

6. Section 7 presents an example of a specific application. Finally, conclusions are presented in section 8.

2. Related Work

Previous research on pattern extraction, such as [1], [2], [3], is characterized by combined pattern matching and pattern generation for ASIPs. In [1], this is achieved with clustering that uses information on frequency of node type successions. Authors of [2] and [3] use an incremental clustering that uses different heuristic approaches with the common aim of identifying frequently occurring patterns.

Another method is presented in [4] where the pattern searching algorithm identifies a big pattern using convexity and input/output constraints. Some improvements of this method were proposed in [5]. Pattern searching under input/output constraints is also used in [6]. The basic algorithm starts from each exit node of the basic block and constructs a sub-graph by recursively trying to include parent nodes. The assembled sub-graph is considered as a potential new instruction. The quality of this instruction is then determined by their system. In [7], a set of Multiple Input Single Output sub-graphs (MaxMISO) is identified first. Each MaxMISO sub-graph is not contained in any other MISO sub-graph. In the next step a candidate set composed of two-input/one-output MISOs found inside the MaxMISO set is selected. Finally, using the selected candidates the application graph is partitioned by a nearly-exhaustive search method using the branch-and-bound algorithm. Recently, in [8], a complete processor customization flow was presented where patterns are clustered, one after the other, making some local decisions.

Our original approach [9] resembles the method presented in [10]. Patterns are incrementally assembled by adding the neighbor nodes to existing matches corresponding to non isomorphic patterns formed in the previous iteration. The difference is the selection of neighbor nodes and new potential patterns. In particular, our approach applies smart filtering of patterns. It decides which new potential pattern is “useful” and can be later extended. The smart filtering uses information derived by a special method that is based on sub-graph isomorphism constraints and constraints programming that is also radically different from the approach proposed in [10]. Our recent method is totally implemented using constraint programming methods. All constraints defining a valid pattern are specified first and patterns that fulfill all these constraints are selected [11].

Pattern selection, binding and scheduling are computationally difficult problems and therefore most researchers use heuristic approaches, such as greedy algorithms, simulated annealing, genetic algorithms and tabu search. Recently several interesting approaches have been proposed. Wang et.al. uses *ACO* (Ant Colony Optimization) algorithm [12] and Guo et.al. [10] a heuristic algorithm based on maximum independent set of a conflict graph. Our approach

is different. The pattern selection, binding and scheduling problem is completely defined using a constraint model. Different approaches can be used here. Our first method uses graph matching constraint together with other binding and scheduling constraints. The other method first identifies possible matches of patterns in an application graph and then uses this information for pattern selection, scheduling and binding. This is also defined as constraint solving problem. The defined problems can be solved using either complete or heuristics methods.

Computational pattern merging has been explored, in first place, in the context of reconfigurable architectures. It can be carried out on fine (circuit or logic) or coarse grain level (functional blocks). In this paper, we are interested in functional reconfiguration that implies coarse grain level only. The patterns for this problem are usually modeled as graphs and graph algorithms can be applied for solving the problem. Clique partitioning of compatibility graphs has been, for example, used for pattern merging in [13]. In our work, we use constraint programming and graph constraint for sub-graph isomorphism and clique finding. In this framework we can combine optimal and heuristic methods for solving pattern merging problem.

A design flow for a simple processor with a dynamically reconfigurable data-path acting as an accelerating co-processor for a specific application domain has been proposed in [14]. The authors reported significant speedup for accelerators that have data-path consisting of hardwired function units and reconfigurable interconnect. Integer Linear Programming (ILP) has been used to solve hardware resource sharing and allocation, and maximum clique finding on compatibility graphs for data-path merging. Our approach also uses compatibility graphs, but we define different design constraints and cost functions to find the best reconfigurable accelerators. We also use a constraint programming approach that makes it possible to use both heuristic and optimal methods in combination with the newest clique finding constraints [15].

In [16] the authors present an efficient heuristic which transforms a set of custom instructions into a single hardware data-path. Their method starts with a set of customized instructions modeled as directed acyclic graphs (DAGs) and the goal is to minimize the area, not the sharing of interconnections. Their approach is based on the classical problems of finding the longest common sequence and substring of two (or more) sequences. The heuristic produces circuits that are much smaller (up to 85.33%) than those synthesized with ILP approach that do not explore resource sharing.

A high level synthesis for data-path-intensive ASIC design has been proposed in [17]. The authors propose performance optimization using template mapping. The key of their algorithm is the introduction of the concept of *bypassability* which allows partial graph matching. A template node is

said to be bypassable on some input if its output value can be set equal to this input by setting the other inputs to constants without inducing side-effects. We use and extend this concept to handle bypassable expressions of a data-path to optimize performance as well as area.

Recently, in [18], a pattern-based high-level synthesis for FPGA resource reduction has been proposed. The paper presents a general pattern-based synthesis framework that extracts similar structures in programs. Their approach benefits of advanced pruning techniques that include extensively sensitive hashing techniques and characteristic vectors to capture similar structures. This is based on notion of graph edit distance. Considering knowledge of previously discovered patterns, the data-path generated at the binding step of the synthesis reduces interconnect costs, but with a latency overhead. We also use pruning techniques but they are incorporated in our constraint programming framework.

Since graphs are the main representation for problems involving patterns, graph algorithm are used to automate many of design activities. This includes (sub-)graph isomorphism and clique finding, for example.

Different types of homomorphism between graphs and sub-graphs have been extensively studied and many algorithms have been proposed. The first algorithm for sub-graph isomorphism has been developed by Ullmann in [19]. Larossa and Valiente [20] studied sub-graph isomorphism problem and methods for solving it using constraint satisfaction. They explored four different solving approaches that have really full look ahead characteristics.

The VF2 algorithm that can be used for both graph and sub-graph isomorphism has been developed by authors of [21]. This algorithm can be described by means of State Space Representation (SSR). In each state a partial mapping solution is maintain and only consistent states are kept. This states are generated using *feasibility rules* that remove pairs of nodes that cannot be isomorphic. We use very similar method to this used by VF2 but we extend the set of rules to be able to handle both undirected and directed graphs as well as different types of “ports” that connect edges in the graph. This is necessary in our case when non-symmetric operations, such as “-” are used.

Graph matching constraint together with constraint satisfaction formulation has been used in electronic design automation area. Teram et. al. [22] have developed the QUEST system for pattern search in hierarchical structural designs. Their system is based on sub-graph isomorphism constraint that uses graph pattern matching methods proposed in [23]. Special search methods are used to find final matching. The QUEST system can handle large real-life graphs representing electronic circuits (up to 30,000 nodes and 87,000 edges).

Results on (sub)graph matching found in the literature indicate that it is possible to build efficient methods that can establish sub-graph isomorphism for practical problems. Application of these methods for computational pattern

selection requires some extensions and refinements of these methods. In this paper, we present this extension together with practical application for selection of computational patterns.

Clique partitioning is often used to find merged patterns. While the problem of finding a maximal cliques is NP-hard there exist many heuristic and non-heuristic algorithms. In our work, we use clique constraint built on principles proposed in [15]. This approach proposes new pruning techniques that make it possible to solve many previously difficult benchmarks.

3. Constraint Programming

In our work we use extensively constraint satisfaction methods implemented in constraint programming environment JaCoP [24].

A *constraint satisfaction problem* is defined as a 3-tuple $S = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ where $\mathcal{V} = \{x_1, x_2, \dots, x_n\}$ is a *set of variables*, $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_n\}$ is a set of *finite domains* (FD), and \mathcal{C} is a set of *constraints*. Finite domain variables (FDV) are defined by their domains, i.e. the values that are possible for them. A finite domain is usually expressed using integers, for example $x :: 1..7$. A constraint $c(x_1, x_2, \dots, x_n) \in \mathcal{C}$ among variables of \mathcal{V} is a subset of $D_1 \times D_2 \times \dots \times D_n$ that restricts which combinations of values the variables can simultaneously take. Equations, inequalities and even programs can define a constraint. Our graph matching constraint is implemented, for example, using a specially developed pruning algorithm presented later.

A *solution to a CSP* is an assignment of a value from variable’s domain to every variable, in such a way that all constraints are satisfied. The specific problem to be modeled will determine whether we need *just one solution*, *all solutions* or an *optimal solution* given some cost function defined in terms of the variables.

The solver is built using constraints own consistency methods and systematic search procedures. *Consistency methods* try to remove inconsistent values from the domains in order to reach a set of pruned domains such that their combinations are valid solutions. Each time a value is removed from a FD, all the constraints that contain that variable are revised. Most consistency techniques are not complete and the solver needs to explore the remaining domains for a solution using search.

Solutions to a CSP are usually found by systematically assigning values from variables domains to the variables. It is implemented as depth-first-search. The consistency method is called as soon as the domains of the variables for a given constraint are pruned. If a partial solution violates any of the constraints, backtracking will take place, reducing the size of the remaining search space.

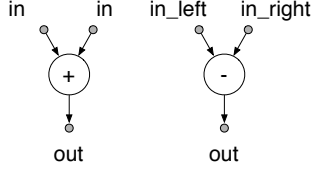


Fig. 5: Representing vertices of HCDG for graph matching constraint.

3.1 Diff constraint

In this paper, we use intensively Diff constraints. Diff constraint takes as an argument a list of 2-dimensional rectangles and assures that for each pair of i, j ($i \neq j$) of 2-dimensional rectangles, there exist at least one dimension k where i is after j or j is after i . The 2-dimensional rectangle is defined by a tuple $[O_1, O_2, L_1, L_2]$, where O_i and L_i are respectively called the origin and the length of the 2-dimensional rectangle in i -th dimension. The Diff constraint is used in this paper for defining constraints for scheduling and resource binding.

The discussion of propagation method for this constraint is beyond the scope of this paper. In general, the constraint propagator uses a method know from computational geometry [25], called *sweep* or *plane sweep* (in two dimensions). Paper [24] discusses the implementation of this constraint for JaCoP solver.

3.2 Graph matching constraint

In our approach we use Hierarchical Dependency Graphs (HCDG) [26] to model designs. Implementation of graph matching constraint requires definition of a graph that can cover basic features of HCDG. We have defined a special *labeled* graph with explicitly defined connecting ports for each node of the graph. Labels are used for defining function of a node (e.g., “+” or “-” in Figure 5) as well as type of the connecting port (e.g. “in”, “out” or “left_in”, “right_in” in Figure 5). Ports are explicitly specified and assigned to nodes. Both nodes and ports can be formally considered as vertices of the graph. More formally our graph is defined as follows.

Definition 1 (Graph): Let $G = (V, E)$ be a graph. We extend our definition to labeled graph $G' = (V, E, \mu, \pi)$, where

$$V = N \cup P, N \cap P = \emptyset \quad (1)$$

$$\mu : V \rightarrow L \quad (2)$$

$$\pi : N \rightarrow \mathcal{P}(P), \pi(n_i) \cap \pi(n_j) = \emptyset \text{ for } n_i \neq n_j \quad (3)$$

$$(p_i, p_j) \in E, \text{ where each port } p_i, p_j \in P \quad (4)$$

is only present in one $e \in E$

The definition defines vertices of a graph as nodes of HCDG graph, N , and connecting ports, P (1). All vertices (nodes and ports) are labeled by labeling function μ using labels from set L in (2). Each node get assigned ports in

(3). Ports are unique and the same port cannot be assigned to more than one node. Finally, edges are defined by connecting two ports (4). It is important to note that each port can be connected to only one other port. Fan-outs must be defined by multiport nodes.

Our graph definition defines always a vertex of a graph as a node and its ports. This makes it possible to distinguish, for example, two input adder from three input adder. Traditional graph definition will define it as a labeled node and will not make it possible to distinguish between these two types of operations explicitly.

Our graph matching constraint, GraphMatch, is defined by a number of rules specifying conditions for graph isomorphism. Our matching function assigns a pattern node to a node of a target graph since it provides an easy way to find a cover of an application graph with a given set of patterns. The matching conditions are defined below for our graph.

Definition 2 (Graph Matching): Given two graphs $G_t = (V_t, E_t, \mu_t, \pi_t)$ and $G_p = (V_p, E_p, \mu_p, \pi_p)$ the matching $f : V_t \rightarrow V_p$ is a non-injective function respecting the following properties:

$$\forall v \in V_t : \mu_t(v) = \mu_p(f(v)) \quad (5)$$

$$\forall n \in N_t \text{ exist a bijective function} \quad (6)$$

$$g_n : \pi_p(f(n)) \rightarrow \pi_t(n)$$

$$\forall u, v \in P_t : (f(u), f(v)) \in E_p \Leftrightarrow (u, v) \in E_t \quad (7)$$

$$\forall u, v, p, q \in P_t, u \neq p, v \neq q : (f(u), f(v)) = (f(p), f(q)) \quad (8)$$

$$\wedge u \in \pi(n_i) \wedge v \in \pi(n_j) \wedge p \in \pi(n_k) \wedge q \in \pi(n_l) \Rightarrow n_i \neq n_k \wedge n_j \neq n_l, \text{ matchings do not overlap}$$

The definition defines conditions for graph isomorphism between target graph G_t and pattern graph G_p . The vertex of pattern graph G_p is mapped into a vertex of target graph G_t if the four conditions are satisfied. It happens when labels of vertices are the same (5) and the ports of pattern graph can be one-to-one mapped into ports of the target graph (6). Condition (7) specifies a constraint on existence of corresponding edges between mapped nodes. Finally, condition (8) specifies a special condition for our method that excludes overlapping patterns in the target graph. This condition is necessary because otherwise one node can be used as an end-node for more than one mapped edge. The developed algorithm implements these conditions as the rules that prune non possible matchings.

Function f can be partial and then $f : V_t \rightarrow V_p \cup \{\perp\}$. This feature is used when we do not want to establish full graph isomorphism. The isomorphism in this case is restricted to parts of the target graph and a pattern graph and establishes sub-graph isomorphism. This is achieved by assigning value \perp for variables representing not mapped parts in the target graph. This sub-graph isomorphism is different than the classical definition from graph theory where sub-graph isomorphism means that a found pattern

```

// target graph  $(V_t, E_t)$ , with nodes  $n_i \in N_t$ 
// pattern graph  $(V_p, E_p)$ , with nodes  $p_i \in N_p$ 

Q ← {n0, ..., nN}
while Q non-empty // queue of changed nodes
  remove ni from Q
  for each pi ∈ D(ni)
    // same node type and port structure
    if ni.label = pi.label ∧ ni and pi have the same ports
      for each port type tp of pi
        P ← nodes connected to pi through port type tp
        T ← ∅, N ← ∅
        for each node mi connected to ni through port type tp
          N ← N ∪ mi
          T ← T ∪ (P ∩ D(mi))
        // the same neighbor structure and not overlapping
        if size of bi-partite matching between T and P ≥ |P|
          for each mi ∈ N that has assigned single value v ∈ P
            remove v from domains of variables of N \ {mi}
        else
          remove pi from domain of D(ni)
      else
        remove pi from domain of D(ni)
    if D(ni) has been changed
      add ni and its neighbors to Q

```

Fig. 6: Consistency algorithm for graph matching.

can be connected arbitrarily to the rest of the target graph. For the purpose of this paper this is more suitable definition but our constraints supports also traditional sub-graph isomorphism and monomorphism, if needed. In practice we use function that defines mapping between nodes only, i.e. $f : N_t \rightarrow N_p \cup \{\perp\}$.

The consistency method for graph matching constraint discussed above is implemented using a consistency algorithm in our solver. The algorithm is depicted in Figure 6. It inspects all nodes in a queue and compares a selected target node to all possible matching nodes specified in its domain. It applies a number of prunings depending on different conditions. If pruning has been applied to a node variable the node and all its neighbors are put back into the queue for recomputation of consistency algorithm.

The first if-statement in the algorithm checks if two nodes have the same node type and port structure. If this is not the case the related pattern node number is removed from the current node variable domain. This removes possible matching between current target node and the pattern node. The second if-statement checks if the structure of neighborhood nodes is the same for both the target graph and the pattern graph. If there is no possible mapping between target graph nodes and pattern nodes representing neighborhood, the related pruning of the current node variable takes place. This step of the algorithm assures also that patterns found in the target graph will not overlap. This is achieved by removing already assigned values from other nodes in the neighborhood.

This constraint is used in several places of our system to support different decisions. Here are some examples of its use.

Example 1. Finding isomorphism between graphs G_1 and G_2 .

```

GraphMatch( $G_1, G_2$ )
search(variables(nodes( $G_1$ )))

```

If there is a solution for this search, graphs G_1 and G_2 are isomorphic otherwise they are not.

Example 2. Finding coverage of graph G with pattern graph P (finite domain variables assigned to nodes of G have in their domain values assigned to nodes of P and a designated value \perp , for example -1). Moreover, we impose Count constraints to limit number of possible matches in the graph to one. Count constraint counts number of values specified by node identifier value(n) on the list of node variables of graph G , variables(nodes(G)), and, in our case, limits them to one.

```

GraphMatch( $G, P$ )
for each n ∈ P
  Count(value(n), variables(nodes( $G$ )), 1)
searchAll(variables(nodes( $G$ )))

```

If there is a number of solutions we have found all possible matchings of pattern P in graph G otherwise there is no matching.

For example, the target graph from Figure 7 can be covered in three different ways by the pattern graph and our solver will produce three possible assignments to node variables [0, 1, -1, -1], [-1, -1, 0, 1] and [-1, 1, 0, -1].

Example 3. Finding cover of graph G with patterns from P . Note that not all graphs of P need to be selected for a valid cover.

```

GraphMatch( $G, P$ )
searchAll(variables(nodes( $G$ )))

```

It will generate all possible covers of graph G with selected patterns from P .

For example, for graphs from Figure 7 our solver will produce a full cover of the target graph represented by assignment [0, 1, 0, 1]. It represents two matches of nodes 0 and 1, and nodes 2 and 3, however, interpretation of this result is not straightforward. One can deduce match of nodes 2 and 1 that is incorrect since nodes 0 and 3 will not have their corresponding matching nodes. We call this kind of match as a phantom match. To avoid such situations we can use several identical patterns with different identifiers together with constraints prohibiting multiple use of the same pattern. In our case, we can use two identical patterns, one numbered 0 and 1 and the other one 2 and 3. Covers [0, 1, 2, 3] or [2, 3, 0, 1] will be found that distinctly identify matches.

To be able to see what are characteristics of graphs representing electronic systems we have checked typical HCDG graphs for programs. It can be noted that these graphs are not tightly connected and has parameter $\eta = \frac{|E|}{|V| \cdot (|V|-1)}$ very low, where $|E|$ is number of edges and $|V|$ is number of vertices in the graph. Usually they have only a small percentage of edges of the fully connected graph with the

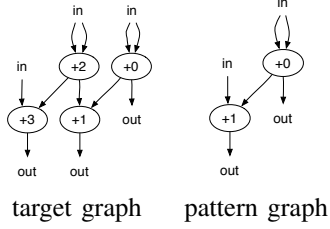


Fig. 7: An example of a target and a pattern graph.

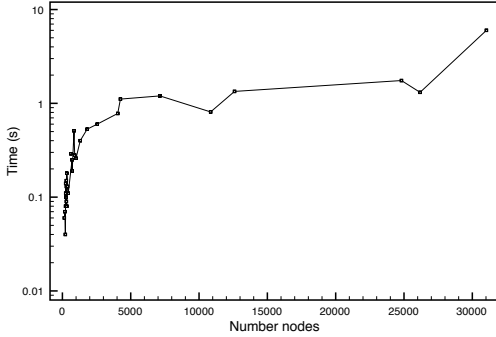


Fig. 8: Runtime for sub-graph matching for MCNC benchmarks.

same number of vertices. Our experiments show that the low value of η is typical for graphs used in electronic design automation. Since we had not large program graphs we have used MCNC Partitioning93 benchmarks to find sub-graph isomorphism. The test was to find a maximum coverage of the graph with four possibly overlapping patterns of different sizes. Overlapping patterns were chosen since distinguishing patterns with similar structures becomes the critical moment in the proposed algorithm. Two patterns of three nodes and two patterns of two nodes were used in the test. The results are presented in Figure 8. Our method works efficiently for these examples. Graphs of size up to 10,000 nodes can be handled in seconds or less than a second. The majority of the graphs from real designs have less than 5,000 nodes as it is reflected in the benchmark. Only extremely large graphs require up to 6 seconds for the algorithm to find the maximal coverage.

3.3 Clique constraint

Clique finding is known to be a difficult problem and maximal clique finding is known to be NP-hard. In our work, we use a clique constraint, `Clique`. This constraint takes as an argument a graph and a finite domain variable defining a size of its clique (K). The constraint assures that 0/1 variables assigned to its nodes are one if related nodes belong to a clique and K defines the size of the clique. Variable K can be used to get a size of a clique, to constraint its size or to find maximal clique by maximizing the value of this variable.

The consistency method implemented by the `Clique` constraint is based on algorithms of [15]. That approach has

proved to solve many difficult problems as well as, for the first time, solve a number of problems with unknown results. In our experiments, we have solved many large graphs rather quickly. For example, a weighted clique is found in a graph with 2,122 nodes and 2,116,470 edges (94.05% edges of fully connected graph) in $\sim 2s$.

4. Pattern Generation

We have experimented with different pattern generation methods based on incremental pattern creation as well as pure constraints defined search.

The first method, implemented in UPaK system, is an iterative process (for details see [27]). Each iteration of this method explores larger patterns (incremented by one node) and identifies candidates for inclusion in a set of patterns. This is achieved by calling in each iteration *pattern searching algorithm*, depicted in Figure 9. Each time the algorithm is called it generates a *Next Pattern Set (NPS)*. The method stops when an *end condition* is reached. In our experiments, the algorithm stops when the pattern size becomes K but other stop criteria are possible.

The inputs to the pattern searching algorithm are: Definitely Identified Pattern Set (*DIPS*), Current Pattern Set (*CPS*) and Architecture Model (*AM*). Initially, *DIPS* and *CPS* sets contain the same set of one-node patterns (Figure 9). The Architecture Model is used to determine whether the newly created pattern tp can be executed on the selected architecture (function *PatternAccepted(tp, AM)*).

```

// Inputs: DIPS-- Definitely Selected Pattern Set,
//         CPS-- Current Pattern Set, APS-- Auxiliary Pattern Set
// Other variables: TPS-- Temporary Pattern Set
//         NPS-- Next Pattern Set, RPS-- Reduced Pattern Set
//         SN-- Set of Nodes

NPS ← ∅
for each  $p_i \in CPS$ 
  TPS ← ∅
  InsertPattern ← true
  for each  $m_j \in N$  corresponding to  $p_i$ 
    SN ← ReturnAllNodesConnectedToMatch( $m_j, G$ )
    for each  $n_k \in SN$ 
       $tp \leftarrow CreateNewPatternFrom(m_j^{p_i}, n_k, sn_j^{p_i})$ 
      if PatternAccepted( $tp, AM$ )
        // check if a new pattern  $tp$  is not isomorphic ( $\cong$ )
        // to any existing pattern
         $NMP_{tp} \leftarrow 0$ 
        if  $\forall_{pattern \in TPS} tp \not\cong pattern$ 
           $TPS \leftarrow TPS \cup \{tp\}$ ,
           $NMP_{tp} \leftarrow |FindAllMatches(G, tp)|$ 
    APS ← ∅
    // smart filtering
    for each  $tp_k \in TPS$ 
      if  $coef1 * NMP_{p_i} \leq NMP_{tp_k}$ 
         $APS \leftarrow APS \cup \{tp_k\}$ 
      if  $coef2 * NMP_{p_i} \leq NMP_{tp_k}$ 
        InsertPattern ← false
    TPS ← APS
  for each  $tp_k \in TPS$ 
    if  $\forall_{pattern \in NPS} tp_k \not\cong pattern$ 
       $NPS \leftarrow NPS \cup \{tp_k\}$ 
  if InsertPattern
     $RSP \leftarrow RSP \cup \{p_i\}$ 
   $DSPS \leftarrow RSP \cup DSPS$ 
return NPS

```

Fig. 9: Pattern searching algorithm.

The pattern searching algorithm searches for new patterns that can be synthesized from all possible matches in application graph G corresponding to each pattern in CPS (see loop, line 8). A new temporary pattern tp is created from current match $m_j^{p_i}$ of pattern $p_i \in CPS$ and the selected node $n_i \in N$ where N is a set of the nodes directly connected to $m_j^{p_i}$ in G . The search is organized both upward and downward to best explore new patterns. A new pattern tp is accepted if it is not isomorphic to any of already found patterns.

A smart filtering (line 24) then decides whether pattern tp should be saved in set NPS or not. It also determines inclusion of current pattern p_i to the $DIPS$. This filtering process uses information about the number of matches in an application graph corresponding to patterns p_i and tp . The decision is based on a heuristic when two coefficients $coef1$ and $coef2$ are used. Their values have been experimentally selected ($coef1, coef2 \in [0..1], coef1 \leq coef2$). When all patterns from the CPS set has been processed the pattern generation algorithm stops the current iteration by copying the NPS content to the CPS .

Although the presented method is simple, it enables generation of small numbers of high quality patterns. This is achieved since we use very precise information on sub-graph isomorphism in our algorithm. Our constraint programming methods check number of matches of partially created patterns in an application graph as well as decide if a current pattern is already isomorphic to existing patterns.

The second method, implemented in DURASE system, uses pure constraint programming definition for pattern generation (for details see [11]). In this method, for each seed node, we first specify constraints defining valid patterns as well as architectural and technological constraints for generated patterns. Then standard constraint programming search produces all possible patterns fulfilling the defined constraints.

In this approach, the pattern generation is defined, for an acyclic application graph $G = (N, E)$ where N is a set of nodes and E is a set of edges. A pattern is a subgraph $P = (N_p, E_p)$ of graph G where $N_p \subseteq N$ and $E_p \subseteq E$. Pattern P is also sub-graph isomorphic to graph G . This sub-graph isomorphism is found, in our system, by defining a set of constraints and finding solutions to them. To be able to define this constraints we introduce a number of definitions.

A set of *successor* nodes of node n is defined by $succ(n) = \{n' : (n, n') \in E\}$. Similarly, we define *predecessors* of node n as $pred(n) = \{n' : (n', n) \in E\}$. All successors of node n are defined recursively as $allsucc(n) = \{n' \cup allsucc(n') : (n, n') \in E\}$. A *path* between two nodes n_i and n_j in graph G is defined as a path in non-directed graph, i.e., as a string of nodes as follows $path(G, n_i, n_j) = (n_i, n_{i+1}, \dots, n_j)$ where each consecutive nodes n_k, n_{k+1} in this string satisfy $(n_k, n_{k+1}) \in E$ or $(n_{k+1}, n_k) \in E$.

Pattern generation iterates over all nodes in G . During

each iteration, all computational patterns formed around node $n_s \in N$, called seed nodes, satisfying all of the architectural and technological constraints are identified. They form a current pattern set (CPS). In the next step, the CPS is reduced to only non-isomorphic patterns. Finally, patterns whose numbers of matches in the application graph are high enough compared to the number of matches obtained for related single seed node patterns are identified. These patterns are added to the definitively identified pattern set ($DIPS$). The number of matches of a given pattern in the application graph is obtained using the constraint programming method described in section 3).

The computational pattern created around seed node n_s in application graph G is an acyclic graph $P_{n_s} = (N_p, E_p)$. The constraint program builds patterns by imposing constraints that define valid patterns. Only solutions that fulfill all these constraints are accepted. Constraints (9)-(13) define valid computational patterns.

Constraint (9) states that each node $n \in N_p$, different from seed node n_s , must form a path in pattern graph P_{n_s} starting at this node and leading to the seed node. We consider here non-directed pattern graphs.

$$\forall n \in N_p \wedge n \neq n_s \quad \exists path(P_{n_s}, n, n_s) \quad (9)$$

Furthermore, each node $n \in N$ is modeled by one finite domain variable n_{sel} . The value of this variable is 1 if node $n \in N_p$ or 0 otherwise. $n_{sel} = 1$ because the seed node n_s always belongs to the pattern created around it. The constraints for all nodes in a pattern are defined in formulas (10) - (13).

$$\forall n \in (N - (allsucc(n_s) \cup n_s)) : \\ n_{sel} = 1 \Rightarrow \sum_{m \in succ(n)} m_{sel} \geq 1 \quad (10)$$

$$\forall n \in (N - (allsucc(n_s) \cup n_s)) : \\ \sum_{m \in succ(n)} m_{sel} = 0 \Rightarrow n_{sel} = 0 \quad (11)$$

Formulas (10) and (11) express two constraints between each node $n \in N - (allsucc(n_s) \cup n_s)$ and its direct successors. Constraint (10) requires selection of at least one node among successors of node n if this node is selected to be part of the pattern. Constraint (11), on the other hand, prohibits node n to belong to the pattern if none of its successors belongs to this pattern.

$$\forall n \in allsucc(n_s) : \\ n_{sel} = 1 \Rightarrow \sum_{m \in (pred(n) \cap (allsucc(n_s) \cup n_s))} m_{sel} \geq 1 \quad (12)$$

$$\forall n \in allsucc(n_s) : \\ \sum_{m \in (pred(n) \cap (allsucc(n_s) \cup n_s))} m_{sel} = 0 \Rightarrow n_{sel} = 0 \quad (13)$$

Table 1: Results for MediaBench applications for patterns with 7 nodes limit for UPaK system and patterns with 4 input/2 output limit for DURASE system.

	Nodes	UPaK			DURASE		
		identified	selected	coverage	identified	selected	coverage
JPEG Write BMP Header	106	6	4	90%	139	10	100%
JPEG Smooth Downsample	51	8	4	74%	128	11	96%
JPEG IDCT	134	7	5	69%	122	16	87%
MPEG Motion Vector	32	8	3	93%	37	4	100%
EPIC Collapse	56	7	4	69%	128	15	100%
MESA Smooth Triangle	197	7	3	73%	49	8	99%
MESA Horner Bezier	18	9	3	83%	35	4	94%
MESA Interpolate Aux	108	3	2	85%	37	4	100%
MESA Feedback Points	53	4	2	80%	39	8	100%
FIR	44	7	4	72%	14	3	100%
Elliptic Wave Filter	34	9	4	67%	64	10	100%
Auto Regression Filter	28	4	3	96%	51	3	100%
Cosine	66	8	3	50%	96	11	100%
Average		6.4	3.3	74%	72.2	8.23	98.2%

Constraints (12) and (13) define relations between each node $n \in allsucc(n_s)$ and its direct predecessors. Constraint (12) requires selection of at least one node from the predecessor nodes set reduced to nodes in the $allsucc(n_s) \cup n_s$ if node n is selected. Constraint (13) prohibits nodes from predecessor set of node n reduced to nodes in $allsucc(n_s) \cup n_s$ to belong to the pattern if node n does not belong to this pattern.

Computational patterns identified with previously defined constraints can have any number of inputs and outputs. To fulfill architectural interface requirements we need to control pattern number of inputs and outputs and to achieve this we impose additional constraints. To be able to define these constraints, each node $n \in N$ has two associated constants $indegree_n$ and $outdegree_n$ defining its number of input edges and output edges respectively. Some of these edges can become pattern external inputs or outputs. We also assume that identified patterns cannot exceed number of inputs $PatternInputs$ and number of outputs $PatternOutputs$.

$$\forall n \in N : n_{in} = indegree_n - \sum_{m \in pred(n)} m_{sel} \quad (14)$$

$$\sum_{n \in N} (n_{sel} \cdot n_{in}) \leq PatternInputs \quad (15)$$

$$\forall n \in N : n_{out} = outdegree_n - \sum_{m \in Succ_n} m_{sel} \quad (16)$$

$$\sum_{n \in N} (n_{sel} \cdot n_{out}) \leq PatternOutputs \quad (17)$$

An edge becomes pattern input (output) if node $n \in N_p$ has an external input (output) edge or a source (destination) node for this edge does not belong to the pattern. Constraints (15) and (17) limit the number of inputs and outputs of the created pattern for nodes belonging to the pattern.

Pattern critical path defines clock cycle requirements for the entire architecture and therefore needs to be controlled.

To be able to control the delay of critical path, additional constraints can be imposed (see for detail [11]).

In some cases, we would like to control not only the delay of the critical path but also size of a pattern. Constraint (18) imposes the maximal number of nodes in the created pattern. It is possible to control it even better using weighted sum and applying different weights to different node types. In this way, area requirements can be taken into account.

$$\sum_{n \in N} n_{sel} \leq PatternNumberOfNodes \quad (18)$$

Table 1 shows the quality of the generated patterns using methods provided by both *UPaK* and *DURASE* systems. The results were obtained for applications from the *MediaBench* benchmark set under different constraints. The *UPaK* method has been configured to accept patterns composed of not more than 7 nodes. For the *DURASE* method, number of nodes in each pattern cannot exceed 7, a critical path is limited to 15ns and the patterns must have at most four inputs and two outputs. The *DURASE* method delivers, in average, better graph coverage but selects more patterns.

5. Pattern Selection and Application Scheduling

When computational patterns are identified we can start process of selecting patterns for application execution and scheduling. It can be noted that depending on different design constraints, such as performance or cost, different patterns and schedules can be best suited. The process of match selection, component binding and scheduling can also be solved using constraints programming methods. The input to this design step consists of an application graph G , a target architecture model and a selected pattern set $DIPS$. Graph G is used to generate operation precedence constraints and dependence requirements while the architecture model and pattern set $DIPS$ are used to find actual matchings. These constraints are defined as inequalities and specialized constraints as specified in section 5.2. The constraint solving technique is then used to find an optimal or suboptimal solution which satisfies the given constraints and optimizes a given cost function. We use a branch-and-bound (*B&B*) algorithm to find the pattern selection and the schedule.

5.1 Architecture Model

Our approach can model abstract architectures composed of interconnected cells where each cell can execute a single pattern or a set of patterns. In general, all cells of an abstract architecture can work concurrently with each other but each cell can execute only one pattern at a time. *ASIP* processor, for instance, can assign a separate instruction to each identified computational pattern and execute it sequentially with all other instructions or patterns. The corresponding

abstract architecture, in this case, is reduced to a single cell containing all patterns. One can also combine a number of patterns and execute them in parallel using a single instruction. In this case, an abstract architecture is composed of one cell containing all one-node patterns corresponding to the original instruction set and a set of cells representing different patterns from the *DIPS* set. We can also manage several different cells containing the same pattern in order to speed-up the execution. In this paper, we consider both cases (sequential and parallel) and report the related experimental results.

5.2 Pattern Selection and Scheduling Model

The inputs for pattern selection and scheduling are the application graph $G = (N, E)$ (where N is a set of nodes and E is a set of edges), the *DIPS* set and the abstract architecture model. During pattern selection and scheduling, the system is looking for sub-graphs of graph G corresponding to pattern graphs from the *DIPS* set. These sub-graphs are called *matches* of pattern graphs because they match these patterns. Some of the patterns from the *DIPS* set whose matches have been selected during the optimization process are later implemented by computational cells in order to build a target architecture. When a cell contains several implemented patterns, each pattern must be used exclusively. Moreover, the final system implementation can use different numbers of copies of the selected cells. The defined constraints must follow these requirements.

We first define finite domain variables which are used to model pattern selection and scheduling. We use variable T to denote the start time of a given node. The subscripts are used to identify related graph nodes. For example, T_i denotes a start time of node $i \in N$. Node dependencies in G are defined by the inequalities as follows.

$$\forall (i, j) \in E : T_i + D_i \leq T_j \wedge D_i \in DS \quad (19)$$

where DS is a set of delays corresponding to all cells used in the system. A delay of node i can vary since it can be a part of different matches. This is handled by other constraints defining mapping between a selected cell and its delay.

Matches in graph G , corresponding to the patterns from the *DIPS* set, are identified using our graph matching constraint.

$$\text{GraphMatch}(G, DIPS) \quad (20)$$

This constraint implies that only a limited number of matches can coexist in G . The existing matches, represented by finite domain variables m_0, \dots, m_n , are used in the following constraints for imposing timing and resource constraints.

It is required that all input signals of a match are available before its cell starts execution. To model this we divide nodes of each match of a pattern into a set of input nodes V_{in} and a set of non-input nodes V_{int} . We also introduce a

new set of dummy nodes V_d . The input node has no input edges connected to other nodes in the same match. Non-input nodes have inputs from other nodes in the match but they can have a number of non-connected input edges as well. Dummy nodes are added, in such cases, to assure the right timing for input signals. Match timing model is depicted in Figure 10 while the following constraints define this model for nodes identified for match m .

$$\forall i \in V_{in}, j \in V_d : T_i = T_j = T_m \quad (21)$$

$$\forall i \in V_{in}, j \in V_d : D_i = D_j = D_m = D_{Cell_k}$$

$$\forall i \in V_{int} : D_i = 0$$

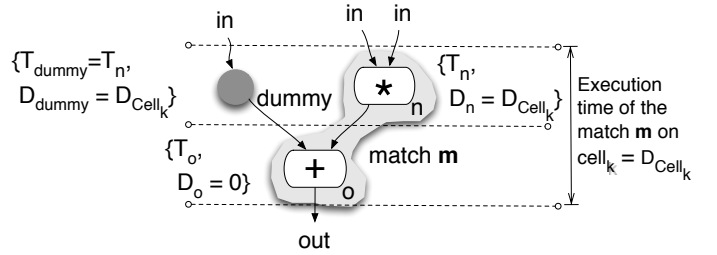


Fig. 10: Match timing modeling.

Delay D_{Cell_k} is the time needed by the $Cell_k$ to execute a selected match m . Thus the match delay $D_m = D_{Cell_k}$. Thanks to the constraints described by equations (21) the match is considered by the remaining nodes of graph G as a separate unit executed at time T_m with delay D_m . The delay from any of the inputs or dummy nodes to any of the outputs is constant and is therefore equal to D_m . The constraints rule-out also all non-convex matches.

If two or more matches in graph G can be implemented using the same cell type they must either use different physical cells or their executions must not overlap. We model execution of a match on a cell through the rectangle as it is often done in scheduling (Figure 11) and use *Diff* constraints. This constraint assures that two dimensional rectangles do not overlap that fits our purpose. For each set of matches executed on a given cell q we impose *Diff* constraint. Below a formulation for three matches m_i, m_j and m_k executed on cell type q is presented. Figure 11 illustrates this constraint.

$$\text{Diff}([T_{m_i}, Cell_{qi}, D_{m_i}, 1], [T_{m_j}, Cell_{qj}, D_{m_j}, 1], [T_{m_k}, Cell_{qk}, D_{m_k}, 1]) \quad (22)$$

Domains for variables $Cell_{qi}, Cell_{qj}$ and $Cell_{qk}$ are defined as $\{1 \dots |Cell_l|\}$ indicating possible selection of a cell numbered from 1 to $|Cell_l|$ for execution of each match.

In resource-constrained scheduling the solver minimizes the schedule length under given resource constraints. Therefore we defined our cost function for minimization as the latest completion time (LCT) for nodes of graph G . The constraint can be simplified by defining it for nodes that

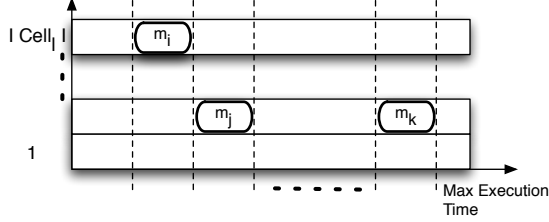


Fig. 11: Modeling of resource constraints.

are last to execute. Minimization of LCT will provide the shortest schedule. Our cost function is defined below.

$$LCT = \max(T_0 + D_0, \dots, T_n + D_n) \quad (23)$$

where T_i and D_i are respectively the start time and delay of node i .

In time-constrained scheduling we minimize the number of different cells used in an implementation and therefore we defined our cost function as follows.

$$Resources = \sum_k \begin{cases} 1 & \text{if exists } m_i \text{ that uses } Cell_k \\ 0 & \text{otherwise} \end{cases} \quad (24)$$

Time-constrained scheduling is done under execution time constraint defined as follows.

$$LCT \leq ExecTimeLimit \quad (25)$$

In the architecture model from Figure 4, registers store intermediate results that reduces data transfers between the architecture extensions and the processor register file. However, it may happen that the extensions need additional transfers because of limitations imposed by an ASIP processor architecture. To model that, additional constraints are needed. In this case, match delay D_m expressed in processor cycles, is composed of three parts as depicted in equation (26).

$$m_{delay} = \delta_{in_m} + D_{Cell_k} + \delta_{out_m} \quad (26)$$

where D_{Cell_k} is execution time for match m on the $Cell_k$ while δ_{in_m} and δ_{out_m} represent read and write time of input and output operands for match m respectively. The read and write transfer times are variable. Read transfer time is specified in equations (27)-(28) and equations (29)-(31) specify write time.

$$IN = \sum_{n \in pred1(m)} n_{sel} \quad (27)$$

$$\delta_{in_m} = \lceil IN / in_PerCycle \rceil - 1 \quad (28)$$

$$\forall n \in last(m) : \sum_{m \in succ1(n)} m_{sel} > 0 \Leftrightarrow B_n = 1 \quad (29)$$

$$OUT = \sum_{n \in last(m)} B_n \quad (30)$$

$$\delta_{out_m} = \lceil OUT / out_PerCycle \rceil - 1 \quad (31)$$

where $pred1(m)$ defines a set of one node matches that are predecessors of match m in graph G , $succ1(n)$ defines a set of one node matches that are successors of node n in graph G . The value of variable IN is equal to the number of required read operations from processor internal registers. These accesses are necessary because the data was produced by processor instructions corresponding to one node matches. The value of variable OUT is equal to the number of write operations to processor internal registers. These accesses are necessary because the data will be used by standard processor instructions corresponding to one node matches (for more details see [28]).

Table 2 presents the results obtained for the applications from the *MediaBench* benchmark set. We present the number of patterns identified by our algorithm, the number of patterns that are actually selected for maximum coverage of the graph as well as the graph coverage.

To explore possible speed-ups we consider two scheduling methods for a selected set of computational patterns. The first method schedules all matches and nodes assuming sequential execution of all operations while parallel scheduling method assumes most parallel execution. Both methods are compared to a sequential execution of the original graph. We consider that each match (up to 7 nodes) can be executed during one clock cycle by the corresponding cell. The clock cycle is determined for the MIPS processor used as the ASIP's processor core (Figure 4) and implemented on Altera Cyclon II FPGA devices running at 25 MHz.

6. Pattern Merging

Identified and selected patterns need to be merged into one reconfigurable unit. This is achieved by the pattern merging

Table 2: Application graph coverage and speed-up for MediaBench applications for patterns of with 7 nodes limit.

Application	Nodes	time (s)	patterns			speed-up	
			identified	selected	coverage	sequential	parallel
JPEG Write BMP Header	106	4.99	6	4	90%	2.83	10.60
JPEG Smooth Downsample	51	3.36	8	4	74%	1.96	3.70
JPEG IDCT	134	20.90	7	5	69%	2.03	2.48
MPEG IDCT	114	5.30	3	2	54%	1.78	2.03
MPEG Motion Vector	32	0.90	8	3	93%	4.57	8.00
EPIC Collapse	56	2.09	7	4	69%	2.24	2.64
MESA Smooth Triangle	197	120.00	7	3	73%	1.68	3.20
MESA Horner Bezier	18	0.36	9	3	83%	3.00	6.00
MESA Interpolate Aux	108	22.80	3	2	85%	3.37	6.30
MESA Matrix Multiplication	109	28.4	7	4	56%	1.57	2.80
MESA Feedback Points	53	1.70	4	2	80%	2.73	5.00
FIR	44	12.5	7	4	72%	2.44	7.30
Elliptic Wave Filter	34	2.20	9	4	67%	2.01	2.60
Auto Regression Filter	28	3.30	4	3	96%	3.50	5.60
Cosine	66	7.05	8	3	50%	1.57	1.70

algorithm that accepts *input pattern set* and produces a *merged pattern*. It is an iterative algorithm that in each iteration merges two patterns: the *temporary merged pattern* and the next pattern selected from the pattern set. The merging method is composed of several steps, such as compatibility graph generation, constraint generation for weighted clique problem, constraint generation for critical path problem, constraint generation for multiplexer minimization problem and optimization. The optimization step uses the JaCoP constraint solver and works on compatibility graph and generated constraints.

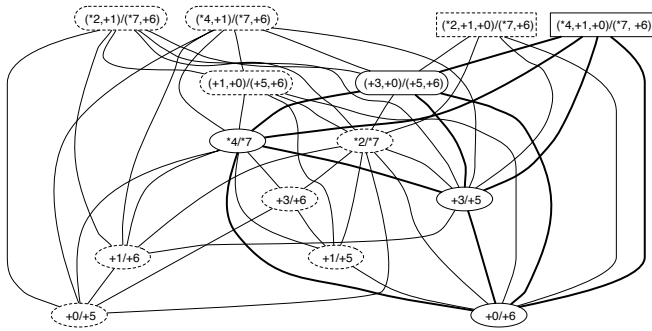


Fig. 12: Compatibility graph and maximal weighted clique for pattern set from Figure 13.

The *compatibility graph* is undirected graph $CG = (V_c, E_c)$, where V_c is a set of vertices and $E_c \subseteq V_c \times V_c$ is a set of edges. Assuming two pattern graphs, $G_i = (V_i, E_i)$ and $G_j = (V_j, E_j)$ the *compatibility graph* contains three types of nodes: *regular*, *edge* and *path* nodes. The regular node, denoted as u_i/u_j , is defined for compatible nodes $u_i \in V_i$ and $u_j \in V_j$ of pattern graphs. Two nodes are compatible if they have the same type and the same number of inputs. Similarly, edge nodes are defined for compatible edges. Two edges (u_i, v_i) and (u_j, v_j) are compatible if node u_i is compatible with node u_j and node v_i is compatible with node v_j . Finally, path nodes define compatibility relations between a path and an edge. Path (u_i, w_i, \dots, v_i) is compatible with edge (u_j, v_j) if u_i is compatible to u_j , v_i

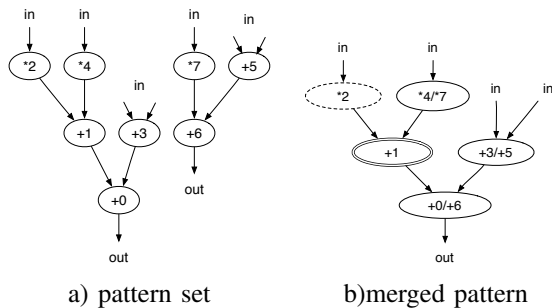


Fig. 13: An example of a set of patterns and a merged pattern.

is compatible to v_j and all other nodes on the path can be bypassed, i.e. their inputs can be set to such values that these nodes transfer data from input to output.

An edge in CG defines *mapping compatibility* between two nodes. Two CG nodes are not compatible whenever they map the same element of graph G_i (node, edge or path) to two different elements of graph G_j , or vice-versa. An example of a compatibility graph for the two patterns from Figure 13.a is presented on Figure 12. The ellipse nodes represent regular nodes, squashed rectangles nodes represent edge nodes and square nodes are used to represent path nodes. Nodes encapsulated with solid lines in the CG graph represent the maximal weighted clique found for the merged pattern from Figure 13.b.

Each node in CG has an associated weight. The weight of a node expresses the *area reduction* of the merged pattern if a given node is selected for merging. For instance, if node u_i/u_j is selected, the corresponding area reduction is $Area(u) + Area(u) - Area(u) - Area(Mux)$, assuming $Area(u) = Area(u_i) = Area(u_j)$ since two nodes were replaced with one node and a multiplexer.

To be able to compute a maximal weighted clique of CG graph each node $u \in V_c$ is modeled by finite domain variable $Sel_u = \{0, 1\}$. Variable $Sel_u = 1$ if the node is a member of a maximal weighted clique and 0 otherwise. A clique in CG is defined by constraint (32). This constraint imposes a condition for each two not connected nodes in the CG graph. At most one of these nodes can have its variable $Sel_u = 1$. In order to find the maximal weighted clique in CG the *Sum* variable defined by constraint (33) must be maximized. Each $weight(u)$ is defined as area reduction for this node if implemented in a merged pattern.

$$\forall (u_c, v_c) \notin E_c : Sel_{u_c} \neq 1 \vee Sel_{v_c} \neq 1 \quad (32)$$

$$Sum = \sum_{u \in V_c} Sel_u \cdot weight(u) \quad (33)$$

In practice, we do not use formulation specified in equation (32) but we use our **CLIQUE** constraint instead. This makes it possible to handle large clique graphs.

The presented model generates merged patterns that maximize weighted clique, i.e. maximizes the possible area reduction for the selected solution. This does not provide control over other important parameters such as critical path length or number of additional multiplexers on a critical path. In constraint based approach, this can be easily incorporated in the model by adding additional constraints. More information on that can be found in [29].

7. Example

Figure 14 presents a simple example of the “Motion Vector” application from the MediaBench set. The figure depicts application graph AG covered with automatically generated patterns and the set of selected patterns, PS . The

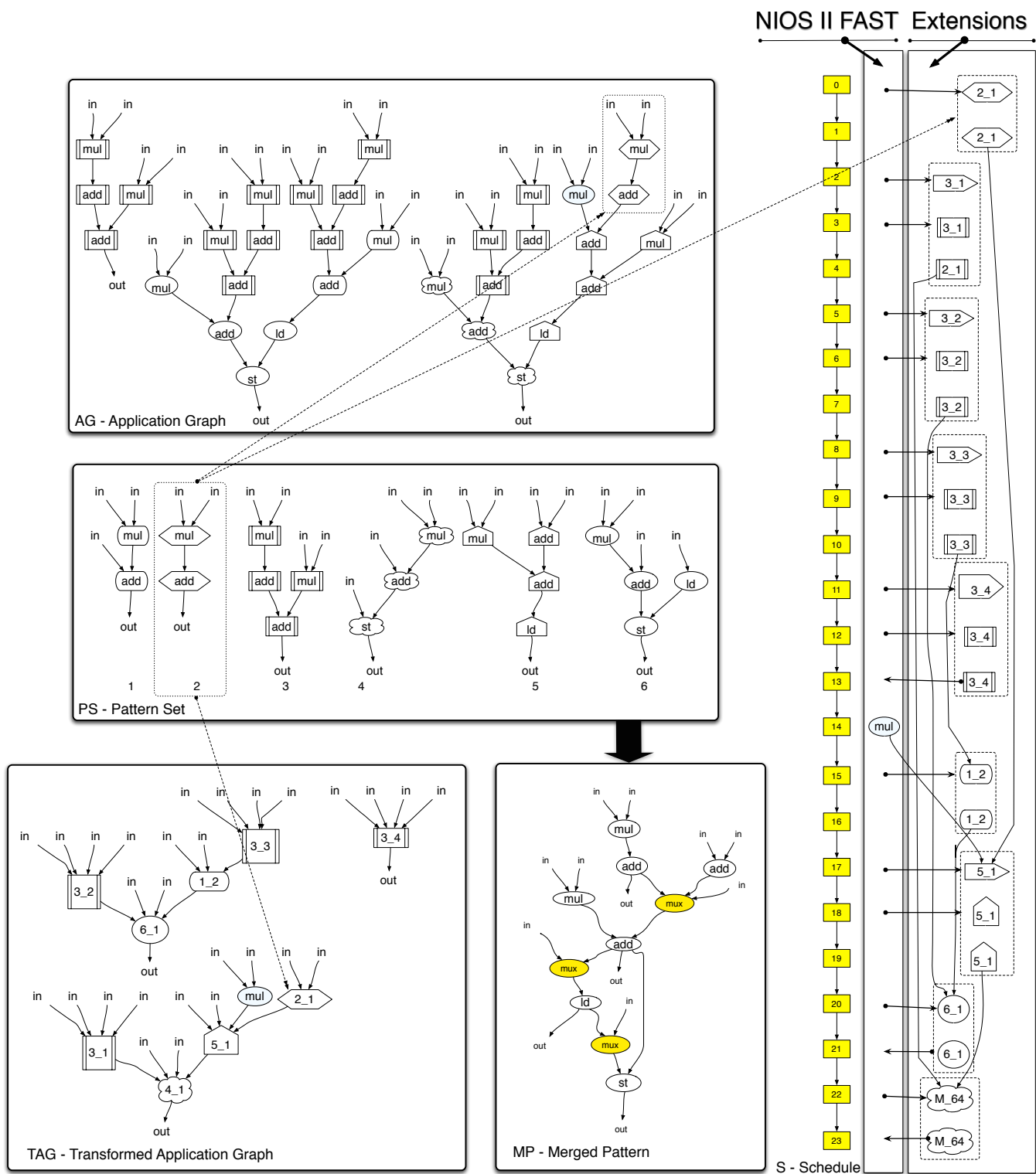


Fig. 14: Example of the “Motion Vector” application.

transformed application graph, *TAG*, represents the original application graph *AG* covered with selected pattern matches

from set *PS*. The resulting merged pattern, *MP*, is built by combining patterns from the selected pattern set. Our pattern

merging method produced, in this case, the merged pattern that has 74% area reduction over the total area of selected patterns.

Figure 14 presents also schedule S obtained for the sequential execution of the application on the processor and its extension, implementing merged pattern MP . In our model, computational patterns, implemented as a processor extension, communicate directly with each other using internal registers. Matches depicted in schedule S , executed on the processor and on the processor extension, are grouped separately. The figure also presents additional cycles needed for data communication between the processor and its extension as well as internal data transfers.

The design has been obtained under the following constraints. The maximal number of pattern nodes does not exceed 10 and the critical path of a pattern is not longer than 15ns. The critical path corresponds to three processor cycles for the Nios2Fast processor running at 200MHz on Stratix2 Altera FPGA. This simple application runs 30% faster with the implemented processor extension.

8. Conclusions

We have shown different tasks from our generic design flow for automatic generation of reconfigurable processor extensions. These tasks include computational pattern identification, selection, binding scheduling and merging. All these tasks are modeled and solved using constraints programming methods developed by us. Our approach makes it possible to take into account many architectural and technological constraints during design optimization. Moreover, many problems can be solved in one optimization step, as for example pattern selection, binding and scheduling that has not been possible with standard approaches. We can optimize either application speed-up or hardware resources. Thus, space exploration is possible. The results for the MediaBench applications show high quality of generated patterns and high application speed-up for both sequential and parallel execution models.

References

- [1] R. Kastner, A. Kaplan, S. O. Memik, and E. Bozorgzadeh, "Instruction generation for hybrid reconfigurable systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 7, no. 4, 2002.
- [2] M. Arnold and H. Corporaal, "Designing domain specific processors," in *Proceedings of the 9th International Workshop on Hardware/Software CoDesign*, Copenhagen, April 2001, pp. 61–66.
- [3] N. Clark, H. Zong, and S. Mahlke, "Processor acceleration through automated instruction set customization," in *36th Annual International Symposium on Microarchitecture*, 2003.
- [4] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instructionset extensions under microarchitectural constraints," in *40th Design Automation Conference (DAC)*, 2003.
- [5] P. Biswas, S. Banerjee, N. Dutt, L. Pozzi, and P. Ienne, "ISEGEN: Generation of high-quality instruction set extensions by iterative improvement," in *42nd Design Automation Conference (DAC)*, 2005.
- [6] A. Peymandoust, L. Pozzi, P. Ienne, and G. De Micheli, "Automatic instruction set extension and utilization for embedded processors," in *ASAP*, 2003.
- [7] Y. Guo, "Mapping applications to a coarse-grained reconfigurable architecture," in *PhD Thesis, University of Twent, Eindhoven, Netherland*, Sep. 8, 2006.
- [8] R. Leupers, K. Karuri, S. Kraemer, and M. Pandey, "A design flow for configurable embedded processors based on optimized instruction set extension synthesis," in *DATE*, 2006.
- [9] C. Wolinski and K. Kuchcinski, "Automatic selection of application-specific reconfigurable processor extensions," in *Proc. Design Automation and Test in Europe*, Munich, Germany, Mar. 10-14, 2008.
- [10] Y. Guo, G. Smit, H. Broersma, and P. Heysters, "A graph covering algorithm for a coarse grain reconfigurable system," in *Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, San Diego, California, June 11-13, 2003.
- [11] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch, and F. Charot, "Constraint-driven identification of application specific instructions in the DURASE system," in *SAMOS IX: International Workshop on Systems, Architectures, Modeling and Simulation*, Samos, Greece, Jul. 20-23, 2009.
- [12] G. Wang, W. Gong, and R. Kastner, "System level partitioning for programmable platforms using the ant colony optimization," in *International Workshop on Logic & Synthesis (IWLS'04)*, Temecula, California, June 2-4, 2004.
- [13] N. Moreano, E. Borin, C. de Souza, and G. Araujo, "Efficient datapath merging for partially reconfigurable architectures," *IEEE Trans. Computer-Aided Design*, vol. 24, no. 7, pp. 969–980, Jul. 2005.
- [14] Z. Huang, S. Malik, N. Moreano, and G. Araujo, "The design of dynamically reconfigurable datapath coprocessors," *ACM Transaction in Embedded Computing Systems*, vol. 3, no. 2, May 2004.
- [15] J.-C. Régin, "Solving the maximum clique problem with constraint programming," in *Proc. CPAIOR*, 2003.
- [16] P. Brisk, A. Kaplan, and M. Sarrafzadeh, "Area-efficient instruction set synthesis for reconfigurable system-on-chip designs," *Proc. 41st Design Automation Conference*, pp. 395–400, 2004.
- [17] M. Corazao, M. Khalaf, L. Guerra, M. Potkonjak, and J. Rabaey, "Performance optimization using template mapping for datapath-intensive high-level synthesis," *IEEE Trans. Computer-Aided Design*, vol. 15, no. 8, pp. 877–888, Aug. 2004.
- [18] J. Cong and W. Jiang, "Pattern-based behavior synthesis for FPGA resource reduction," in *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 2008, pp. 107–116.
- [19] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, 1976.
- [20] J. Larrosa and G. Valiente, "Constraint satisfaction algorithms for graph pattern matching," *Mathematical Structures in Computer Science*, vol. 12, pp. 403–422, 2002.
- [21] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 10, pp. 1367–1372, 2004.
- [22] Z. Terem, G. Kamhi, M. Vardi, and A. Iron, "Pattern search in hierarchical high-level designs," in *Proc. of the 11th IEEE Intl. Conference on Electronics, Circuits and Systems*, 2004., Dec. 2004.
- [23] J. Larrosa and G. Valiente, "Graph pattern matching using constraint satisfaction," in *Proc. Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, Berlin, Germany, Mar. 25–27 2000, pp. 189–196.
- [24] K. Kuchcinski, "Constraints-driven scheduling and resource assignment," *ACM TODAES*, vol. 8, no. 3, pp. 355–383, Jul. 2003.
- [25] F. P. Preparata and M. I. Shamos, *Computational Geometry. An Introduction*. Springer-Verlag, 1985.
- [26] A. Kountouris and C. Wolinski, "Efficient scheduling of conditional behaviors for high level synthesis," *ACM TODAES*, vol. 7, no. 3, pp. 380–412, Jul. 2002.
- [27] C. Wolinski and K. Kuchcinski, "Computation patterns identification for instruction set extensions implemented as reconfigurable hardware," in *ERSA*, Las Vegas, Nevada, USA, Jun. 25-28, 2007.
- [28] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch, and F. Charot, "Constraint-driven instructions selection and application scheduling in the DURASE system," in *ASAP*, July 2009.
- [29] C. Wolinski, K. Kuchcinski, and E. Raffin, "Architecture-driven synthesis of reconfigurable cells," submitted for publication, Mar. 2009.