation. Graphs are represented by a string called K-formula, and this linearizion process is the major drawback since the selection order of nodes can dramatically affect the matching result. The work in [13] tries to improve the quality of logical synthesis by considering patterns at the behavior synthesis step, and the pattern library is given by users. Other interesting works include scheduling and binding algorithms with patterns [6, 25], which both assume that patterns are given in advance. Pattern-based synthesis techniques can also be found in custom instruction set generation like [7, 10, 35, 4, 5]. In [10], patterns are restricted to have only one output. Atasu et al. developed an exhaustive binary-tree search algorithm [4]; however, the connectedness of the subgraph is not considered. The work in [5] proposed a polynomial time algorithm with respect to the input/output port number. Despite the limit of the size on input/output ports, the enumeration process is not purely incremental since the subgraphs grow with an arbitrary size. Another approach in [35] constructs subgraphs by combining single input cones, which shares the same problem with [5]. Also, most of the works avoid the duplicate enumeration using hash tables; therefore the performance is dependent on the number of patterns

and implementation of the hash tables. Our proposed algorithm can discover connected and convex patterns from the original specification in a complete, efficient and incremental way; moreover, duplication checking in our approach is independent to number of patterns which results in dramatic performance improvement and storage space reduction.

There is extensive literature on general binding algorithms in high-level synthesis like [31, 18, 22, 8, 11]. For example, the weighted bipartite matching approach [18] tries to minimize the multiplexers following a step-by-step method, and it is later enhanced by the co-family based approach in [8]. In [11], a distributed register architecture is proposed to reduce the interconnect cost by using abundant embedded memory blocks on FPGAs. The common problem in these approaches is that regularity is not considered and maintained before the binding step. In contrast, our pattern-based approach discovers and preserves the regularity in the whole design flow.

3. PRELIMINARIES AND PROBLEM FORMULATION

This section presents the preliminaries and the problem formulation of our pattern-based behavior synthesis flow.

3.1 Preliminaries

DEFINITION 1. *A **data flow graph (DFG)** is a directed acyclic graph $G(V_o, E_d)$, where the vertices in V_o represent the operation nodes, and the edges in E_d represent the data dependencies (or data transfers) between operations. A directed edge $e(v_i, v_j)$ denotes that operation v_i produces one of the input operands for operation v_j.*

For computation-intensive designs, we typically use data flow graphs to capture the computation kernels. However, our approach is not limited to pure data flow applications only; designs with moderate control flows can be handled in the sense that the designs are treated as a collection of data flow graphs by treating each basic block inside the original program as a DFG.

DEFINITION 2. *A labeled graph G is a five-element tuple $G = \{V, E, \Sigma_V, \Sigma_E, l\}$ where V is a set of vertices and E is a set of edges. Σ_V and Σ_E denote the sets of vertex labels and edge labels respectively. The label function l defines a mapping from vertices and edges to labels.*

The labeled graph provides a proper representation to capture the structure and properties of the data flow graphs. In this paper labeled graphs are used to describe patterns, and naturally we can easily derive a labeled graph of a DFG by choosing the label of a node to be the type of the respective operation (addition, multiplication, etc.).

DEFINITION 3. *The **editing distance d(G_1,G_2)** [27] of two labeled graphs G_1 and G_2 is the minimal sequence of edit operations(relabel of node, insert a node/edge or delete a node/edge) that transform G_1 to G_2.*

Unlike the aforementioned exact matching approaches, we exploit the concept of *editing distance* to describe the similarity of graphs. Please notice that each edit operation can be associated with a cost, and editing distance is redefined to be the minimal total cost of edit operation sequences. Editing distance provides great flexibility for handling variations in the practical designs. Figure 2 shows three kinds of common variations for data-path generation. Structural variation occurs when two DFGs have a small distortion in the structures, such as the operation mismatches in Figure 2(a). In (a), the cost of combining the functionality of a comparator and a subtractor in hardware is marginal, which leads to the optimized data-path on the right. Similarly, bit-width variation and port variation can be perfectly captured using the editing distance metrics in (b) and (c). Furthermore, these decisions are made at runtime which is more flexible than making an arbitrary decision by preprocessing. For instance, we can merge all adders/subtracters to unified addsub operations. However, addition operations are commutative while subtraction operations are not. Conservatively, if we set the addsub operations to non-commutative, potentially many patterns can not be found due to the restrictions. With the concept of edit distance, we can dynamically merge them without losing any pattern.

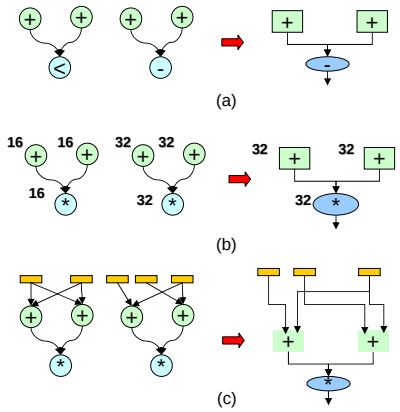

Figure 2: Design variations and corresponding datapaths. (a) structural variation, (b) bit-width variation, (c) port variation.

With respect to editing distance, pattern and pattern instances are defined as the following:

DEFINITION 4. *Given a set of DFGs $\{G_i | i = 1, 2, ..N\}$, a editing distance threshold l_{dist} and a frequency threshold l_{count}, a labeled graph $\mathbb{P}$ is called a **pattern** if $\exists$ set $\mathbb{S} = \{SG_j\}$: (1) $|\mathbb{S}| \geq l_{count}$; (2) $\forall SG_j \in \mathbb{S}$,$\exists G_i$ such that SG_j is a subgraph of G_i; (3) $\forall SG_j \in \mathbb{S}$, $d(SG_j, \mathbb{P}) \leq l_{dist}$. Respectively, each subgraph in set $\mathbb{S}$ is called a **pattern instance** of the pattern $\mathbb{P}$; set $\mathbb{S}$ is called a **pattern group**.*

3.2 Problem Formulation

Using the definitions in Section 3.1, we can formulate our pattern-based synthesis problem. Our proposed approach can be partitioned into two main problems: pattern recognition (PR) and pattern-based synthesis for FPGA resource reduction (PBS-RR).

PROBLEM 1. ***Pattern recognition problem (PR)**. Given a behavior specification in a set of DFGs, find all the patterns as well as all the pattern instances with respect to user-defined edit distance limit and frequency limit.*

PROBLEM 2. ***Pattern-based synthesis problem for FPGA resource reduction (PBS-RR)**. Given all the patterns (or part of all patterns) of a behavior description and platform information of the target FPGA, generate the hardware implementation satisfying certain design constraints and minimize the resource usage of the total design.*

4. PATTERN RECOGNITION (PR)

In this section we describe our pattern recognition algorithm for discovering all feasible patterns inside DFGs. Our approach is efficient and scalable by considering the property of DAGs and using advanced pruning techniques. First, we will introduce some important techniques to solve the **PR** problem. Next, the overall pattern recognition framework is introduced and two different searching strategies are discussed.

4.1 Techniques

4.1.1 Subgraph Enumeration

Subgraph enumeration plays an important role in the pattern recognition algorithm. A good enumeration algorithm should guarantee completeness while avoiding redundant enumerations. For the purpose of multiplexor reduction, the subgraphs enumerated in our approach are limited to:

- **connected**: patterns with disjoint operations will not help to reduce the multiplexors since there are no interconnects among them.
- **convex**: a subgraph G' of graph G is convex if there is no path in G from node $u \in G'$ to node $v \in G'$ which contains a node $w \notin G'$. The requirement for convexity comes from the consideration of better performance estimation. Usually all pattern instances are scheduled in a uniform way. If a pattern is non-convex, the scheduler can not estimate the latency inside a pattern which may result in very large latency overhead. In addition, non-convex patterns usually introduce more ports which result in more multiplexors in final designs.

Specifically for pattern recognition, the subgraph enumeration process should be incremental, i.e., size $k+1$ subgraphs should be enumerated only after all the size k subgraphs are enumerated. The rational behind this requirement is that the pattern pruning process can safely throw away infrequent size k subgraphs before expanding them to size $k+1$ subgraphs. However, if the enumeration process is not incremental, we can not make early decisions to prune useless subgraphs.

Next, a new subgraph enumeration approach is designed for DAGs which can enumerate convex connected subgraphs in a complete, efficient and incremental way. Before details are discussed, several definitions need to be introduced.

DEFINITION 5. *In a connected DAG **G**, a **bridge** node **bv** is a node in **G** whose removal disconnects the graph.*

DEFINITION 6. *In a DAG **G**, **PIs** are nodes without any input edge and **POs** are nodes without any output edge. A **bridge PI(PO)** is a PI(PO) node which is also a bridge. A **leaf PI(PO)** is a PI(PO) node which is not a bridge.*

Our definition of *bridges* uses a form of connectedness called weakly connected. A weakly connected graph is where the direction of the graph is ignored and the connectedness is defined as if the graph was undirected. An example is shown in Figure 3. In (a), node 1 is a leaf input; nodes 6 and 7 are leaf outputs; node 2 is a bridge input. With the definition of leaf PI/PO, we can proof the following theorems.

LEMMA 1. *A DAG G has at least one leaf PI or one leaf PO.*

PROOF: Assume all the PIs and POs in G are bridges. We randomly select a bridge PI/PO v, and removal of node v will result in two or more subgraphs. Let G' be one of these subgraphs. Every bridge PI/PO in G is still a bridge PI/PO in $G' \bigcup v$. Repeat the same process. Finally we can reach a state where a subgraph has two nodes but still has a bridge PI/PO, which is a contradiction.□

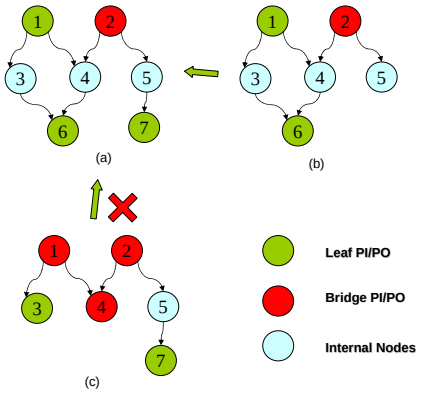

Figure 3: A DAG example and its primary subgraph.

LEMMA 2. *The subgraph G', generated by removing a leaf PI/PO from a convex DAG G, is still convex.*

PROOF: Removing a PI/PO can not destroy the convexity because a PI/PO is not on any path inside G.□

DEFINITION 7. *A graph G' is a primary subgraph of graph G if G' is generated by removing the leaf PI/PO in G which has the biggest ID among all leaf PIs/POs.*

THEOREM 1. *A convex DAG G_{k+1} has a unique primary subgraph G_k that is also convex.*

PROOF: It can be easily seen that there is only one unique way to get the primary subgraph of a DAG. Based on Lemma 2, the primary subgraph is also convex.□

For example, the primary subgraph of the graph in Figure 3(a) is shown in (b), while the graph in (c) is not the primary subgraph because node 7 is the biggest ID among all leaf PIs/POs. Based on the primary subgraph relation, we propose a bottom-up constructive method for the subgraph enumeration problem. At step $k+1$, all the convex subgraphs with k nodes are generated, and they are extended by adding one neighbor in the original DFG. We assume that a convex subgraph G_{k+1} can only be generated by extending its primary subgraph G_k, therefore duplications are avoided. Please be noticed that a different labeling algorithm may change IDs of nodes, therefore the primary subgraph of a subgraph may also be changed; however subgraphs are still guaranteed to be generated only once as long as each ID is unique. Our method only checks a property inside a subgraph, and is totally independent of the total number of patterns. Bridges can be detected by a DFS traverse of the subgraph; in practice, the subgraphs are usually very small, which means the validation of primary subgraph relation literally takes a constant time. Our approach also dramatically reduces the storage requirements since patterns are no longer needed to be kept in memory for duplication checking after they are visited.

4.1.2 Characteristic Vector

Since the computation of graph edit distance is expensive, we compute a signature of a subgraph called *characteristic vector* (CV) similar to the work in [34]. If a

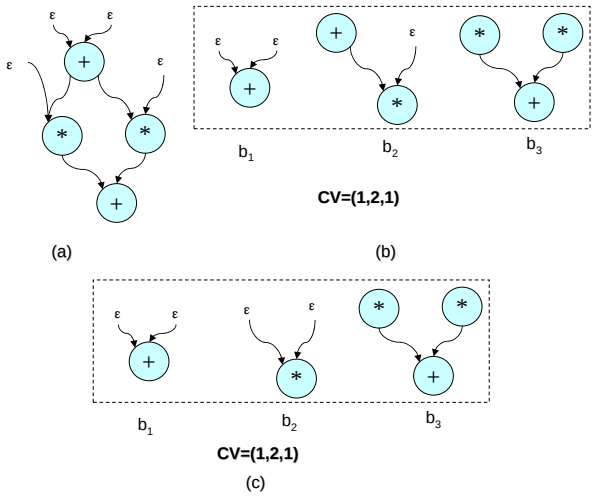

Figure 4: (a) A DFG. (b) Binary branches and CV of the given DAG. (c) Binary branches and CV of the given DAG modified for pattern pruning.

signature of a subgraph is significantly different than the signature of a pattern, this subgraph is not needed for matching with the pattern. Actually, for tree structures, Yang et.al [34] proved that we can estimate the lower bound of edit distance based on CVs. In this section we propose CV generation mechanism for DAGs.

DEFINITION 8. *In a DAG $G = (V,E)$, a **Binary Branch** is a subgraph $\boldsymbol{S} = \{u,l,r\} \subseteq G$, such that edge (u,l) and $(u,r) \in E$.*

DEFINITION 9. *For a given universe of distinct binary branches $\mathbb{U} = \{BB_1, BB_2, ...\}$ with size $|\tau|$, the **Characteristic Vector (CV)** of a DAG G is a vector $(b_1, b_2, ..., b_{|\tau|})$ with each element b_i representing the number of occurrences of the ith binary branch in $\mathbb{U}$.*

In [34], all distinct q-level binary trees serve as the universe $\mathbb{U}$ for *CV* calculation; however it is possible that some binary branches are never used which results in a waste of storage spaces. In our approach, only the binary branches which appear at least once in the given set of DFG are considered. As shown in Figure 4(a)(b), there are three different binary branches in the given graph. Notice that actually b_2 has two instances since multipliers are commutative operations. The symbol ε denotes an empty node if the input of that node is not in this graph. In practice, our method can effectively capture the topological order and labels of a graph since most of the operations in behavior synthesis have less than two inputs. If a node has more than two inputs, we simply ignore this node.

For tree structures, it is easy to see that each node may appear in at most two binary branches; therefore one edit operation, like relabeling, can only slightly change the occurrence of binary branches. However for DAGs the property will not hold since one node may have multiple outputs. To make this relation hold in DAGs, the calculation of CV is slightly changed, as shown in Figure 4(c). If one node has multiple outputs, then this node will be treated as an empty input for all its successors. With this modification, the relation between edit distance and CV can be summarized as the following:

THEOREM 2. *Let $d(G_1,G_2)$ be the edit distance between two DAG G_1 and G_2, $CV(G_1), CV(G_2)$ be the characteristic vectors of G_1 and G_2 respectively, $\|CV(G_1) - CV(G_2)\|_1 \leq 4 * d(G_1,G_2)$.*

PROOF: As we discussed above, one node can only appear in at most two binary branches after we modified the CV calculation method. Since one edit operation (vertex relabeling/insertion/deletion, edge insertion/deletion) can only change at most one node at a time, we can conclude that at most two binary branches are destroyed and at most two new binary branches are generated; i.e., the change in CV in terms of l_1 norm is ≤ 4. Adding all the edit operations together, we come to the conclusion that $\|CV(G_1) - CV(G_2)\|_1 \leq 4 * d(G_1,G_2)$.□

Theorem 2 can help the pattern recognition process to reduce the edit distance calculation. If the edit distance limit is l_{dist}, a subgraph g can not be an instance of pattern p if $\|CV(g) - CV(p)\|_1 > 4 * l_{dist}$.

4.1.3 Locality-Sensitive Hashing(LSH)

After subgraphs are enumerated, they are matched with patterns to see whether they are instances of discovered patterns or not. When the number of found patterns is very large, finding the correct pattern for a given subgraph can be costly using pairwise comparison between subgraphs and patterns. LSH provides a perfect tool to find near-neighbors of a query vector in the Euclidean space, and a characteristic vector serves as the metric for comparison.

DEFINITION 10. *Let $\|\overrightarrow{v}\|_p$ denote the l_p norm of vector $\overrightarrow{v}$; S and U denote vector spaces while $|U| < |S|$. A hashing function family $\mathcal{H} = \{h : S \to U\}$ is called (r_1, r_2, p_1, p_2)-sensitive if for any $u, v \in S$*

- *if $\|u - v\|_p \leq r_1$, then $Pr_{\mathcal{H}}[h(u) = h(v)] \geq p_1$.*
- *if $\|u - v\|_p \geq r_2$, then $Pr_{\mathcal{H}}[h(u) = h(v)] \leq p_2$.*

LSH can hash two similar vectors the same bucket with arbitrary high probability and hash two distant vectors to the same bucket with arbitrary low probability. A LSH implementation can be found in [14]. Using LSH, we can efficiently find all $(1+\varepsilon)$ approximated nearest neighbors of any vector within distance R with arbitrary possibility p in $O(n * \log n)$ instead of $O(n^2)$ for n vectors. Detailed information can be found in [14].

Overall, our pruning process combines LSH and CV to reduce expensive pattern-matching operations. When a subgraph is matched with a set of patterns, we calculate the CV of this subgraph and obtain the nearest neighbors of this subgraph within distance $4 * l_{dist}$ using LSH. As discussed above, the patterns which have CVs out of that range need not to be considered. Experiments show that for each subgraph we only need to calculate edit distance about 2-6 times on average before the corresponding pattern is found.

4.2 Pattern Recognition Algorithm

Having all the important techniques discussed, a pattern recognition (PR) algorithm is proposed in this section. Our algorithm has two phases: a breath-first search step (HPR) and a depth-first search step (VPR). At the beginning, the HPR algorithm exhaustively discovers patterns, level by level. But if there is a big pattern in this design, all its subgraphs are patterns, too; and the potentially exponential number of subgraphs of a big pattern (say 20) can be very huge. If the number of patterns is too large, the pattern recogniion algorithm automatically changes from HPR phase to VPR phase. The VPR algorithm tries to find those big patterns first using depth-first search; therefore, small patterns contained by discovered patterns will not be generated later.

4.2.1 Horizontal Pattern Recognition (HPR)

Algorithm 1 HPR Algorithm

```
1: ℙ          → set of discovered patterns
2: 𝕊_k        → set of size k subgraph
3: INST(P)    → instances of a pattern P
4: l_dist     → edit distance limit
5: l_count    → frequency limit
6:
7: travel all DFGs, add size 1 patterns and instances to ℙ and 𝕊_1
8: for k ← 2, N do
9:    for all s_k ∈ 𝕊_k do
10:      adding a neighbor to expand s_k to s_{k+1}
11:      if s_k is the primary subgraph of s_{k+1} and convex then
12:         calculate CV(s_{k+1})
13:         get list of patterns P_i s.t. ||CV(P_i) − CV(s_{k+1})||_1 ≤ 4 ∗
            l_dist using LSH
14:         calculate edit distance of s_{k+1} with each P_i
15:         if d(P_i, s_{k+1}) < l_dist then
16:            add s_{k+1} to INST(P_i)
17:         else
18:            create a new pattern based on s_{k+1}, add to ℙ
19:         end if
20:         add s_{k+1} to 𝕊_{k+1}
21:      end if
22:   end for
23:   for all new pattern P_i ∈ ℙ do
24:      if |INST(P_i)| < l_count && size(P_i) ≤ (k+1−l_dist) then
25:         remove P_i from ℙ
26:         remove INST(P_i) from 𝕊_{k+1}
27:      end if
28:   end for
29: end for
```

Algorithm 2 VPR Algorithm

```
1: P          → current pattern
2: k          → size of pattern P
3: ℙ_{k+1}    → set of size k+1 patterns
4: 𝕄          → set of maximal patterns
5: INST(P)    → instances of a pattern P
6: l_dist     → edit distance limit
7: l_count    → frequency limit
8:
9: for all s_k ∈ INST(P) do
10:   adding a neighbor expand s_k to s_{k+1}
11:   if s_k is the primary subgraph of s_{k+1} and convex then
12:      similar with HPR algorithm, find pattern for s_{k+1}, or in-
         sert a new pattern to ℙ_{k+1}
13:   end if
14: end for
15: for all new pattern P_i ∈ ℙ_{k+1} do
16:   if P_i is contained by a maximal pattern M ∈ 𝕄 then
17:      continue
18:   else
19:      call VPR(P_i) recursively
20:   end if
21: end for
22: if P is not contained by any pattern in 𝕄 then
23:   add P to 𝕄
24: end if
```

HPR, as suggested by its name, discovers patterns with a breath-first-search approach. The pseudo code of the HPR algorithm is shown in Algorithm 1.

At step $k+1$, all the size k pattern instances are extended by one node using the subgraph enumeration techniques in Section 4.1.1. HPR is complete because subgraphs of a pattern are also patterns. If a subgraph s_k is not a pattern instance of a certain pattern P at step k, it can not be a subgraph of another pattern instance larger than k, which means no further extension is needed for s_k.

Lines $12-20$ show the pruning process in HPR. After a new subgraph s_{k+1} is generated, it is compared to the existing patterns by calculating the edit distance between itself and existing patterns. However, most of the calculation can be avoided using the CV and LSH techniques in Section 4.1.2 and Section 4.1.3. The CV of a subgraph is calculated and used as a key to find the patterns which have close CVs, as shown in line 12. This can be solved by finding the nearest neighbors using LSH in line 13. After getting the list of possible pattern candidates, edit distances are calculated. If s_{k+1} matches a pattern P, it will be added to the pattern instance list of P; otherwise a new pattern will be generated based on s_{k+1}.

When all the subgraphs are processed, each newly generated pattern will be examined to see whether it satisfies the frequency limit. If not, it will be removed together with all its instances. Edit distance further complicates the pruning process since pattern instances may have different node sizes because of vertex insertion/deletion. For example, if l_{dist} is 1, we can not remove size k patterns until we finish searching all size $k+1$ subgraphs. The second condition in line 24 make sure no pattern are thrown away until it is really not useful. To overcome this problem, we can either limit l_{dist} or limit the number of vertex insertions/deletions allowed.

4.2.2 Vertical Pattern Recognition (VPR)

As previously mentioned, subgraphs of a pattern are also patterns, and this may result in an explosion in the number of patterns found in HPR. If the number of patterns is too large, we will begin the VPR phase to find maximal patterns.

DEFINITION 11. *Let P_1 and P_2 be two patterns, sets $S_1 = \{p_1^i | i = 1..n\}$ and $S_2 = \{p_2^i | i = 1..n\}$ denote the instances of P_1 and P_2 respectively; $P_1 \prec P_2$ if P_1 is a subgraph of P_2 and there is bijection between elements of S_1 and S_2: $p_1^i \rightarrow p_2^j$ such that p_1^i is a subgraph of p_2^j.*

DEFINITION 12. *A pattern P is a maximal pattern with respect to a set of DFGs if there is no other pattern P' in the same set of DGFs that satisfies $P \prec P'$.*

Intuitively, a maximal pattern can not be contained by other patterns, and each pattern is a subgraph of a certain maximal pattern. In our VPR algorithm, a depth-first approach is deployed to find maximal patterns; the pseudo-code of VPR is shown in Algorithm 2. The pattern recognition is similar to HPR except that we extend one pattern group at a time instead of a set of pattern groups with size k. At each step, a larger pattern which contains the current pattern will be discovered. VPR can find maximal patterns in a short time using depth-first search. After the maximal patterns are discovered, we can avoid enumerating all their sub-patterns during the search steps by checking to see if the current pattern is contained by certain maximal patterns or not, as shown in line 22. In practice, HPR algorithm will get stuck with large patterns in DFGs; however VPR solves the problem caused by the explosive growth of pattern numbers, and can efficiently find all maximal patterns.

5. PATTERN-BASED SYNTHESIS FLOW FOR FPGA RESOURCE REDUCTION

Our pattern recognition framework can be applied in many practical problems, such as the FPGA resource reduction problem (PBS-RR) discussed in this paper. If all pattern instances are scheduled and bounded in a uniform way, the internal data flows are free of multiplexors (except the multiplexors generated due to resource sharing among nodes inside a single pattern instance). Based on this observation, a pattern-based behavior synthesis flow is proposed in this section for FPGA resource reduction.

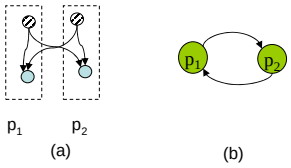

Figure 5: (a) A DFG covered by two pattern instances p_1 and p_2. (b) The reduced graph.

Specifically for the PBS-RR problem, only vertex relabeling is allowed in edit distance calculation. The reason is that vertex insertion/deletion not only increases the resource usage of a single pattern with additional multiplexors to handle variations among pattern instances, but also complicates the scheduling algorithm by introducing latency variations. With this restriction, all the instances of a pattern have the same number of nodes, and the pattern itself can be viewed as a complex operation. Therefore, our pattern recognition framework can be easily integrated into any existing behavior synthesis system, and no specific algorithm for pattern-based synthesis is needed.

However, if patterns are viewed as complex nodes, instances of a same pattern may not be scheduled to the same time step in order to share them like the regular operations. This seemingly restrictive limitation is required because it guarantees that the resource constraints in the original design are maintained. With this restriction, the following issues need to be addressed:

- If instances of a common pattern are required to be scheduled in the same way, there is a possibility that no legal schedule can be obtained. An example is shown in Figure 5. The DFG in Figure 5(a) is covered by two pattern instances p_1 and p_2; we can see that if p_1 and p_2 are scheduled at different time steps, they cannot be scheduled in the same way. If we view each pattern instance as a big operation, the reduced subgraph in (b) has a loop, which means p_1 and p_2 can only be scheduled at the same step. To fix this problem, the pattern selection process will only pick one of these two pattern instances.

- The latency of the design may increase after pattern-based synthesis. As shown in Figure 6, the given DFG can be covered by instances of two patterns. If the pattern in (a) is chosen, the resulting scheduling solution is shown in (b) with the restriction that the execution time of p_1 and p_2 cannot overlap. However, if we choose the pattern in (c), the latency of the final synthesis result in (d) is almost half of the latency in (b). This example suggests that we should choose patterns with shorter critical paths if possible to reduce latency overhead.

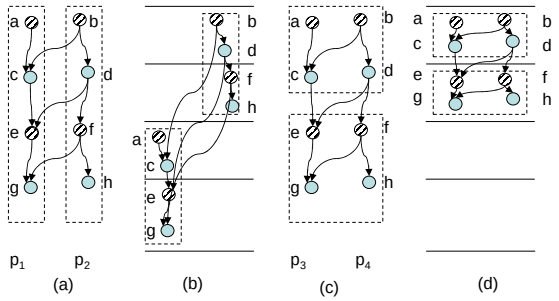

Figure 6: Latency with different pattern selection strategy.

5.1 Pattern Selection

Pattern selection attempts to find an appropriate set of pattern instances which minimize resource usage and latency overhead. In this paper, a greedy algorithm is used. At each step, the best pattern is chosen based on some metrics, and all its pattern instances are removed from the DFGs. The whole process is repeated until the available patterns or nodes are empty. Please take notice that not all instances of pattern P can be chosen, because the pattern instances may not be compatible. Two pattern instances p_1 and p_2 are compatible to each other if p_1 and p_2 do not share common nodes, and there is no loop in the reduced graph as discussed previously in this section.

For the PBS-RR problem, the following metric is used for a given pattern P with N compatible pattern instances:

$$\frac{N * mux(io) + area(P)}{N * (mux(io) + mux(internal)) + area(P)} + \alpha * \frac{|P|}{latency(P)} \quad (1)$$

In Equation 1, the first part of the metric is the area saving with the assumption that each internal edge will not have multiplexors after PBS-RR. The function $mux(e)$ returns the estimated area of 1-input multiplexor needed for data flow edge e; $N * mux(io)$ and $N * mux(internal)$ are estimated areas of N-input multiplexors at PI/POs and internal data flows of pattern P respectively. The second part of the metric is a measurement of "flatness" of P. A pattern P is "flat" if the critical path of P is very small compared to the total nodes of P, and flat patterns are good candidates to reduce the latency overhead. The variable α is a parameter which users can adjust based on their requirements to trade off between latency overhead and resource reduction.

Experimental results show that the greedy pattern selection process is efficient enough to find good pattern candidates. However, it is not hard to further improve the pattern selection process using a mathematical programming based algorithm.

5.2 Scheduling and Binding

After pattern selection, the scheduling and binding algorithms are fairly easily designed to solve the PBS-RR problem. Briefly, each pattern is scheduled and bounded based on the resource constraints in advance to get the respective hardware implementation. Next, patterns are viewed as complex multi-cycle operations, and any state-of-the-art behavior synthesis algorithm can be easily adapted for PBS-RR problem.

In the scheduling step, each pattern instance can be scheduled in the same way by adding relative timing constraints between a pivot node and all the other nodes. The pivot node can be any node in this pattern instance, and the relative timing between a pivot node and other nodes is determined based on the initial scheduling result of the corresponding pattern. For the example in Figure 1, let

s_v denote the time step of node v in the scheduling solution; we can add the following relative timing constraints to make sure that pattern instances $\{2,3,4\}$ and $\{5,6,7\}$ of pattern P are scheduled alike: $s_4 - s_3 = 1, s_4 - s_2 = 1, s_7 - s_5 = 1, s_7 - s_6 = 1$. Nodes 4 and 7 are called pivot nodes because once they are scheduled, the scheduling results of all the other nodes are known as well. Additional constraints are added to make sure that execution times of pattern instances do not overlap; these constraints are similar to resource constraints of normal operations if we treat pattern instances as complex operations. In our approach, the SDC scheduling in [12] is used; it can handle relative timing constraints and other general constraints.

The pattern-based resource binding algorithm should guarantee that the corresponding nodes of pattern instances are assigned to the same function unit. For the scheduling results in Figure 1(c), once node 2 is bound to function unit FU_i, node 4 should be bound to FU_i as well (but if there is a mismatch between node i and node j, these two nodes may not be bound to the same function unit); and this kind of constraint should be considered in our pattern-based resource binding step. In this paper an extension [11] of the iterative bipartite matching algorithm in [18] is adapted for PBS-RR problem. The work in [11] was originally designed for distributed register files; however the core algorithm can be applied to general binding problems as well. At each time step, if a node of a pattern instance knows that the corresponding node in another pattern instance is already bound, the binding solution of this node is known and does not to be considered; other operations are bound using a bipartite-matching formulation found in [18].

6. EXPERIMENTAL RESULTS

6.1 Experiment Setup

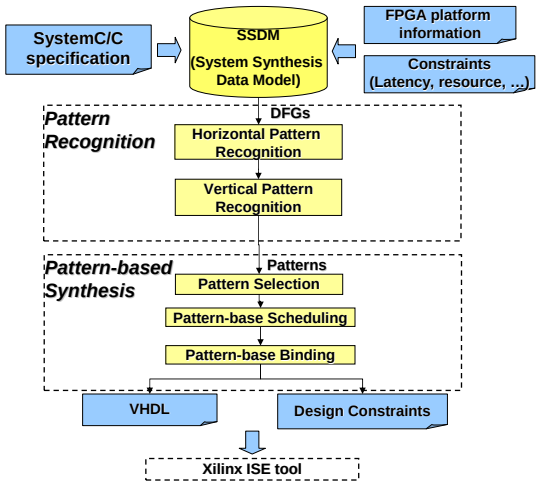

Figure 7: Pattern-based behavior synthesis flow.

Our pattern-based synthesis flow has been implemented in the *xPilot* behavior synthesis system [9]. The whole design flow is shown in Figure 7. *xPilot* takes behavioral languages like C as input and parses them into control data flow graphs. The control data flows graphs are viewed as collections of data flow graphs for pattern recognition. The GMT toolkit [1] is used for graph edit distance calculation, and the LSH implementation can be found at [15]. The synthesis engine will then perform the pattern-based synthesis flow to reduce the resource usage with certain design constraints. The synthesis results are dumped into RT-level VHDL and accepted by the downstream RTL synthesis tools. Our experiments use the Xilinx Virtex-4 FPGA and ISE 9.1 tool [16].

Our test cases include a set of real-life computation-intensive programs: CHENDCT, CHEM, DIR, LEE, PR, FFT and IDCT. The first five test cases are DSP kernels with pure data flow; IDCT and FFT have moderate control flows. Those test cases feature abundant arithmetic computations, and sizes of their corresponding DFGs range from tens to hundreds.

6.2 Effectiveness of Pruning Techniques in Pattern Recognition

The effectiveness of our proposed pruning techniques using CV and LSH is shown in Table 1. The test case in this experiment is CHENDCT, and the edit distance limit l_{dist} is set to 1. The first column lists the size of the patterns. For each row with pattern size k, *#Subgraph* is the number of subgraphs with the size k in the given design; *#Pattern* and *#Inst* are the number of patterns and their instances respectively; and *#Calc* is the average number of edit distance calculations needed before a subgraph matches with one certain pattern.

Table 1: Pattern recognition results on CHENDCT.

Size	#Subgraph	#Pattern	#Inst	#Calc
2	62	3	62	0.96
3	108	12	108	1.08
4	195	20	161	1.48
5	366	26	248	1.49
6	701	35	404	1.9
7	1357	58	579	2.6
8	2533	76	714	3.18
9	4517	86	762	3.82
10	7800	94	793	4.43
11	13112	101	668	7.04
12	21365	73	348	7.89
13	33316	32	87	5.03
14	49040	3	6	1.7

Several conclusions can be drawn from Table 1. Our pattern synthesis algorithm can effectively recognize patterns from an enormous searching space in a complete and systematic way. This example also demonstrates the existence of fairly big patterns in real programs. Moreover, the results show that our proposed pruning algorithm can dramatically reduce the number of expensive edit distance calculations in pattern matching using CV and LSH; on average only about 2-6 calculations are needed with hundreds of pattern candidates.

6.3 FPGA Resource Reduction Results

In this section, the pattern-based FPGA resource reduction algorithm is tested on all seven of the test cases mentioned before. Our algorithm is compared to a traditional behavior synthesis flow deploying state-of-the-art scheduling and binding algorithms in [12, 11]. For a fair comparison, our pattern-based synthesis algorithm extends the same algorithms discussed in Section 5.2. There are several important parameters to be determined in this experiment, such as the edit distance limit l_{dist} and frequency limit l_{count}. The parameter l_{count} is dynamically adjusted in our implementation; i.e., if the number of patterns with size l_{count} is neglectful to the number of whole patterns, l_{count} will be increased. The parameter l_{dist} affects both the runtime and the quality of our algorithm; for the FPGA resource reduction problem, our experiment shows that $l_{dist} \leftarrow 1$ is usually good in the respect that a larger edit dis-

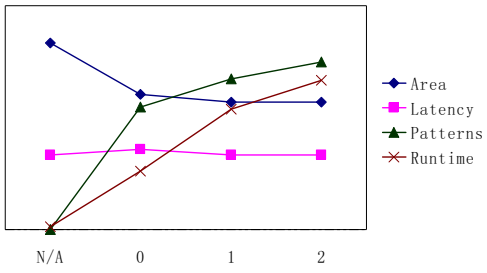

Figure 8: Trade-offs with different l_{dist} value (N/A denotes the results without pattern optimizations) .

tance usually results in more multiplexing overhead in pattern implementations. The tradeoff with l_{dist} is shown in Figure 8 on example DIR, where area/performance data are compared with different l_{dist}. Other test cases show similar behavior as DIR. So, l_{dist} is assigned to be 1 in the subsequent experiment.

Table 2 shows the QoR of our proposed pattern-based synthesis algorithm compared to the flow without pattern optimization. The first column lists names of test cases. For each test case, the second and third columns are the register usage of the synthesis result without and with pattern optimization respectively; the fourth column is the comparison between two algorithms. Similarly, the fifth to seventh columns are the numbers and comparison of LUT usage; the eighth to tenth columns are the numbers and comparison of SLICEs. Table 2 also lists the number of pattern instances in the corresponding design and the maximal pattern size in columns "PINSTS" and "MAX". The numbers inside parenthesis in column "MAX" are the maximal pattern sizes being actually used. For comprehensive comparison of overall resource usages, we enforce multipliers to be implemented by LUTs, not DSP blocks in Xilinx FPGAs.

Overall, our pattern-based synthesis flow can achieve about an average 20% reduction over the traditional behavior synthesis flow. Data in Table 2 also suggests a high correlation between regularities of given programs (number of patterns found) and resource reductions. For other important metrics, the pattern-based synthesis algorithm achieves the aforementioned resource reduction with an averagely marginal $\pm$2% frequency variation and a reasonable 7% latency overhead on average. The pattern optimization flow is also very efficient in runtime; most of the test cases can be finished within 1 minute, with the largest one (FFT) in 5 minutes.

The efficacy of our approach is further demonstrated by a case study of the FFT testbench. The FFT design contains $\sim$200 lines of C code, and about $\sim$500 nodes in the translated CDFG ($\sim$60 multipliers and $\sim$200 adders respectively). The biggest pattern found has size 34, and totally 8657 patterns are discovered. The *HPR* phase discovers $\sim$7500 patterns with size up 11, while the *VPR* phase discovers $\sim$ 1000 patterns and $\sim$80 of them are maximal patterns. The whole algorithm finishes within 5 minute. The runtime of the *PR* algorithm is mainly determined by the size of the biggest pattern, therefore the *PR* algorithm should be able to handle most programs in practice. By utilizing the detailed pattern information, the pattern-based synthesis engine can reduce 20% of the total area compared to the regular synthesis flow, as shown in Table 2 (the line begins with FFT). Also, the clock period is reduced by 10% and the number of clock cycles is slightly increased by 6%.

7. CONCLUSIONS AND ONGOING WORK

In this paper we present a general pattern-based behavior synthesis framework which can efficiently extract patterns from behavior specifications. Our approach exploits advanced subgraph enumeration and pattern pruning techniques to efficiently recognize patterns from an enormous search space. Further, the pattern recognition framework is applied to solve the resource optimization problem on FPGA platforms. Experiment shows the efficacy of both the pattern recognition algorithm and the resource reduction algorithm. Our future work includes the support of patterns with control flows.

8. ACKNOWLEDGMENT

This work is partially supported by MARCO/DARPA Gigascale Silicon Research Center (GSRC), the SRC GRC contract 2006-TJ-1400, the NSF grant CCF-0530261, and a grant from Xilinx Inc. and Magma Inc. under the California MICRO program.

9. REFERENCES

[1] GMT toolkit. http://www.cs.sunysb.edu/ algorith/implement/gmt/implement.shtml.

[2] M. A. Abdulrahim and M. Misra. A graph isomorphism algorithm for object recognition. *Pattern Analysis and Applications*, 1(3), 1998.

[3] C. Alias. *Program Optimization by Template Recognition and Replacement*. PhD thesis, 2005.

[4] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 256–261, New York, NY, USA, 2003. ACM Press.

[5] P. Bonzini and L. Pozzi. Polynomial-time subgraph enumeration for automated instruction set extension. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1331–1336, New York, NY, USA, 2007. ACM Press.

[6] O. Bringmann and W. Rosenstiel. Resource sharing in hierarchical synthesis. In *ICCAD '97: Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 318–325, Washington, DC, USA, 1997. IEEE Computer Society.

[7] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *Proc. of CASES*, 2002.

[8] D. Chen, J. Cong, and Y. Fan. Low-power high-level synthesis for FPGA architectures. In *Proc. International Symposium on Low Power Electronics and Design*, 2003.

[9] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. Platform-based behavior-level and system-level synthesis. In *Proceedings of IEEE SOCC*, 2006.

[10] J. Cong, Y. Fan, G. Han, and Z. Zhang. Application-specific instruction generation for configurable processor architectures. In *Proc. of the ACM International Symposium on Field-Programmable Gate Arrays*, pages 183–189, 2004.

[11] J. Cong, Y. Fan, and W. Jiang. Platform-based resource binding using a distributed register-file microarchitecture. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 709–715, New York, NY, USA, 2006. ACM Press.

[12] J. Cong and Z. Zhang. An efficient and versatile scheduling algorithm based on SDC formulation. In *Proceedings of*

Table 2: Resource reduction on all testcases.

	FF	**FF(p)**	CMP	**LUT**	**LUT(p)**	CMP	**SLICE**	**SLICE(p)**	CMP	**PINSTS**	**MAX(u)**
CHEM	1729	1191	-31.12%	4480	3355	-25.11%	2611	2105	-19.38%	2709	16(6)
CHENDCT	1963	1625	-17.22%	3087	2129	-31.03%	1880	1672	-11.06%	4940	14(4)
DIR	2012	1413	-29.77%	2767	1926	-30.39%	2347	1543	-34.26%	5415	15(15)
LEE	1787	1809	1.23%	3550	3217	-9.38%	2010	1987	-1.14%	641	9(4)
PR	1764	1443	-18.20%	3811	2403	-36.95%	2163	1652	-23.62%	1601	13(3)
FFT	8117	6371	-21.51%	12494	10436	-16.47%	9390	7514	-19.98%	8657	34(12)
IDCT	1917	1978	3.18%	4730	3362	-28.92%	3236	2575	-20.43%	8318	40(4)
average			-16.20%			-25.47%			-18.55%		

Design Automation Conference, July 2006.

[13] M. R. Corazao, M. A. Khalaf, L. M. Guerra, M. Potkonjak, and J. M. Rabaey. Performance optimization using template mapping for datapath-intensive high-level synthesis. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 15(8):877–888, 1996.

[14] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, New York, NY, USA, 2004. ACM Press.

[15] http://www.mit.edu/ andoni/LSH/. *LSH Algorithm and Implementation*.

[16] http://www.xilinx.com. *Xilinx Website*.

[17] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *ICDM*, pages 549–552, 2003.

[18] C.-Y. Huang, Y.-S. Chen, Y.-L. Lin, and Y.-C. Hsu. Data path allocation based on bipartite weighted matching. In *Proc. of the 27th Conference on Design Automation*, pages 499–504, 1990.

[19] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD*, pages 13–23, 2000.

[20] C. Keler. Pattern-driven automatic parallelization. In *Scientific Programming, 5:251–274*, 1996.

[21] K. Keutzer. DAGON: Technology binding and local optimization by DAG matching. In *Proc. of the 24th Design Automation Conference*, pages 341–347, 1987.

[22] T. Kim and C. Liu. An integrated data path synthesis algorithm based on network flow method. *Custom Integrated Circuits Conference, 1995., Proc. of the IEEE 1995*, 1-4:615–618, May 1995.

[23] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM*, pages 313–320, 2001.

[24] T. Kutzschebauch and L. Stok. Regularity driven logic synthesis. In *ICCAD*, pages 439–446, 2000.

[25] T. Ly, D. Knapp, R. Miller, and D. MacMillen. Scheduling using behavioral templates. In *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 101–106, New York, NY, USA, 1995. ACM Press.

[26] B. D. Martino and G. Iannello. PAP recognizer: A tool for automatic recognition of parallelizable patterns. In *Proc. of IWPC*, 2004.

[27] B. T. Messmer and H. Bunke. A new algorithm for error-tolerant subgraph isomorphism detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20(5):493–504, 1998.

[28] R. Metzger and Z. Wen. *Autamatic algorithm recognition and replacement: a new approach to program optimization*. MIT Press, Cambridge, MA, USA, 2000.

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

[30] S. Theodoridis and K. Koutroumbas. *Pattern recognition*. Academic Press, 1999.

[31] C.-J. Tseng and D. Siewiorek. Automated synthesis of data paths in digital systems. 5(3):379–395, July 1986.

[32] L. M. Wills. *Automated program recognition by graph parsing*. PhD thesis, 1992.

[33] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.

[34] R. Yang, P. Kalnis, and A. K. H. Tung. Similarity evaluation on tree-structured data. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 754–765, New York, NY, USA, 2005. ACM Press.

[35] P. Yu and T. Mitra. Scalable custom instructions identification for instruction-set extensible processors. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 69–78, New York, NY, USA, 2004. ACM Press.