Graph constraints in embedded system design

Wolinski, Christophe; Kuchcinski, Krzysztof; Martin, Kevin; Floch, Antoine; Raffin, Erwan; Charot, Francois

2010

Document Version:
Early version, also known as pre-print

Graph Constraints in Embedded System Design

CHRISTOPHE WOLINSKI
University of Rennes I/IRISA/INRIA Centre Rennes - Bretagne Atlantique, France
KRZYSZTOF KUCHCINSKI
Lund Univeristy, Sweden
KEVIN MARTIN, ANTOINE FLOCH, ERWAN RAFFIN and FRANÇOIS CHAROT
University of Rennes I/IRISA/INRIA Centre Rennes - Bretagne Atlantique, France

In this paper, we present application of graph constraints combined with finite domain constraints for embedded system optimization problems. In particular, we present methods for identification and selection of computational patterns as well as application scheduling with these patterns that has direct application in ASIP processor design. In this work we use *connected component*, *(sub)graph isomorphism* and *clique* constraints. Our experimental results show that these methods work for relatively large examples and provide much better results than previous heuristic based approaches.

Categories and Subject Descriptors: D3.3 [**Programming Languages**]: Language Constructs and Features—*constraints*

Additional Key Words and Phrases: constraint programming, reconfigurable architectures, resource assignment, scheduling, system-level synthesis

1. INTRODUCTION

Embedded systems are designed carefully to optimize different design parameters such as cost, performance and power consumption. Tools used for these kind of optimisations use different models. Most popular models are graph based. Annotated graphs are used to represent basic features of embedded systems and then design tools work on these models to carry out different kind of optimizations. They use different properties of graphs and solve different graph problems, such as (sub)graph isomorphism, simple path, clique and connected components. Most of these problems are NP-complete or NP-hard and heuristic ad-hoc solutions are usually used.

In this paper, we use graph constraints for modeling and solving a problem of extending standard processors with specialized instructions to improve their performance for selected applications. This problem can be defined as finding *computational patterns*, implemented as special instructions, that provide performance improvement. These patterns can be in turn defined as subgraphs of an application graph. We are interested in subgraphs that provide more efficient implementation than decomposition to a sequence of instructions and subgraphs that are present in many different parts of an application graph. Moreover, other technological constraints may apply. We propose to use *graph constraints* combined with other finite domain constraints to define and solve this problem. For this purpose we use JaCoP constraint solver [Kuchcinski 2003] extended with graph constraint library.

Design of application specific processors (ASIPs) or processor extensions involves

identification of new instructions and their implementation as computational cells. This process can be defined using graphs and graph operations. An application is represented as a graph and new instructions as subgraphs of this graph. In our case, we use Hierarchical Conditional Dependency Graphs (HCDGs) [Kountouris and Wolinski 2002] but other representations are also possible. Using this representation, subgraph isomorphism and connected components constraints together with constraint programming are used to identify and select patterns as well as schedule applications [Wolinski and Kuchcinski 2008]. The result is a set of instructions defined as subgraphs, often called *computational patterns*. Direct implementation of all these patterns is area inefficient and therefore they need to be merged to form a *reconfigurable cell*. In this paper, we define and solve this problem using constraint programming. We use *clique* constraint combined with other constraint that define architectural requirements for merged patterns.

Related work in this area uses usually heuristic approaches for *identification of computational patterns*. Iterative heuristics (e.g., [Galuzzi et al. 2007]) progressively generate a set of patterns that fulfill certain constraints, such as number of inputs/outputs or connectivity. Other approaches [Dinh et al. 2008] combine simple patterns with more complex ones. Recent research [Bonzini and Pozzi 2008a] uses a method that performs an exhaustive exploration of patterns with input/output constraints.

Incremental algorithms are also used for *pattern selection*. Guo et al. [2003] proposes an iterative algorithm based on a conflict graph. Bonzini and Pozzi [2008b] proposes a toolchain that first determines the best pattern to cover a graph and then computes a schedule. Other approaches deal with the pattern selection problem globally. For example, Clark et al. [2005] defines a method based on dynamic programming to select the best occurrences under area constraints.

2. GRAPH CONSTRAINTS

Graph constraints work on graphs that can define different types of graphs used in embedded system design. In our work. we have defined an annotated graphs for this purpose. Graph $G = (N, E)$ contains nodes N and edges $E = (n_i, n_j)$, where $n_i, n_j \in V$. Each node has an assigned label that basically defines a type of a node. It can represent an operation of a node (e.g., +, – or *). A node has also assigned *connectors* that are used to connect this node to other nodes or define its connection structure. Each connector can be connected to at most one other connector. The connectors are also labeled defining types of connections. Therefore, we can define directed graphs by specifying *in* and *out* connectors or non-directed graphs where all connectors have the same type. This also makes it possible to distinguish non-symmetrical operations, such as "–" or "/". They will have two different labels for their input connectors to distinguish them. Similarly, we can define a node that has two data inputs and one control input. Fan-out connections must be defined as special fan-out nodes. This graph definition makes it possible to represent HCDG graphs using our graph representation and then use graph constraints to compute different properties of these graphs.

The constraints defined for our labeled graphs define for each node a finite domain variable. These variables represent valid solutions for the constraints. For exam-

```
DIPS ← ∅
for each n_s ∈ N
  TPS ← ∅
  CPS ← FindAllPatterns(G, n_s)
  for each p ∈ CPS
      if ∀_{pattern∈TPS} p ≢ pattern
          TPS ← TPS ∪ {p},
          NMP_p ← | FindAllMatches(G, p) |
  NMP_{n_s} ← | FindAllMatches(G, n_s) |
  for each p ∈ TPS
      if coef · NMP_{n_s} ≤ NMP_p
          DIPS ← DIPS ∪ {p}
return DIPS
```

Fig. 1. Pattern identification algorithm.

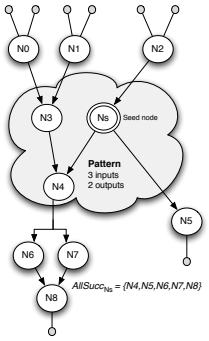

Fig. 2. Pattern example.

ple, a *connected component* constraint assigns a 0/1 variable to each node. This variable is 1 if a node belongs to a connected component and 0 otherwise. Having finite domain variables assigned to graph nodes makes it possible to combine these variables with other finite domain variables of the model. This approach is different than [Dooms et al. 2005] where a separate graph domain is introduced but fits well our purpose since we mix finite domain constraints and graph constraints.

3. CONNECTED COMPONENTS

Connected components constraint is a basic mechanism used for identification of computational patterns. Pattern identification is defined, in our approach, 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 $E_p \subseteq E$. In our approach, pattern P is a connected sub-component of graph G. It is also subgraph isomorphic to graph G.

The pattern identification algorithm is depicted in Figure 1. It finds first all patterns in the graph around seed node $n_s \in N$. This is achieved by finding connected components in a non-directed graph satisfying additional constraints (function $FindAllPatterns(G, n)$). We examine all nodes as seed nodes but more selective approaches can also be used. The found patterns can be identical (subgraph isomorphic) with already identified patterns and therefore our algorithm checks this using graph isomorphism constraint discussed in the next section. Finally, we use a heuristic to accept only patterns whose numbers of matches in the application graph is high enough. The number of matches in the application graph is also obtained by finding all subgraphs isomorphic to patterns in the application graph.

Additional constraints that are used together with the connected component constraint can be of different kind. First, we always define constraints that limit number of inputs and outputs of the patterns. This is defined by an architecture and need to be followed by identified patterns. We also have possibility to add constraints that will limit the length of the critical path of a newly created pattern and thus influence the timing of the extensions.

Figure 2 depicts an example computational pattern identified around seed node Ns with constraints that number of inputs cannot be greater than three and number of outputs cannot exceed two.

4. (SUB)GRAPH ISOMORPHISM

Subgraph and graph isomorphism are defined in our CP framework as a `GraphMatch` constraint. This constraint takes as arguments two graphs, *target* graph G_t and *pattern* graph G_p, and establishes (sub)graph isomorphism using a matching function $f : N_t \rightarrow N_p$. Moreover, we assign finite domain variable var_i to each node $n_i \in N_t$ of the target graph. The domain of this variable is denoted as $D(n_i)$ and defines matching f, i.e., if variable var_i contains in its domain value p then pattern graph node numbered p possibly matches target node i. Since we use finite domain variable it has initially all nodes in its domain. The domain of this variable is then pruned by our consistency algorithm and will contain only compatible nodes.

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 subgraph isomorphism. This is achieved by assigning value $\perp$ for variables representing not mapped parts in the target graph. This subgraph isomorphism is different than the classical definition from graph theory where subgraph isomorphism means that a found pattern 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 subgraph isomorphism and monomorphism, if needed.

Our `GraphMatch` constraint is used during computational pattern identification, as discussed in previous section, in several situations. First, we find whether a newly identified pattern is isomorphic to already identified patterns. If this is the case we do not store this pattern in our pattern set. Second, we find how many isomorphic matches of a given pattern exist in a target graph and select patterns that occur often in a target graph. This helps to find representative patterns and reduces the number of "useful" patterns.

The subgraph isomorphism constraint can also be used for binding and scheduling. In our early work [Fuentes Martínez and Kuchcinski 2007], we have used this constraint together with ordinary scheduling constraints (precedence and resource constraints) to select computational patterns, do binding and scheduling in a single optimization step. Our experiments carried out on classical high-level synthesis benchmarks showed good results. We have obtained (proved) optimal results for all benchmarks for graphs of size up to 50 nodes. In this project, however, we use slightly different formulation to handle larger and more complicated examples.

For pattern selection and application scheduling we use two step process. First, using subgraph isomorphism constraint and search for all solutions, we determine all possible matches of a given pattern in an application graph. A match is a subgraph of an application graph that is isomorphic to a given pattern. After execution of this step, each node $n \in N_t$ has an associated set $matches_n$ containing all matches that can cover it. Obviously, in the final covering of graph G each node $n \in N$ can only be covered by one match since we do not allow overlapping matches.

In the second step we carry out actual pattern selection and scheduling. Our methods support both sequential and parallel execution of matches by using slightly different models. In sequential scheduling case, we model selection of a given match in a final schedule using finite domain variable m_{sel} associated to each match $m \in M$, where M is a set of all matches. The value of variable $m_{sel} = 1$ if match m

Table I. Results obtained for sequential scheduling and parallel scheduling.

			Sequential scheduling				**Parallel scheduling (model A)**							**Parallel scheduling (model B)**					
Benchmarks	|V|	Cycles	Selected	Cycles	Speedup	Runtime (s)	Selected	Cycles	Parallel memory accesses	Local variables	Number cells	Speedup S_1	Runtime (s)	Selected	Cycles	Number cells	Speedup S_2	Runtime (s)	Percent of S_1
JPEG IDCT	200	254	6	195	1.3	18.9	28	22	13	78	8	11.55	14.2	28	34	5	7.47	10.2	64%
BF	330	340	9	163	2.09	0.1	7	81	3	152	4	4.2	13.2	7	98	4	3.47	0,01	82%
MESA invert matrix	278	334	7	148	2.26	3.8	9	25	38	134	8	13.36	0.6	9	78	4	4.28	0.5	32%
with mem. access ≤ 10							9	29	10	134	8	11.52	1.99						
MCRYPT cast128	424	464	29	240	1.83	50.5	18	202	8	155	6	2.3	6.6	18	219	3	2.12	15.9	92%
GSMenc	387	433	8	117	3.7	0.1	9	16	15	132	8	27.06	6.0	9	68	2	6.37	7.2	23%
POLARSSL aes	1350	1658	12	316	2.28	6.2	15	280	27	739	8	5.92	20.2	10	482	5	3.44	43.6	58%
POLARSSL des3	398	530	15	295	1.76	5.2	26	95	6	156	5	5.58	23.9	25	109	4	4.86	24.8	87%
Maximal Speedup					3.7							27.06					7.47		
Average Speedup					2.17							9.98					4.57		

is selected or 0 otherwise. The schedule length for this case can be easily defined as a sum of all execution times using equation (1). Minimization of variable T_{seq} provides the fastest schedule with selected computational patterns. The selected patterns define also most efficient patterns to obtain this schedule. In parallel scheduling case, we minimize a different variable that defines schedule length for this case. It is defined in equation (2), where $m_{i_{start}}$ and $m_{i_{delay}}$ define match start time and its delay.

$$T_{seq} = \sum_{m \in M} m_{sel} \cdot m_{delay} \tag{1}$$

$$T_{par} = \max_{1 \leq i \leq N} (m_{i_{start}} + m_{i_{delay}}) \tag{2}$$

Our methods make it possible to model two types of architectures for processor extensions. One with local memories (model A) and one without them (model B). When an extension does not have local memories (model B) all data transfers are carried out to and from a processor. Obviously, model A can provide better performance for the cost of additional registers.

We have carried out experiments with multimedia applications. Table I presents results obtained for selected applications from *MediaBench*, *MiBench* and *Cryptographic Library* benchmark sets. These applications are written in C and compiled using our design flow for the ALTERA NIOS target processor.

The experiments were carried out for two architecture models. One with local memories (model A) and one without local memories (model B). The number of parallel memory accesses was not constrained to obtain the maximal speed-up. As table I depicts, a significant speedup was obtained where local memories were used. We also present, as an example, the results obtained for the "MESA invert matrix" application when the number of parallel accesses was limited to 10.

5. CLIQUE

The selected computational patterns, represented as subgraphs of an application graph, need to be implemented as processor extensions. Implementation of all

patterns will be expensive and inefficient and therefore we have developed a method for merging the patterns into a single reconfigurable pattern.

The problem of merging computational patterns can be modeled using graphs. Finding a suitable merge can be defined as finding the maximum common sub-graph isomorphism (MCS). This is an optimization problem that is known to be NP-hard. Formally, the problem, defined for two graphs (G_1 and G_2), is to find the largest induced sub-graph of G_1 isomorphic to a sub-graph of G_2. One possible solution for this problem is to build a modular product graph [Brandes and Erlebach 2005], in which the largest clique represents a solution for the MCS problem. The other method is to use a kind of backtracking algorithm that iteratively adds vertices which does not violate the common sub-graph condition. We use the first method and build first a *compatibility graph* between two patterns and then find a *clique* that maximizes a given cost function.

Our method iteratively merges two computational patterns represented by graphs. In each step, a pattern selected from a pattern set and a already partially merged pattern are used to produce a new merged pattern. *Compatibility graphs* (CG) are created for this purpose for pairs of computational patterns. The nodes of the compatibility graph are created for *shared nodes*, *shared connections* and *shared paths* with *bypassed* nodes. Shared nodes, for example, are nodes that have the same type and the same number of inputs/outputs. Similar considerations are used to define shared connections and paths. An edge in CG defines *mapping compatibility* between two nodes. Mapping compatibly respects a number of conditions between these nodes that are specified for all types of nodes. For instance, an edge exists between between two shared nodes of CG if both nodes are not constructed from the same nodes of pattern graphs. This removes possibility to map the same nodes of pattern graphs more than once. Similar rules apply to shared connections and shared path nodes. Finally, the *clique partitioning* with a given cost function optimizing specific design features under architectural constraints is applied to a compatibility graph.

A unique feature of our approach is that we can extend our model that contains compatibility graph with other constraints. In this way we can search for solutions that not only maximize sharing but also respect architectural constraints. This is not possible in other approaches. Consider, for example two computational patterns depicted in Figure 3. Traditional methods for pattern merging, such as [Moreano et al. 2005] produce a solution depicted in Figure 4.a. The pattern from Figure 4.b, was obtained with our method under conditions that the length of a critical path is three, the number of multiplexers on the critical path is zero and the *bypassed nodes* are allowed. Node "+1" in Figure 4.b is a *bypassed* node, which only passes data without any processing for the second pattern from Figure 3. The quality of the design has been significantly improved by applying additional architectural constraints. Both the area and the critical path are optimized. This simple example shows that the standard approach largely used in the past is not always efficient.

Table II shows different results obtained for the set of patterns identified by our system for DSP applications from the MediaBench test suite [Lee et al. 1997]. The area reduction is specified in relation to the area of the set of patterns, and it is expressed in the number of combinational atoms (denoted as CA in Table II) for

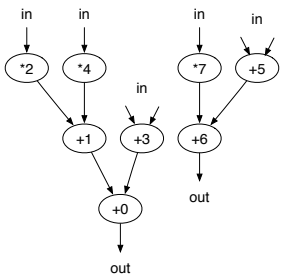

Fig. 3. Set of patterns.

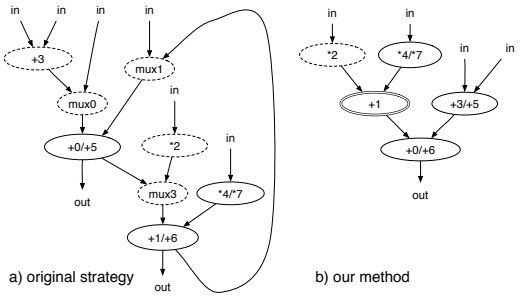

Fig. 4. Two cases of a merged pattern.

32 bits operators. We also specify the number of edges in the merged patterns. For each application from the MediaBench test suite, five experiments with the following options have been carried out. N- node sharing, E- edge sharing, P(n)- path sharing; n- maximum number of bypassed nodes, CP- critical path cannot increase and NM(n)- maximum of number of n multiplexers allowed.

Exp.1 N=Yes
Exp.2 N=Yes E=Yes P=2
Exp.3 N=Yes E=Yes NM=0
Exp.4 N=Yes E=Yes P=2 CP=Yes NM=0
Exp.5 N=Yes E=Yes P=2 CP=Yes NM=2

All weighted cliques found during pattern merging were proved optimal. The runtime for all experiments, including the time for the optimality proof, for most restrictive Exp.4 experiment was 0.3s and the average runtime is only 0.1s.

6. CONCLUSIONS

In this paper, we have presented our approach how to use graph constraints in embedded system design. We have first presented the graphs used in our work and indicated that they can represent a general design representation know as Hierarchical Conditional Dependency Graphs. In particular, we have shown application of *connected component*, *(sub)graph isomorphism* and *clique* constraint for the problem of identification and selection of computational patterns as well as application scheduling with these patterns. This problem is important for design of application specific processor where processor accelerators can be automatically determined

Table II. Pattern merging results for pattern sets identified by our system for MediaBench test suite.

Application	Nb. patterns	Area reduction in %					Area in CA for Altera Stratix2 EP2560					Number of Edges in merged pattern				
		Exp. 1	Exp. 2	Exp. 3	Exp. 4	Exp. 5	Exp. 1	Exp. 2	Exp. 3	Exp. 4	Exp. 5	Exp. 1	Exp. 2	Exp. 3	Exp. 4	Exp. 5
Auto Regression Filter	3	47	49	1	48	48	2180	2084	4168	2048	2048	20	14	24	14	14
Cosine	9	81	86	9	10	84	1476	1412	3164	2473	1473	35	36	67	48	44
Elliptic Wave Filter	8	80	80	72	73	78	1442	1442	2067	2003	1600	43	37	54	45	44
EPIC Collapse	7	67	68	13	23	67	1127	1059	2882	2576	1092	38	37	56	48	40
FIR	8	81	82	72	80	80	1378	1250	1971	1410	1378	35	24	41	35	35
JPEG IDCT	7	75	76	56	66	75	1379	1347	2469	1948	1379	36	32	47	42	37
JPEG Smooth Downsample	9	64	64	51	53	66	448	448	608	576	436	36	31	44	39	31
JPEG Write BMP Header	7	73	73	12	12	71	1073	1073	5548	4045	1137	26	26	64	62	32
MESA Feedback Points	5	50	54	38	40	54	1843	1715	2308	2212	1715	33	26	34	28	26
MESA Horner Bezier	5	59	60	35	48	60	1683	1619	2677	2148	1619	19	15	21	17	15
MESA Interpolate Aux	4	23	23	22	23	23	1684	1684	1716	1700	1684	19	19	22	20	19
MESA Matrix Multiplication	3	76	77	55	75	75	2340	2318	4520	2436	2436	35	32	56	41	40
MESA Smooth Triangle	9	78	79	66	66	75	2370	2278	3835	3770	2808	37	30	40	36	33
MPEG IDCT	6	63	63	51	61	63	1810	1804	2390	1861	1804	54	54	66	60	54
MPEG Motion Vector	7	81	81	63	79	80	1218	1218	2340	1314	1282	22	21	33	26	24
Average		66.53	67.67	41.07	50.47	66.6										

and designed.

Other graph constraints can also be used for other applications. For example, *simple path* constraint has been used by us to minimize area and reconfiguration time of the communication network in regular 2D reconfigurable architectures [Wolinski et al. 2008]. The constraint helps us to select routing for different paths for different communications.

REFERENCES

Bonzini, P. and Pozzi, L. 2008a. On the Complexity of Enumeration and Scheduling for Extensible Embedded Processors. Tech. Rep. 2008/07, University of Lugano. Dec.

Bonzini, P. and Pozzi, L. 2008b. Recurrence-aware instruction set selection for extensible embedded processors. *IEEE Trans. Very Large Scale Integr. Syst. 16,* 10, 1259–1267.

Brandes, U. and Erlebach, T. 2005. *Network Analysis: Methodological Foundations*. Springer.

Clark, N., Zhong, H., and Mahlke, S. 2005. Automated custom instruction generation for domain-specific processor acceleration. *Trans. on Computers 54,* 10, 1258–1270.

Dinh, Q., Chen, D., and Wong, M. D. F. 2008. Efficient asip design for configurable processors with fine-grained resource sharing. In *FPGA '08: Proc. of the 16th intl. ACM/SIGDA symposium on Field programmable gate arrays*. 99–106.

Dooms, G., Deville, Y., and Dupont, P. 2005. CP (Graph): Introducing a graph computation domain in constraint programming. In *Principles and Practice of Constraint Programming - CP 2005*. Springer, New York, NY, USA, 211–225.

Fuentes Martínez, A. and Kuchcinski, K. 2007. Graph matching constraints for synthesis with complex components. In *Proc. 10th Euromicro Conference on Digital Systems Design*. Lübeck, Germany.

Galuzzi, C., Bertels, K., and Vassiliadis, S. 2007. The spiral search: A linear complexity algorithm for the generation of convex MIMO instruction-set extensions. In *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*. 337–340.

Guo, Y., Smit, G. J., Broersma, H., and Heysters, P. M. 2003. A graph covering algorithm for a coarse grain reconfigurable system. In *LCTES '03: Proc. of the ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*. 199–208.

Kountouris, A. and Wolinski, C. 2002. Efficient scheduling of conditional behaviors for high level synthesis. *ACM Transactions on Design Automation of Electronic Systems 7,* 3 (July), 380–412.

Kuchcinski, K. 2003. Constraints-driven scheduling and resource assignment. *ACM Trans. Des. Autom. Electron. Syst. 8,* 3, 355–383.

Lee, C., Potkonjak, M., and Mangione-Smith, W. H. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*. 330–335.

Moreano, N., Borin, E., de Souza, C., and Araujo, G. 2005. Efficient datapath merging for partially reconfigurable architectures. *IEEE Trans. Computer-Aided Design 24,* 7 (July), 969–980.

Wolinski, C. and Kuchcinski, K. 2008. Automatic selection of application-specific reconfigurable processor extensions. In *Proc. Design Automation and Test in Europe*. Munich, Germany.

Wolinski, C., Kuchcinski, K., Teich, J., and Hannig, F. 2008. Area and reconfiguration time minimization of the communication network in regular 2D reconfigurable architectures. In *Proc. of the International Conference on Field Programmable Logic and Applications (FPL)*. Heidelberg, Germany.