Peeling Algorithm for Custom Instruction Identification

Kang Zhao and Jinian Bian
EDA Lab, Dept. Computer Science & Technology
Tsinghua University, Beijing 100084, China
Email: zhaokang@tsinghua.edu.cn; bianjn@tsinghua.edu.cn

***Abstract*—To speedup the custom instruction identification for the application specific instruction-set processor (ASIP), this paper proposes a peeling algorithm. It starts with the maximal valid pattern, and then gets a set of valid sub-patterns through deleting the source and sink respectively. Furthermore, a local priority is proposed for the exhaustive pruning. The final experiment indicates a distinct speedup compared to the fastest deterministic algorithm.**

I. Introduction

Application specific instruction-set processor (ASIP) can obtain a tradeoff between the efficiency and flexibility for the embedded system design, which is designed facing the challenges of both high efficiency and time-to-market pressure. ASIP can improve the processor performance through the custom instructions, which can be viewed as the special hardware in the processor [1].

To implement the instruction-set extension (ISE), two sub-problems should be solved: 1) custom instruction identification, which enumerates all feasible subgraphs from the program's DFG (data flow graph); 2) instruction selection, which selects an optimal instruction set under various constraints [2]. In this flow, candidate custom instructions are first enumerated via the identification process, and then the instruction subset which satisfies the performance/cost constraints will be selected via the selection process. This paper will only focus on the first sub-process: instruction identification. Its motivation is to enumerate all the candidate custom instructions under the architectural constraints. It is a subgraph enumeration problem, and there are two choices for each node in the DFG: included or not by the subgraph. Therefore, the search space will be 2^n.

To settle this issue, researchers have proposed many identification methods. [3] used an interesting strategy based on maze search. It started with a selected node, and then grew it in a better direction. [4] proposed an algorithm based on subgraph isomorphism. However, such methods can only get segmental results, and may miss some better solutions. In addition, [5] proposed an algorithm based on binary-decision-tree (BDT). To reduce the space, the choices which violate the constraints will be pruned in the BDT. However, it cannot deal with large DFGs. Furthermore, Yu proposed a fast algorithm to enumerate the connected [2] patterns. It defined upward and downward corns, and obtained the feasible subgraphs using corns' combination. However, this algorithm often considered a pattern more than once. To settle this issue, [6] proposed an algorithm based on exhaustive pruning. It started with an empty set and combined the other nodes under several constraints. Once the constraints were violated, this direction stopped.

However, the previous identification efficiency is still low when enumerating all feasible subgraphs. Some work speed up via strict constraints, some used heuristic methods to get a subset result, and some only enumerated the maximal patterns instead of all patterns. To settle these issues, this paper proposes a peeling algorithm, which can enumerate all valid patterns as the candidates with a high speed.

The rest is organized as follows. Section 2 presents the problem formulation. Section 3 proposes the details of the peeling algorithm. Section 4 verifies the algorithm with the experiment. Finally, Section 5 presents the conclusion.

II. Problem Formulation

The custom instruction identification can be formulated as a subgraph enumeration problem. Let $G(V, E)$ be a directed acyclic graph (DAG), where V is the nodes which denote operations, and E is the arcs which represent the data dependencies. Custom instructions is the combination of primitive operations. Since the primitive operation is represented as a node, the custom instruction will be mapped to a subgraph. Here we call the subgraph a pattern. Furthermore, the indegree and outdegree of the subgraph will denote the operand numbers of the custom instructions.

Definition 1: Let $G'(V', E')$ be a subgraph of $G(V, E)$. If $\forall v_1 \in V' \wedge \forall v_2 \in V' \wedge \forall v_3 \in Path(v_1, v_2) \rightarrow v_3 \in V'$, then the subgraph G' is convex; otherwise, it is nonconvex.

$Path(v_1, v_2)$ denote the nodes on the paths from v_1 to v_2. The subgraph for the custom instructions must be convex [6]. We will loose the I/O constraints, and only consider the convexity and the connectivity constraints. In conclusion, the instruction identification problem can be presented as:

Problem: Given a DAG $G(V, E)$, find out all the valid patterns $\{G'\}$ exhaustively, which satisfy the two constraints simultaneously: (1)G' is convex; (2)G' is connected.

III. Peeling Algorithm

A DFG example is shown in Fig. 1. This section will propose the peeling algorithm based on this example.

978-1-4244-7456-1/10/$26.00 ©2010 IEEE

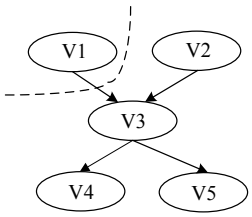

Fig. 1. Example of a DFG.

1. patterns$\Leftarrow \emptyset$;
2. patterns.add($\widehat{G_i}$);
3. $X \Leftarrow \emptyset$;
4. partition($\widehat{G_i}, X$);
5. **return** patterns;

Fig. 2. An overview of the peeling algorithm.

A. Overview

Let $G(V, E)$ be a DAG. For the nodes $u, v \in V$, the relations between u, v and G are:

$$Pred(G,u) = \{v | v \in V \wedge Path(v,u) \neq \emptyset\}$$
$$Succ(G,u) = \{v | v \in V \wedge Path(u,v) \neq \emptyset\}$$
$$Disc(G,u) = \{v | v \in V \wedge Path(u,v) = Path(v,u) = \emptyset\}$$

The DFG G may be disconnected. The precondition of the peeling algorithm is a connected pattern, therefore, G must be cut into several connected components, and we call each connected pattern a cut of G.

Based on the above definitions, Fig. 2 presents an overview of the peeling algorithm. Given a connected DFG G_0, it will be the initial graph during the partitioning. X is the track of the priority list for pruning, which is initialized to be empty. Finally the valid pattern list is returned via $patterns$.

B. Partition

The motivation of the $partition$ function in Fig. 2 is to divide the pattern into several subsets, and search for the valid patterns recursively.

Theorem 1: G is a valid pattern. For $\forall u \in G$, Pred(G, u), Succ(G, u) and Disc(G, u) are all convex patterns.

Proof: By contradiction, let Pred(G, u) be nonconvex. Then, we can get that $\exists a \exists b \exists c (a \in$Pred$(G, u) \wedge b \in$Pred$(G, u) \wedge c \notin$Pred$(G, u) \wedge Path(a, c) \neq \emptyset \wedge Path(c, b) \neq \emptyset)$. Since there is a path from c to b, c must also be the predecessor of u, i.e. $c \in$Pred(G, u). This conflicts with the precondition, and then Pred(G, u) must be convex. It is similar for Succ(G, u). It is obvious that Disc$(G, u) \cap (\{u\} \cup$ Pred$(G, u) \cup$ Succ$(G, u))=\emptyset$. By contradiction, if Disc(G, u) is nonconvex, there must be a node $x \notin G$, which is on the path between two nodes in Disc(G, u). Then, G is nonconvex. This conflicts with the definition of the valid pattern G. Therefore, Disc(G, u) is also convex.

Based on Theorem 1, the graph can be partitioned into at least three parts: Pred(G, u), Succ(G, u) and Disc(G, u). Then, there is no need to examine the convexity. However, if the current pattern is partitioned based on each node, there will be many repeated enumerations during the recursive partitions. Therefore, the source and the sink will be selected instead of each node as the start for the partitioning. The successors of the source and the predecessors of the sink will include other nodes' successors and predecessors. In the following the source and sink nodes are called peeling node.

Theorem 2: Let G be a valid pattern, and u be its peeling node. The subgraph $G/\{u\}$ is denoted as Dele(G, u). Then, Dele(G, u) is a convex pattern.

$$Dele(G,u) = G - \{u\}$$
$$= \begin{cases} Succ(G,u) \cup Disc(G,u) & \text{if } u \in \text{source}(G) \\ Pred(G,u) \cup Disc(G,u) & \text{if } u \in \text{sink}(G) \end{cases}$$

Proof: Let u be the sink node of G. Then, Dele(G, u)= Pred$(G, u) \cup$Disc(G, u). Since Pred(G, u) and Disc(G, u) are both convex (Theorem 5), we only need to prove that all the nodes on the path "$x \rightarrow y$" belongs to this subgraph ($x \in$Pred$(G, u) \wedge y \in$Disc(G, u)). By contradiction, if $\exists z (z \notin$ Dele$(G, u) \wedge$ (there is a path "$x \rightarrow z \rightarrow y$")), Dele$(G, u)$ will be nonconvex. According to the definition of Disc(G, u), $z \neq u$ and $z \notin G$. Based on Definition 1, z will bring G into nonconvex. This is a contradiction. Therefore, Dele(G, u) must be a convex pattern.

In conclusion, the current pattern can be partitioned via deleting the peeling nodes, as the algorithm shown in Fig. 3. The first step is to find the peeling nodes and delete them respectively. Since $Dele(G, u)$ may be disconnected, function $disconnected$ is used to examine the generated $Dele$. Furthermore, a priority is defined to avoid repeated enumeration. The function $violate\_priority$ is to examine whether the current peeling node violates the priority or not. After the partitioning, the priority list X will be updated according to the set of the peeling nodes, and the generated $Dele$s will be partitioned recursively under the new priority X'. This partitioning seems to be a peeling process, so its name is peeling algorithm.

Although this algorithm can enumerate all valid patterns, it may bring repeated enumerations. As the example shown in Fig. 4, subset {3, 4} can be generated through the path (a)(c)(f), and also the path (a)(d)(g). Except for the paths starting with (a), {3, 4} can also be generated starting with (b), because {3, 4} is also the subset of {1, 2, 3, 4}. Therefore, a priority in the $partition$ function is proposed to avoid repeated enumerations.

C. Pruning

As shown in Fig. 4, (c) and (d) have the same subset {3, 4}. How to avoid such repeated enumerations? For the pattern (d), its only "advantage" is that (d) has {2} but (c) does not. Therefore, if (d) deletes {2}, it will have no "advantage" and a repeated enumeration generates. Such comparison only exists between the brother patterns which have the same parent. Under this strategy, a local priority is proposed.

partition(G, X)

1. **if**$(G = \emptyset)$ **return**;
2. $P \Leftarrow$ source$(G) \cup$ sink(G);
3. **for**(each node $u \in P$)
4. **if**(violate_priority(u, X)) **or** disconnected(G, u)
5. $P \Leftarrow P - \{u\}$;
6. **else**
7. patterns.add(Dele(G, u));
8. **for**(each node $u \in P$)
9. $X' \Leftarrow$ change_priority(P, u, X);
10. partition(Dele$(G, u), X'$);

Fig. 3. Algorithm of the *partition* function.

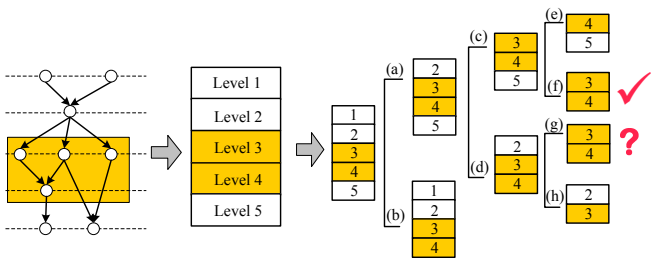

Fig. 4. Illustration for the peeling process.

Definition 2: Let u and v be two peeling nodes for the pattern G. If Dele(G, u) is enumerated earlier than Dele(G, v), the priority of u will be higher than v. This order is named as the local priority.

The earlier a peeling node is deleted, the higher its priority will be. As the example shown in Fig. 5, which presents the spanning tree for the DFG in Fig. 1. In this tree, each valid tree node is distributed with a number, and this number means the tracking order. This tree is partitioned into many levels, and the tree nodes on the same level have the same length. For example, the lengths for the nodes (3) and (4) are both 4. Here, the path in the spanning tree means the one from the root to the leaf.

Theorem 3: The local priority has three characters: 1) the partial order; 2) the local effect; 3) the genetic.

Proof: 1) Since the priority is decided by the enumeration order, $priority(a) > priority(b) \rightarrow time(a) < time(b)$. Then, if $priority(a) > priority(b) \wedge priority(b) > priority(c)$, we can get $time(a) < time(b) < time(c)$ and $priority(a) > priority(b) > priority(c)$. As shown in Fig. 5, the order of the peeling nodes are "$V1 \rightarrow V2 \rightarrow V4 \rightarrow V5$" on the level 1, so $priority(V1) > priority(V2) > priority(V4) > priority(V5)$. 2) Since the priority order focuses on the current pattern, it only appears between the peeling nodes of the same pattern. In the spanning tree, this is represented that the priority only appears between the brother nodes with the same parent. As shown in Fig. 5, the priority order exists between the tree nodes (5), (6) and (7), because they have the same parent (1). 3) The genetic character means that $priority(u_1)$ will be higher than any successor nodes of u_2 if $priority(u_1) > priority(u_2)$. For the node x in the spanning tree, its subtree means deleting other nodes after deleting x. If $priority(y) > priority(x)$, the subtree of y will be enumerated earlier, and y will be deleted earlier than x. If y is in the subtree of x, it means that y will be deleted later than x, which will generate the same enumeration. As shown in Fig. 5, since $priority(V1) > priority(V2) > priority(V4)$ in the subtree of the tree node (15), the candidate $\{V3, V5\}$ will be deleted. Therefore, the priority is genetic.

Theorem 4: The pruning based on the local priority can avoid repeated enumerations exhaustively.

Proof: By contradiction, let u and v be two nodes with the same value in the spanning tree. Since different levels have different lengths, u and v must belong to the same level in the spanning tree. We can refer to the example in Fig. 5. There are two relations with the positions of u and v: 1) u and v have the same parent; 2) u and v have different parents. For the case 1), based on Definition 2 the repeated enumeration does not exist; for the case 2), although u and v have different parents, they must have the same predecessor p. If the values of u and v are the same, the paths "$p \rightarrow u$" and "$p \rightarrow v$" must be the same. The only difference between the two paths is the erasing order. If u' is deleted earlier than v' on the first path, v' will appear earlier than u' on another path. Based on Definition 2 and Theorem 3, the second path should be deleted. As a consequence, the repeated enumeration disappears. As shown in Fig. 5, there are two paths J and K with the same value at level 3. The path y will be deleted due to the $V1$'s high priority. This is just the genetic character of the local priority.

IV. Experiment

To verify the feasibility of the peeling algorithm, we have implemented it in C++ on a Linux machine. This machine has an Intel Xeon 3-GHz CPU and 4-GB memory, and the OS is Red Hat Enterprise Linux AS release 3. The implemented programs were compiled using GCC 2.96 with option -03.

The MESA benchmarks from MidiaBench [7] are used as the inputs. Table I presents the functions of MESA. The first and second columns denote the IDs and names of the DFGs, and the next two columns present the numbers of their nodes and edges. Those DFGs were represented with DOT format, therefore, a DOT compiler was implemented to identify and import those graphs. The back-end focused on the identification time, which was measured using the standard C++ function $clock()$ and the macro CLOCKS_PER_SEC. We will compare the peeling algorithm with [6], which has achieved the same exhaustive enumeration. Since it used the I/O constraints for pruning, such constraints were removed for an equitable comparison. The algorithm in [6] ("CH") is used to compare with the peeling algorithm ("PE").

In Table I the total number of valid patterns are summarized in the fifth column. To compare the efficiency of those algorithms, the sixth and seventh columns present the identification time and the speedup respectively. The precision floating-point is three. Therefore, if the identification time is 0.000s, it means

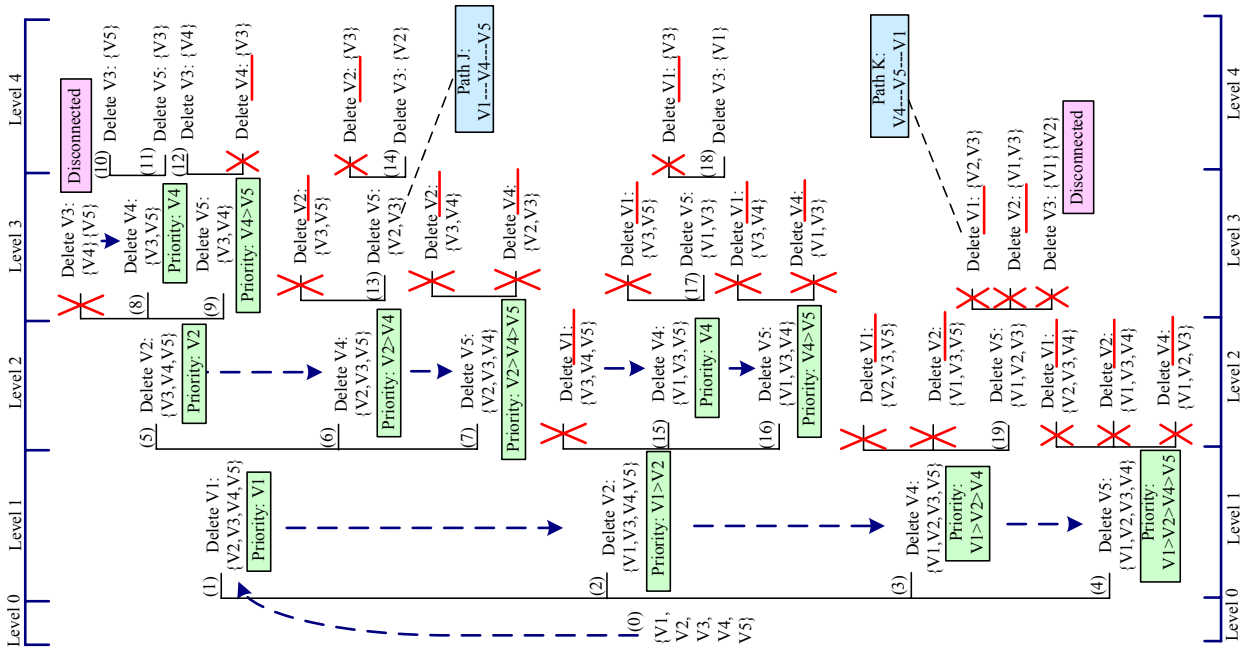

Fig. 5. The enumeration and pruning details for the example in Fig. 1.

TABLE I
EXPERIMENTAL RESULTS AND COMPARISON.

ID	MESA Functions	Node	Edge	pattern	Identification time (s)		Speedup (×)
					CH	PE	PE/CH
1	Horner Bezier	18	16	34	0.010	0.000	11.11
2	Feedback Points	53	50	114	0.030	0.000	33.33
3	Invert Matrix	333	354	9786	122.870	1.980	62.06
4	Smooth Triangle	197	196	329	0.840	0.010	84.00
5	Matrix Multiplication	109	116	625	0.590	0.040	14.75

that the time is too short to represent, and we will use the value of 0.0009 to calculate the speedup. Table I indicates that PE is faster than CH; however, the speedup effect is not super. The reason is that they both used the predecessor and the successor to satisfy the convexity constraint. However, CH starts the exploration with an empty set, and grows the pattern through combination. Instead, PE starts with a maximal valid pattern, and gets the smaller patterns through partitioning. The start of CH is arbitrary, which may bring a larger search space for the identification; however, the start of PE is easier because the maximal valid pattern is fixed. Therefore, PE can achieve a faster identification speed than CH.

V. CONCLUSION

This paper proposes a peeling algorithm for the custom instruction identification. This algorithm proposes a local priority for an exhaustive pruning. The experiments indicate that the peeling algorithm can speedup the identification and can identify the custom instructions quickly.

ACKNOWLEDGMENT

This work was supported by National Natural Science Foundation of China under grant NSFC-90207017, NSFC-90607001 and NSFC-60876030; National Basic Research Program of China (973) under grant 2005CB321605; National Postdoctoral Sustentation Fund under grant 023250010.

REFERENCES

[1] F. Sun, and et al, "Application specific heterogeneous multiprocessor synthesis using extensible processors," *IEEE Trans. CAD*, pp.1589-1602, vol. 25, No. 9, 2006.
[2] Y. Pan and T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors", *Proc. CASES*, Washington DC, pp. 69-78, 2004.
[3] N. Clark, H. Zhong and S. Mahlke, "Automated Custom Instruction Generation for Domain-Specific Processor Acceleration", *IEEE Transactions on Computers*, Vol. 54, No. 10, pp. 1258-1270, 2005.
[4] C. Wolinski and et al, "Identification of application specific instructions based on sub-graph isomorphism constraints", *Proc. ASAP*, pp. 328-333, 2007.
[5] L. Pozzi and et al. "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Trans. CAD*, vol. 25, No. 7, 2006.
[6] X. Chen, et al, "Fast identification of custom instructions for extensible processors", *IEEE Trans. CAD*, Vol. 26, No. 2, pp. 359-368, 2007.
[7] http://express.ece.ucsb.edu/benchmark/