Instruction-level Hardware/Software Partition through DFG Exploration

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

Abstract— **To reduce the huge search space when customizing instruction-level accelerators for the application specific instruction-set processor (ASIP), this paper proposes an automated instruction-level hardware/software partition method based on the data flow graph exploration. This method integrates the instruction identification and selection using an iterative improvement strategy. The search space is reduced via considering the performance factors during the identification.**

Keywords-Hardware/software partition, ASIP, DFG exploration

I. INTRODUCTION

The application specific instruction-set processor (ASIP) can increase the processor performance through the specific instruction sets customization, which can be viewed as the special hardware in the processor [1]. Except for the instruction set architecture (ISA), ASIP also allows designers to customize the underlying microarchitecture for a specific domain.

To improve the efficiency, application programs are usually implemented through a hardware/software partition: if a complex operation is used frequently in the program, custom instructions will be extended to speedup the computation; if not, it will be compiled with the primitive instructions. In this process, the instruction set extension (ISE) plays a very important role. The basic strategy is to combine the primitive operations in the data flow graph (DFG). To implement it, two problems should be solved: 1) instruction identification, which enumerates all feasible subgraphs from the program's DFG; 2) instruction selection, which selects an optimal instruction set under various constraints [2]. In the traditional flow, candidate instructions are first enumerated via the identification process, and then the instruction subset which satisfies the performance constraints will be selected via the selection process. In this flow the identification guarantees the architectural constraints, and the selection guarantees the system constraints.

However, there are some limitations. The flow is divided into two processes, which may bring much redundant work. For example, the first process does not consider the performance constraints; instead, it only considers the architecture constraints. After a full enumeration, the second process must deal with huge quantities of candidates. If the system constraints can be considered in the identification, there will be fewer redundant candidates. Furthermore, the search space is exponential (2^n) and it is a serious problem for a full identification. If we insert the system constraints into the identification process, it may be able to reduce the search space.

To settle this issue, this paper will propose a methodology based on the graph exploration. How to enhance the efficiency and reduce the search space are two focuses. Section II will discuss the related work. In Section III, the problem will be formulated. Section IV proposes a novel methodology and the details are presented in Section V. Section VI will present the experimental results. The conclusion is drawn in Section VII.

II. RELATED WORKS

The instruction-level partitioning includes two processes: instruction identification and instruction selection.

1) The motivation of the identification is to enumerate all feasible subgraphs. It has no relation with the system performance; instead, it focuses on the architectural constraints. Atasu first proposed a breadth search method in [3], which used a full binary tree model. To reduce the search space, [3] used a pruning strategy, which can avoid enumerating invalid subgraphs. Then Pozzi [1] improved it by adding a pruning criterion based on the input number constraints. The limitation of this algorithm is that the search space was still exponential. [2] proposed an algorithm which defined the upward and downward corns, and obtained the feasible subgraphs using corns' combination. However, this method can only get segmental results compared to the exhaustive algorithms. Then successive work by Chen [4] accelerated the algorithm by reducing the invalid patterns and considering their emergence frequencies. And Atasu [5] also proposed an enumeration algorithm which only resolved the maximum convex subgraphs.

2) Instruction selection is to select a subset from the candidates to satisfy the system constraints. [6] first described a formal method based on integer linear programming (ILP) and maximize the chip performance by using a branch-and-bound algorithm. Similarly, Lee [7] also formulated the instruction selection problem with ILP and presented an effective heuristic algorithm. To reduce the huge space, many researchers proposed heuristic algorithms. In [8] Atasu resolved the problem by restricting the input and output constraints.

In conclusion, the previous works have two limitations. First, the instruction identification did not consider the system constraints and many idle candidates were enumerated, which might reduce the efficiency. Second, the exponential search space was not reduced effectively. To settle the issue, we propose a methodology which ties the identification and selection processes together.

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.

978-1-4577-0387-4/11/$26.00 ©2011 IEEE

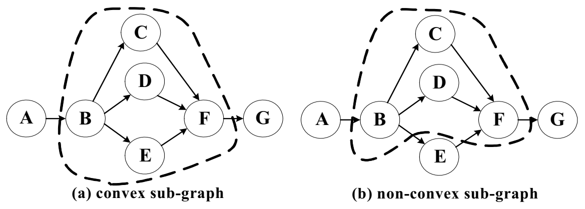

Fig. 1. Examples for the convex and non-convex subgraphs.

III. Problem Definition

A. Preliminary Definitions

To clearly explain the problem, we first present several preliminary definitions.

Definition 1: $G(V, E)$ is a directed acyclic graph (DAG), where V is the nodes which denote operations, and E is the arcs which represent the data dependencies.

Here DAG will be used to represent the data flow graph (DFG). DFGs will be the initial input for the problem. Let $G'(V', E')$ be a subgraph of $G(V, E)$. Its indegree and outdegree will be represented as $IN(G')$ and $OUT(G')$. The custom instructions are usually generated through combining primitive operations. Since the primitive operation is represented as a node in the DFG, the custom instruction will be mapped to a subgraph.

Let $G(V, E)$ be a DAG. For $v \in V$, if $IN(v)>1$, the node v is an upward node; if $OUT(v)>1$, then it is a downward node. For $v \in V$, if $IN(v) \leqslant 1$ and $OUT(v) \leqslant 1$, the node v is a SISO (single in single out) node; otherwise, it is a MIMO (multiple in multiple out) node.

Definition 2: $G'(V', E')$ is a subgraph of $G(V, E)$. If the condition is satisfied, G' is convex; otherwise, it is nonconvex:

$$\forall v_1 \in V' \wedge \forall v_2 \in V' \wedge \forall v_3 \in Path(v_1, v_2) \rightarrow v_3 \in V' \quad (1)$$

Fig. 1 presents two different subgraphs. Here the subgraphs are surrounded by the broken lines. The subgraph (a) is convex but (b) is not, because in (b) the node E is on the path from B to F, however, it is not included in the subgraph (b).

B. Problem Formulation

The instruction-set extension problem can be described as: given an application program which has a set of hot spots consuming much computing time, customize an extensible instruction set under multiple constraints so that the execution time of those hot spots can be minimized.

To resolve this problem, it is extracted based on the DFG. First, the program will be converted into the DFG, and each instruction in the assembler is mapped to the node in the DFG. Second, the candidate feasible subgraphs will be enumerated. Finally, the custom instructions will be selected out to minimize the execution time.

The feasible subgraph must satisfy three conditions. First, its indegree and outdegree cannot exceed the maximum value. For a processor, the register file has decided the operand number, so the indegree and outdegree of the feasible patterns should also satisfy this constraint. Second, the feasible subgraph must be convex. If the candidate subgraph is nonconvex, there will be a data dependency loop. This means that the hardware (custom instruction) will have data transfer relations with the software, which is not feasible. Third, the capability of candidate patterns cannot exceed the value Z_m. The motivation is to restrict the cost of synthesized accelerators.

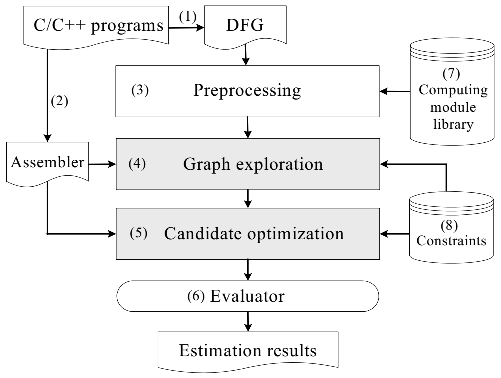

Fig. 2. The proposed method based on DFG exploration.

Problem: Given a directed acyclic graph $G(V, E)$, find out a set of feasible subgraphs $\{G'\}$ which satisfy the following conditions simultaneously, so that the total execution time T can be minimized:

- $IN(G') \leqslant in_{max}$ and $OUT(G') \leqslant out_{max}$;
- G' is convex;
- $Path(G') \leqslant Z_m$.

IV. Methodology

This section will present an overview of the proposed method. The whole flow is presented by Fig.2. It includes three main steps: preprocessing, graph exploration and candidate optimization. To present a clear explanation, Fig. 3 presents a simple demo for this workflow. Then we will present the motivations for each subprocess.

The motivation of the preprocessing is to explore the instruction-level parallelism. Its main task is to allocate the operations into a set of limited computing elements, and then schedule the instructions to reduce the total running cycles. This step will provide the initial order as the input of the following steps. For example, we assumed that there are one adder and one multiplier. So if we use the first order of Fig.3(1), the multiplier can only deal with one multiplication each time. To release the parallelism, the order is scheduled as shown in Fig. 3(2), and then the multiplier and adder elements can run simultaneously.

To extract the candidate subgraphs, a strategy based on the graph exploration will be proposed. A priority for each node will be defined, and then the subgraph grows via combining the neighbor nodes from high priority to low priority. This strategy is named as seed-growth. To reduce the search space, an intelligent guide function will be presented to decide which direction to grow. After the graph exploration, we will get a set of feasible candidates, as shown in Fig. 3(3). However, those

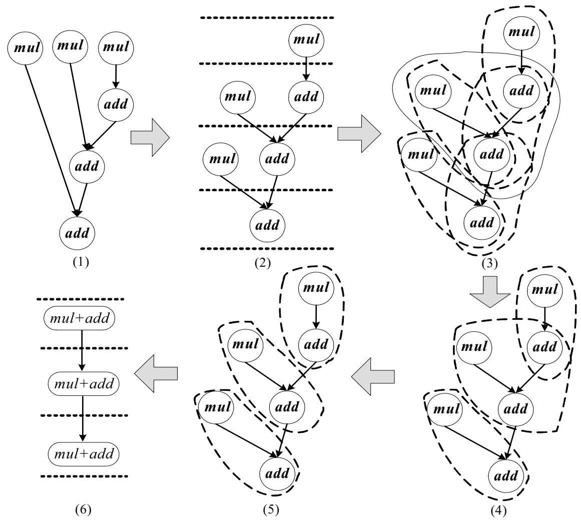

Fig. 3. An example for the work flow in Fig. 2.

candidates may intersect. Therefore, to reduce the total cost, those candidates must be optimized.

The candidate optimization focuses on graph splitting, as shown in Fig. 3(4)(5). After the graph exploration, the generated candidates may have functional overlap. For example, there are three candidates: mul_add, add_add, and add_mul_add. However, each node cannot be represented by more instructions, so a splitting process is needed.

V. Data Flow Graph Exploration

A. Preliminary

Definition 3: Let $G'(V', E')$ be a subgraph. P is the critical path of G', and u and v are its source and sink. If $|P|= Z_m$, this path cannot be lengthened more. Then u is called as a $stop_{in}$, and v is a $stop_{out}$:

$$u \in stop_{in} \qquad v \in stop_{out} \qquad (2)$$

Here $stop_{in}$ and $stop_{out}$ will guarantee $Path(G') \leqslant Z_m$.

Definition 4: Let $G(V, E)$ be a DAG. For each node $v \in V$, its priority value will be:

$$Priority(v) = IN(v) + OUT(v) + IN(v) \times OUT(v) \quad (3)$$

The node with higher degree is more complicated than the one with lower degrees. In this definition, the third item is used to reduce the priority of the nodes which have no incoming or outgoing arcs. For example in Fig. 4, if the node has higher in/out degree, the value of the priority will be higher: *priority*(*op*5)>*priority*(*op*4); if the node is near to the boundary, its priority may be lower: *priority*(*op*4)>*priority*(*op*2).

B. Strategy

The strategy is to select a node as the seed and then grow through combining its neighboring nodes. However, which direction to combine is the best choice, and how to ensure that

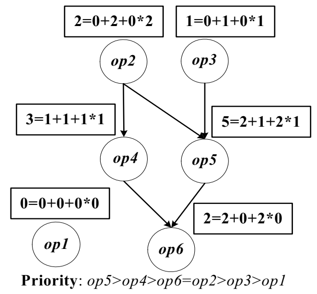

Fig. 4. An example to show the priority for each node.

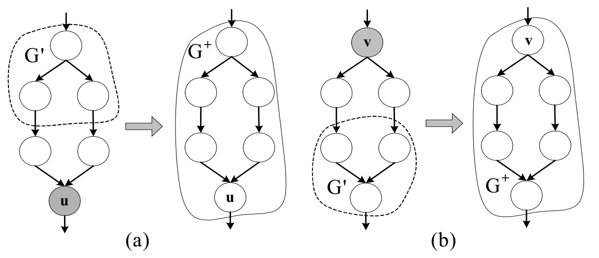

Fig. 5. The examples to explain Lemma 2. The subgraphs are surrounded by the broken lines, and the node with shadow is the focus to be found. a) $|OUT| > out_{max}$; b) $|IN| > in_{max}$.

the generated subgraphs satisfy the constraints proposed in Section III.B? This will be the key.

Capability - The capability constraint can limit the size of generated subgraphs. Since it is related with the longest path, we will use the definition of $stop_{in}$ and $stop_{out}$.

Lemma 1: Let $G(V, E)$ be a DAG, and $G'(V', E')$ be its subgraph. Assumed that $u, v \in V' \wedge u \in stop_{in} \wedge v \in stop_{out}$, if $t, r \in V \wedge t, r \notin V'$, we can have:

- If $t \in$ Father(u), t cannot be combined with G';
- If $r \in$ Son(v), r cannot be combined into G'.

Proof: Based on Definition 1, there must be a longest path from u, and its length is Z_m. Therefore, if we select a node t from the set Father(u) and combine it with G', the length of its critical path must be Z_m+1. This case will violate the constraint of Z_m, so t cannot be combined. Similarly, there must be a longest path end with v. If we contained the node v, the constraint of $Path(G') \leqslant Z_m$ will be violated.

Input/Output - The input and output degrees of the subgraphs are limited. They cannot exceed in_{max} and out_{max}. When combining subgraphs, how to deal with the subgraph which violates the maximum input/output constraint?

Lemma 2: Let $G'(V', E')$ be a subgraph of $G(V, E)$. If $|OUT(G')|>out_{max}$, we should find a node $u \in V \wedge u \notin V'$, which satisfies that u is the progeny of ($|OUT(G')|-out_{max}+|OUT(u)|$) outgoing nodes at least. If $|IN(G')|>in_{max}$, we should find a node $v \in V \wedge v \notin V'$, which satisfies that v is the ancestor of ($|IN(G')|- in_{max} +|IN(v)|$) incoming nodes at least.

Proof: If $|OUT(G')|>out_{max}$, G' will be an infeasible subgraph. So to reduce the outgoing arcs, it is a good choice to find a progeny which can eliminate unwanted arcs via

```
Find(Graph $G$, Graph $P$, Boolean $f$)
 1. if($f$ is true){
 2.    $Pro \Leftarrow \{v|v \notin P \wedge \exists u(u \in P \wedge u = ancestor(v))\}$
 3.    for($i \Leftarrow 1$; $i \leq Node_{num}(P)$; $i++$)
 4.       if($\exists j(j \in Pro \wedge i = Ancestor(j))$)
 5.          $num_j++$;
 6.    $\exists x \forall j(num_x \geq num_j \wedge j \in Pro \wedge x \in Pro)$;
 7.    return $x$;
 8. }endif
 9. else if($f$ is false){
10.    $Anc \Leftarrow \{v|v \notin P \wedge \exists u(u \in P \wedge u = progeny(v))\}$
11.    for($i \Leftarrow 1$; $i \leq Node_{num}(P)$; $i++$)
12.       if($\exists j(j \in Anc \wedge i = Progeny(j))$)
13.          $num_j++$;
14.    $\exists y \forall j(num_y \geq num_j \wedge j \in Anc \wedge y \in Anc)$;
15.    return $y$;
16. }endelse
```

Fig. 6. Lemma 2 implementation. *P* is the subgraph of *G*.

combination. We assume that the outdegree of this progeny node *u* is u_0. If combining *u*, we must also combine all the nodes on the paths from *G*' to *u*, and their outgoing arcs after combination will be *temp*. Therefore, we can get: $|OUT(G^+)|=|OUT(G')|+|OUT(u)|+|temp|-|Result| \leqslant out_{max}$. Here *Result* means that how many paths in *G*' has the same progeny *u*. Since $|temp| \leqslant 0$, it is obvious that $|Result| \geqslant |OUT(G')|+|OUT(u)|+|temp|-out_{max} \geqslant |OUT(G')|+|OUT(u)|-out_{max}$. Therefore, *u* is the progeny of $(|OUT(G')|-out_{max}+|OUT(u)|)$ outgoing nodes at least. In addition, for the case of $|IN(G')| > in_{max}$, the proof is similar. As the example shown in Fig. 5, we assumed that $in_{max}=out_{max}=1$ for simplicity. In the first figure, we should explore downwards to reduce the outdegree and find a node *u* which is the progeny of $P \subseteq G'$. This possible node may lead to a fewer output. If *u* is combined, the paths from *G*' to *u* must also be included. Also, if the current subgraph violates the input constraint as shown in the second figure, we will search upward and find a possible ancestor *v*. This will be a choice for the I/O constraints.

Fig. 6 presents the implementation of Lemma 2. If the parameter of *f* is true, it will find the proper progeny node of the subgraph *P*; if *f* is false, it will find the ancestor node.

Convex - Combining a node with the current subgraph may generate a nonconvex subgraph. So this step is to find out those hidden nodes and combine them to the current subgraph.

Lemma 3: *G*'(*V*', *E*') is a convex subgraph of *G*(*V*, *E*). Given a node $v \in V \wedge v \notin V'$, then after combining *v* with *G*', we can obtain a convex subgraph G^+, which satisfies the conditions:

- If the condition of $P \subseteq G' \wedge P = Ancestor(v)$ is satisfied, $G^+ = G' \bigcup \{v\} \bigcup (Progeny(P) \bigcap Ancestor(v) - G')$.
- If the condition of $P \subseteq G' \wedge P = Progeny(v)$ is satisfied, $G^+ = G' \bigcup \{v\} \bigcup (Ancestor(P) \bigcap Progeny(v) - G')$.

Proof: Since the two conditions in Lemma 3 are similar, we only present the proof for the first item. Here we will use the

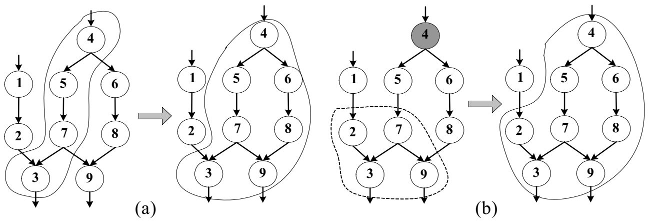

Fig. 7. How to guarantee that the combined graph is convex.

definition of convex subgraph directly. According to Definition 2, for each two nodes in the subgraph if all the paths between those two nodes are also contained in this subgraph, this subgraph will be convex. Since the final result include three items: *G*', {*v*}, and $Progeny(P) \bigcap Ancestor(v) - G'$. It is assumed that $U_m = Progeny(P) \bigcap Ancestor(v) - G'$. If *G*' are not connected directly with *u*, the subgraph $G' \cup \{v\}$ will be discrete and insignificant. Also, *G*' and {*v*} are both convex. Therefore, the task is to verify the following sets: U_m, $\{v\} \cup U_m$ and $G' \cup U_m$. 1) U_m: we assume that for $\forall a,b \in U_m$ there should be a path *Q* between the nodes *a* and *b*. If $c \in Q \wedge c \notin U_m$, $c \notin Progeny(P) \wedge c \notin (Ancestor(v) - G')$. However, the start and the end of *Q* should satisfy the condition that $a,b \in Progeny(P) \bigcap Ancestor(v) - G'$, so each node on *Q* should also satisfy this condition. Then the conflict exists. This reduction to absurdity just proves that U_m is convex. 2) $\{v\} \cup U_m$: for each node $x \in U_m$, it must be the ancestor of *v*. So this path must be contained in $\{v\} \cup U_m$, and this subgraph is convex. 3) $G' \cup U_m$: since each node in U_m is the progeny of *P*, all the paths between each two nodes in $P \cup U_m$ are also contained in this domain; for another part, we have the relation $(G' - P) \bigcap U_m = \varnothing$, so the set (*G*'-*P*) will be not considered. Therefore, it can be proved that $G' \cup U_m$ is convex. 4) $G' \cup \{v\} \cup U_m$: based on the analysis above, we can only prove that all paths between the node *v* and $\forall y \in G'$ are also in this domain. Since the path from *v* to *G*' must pass the node $z \in U_m$, and the subpath *v*-*z* and *z*-*G*' must be contained in the domain, so the total path should also be contained. To explain it clearly, Fig. 7 presents two examples. In the first figure, *G*'={*op*4, *op*5, *op*7, *op*3}, *P*={*op*3}, *v*=*op*9 and U_m={*op*6, *op*8}; in the second figure, *G*'={*op*2, *op*3, *op*7, *op*9}, *P*={*op*2, *op*3}, *v*=*op*4 and U_m={*op*5, *op*6, *op*8}.

Latency - When combining the neighbor nodes, there are many directions to grow, which one is the best choice? The answer is the one which can bring the most performance improvements.

Lemma 4: Given a DAG *G*(*V*, *E*) and its critical path *P*, combining the nodes on *P* could obtain higher performance improvements than others.

Lemma 5: Assumed that *A* and *B* are two connected node, and their latencies are represented as L_A and L_B. Here $L_A \geqslant L_B$. Then the subgraph *AB* is generated via combining *A* and *B*, and its latency is: $L_{AB}=L_A+L_B/k$ $(k \geqslant 1)$. When $Z_m \rightarrow 1$, $k \rightarrow \infty$.

```
Direction(Graph $Seed$, Graph $G(V,E)$)
1. $Q \Leftarrow \{v|u \in Seed \wedge v \notin Seed \wedge (v = ancestor(u) \vee v = progeny(u))\}$;
2. for(each node $v$ in $Q$){
3.     $Temp \Leftarrow$ Unit($Seed, v$);
4.     if(Stop($Temp$) $> 0$) continue;
5.     while(IN($Temp$) $> in_{max}$)
6.         $Temp \Leftarrow$Unit($Temp$, Find($G$, $Temp$, false));
7.     while(OUT($Temp$) $> out_{max}$)
8.         $Temp \Leftarrow$Unit($Temp$, Find($G$, $Temp$, true));
9.     if(Stop($Temp$) $> 0$) continue;
10.    $\triangle latency \Leftarrow$delay($Seed$)-delay($Temp$);
11.    $Fun(v) \Leftarrow \triangle latency/(dis(v)+1)$;
12. }endfor
13. Select $j$ in $Q$ with the largest value of $Fun$;
14. if($j$ does not exist) return false;
15. $Seed \Leftarrow$Unit($Seed, j$);
16. return true;
```

Fig. 8. The combination direction selection.

Proof: Each custom instruction will be implemented as a special hardware, so the latencies are related with the detailed implementation of the hardware. It is hard to propose a formulation for their latencies, so this lemma presents an approximate evaluation formula. Because the processor runs the two instructions *A* and *B* in sequence, it will be slower than the hardware implementation, i.e. $L_{AB} \leqslant L_A + L_B$. For the operation *A*, it adds another operation *B*, so it is not likely to reduce its latency than the original value, i.e. $L_{AB} \geqslant L_A$. Therefore, $L_A \leqslant L_{AB} \leqslant L_A + L_B$, and also $L_{AB} = L_A + L_B/k$ ($k \geqslant 1$). For simplicity, it is assumed that during the hardware implementation we can use any kinds of optimization method, such as parallelism and pipelining. Then the operation *B* can be fused into *A* completely, i.e. $L_{AB} = L_A$. However, $L_{AB} = L_A$ is only available when the Z_m is small enough. If $Z_m \to \infty$, the $L_{subgraph}$ will not be equal to its member with the maximum latency.

Based on the two lemmas above we propose a guide function *Fun*, which decides the best direction to combine with the seed node:

$$Fun(v) = \frac{\Delta latency(v)}{dis(v)+1} \tag{4}$$

Here $\triangle latency(v)$ means the latency reduction after combining the operation *v*, and *dis*(*v*) is the distance from the critical path. If the value of $\triangle latency(v)$ is bigger and *dis*(*v*) is smaller, the probability that *v* is combined with the current node will be higher. In this equation the item of *dis*(*v*)+1 is to prevent the case that the node *v* is just on the critical path, i.e. *dis*(*v*)=0. Therefore, when exploring candidate subgraphs we will use Eq.(4) to decide the best combination direction. If *Fun*(*v*) is bigger, its probability to be combined will be higher.

C. MIMO Node Combination

The strategy is to select a node as the seed first, and then grow it through combing the neighbor nodes. However, which node should be the seed node, and which node has the highest priority to being the seed? As mentioned in Section III.A, the nodes can be divided into two types: MIMO and SISO. The case of SISO is easier due to the single data relationship. So we will deal with the MIMO nodes in detail.

```
MIMO_d(Graph G)
1. $Candidate \Leftarrow \phi$;
2. $N \Leftarrow$all nodes in $G$ from high priority to low priority;
3. for($i \Leftarrow 1$; $i \le N_{length}$; $i++$)
4.     if($N[i]$ is not visited)
5.         $Temp \Leftarrow$Direction($\{N[i]\}, G$);
6.         All the nodes in $Temp$ are defined to be visited;
7.         $Candidate \Leftarrow Candidate \cup \{Temp\}$;
8.     }endif
9. return $Candidate$;
```

Fig. 9. The MIMO node combination.

Fig.8 and Fig.9 present the algorithms of the MIMO node combination. The function "*MIMO_d*" is used to select the seed node, and "*Direction*" is used to combine the neighbor nodes with the seed. From Fig. 9 we can see that the choice of seed nodes is scheduled from high *priority* to low *priority*, so we will deal with the node with higher indegree and outdegree first. The strategy of Fig. 8 is to scan the neighbor nodes stage by stage, and verify whether it can bring an efficient performance improvement after combining. If the generated subgraph violates the constraints, the corresponding lemmas proposed in Section V.B will be used.

We will compare the search space of the proposed algorithm with the previous exhaustive enumeration method. For the previous exhaustive algorithm, its search space is $O(2^n)$, where *n* is node number of the graph. To generate a subgraph there are two choices for each node, so its search space is 2^n. Furthermore, there are 2^n subgraphs in all, so the total search space is: $2^n \times 2^n = 2^{2n}$. For our proposed algorithm, we assume that there are (*m*-1) neighbor nodes for each seed node; therefore, to generate a subgraph it needs a search space of 2^{m-1}. Here we assume that *S*(*n*) is the search space for the graph with *n* nodes. So based on the features of the proposed algorithm, we can get the relation:

$$S(n) = \begin{cases} 0 & if\ n = 1 \\ S(n-m) + 2^{m-1} & if\ 1 \le m \le n-1 \end{cases} \tag{5}$$

Here the value of *m* is related with Z_m. If Z_m is higher, *m* will be bigger. Therefore, we can analyze two extreme cases: *m*=1 and *m*=*n*-1. When *m*=*n*-1, the final result is 2^{n-2}; when *m*=1, *S*(*n*)=*S*(*n*-1)+1=*S*(*n*-2)+2=…=*S*(1)+(*n*-2)=*n*-2. From the two extreme cases, we can see that the proposed algorithm must have a lower search space than the previous algorithms.

D. Candidate Optimization

After the data flow graph exploration, we have obtained a set of feasible candidates. However, this is not the final result, because there may be two candidates with intersection. We have implemented a greedy method to split the candidates. The details will be omitted due to the paper length.

TABLE I. BENCHMARKS IN THE EXPERIMENTS

ID	Benchmark	Node	Edge
1	Auto Regression Filter	28	30
2	MPEG:Inverse Discrete Cosine Transform	114	164
3	JPEG:Forward Discrete Cosine Transform	134	169
4	EPIC:Collapse_pyr	56	73
5	MESA:Invert Matrix	333	354

VI. EXPERIMENT

To verify the feasibility of the proposed method, we have implemented it with C++ and done some tests on a v880 machine running Sun Solaris. The front-end mainly used the benchmarks from MidiaBench [9]. Table I presents the details. The back-end used the simulator-based evaluation values. Firstly, the latency for each primitive instruction was simulated using Altera Quartus II. Those operations were implemented with VHDL and then simulated to get the running time, which will be used to evaluate the latencies of custom instructions.

We will compare the proposed algorithm (referred to as "GA") with the algorithm in [1] (referred to as "EX") and [4] (referred to as "SP"). EX is an exhaustive algorithm, which will enumerate all the valid subgraphs. Therefore, it will consume many search space. We have implemented it based on the binary decision tree (BDT) model, and used a branch-and-bound strategy to omit the patterns which violate the constraints. Since [1] and [4] did not consider the constraint of Z_m, we added it in the implementation. SP is also an exhaustive algorithm, which identified the next node for inclusion to the current subgraph. So it is similar to GA.

Table II presents the results in detail. The first column is the IDs of the benchmarks, and the following three columns present the $in_{max}/out_{max}/Z_m$ constraints. Here we compared four kinds of constraints: 2/1/2, 4/1/2, 4/2/2 and 4/2/3. Then the search spaces for the three algorithms are compared. Here the search space is the total scanned nodes to generate the current valid subgraph. Then the next two columns are the numbers of valid candidate subgraphs. EX and SP are both exhaustive algorithms, so their results are the same. GA considered the performance factor in the enumeration stage, and it will omit many valid candidates which bring less performance enhancement, so its result is less than EX and SP. Finally the last two columns present the improvements over EX and SP, which are the ratios between the search spaces. The function of EX and SP was to enumerate all valid patterns, and they did not consider any performance factors. However, GA considers the performance in the identification stage, so it can omit many candidates and reduce the search space. But the sequel is that GA will do more work than EX and SP, so it is not significant to compare their execution time directly. Instead, we will analyze their search space. From Table II we can see that for the same benchmark and constraints, the proposed algorithm will use much less search space than the previous algorithms.

VII. CONCLUSION

This paper presents an instruction level partition method based on the DFG exploration. To reduce the huge search space, this method combined both the instruction identification and the selection. The experiments indicated that the performance is comparable to the exhaustive algorithms.

TABLE II. EXPERIMENTAL RESULTS UNDER DIFFERENT CONSTRAINTS.

ID				Search Space			Candidates		Final	Improve(x)	
	in	out	Z_m	EX	SP	GA	EX&SP	GA		EX	SP
1	2	1	2	131	42	18	18	6	6	7.3	2.4
	4	1	2	236	66	30	22	6	4	7.9	2.2
	4	2	2	841	196	50	83	9	4	16.9	3.9
	4	2	3	1351	342	48	106	9	4	28.2	7.1
2	2	1	2	5277	471	201	129	30	21	26.3	2.3
	4	1	2	7153	574	273	147	41	32	26.2	2.1
	4	2	2	7802	688	297	149	42	33	26.3	2.3
	4	2	3	11599	1054	535	185	29	27	21.7	1.9
3	2	1	2	10399	495	202	139	48	27	51.5	2.5
	4	1	2	11903	562	269	148	57	27	44.2	2.1
	4	2	2	12278	719	376	191	96	33	32.7	1.9
	4	2	3	17871	1255	432	271	136	33	41.4	2.9
4	2	1	2	1169	119	48	24	19	14	24.4	2.5
	4	1	2	2211	255	86	37	30	20	25.7	3.0
	4	2	2	3398	427	139	61	54	29	24.4	3.1
	4	2	3	12142	1036	328	79	23	14	37.0	3.2
5	2	1	2	27355	5347	1265	531	114	87	21.6	4.2
	4	1	2	79961	15292	2016	616	140	67	39.7	7.6
	4	2	2	80615	15489	2064	632	156	67	39.1	7.5
	4	2	3	1993072	45377	1781	2349	220	52	1119.1	25.5

EX: [1]; SP: [4]; GA: the proposed method.

ACKNOWLEDGMENT

This work was supported by National Natural Science Foundation of China under grant NSFC-90207017, NSFC-90607001 and NSFC-60876030.

REFERENCES

[1] L. Pozzi, K. Atasu and P. Ienne. "Exact and approximate algorithms for the extension of embedded processor instruction sets," IEEE TCAD, vol. 25, No. 7, pp. 1209-29, July, 2006.

[2] Y. Pan, T. Mitra, "Scalable Custom Instructions Identification for Instruction-Set Extensible Processors", Proc. conference on Compiler, Architectures and Synthesis for embedded systems, pp. 69–78, 2004.

[3] K. Atasu, L. Pozzi and P. Ienne. "Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints," Proc. the Design Automation Conference (DAC), pp. 411–418, June 2003.

[4] X. Chen, and et al, "Fast Identification of Custom Instructions for Extensible Processors", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 26, No. 2, pp. 359–368, 2007.

[5] K. Atasu, O. Mencer, and et al. "Fast Custom Instruction Identification by Convex Subgraph Enumeration, " Proc. Conference on Application-specific Systems, Architectures and Processors, Leuven, pp. 1–6, 2008.

[6] A. Alomary, T. Nakata, and et al, "An ASIP instruction set optimization algorithm with functional module sharing constraint", IEEE/ACM international conference on Computer Aided Design, Nov. 1993.

[7] J. Lee, K. Choi, N. Dutt, "Automatic Instruction Set Design Through Efficient Instruction Encoding for Application-Specific Processors". TR 02-23, August 2003.

[8] K. Atasu, G. Dundar and C.Ozturan. "An Integer Linear Programming Approach for Identifying Instruction-Set Extensions", Proc. International CODES+ISSS, Jersey, September, 2005.

[9] http://express.ece.ucsb.edu/benchmark/