A Fast Custom Instructions Identification Algorithm based on Basic Convex Pattern Model for Supporting ASIP Automated Design

Kang Zhao, Jinian Bian, Sheqin Dong*
Dept. of Computer Science and Technology, Tsinghua University, Beijing, P.R. China
zhao-k04@mails.tsinghua.edu.cn; bianjn@tsinghua.edu.cn; dongsq@tsinghua.edu.cn

Abstract

To improve the computation efficiency of application specific instruction-set processor (ASIP), a strategy of hardware/software collaborative design is usually utilized. In this process, the auto-customization of instruction set has always been a key part to support the automated design of ASIP. The key issue of this problem is how to effectively reduce the huge exponential exploration space in the instruction identification process. To address this issue, we first formulate it as a sub-graph enumeration problem under multi-constraints, and then propose a fast instruction identification algorithm based on basic convex pattern (BCP) model. The kernel technique in this algorithm is the transformation from the graph exploration to the formula-based computations. Experimental results have indicated that the proposed algorithm has a distinct reduction on the execution time.

Keywords: BCP, Instruction identification, CSCW design, Application Specific Instruction-set Processor.

1. Introduction

Application Specific Instruction-set Processor (ASIP) is a processor designed for a set of particular applications, which provides a good tradeoff between efficiency and flexibility [1]. In response to the challenges of high efficiency and time-to-market pressure, customizable ASIP has been frequently investigated and become an attractive technology to solve these issues.

To improve the computation efficiency of micro-processors, application program is usually implemented through a hardware/software collaborative design: if a complex operation is utilized frequently in the program, specific instruction will be customized to speed up the computation, i.e. through hardware; if not, it will be compiled with the basic instruction-set, i.e. through software. In this process the specific instruction-set customization plays a very important role, since it can greatly affect the performance of ASIP. The basic strategy to achieve instruction automated design is to combine basic operations in the data flow graph (DFG), which can be formulated as a sub-graph selection problem. To implement this strategy, two important problems must be solved: (1) custom instruction identification, which enumerates all feasible sub-graphs from the DFG extracted from the program; (2) instruction selection, which selects an optimal instruction set under various constraints [2]. In this paper, we only focus on the first problem, i.e. custom instruction identification.

Recently, many academic researches have addressed this issue. However, it is still far from the exploration space in custom instruction identification to be effectively solved. Kubilay Atasu [3] first presents an algorithm based on a BDT (binary decision tree) model. This algorithm utilizes a pruning strategy to reduce the design space, and thus the automated instruction identification can be facilitated. A similar approach is introduced in [2]. It reduces the exploration space by defining the constraint of invalid nodes and enumerating patterns based on the operations on upward and downward cons. Also, some other researchers deal with intractability by using very strict constraints [4], and propose heuristic algorithms to limit the search space [5, 6]. Besides, the strategy that combines instruction identification with selection is also investigated in [7]. However, those algorithms cannot enumerate all feasible sub-graphs, and the exponential space have not been reduced drastically yet.

There are two intrinsic limitations for these proposed techniques. First, the exploration space is exponential and cannot be reduced easily. Second, the running time will be beyond endurance when the size of problem is large.

To solve this problem, we propose a fast algorithm based on a novel model BCP (*Basic Convex Pattern*). First, feasible BCPs are determined by constructing a multi-stage graph, which can effectively reduce the complexity. Then, feasible BCPs are extracted through adjacency matrix multiplication between adjoining stages. Finally, all valid sub-graphs are enumerated using the combination and partition operations on BCPs.

* Work supported in part by National Natural Science Foundation of China under grand NSFC-90207017, and NSFC-60236020; National Basic Research Program of China (973) under grand 2005CB321605.

1-4244-0963-2/07/$25.00 ©2007 IEEE.

The rest of the paper is organized as follows. In Section 2, the formulation of custom instruction identification is first introduced. Then in Section 3, the previous BDT-based algorithm for custom instruction identification is described and compared. In Section 4, our algorithm is then illustrated as well as how it solves the enumeration problem. Finally experimental results are shown in Section 5 to support our algorithm. Our conclusion and future works are presented in Section 6.

2. Problem Formulation

As discussed, the custom instruction identification process can be formulated as a sub-graph enumeration problem under multi constraints. To clearly illustrate the formulation, we first present several preliminary definitions:

Definition 1: G (V, E) is a directed acyclic graph (DAG), where V is the set of nodes which denote basic operations in the program, and E is the set of arcs which represents the data dependency relations between operations.

Definition 2: Let $G'(V', E')$ be a sub-graph of $G(V, E)$. For $\forall e$ ($e \notin E' \wedge$ e connects with V'), the number of the incoming arcs is input degree, and the number of outgoing arcs is output degree.

Definition 3: Let $G'(V', E')$ be a sub-graph of $G(V, E)$. For $\forall v_1, v_2 \in V'$, if all the nodes on the paths between v_1 and v_2 are contained in V', G' is convex; otherwise, G' is non-convex.

For example, Figure 1 shows two different sub-graphs of the same graph, where the nodes and arcs of a sub-graph are surrounded by the broken line. The sub-graph (a) is a convex sub-graph but the sub-graph (b) is not, because the node E is on the path from node B to node F, but it is not contained in sub-graph (b).

Thus, the basic operations can be mapped to the nodes of DAG, and the candidate special instructions can be mapped to valid sub-graphs. In this way, the custom instruction identification problem can be converted into a sub-graph enumeration problem, which is specified as follows:

Problem: Given a directed acyclic graph G (V, E), find all the feasible sub-graphs that satisfy the following two conditions:

- The input and output degrees of the sub-graph cannot exceed the maximum number, since the maximum numbers of input and output operands in custom instructions are constrained.
- Only the convex sub-graph is feasible. If the candidate sub-graph is non-convex, there must be two pairs of incoming and outgoing arcs between this sub-graph and another one. This cannot help the achievement of the automated customization for the special instruction corresponding to this sub-graph.

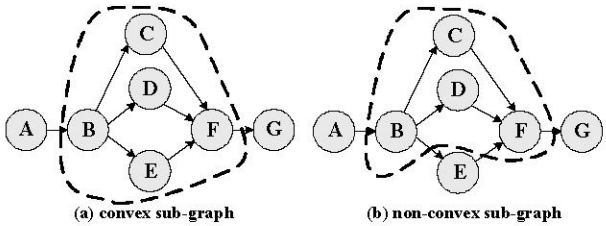

Fig. 1. Definition of convex and non-convex sub-graphs.

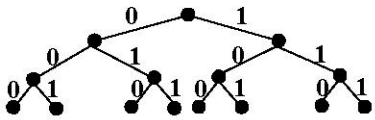

Fig. 2. An example of the binary decision tree.

3. Previous Algorithm

The custom instruction identification problem has attracted lots of attention in recent years. Some promising algorithms are correspondingly proposed. The most typical algorithm is introduced in [3], which utilizes a binary decision tree (BDT) and transforms the sub-graph enumeration problem into a BDT path selection problem.

In this approach, DFG is first converted into BDT. For each node in DFG, it is mapped to a certain level in BDT. Each binary branch in BDT denotes whether the target node is selected or not when generating sub-graphs. As depicted in Figure 2, this BDT has three levels corresponding to three nodes in DFG. For each branch, 1 represents that this node is contained and 0 represents not. Therefore, every path from top to bottom in BDT is one selection for a candidate sub-graph. Obviously the exploration space is exponential, since there will be 2^n paths in BDT if the number of nodes in DFG is n. To reduce the exploration space, it utilizes a branch-and-bound strategy. In the BDT generation process, the nodes are first arranged with reverse topological sorted order. Thus, if one path in BDT violates the two conditions presented in section II, the exploration on the following sub-tree is stopped.

The motivation of this algorithm is to enumerate all the candidate solutions with BDT format, and then select the feasible ones with the pruning strategy. However, the space is exponential and this intrinsic limitation will lead to unendurable runtime performance when the graph size is large. To speed up the calculation, we present a fast algorithm based on a novel model BCP.

4. Our Fast Algorithm

To reduce the exponential exploration space of the instruction identification problem, we first define a novel model Basic Convex Pattern (BCP), and then present further details about the main flow based on this model.

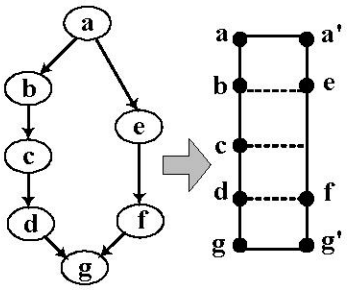

Fig. 3. BCP model definition.

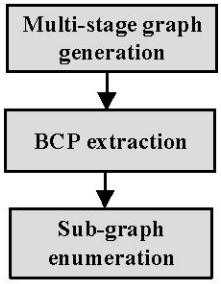

Fig. 4. Main flow

4.1. BCP Model

Definition 4: Let $G(V, E)$ be a DAG. For $\forall v_1, v_2 \in V$, if the number of paths between v_1 and v_2 is bigger than one, the sub-graph is in the basic convex pattern that is composed of the nodes on all the paths between v_1 and v_2.

From this definition it is obvious that each BCP is a convex sub-graph. This feature can be used to find all the feasible sub-graphs through combining and partitioning operations on BCP. The BCP model is presented to satisfy the conditions first in a local region, which may reduce the exploration space.

To emphasize the hierarchical feature, we abstract BCP into a multi-level rectangle format for convenient analysis, as shown in Figure 3. All the nodes in DFG are mapped to the points in the rectangle. In addition, the arcs between different levels are mapped to the arcs in the DFG and their directions are all downwards. However, if two points at the same level are connected, then these two points will correspond to the same node in DFG. For example, points *a* and *a'* are mapped onto the same node *a* in original DFG. The rectangle specification is used to obtain a regular form, which can bring a bright sight in the analysis on the BCP partition and combination processes.

4.2. Main Flow

The strategy of our method is to find all the feasible BCPs first, and then enumerate feasible sub-graphs through BCP combining and partitioning operations.

Figure 4 shows the main design flow. To reduce the space of BCP searching, we first convert the ordinary DAG into a pre-defined multi-stage graph format. Then, through calculating the connected relation matrices between each two stages in the multi-stage graph, feasible BCPs are extracted out. Finally, feasible sub-graphs are enumerated through BCP partition and combination operations. The advantage of this flow is that it can reduce the execution time in three sub-processes. These algorithms will be presented in detail in the following sub-sections.

4.3. First Phase: Multi-stage Graph Generation

Definition 5: Let $G(V, E)$ be a DAG. If G satisfies the three conditions below, it is a multi-stage graph:

- Graph G can be partitioned into m stages ($m>1$) and the node number of each stage is more than 0.
- If two nodes belong to the same stage, there will be no arcs between them.
- Each arc only exists between neighboring stages. Separate stages have no arcs between them.

The motivation of converting G to multi-stage graph is to reduce the complexity. To find out all BCPs, the paths between each two nodes in G must be determined. Further, if the adjacency matrix $A_{n\times n}$ is used, $(A_{n\times n})^n$ should be calculated out to determine the paths between each two nodes. And its complexity is $n \cdot O(n^3) = O(n^4)$. However, if multi-stage graph is utilized, there is no need to calculate the paths between each two nodes; instead, we can calculate each adjacency matrix from a low stage to another high stage. Its complexity is:

$$((m-1)+(m-2)+\cdots+1)\times O\left(\left(\frac{n}{m}\right)^3\right) = O\left(n^3\left(\frac{1}{2m}-\frac{1}{2m^2}\right)\right)$$

where m denotes the graph's stage number.

In the generation process of multi-stage graph, new nodes should be added, such as the black nodes shown in Figure 5. And we name those nodes as rubbish node. For example, there is an arc between node *A* and node *C* in Figure 5, but they cannot be connected because they belong to detached stages. So there must be one rubbish node which belongs to the same stage with node *B*.

The multi-stage graph generation procedure is described as follows:

Procedure Multi_Stage_Graph (Graph)
  Find the topological sorted order (*Depth-First-Search*);
  For each node *v* **Do**
    stage(*v*)=0; //Initialize stage value of each node *v*
  End For;
  For each node *v* in topological sorted order **Do**
    S ⇐ *v*; // *S* is a temporary node set
    For each node *p*∈(*V*-*S*) **Do**
      stage(*p*) ⇐ max{stage(*p*), stage(*p*)+ d(*v*, *p*)};
        //*d* represents the adjacency relation, 0 or 1
    End For;
  End For; // Determine the stages of all nodes

  For each arc *e* in graph *G* **Do**
    IF *e* spans two detached stages **Then**
      Calculate minus stages *minus_stage*;
      Add *minus_stage*-1 rubbish nodes;
    End IF;
  End For; // Add necessary rubbish nodes

In procedure Multi_Stage_Graph, it first arranges the nodes with a topological sorted order, and its complexity is $O(|E|+|V|)$; then the stage value for each node is determined through an $O(|E|log|V|)$ algorithm similar to Dijkstra algorithm. Finally the rubbish nodes

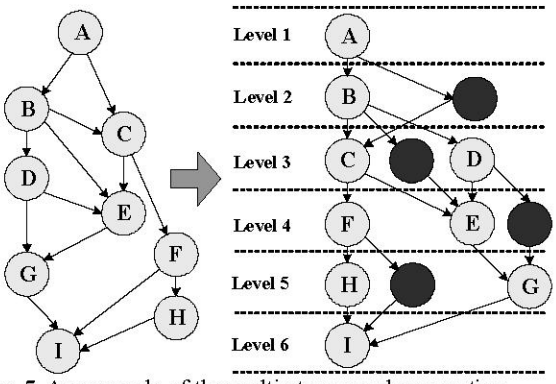

Fig. 5. An example of the multi-stage graph generation.

are added to the graph. The complexity of the multi-stage graph generation algorithm is polynomial.

4.4. Second Phase: BCP Extraction

After the multi-stage graph generation, it comes to the BCP extraction. As explained above, using multi-stage graph can reduce the complexity from $O(n^4)$ to $O(n^3)$. To extract feasible BCP, the paths between two target nodes must be determined, so the adjacency matrix between two stages is needed.

If there are three sets of nodes A, B and C:

$$A=\{a_1,a_2\}\quad B=\{b_1,b_2,b_3\}\quad C=\{c_1,c_2\}$$

Their adjacency matrices are represented with M. For example M_{AB} stands for the path numbers between nodes in set A and B. It is straightforward to get the following result:

$$\because M_{AB}=\begin{bmatrix} ab_{11} & ab_{12} & ab_{13} \\ ab_{21} & ab_{22} & ab_{23} \end{bmatrix}\quad M_{BC}=\begin{bmatrix} bc_{11} & bc_{12} \\ bc_{21} & bc_{22} \\ bc_{31} & bc_{32} \end{bmatrix}$$

$$\therefore M_{AC}=\begin{bmatrix} ac_{11} & ac_{12} \\ ac_{21} & ac_{22} \end{bmatrix}=\begin{bmatrix} \sum_{i=1}^{3} ab_{1i}\times bc_{i1} & \sum_{i=1}^{3} ab_{1i}\times bc_{i2} \\ \sum_{i=1}^{3} ab_{2i}\times bc_{i1} & \sum_{i=1}^{3} ab_{2i}\times bc_{i2} \end{bmatrix}$$

$$\therefore M_{AC}=M_{AB}\times M_{BC}$$

So the path numbers can be calculated using matrix multiplication. Furthermore, the nodes on paths can also be tracked along the opposite direction of matrix multiplication. To reduce the amount of calculation, we can operate from high levels to low levels so that the values that are already worked out can be utilized. Take Figure 5 for example, M_{56} and M_{45} are calculated first, and then M_{46} can be got using the value of M_{56} and M_{45} with the following procedure:

Procedure Matrix(*Multi_stage_graph*)
For *i* **from** *stage_num*-1 **downto** 1 **Do**
  For *j* **from** 1 **to** *stage_num*-i **Do**
    //variable *stage_num* represents the number of stage
    IF value of *j* is 1 **Then**
      M(*i*, *i*+*j*) ⇐ {0, 1} ;
      // values of matrix M(*i*, *i*+*j*) are 0 or 1
      //according to the connected relations straight
    ELSE
      M(*i*, *i*+*j*) ⇐ M(*i*, *i*+*j*-1)*M(*i*+*j*-1, *i*+*j*);
    End IF;
  End For;
End For;

According to the adjacency matrices, BCPs can be extracted. If there are more than two paths between the two target nodes and the sub-graph also satisfies the operand number limitation, it is a feasible BCP.

4.5. Third Phase: Sub-graph Enumeration

Each BCP is a feasible sub-graph, but there are still many valid sub-graphs which are not BCP. For example, in Figure 5 A, B, C and D compose a feasible sub-graph, but unfortunately it is not a BCP. So besides BCPs, there are still many sub-graphs need to be found out.

However, there has been no need to explore the entire graph. Since the useful information of BCPs has been extracted, we only need to find out the hidden mathematical rules and calculate the feasible sub-graphs instead of searching. This is the key technique that reduces the execution time of program.

We have summarized two cases that should to be considered:

(1) BCP partition: For each BCP, if one of its subsets does not contain both the source and sink nodes, it can turn out to be a feasible sub-graph. So our main task is to partition it into several regions and analyze the number of these candidate subsets. This sub-process involves several sophisticated implementations of combinatorics related techniques. Suppose a BCP has p paths, and there are q_i nodes on path i, as shown in Figure 6. In this figure, the points surrounded by circles will be selected into the candidate subset.

For simplification we suppose that the selected nodes must be connected. So when the source and sink nodes are neither selected, the number of the candidate subsets will be:

$$\sum_{i=1}^{p}\left(C_{q_i-2}^{1}+C_{q_i-2}^{1}+\cdots+C_{q_i-2}^{q_i-2}\right)=\sum_{i=1}^{p}(2^{q_i-2}-1)$$

And when only one node is selected from source or sink nodes, the subset number is:

$$(q_1-2+1)(q_2-2+1)\cdots(q_p-2+1)=\prod_{i=1}^{p}(q_i-1)$$

On the beginning part of the formula above, adding 1 in each bracket is out of the consideration that none of nodes on this path is selected. Therefore, the total number of candidate subsets for this BCP is

$$\sum_{i=1}^{p}(2^{q_i-2}-1)+\left(2\prod_{i=1}^{p}(q_i-1)-1\right)$$

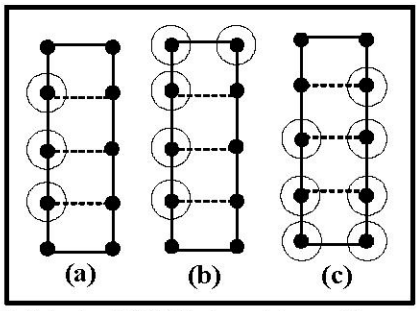

Fig. 6. (a) Subset of BCP that contains neither source node nor sink node; (b) Subset that only contains source node; (c) Subset that only contains sink node.

(2) BCP combination: If two BCPs are combined together, there may be more than one source node or sink node, but it still can turn out to be a feasible sub-graph, as shown in Figure 7. So we should combine the connected BCPs and check the result out. In Figure 8, the points surrounded by circles represent the selected nodes. In two BCPs' combination, there must be one public node selected during enumeration. If none public node is selected, there will be only one subset of the small BCP and the enumeration will be repeated.

Without loss of universality, Let the path numbers of PCB A and B be P_A and P_B, the node amount of each path be q_i^a and q_j^b respectively ($1 \le i \le P_A$, $1 \le j \le P_B$), and the public nodes belong to both path k_A and k_B. We assume that $q_{k_A}^a \ge q_{k_B}^b$, and the number of feasible sub-graphs is:

$$Result = Num(A) \times 2(q_{k_B}^b - 1) \times Num(B) + Subset(A)$$

where $Num(A)$ represents the result of region A without the public path k_A, and $Num(B)$ is for region B without k_B. For the public path, if its source and sink nodes are neither selected, it results in repeated account. Therefore, there must be two nodes selected: one from source nodes on the public path and the other from sink nodes. Also, $2(q_{k_B}^b - 1)$ is just the number count for this region. In addition, if the source and sink nodes of the public path are both selected, all nodes of BCP B should be selected and it will be equal to pure subset of BCP A. The pure subset is then mapped to $Subset(A)$ in the formula above.

For the sake of simplicity, we assume the other nodes on path k_A lean to the selected public nodes and must be all pitched on, such as the node a' in Figure 7(a). Then considering the connectivity condition, we have

$$Num(A) = \prod_{1 \le i \le P_A \text{ and } i \ne k_A} \left(q_i^a - 1\right)$$

$$Num(B) = \prod_{1 \le i \le P_B \text{ and } j \ne k_B} \left(q_j^b - 1\right)$$

Subtracting one in the formula is out of the consideration both source node and sink node in this path are selected.

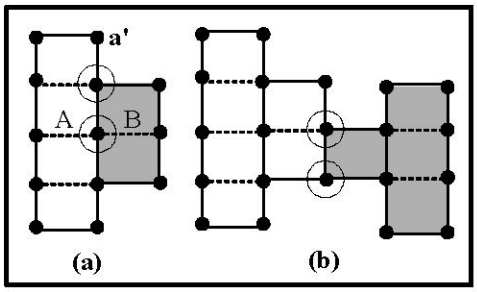

Fig. 7. (a) An example of simple BCPs combination; (b) an example of complex BCPs combination.

$$\because Subset(A) = \sum_{i=1}^{P_A} (2^{q_i - 2} - 1) + \left(2 \prod_{i=1}^{P_A} (q_i - 1) - 1\right)$$

$$\therefore Result = 2(q_{k_B}^b - 1) \times \prod_{1 \le i \le P_A \text{ and } i \ne k_A} \left(q_i^a - 1\right) \times \prod_{1 \le i \le P_B \text{ and } j \ne k_B} \left(q_j^b - 1\right) + \sum_{i=1}^{P_A} (2^{q_i - 2} - 1) + \left(2 \prod_{i=1}^{P_A} (q_i - 1) - 1\right)$$

When the target BCPs are complex, as shown in Figure 7(b), we can first partition each BCP into smaller ones and take calculations on the basis of those smaller BCPs.

5. Experimental Results and Analysis

5.1. Experimental Results

To evaluate our BCP based algorithm, we implement the algorithm with C++ and then take some tests on a v880 machine running Sun Solaris. As we want to verify it with graphs of various sizes, the input graph in our experiment is random generated by TGFF [8]. Although randomly generated, the graph still can reflect the data dependency features of programs and satisfy our essential requirements, because it can generate all types of DAGs in the experiment.

To examine the execution time of the algorithm, we take experiments with various node numbers from 10 to 200. Finally the time consumed is recorded in Table I. For comparison, we also implement the algorithm proposed in [3] under the TGFF input, and then compare its running time with our algorithm. In Table I the first column shows the DAG examples with different node numbers. Then the second and third columns present the numbers of BCP and feasible sub-graphs. The fourth and fifth columns compare the running time of previous algorithm with our proposed algorithm. The last column provides the speedup in execution time. In this experiment, the maximum input and output degrees are assumed to be 3 and 2. From the experimental results we can see that the proposed algorithm has achieved a distinct reduction on the execution time. And the larger the graph size is, the better the speedup effect can be achieved.

5.2. Analysis

There are two main reasons why our proposed algorithm reduces the execution time:

(1) BCP exploration is utilized instead of sub-graph direct exploration. BCP is more regular than sub-graph, and can further be found out by an intermediate model of the multi-stage graph that takes a series of polynomial operations. This can effectively reduce the calculation quantity. Its complicacy is $O(n^4+n^2)$.

(2) Formulas are fully utilized when we perform BCP partitioning and combination, and the complicity in the algorithm is O(1). Since most helpful information has been extracted after BCPs' generation, we can directly operate on these BCPs instead of exploring in the graph.

In addition, its integrality and validity can also be ensured in two aspects:

(1) In BCP extraction sub-process, the adjacency matrices between each two nodes on different stages are considered, which ensures the found BCPs are integrate.

(2) In sub-graph enumeration sub-process, most feasible sub-graphs are found through BCP partition and combination operations. The advantage of BCP is that it satisfies the convex condition directly, which is just the kernel factor to reduce the execution time.

Finally, based on the experimental results, we illustrate the trend on execution rate over graph size in Figure 8. From figure 8 we can find that in the worst case the exploration space is also exponential; however, with the proposed algorithm the execution time has been reduced distinctly.

6. Conclusion and Future Work

In this paper, instruction identification is first formulated as a sub-graph enumeration problem under multi constraints. Then, to reduce the execution time a fast algorithm based on a novel BCP model is presented.

TABLE I
Experimental results on different sizes of graph

Node num.	BCP num	Sub-graph num.	Running time (sec)		Speedup (×)
			previous	novel	
9	9	67	0.03	<0.01	>3
10	15	92	0.08	<0.01	>8
11	20	204	0.26	<0.01	>26
12	25	385	0.58	0.01	58
13	30	287	0.73	0.01	73
14	39	258	0.94	0.02	47
15	30	171	1.01	0.02	50.5
16	27	130	1.59	0.03	53
17	53	982	14.02	0.04	351
18	53	982	14.00	0.04	350
19	53	982	14.10	0.04	352.5
20	69	2493	50.89	0.05	1017.8

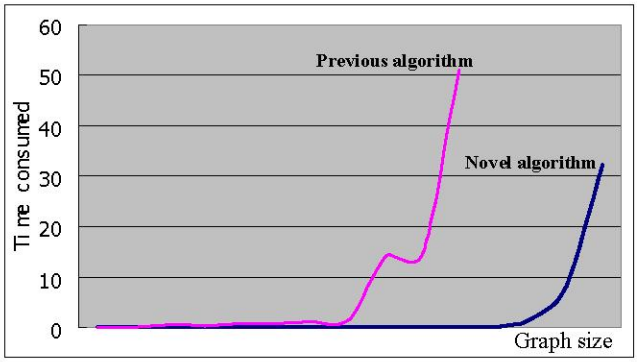

Fig. 8. Trend of execution time over graph size.

This algorithm consists of three phases: multi-stage graph generation, BCP extraction and sub-graph enumeration. Finally, through an experiment, the feasibility of the novel algorithm is verified and the execution time is reduced in a wide scope.

In the future our work will focus on the second sub-problem, i.e. custom instruction selection, which deals with the selection of an optimized instruction set under multi metrics, such as power dissipation and execution time.

References

[1] Manoj Kumar Jain, M. Balakrishnan, Anshul Kumar, "ASIP Design Methodologies: Survey and Issues", *Proceedings of the 14th International Conference on VLSI Design (VLSID '01)*, 2001.

[2] Pan Yu, Tulika Mitra, "Scalable Custom Instructions Identification for Instruction-Set Extensible Processors", *Proceedings of the Conference on Compiler, Architectures and Synthesis for Embedded Systems (CASES)*, 2004.

[3] K. Atasu, L. Pozzi and P. Ienne, "Automatic application specic instruction-set extensions under microarchitectural constraints", *Design Automation Conference (DAC)*, 2003.

[4] D. Goodwin and D. Petkov, "Automatic generation of application specific processors", *Proceedings of conference on Compilers, architectures and synthesis for embedded systems*, pp. 137-147, 2003.

[5] M. Arnold, "Instruction set extensions for embedded processors", *PhD thesis*, Delft University of Technology, 2001.

[6] F. Sun, S. Ravi, A. Raghunathan, and NK Jha, "Synthesis of custom processors based on extensible platforms", *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2002.

[7] N. Clark, H. Zhong, S. Mahlke, "Processor acceleration through automated instruction set customization", *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 129-140, 2003.

[8] R. P. Dick, D. R. Rhodes, W. H. Wolf, "TGFF: Task Graphs for Free", *Proceedings of CODES*, pp.97-101, 1998.