A comparison of heuristic algorithms for custom instruction selection

Shanshan Wang, Chenglong Xiao, Wanjun Liu, Emmanuel Casseau

A Comparison of Heuristic Algorithms for Custom Instruction Selection

Shanshan Wang*, Chenglong Xiao*,Wanjun Liu*,Emmanuel Casseau†
*Liaoning Technical University, China
{celine.shanshan.wang,chenglong.xiao}@gmail.com
liuwanjun39@163.com
†University of Rennes I, Inria, France
emmanuel.casseau@irisa.fr

***Abstract*—Extensible processors with custom function units (CFU) that implement parts of the application code can make good trade-off between performance and flexibility. In general, deciding profitable parts of the application source code that run on CFU involves two crucial steps: subgraph enumeration and subgraph selection. In this paper, we focus on the subgraph selection problem, which has been widely recognized as a computationally difficult problem. We have formally proved that the upper bound of the number of feasible solutions for the subgraph selection problem is $3^{n/3}$, where n is the number of subgraph candidates. We have adapted and compared five popular heuristic algorithms: simulated annealing (SA), tabu search (TS), genetic algorithm (GA), particle swarm optimization (PSO) and ant colony optimization (ACO), for the subgraph selection problem with the objective of minimising execution time under non-overlapping constraint and acyclicity constraint. The results show that the standard SA algorithm can produce the best results while taking the least amount of time among the five standard heuristics. In addition, we have introduced an adaptive local optimum searching strategy in ACO and PSO to further improve the quality of results.**

***Keywords*-Heuristic algorithms; Data-flow graph; Extensible processors; Custom instructions; Custom function units;**

I. INTRODUCTION

Due to a good trade-offs between performance and flexibility, extensible processors have been used more and more in embedded systems. Such examples include Altera NIOS processors, Xilinx MicroBlaze and ARC processors. In extensible processors, a base processor is extended with custom function units that execute custom instructions. Custom function units running the computation-intensive parts of application can be implemented with application-specific integrated-circuit or field-programmable gate-array technology. A custom instruction is composed of a cluster of primitive instructions. In general, the critical parts of an application are selected to execute on CFUs and the rest parts are run on the base processor. With the base processor, the flexibility can be guaranteed. As selected custom instructions usually impose high data-level parallelism [1], executing these custom instructions on CFUs may significantly improve the performance.

Deciding which segments of code to be executed in CFUs is a time-consuming work. Due to time-to-market pressure and requirement for lower design cost of extensible processors, automated custom instruction generation is necessary. Fig 1. shows the design flow for automatic custom instruction generation. Starting with provided C/C++ code, a front-end compiler is called to produce the corresponding control data-flow graph (CDFG). Then, the subgraph enumeration step enumerates all subgraphs (graphic representation of custom instructions) from DFGs inside CDFG that satisfy the micro-architecture constraints and user-defined constraints [2]–[5]. Next, in order to improve the performance, the subgraph selection step selects a subset of most profitable subgraphs from the set of enumerated subgraphs, while satisfying some constraints (e.g. non-overlapping constraint and area constraint). Based on the selected subgraphs, the subgraphs with equivalent structure and function are grouped together. The behavioral descriptions of the selected custom instructions along with the code incorporating the selected custom instructions are finally produced. The crucial problems involved in custom instruction generation are: subgraph enumeration and subgraph selection. In this paper, we focus on the subgraph selection problem. The main contributions of this paper are:

- formulating the subgraph selection problem as a maximum cliques problem;
- an upper bound on the number of feasible solutions is $3^{n/3}$, where n is the number of subgraph candidates;
- adaptation of five popular heuristic algorithms for the subgraph selection problem;
- detailed comparison of these five algorithms in terms of search time and quality of the solutions. Furthermore, we extend PSO and ACO algorithms to include an adaptive local optimum searching strategy. Results show that the quality of the solutions can be further improved.

The rest of the paper is organized as follows. Section 2 reviews the related work on the subgraph selection problem, while section 3 formally formulates the subgraph selection problem as a weighted maximum problem and presents an upper bound on the number of feasible solutions. Section 4 discusses the proposed SA, TS, GA, PSO and ACO algorithms for the subgraph selection problem. In section 5, the experiments with practical benchmarks are performed to evaluate the algorithms in terms of search time and quality

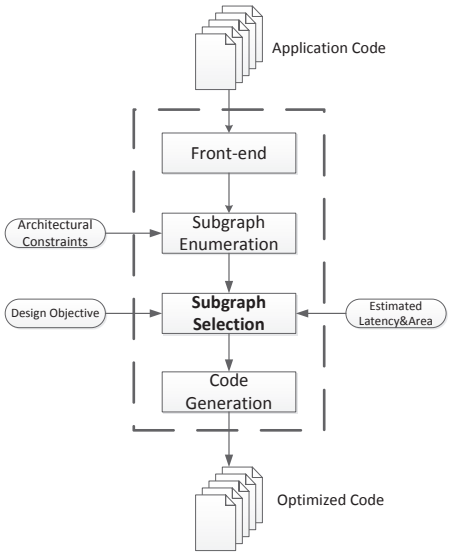

Figure 1. The design flow for custom instruction generation

of the solutions. Some discussion on the future work is given in section 6. Section 7 concludes this paper.

II. Related Work

Prior to review the related work on custom instruction selection (or subgraph selection), we start by discussing algorithms for custom instruction enumeration.

Plenty of previous research on custom instruction enumeration have been presented in recent years. The custom instruction enumeration tries to enumerate a set of subgraphs from the application graph with respect to the micro-architecture constraints or user-defined constraints. The custom instruction enumeration process is usually very time-consuming. For example, it was proven in [6] that the number of valid subgraphs satisfying the convexity and I/O constraints is n^{I+O}, where n is the number of nodes in the application graph, and I and O are the maximum number of inputs and outputs respectively. A set of algorithms with $O(n^{I+O})$ time complexity can be found in existing literature [3], [5], [7]. Some other algorithms targeted to only enumerate maximal convex subgraphs by relaxing the I/O constraints have also been proposed [8], [9]. However, the number of maximal convex subgraphs can still be exponential. In [10], the number of maximal convex subgraphs was formally proven to be $2^{|F|}$, where F is the set of forbidden nodes in the application graph. It can be observed from previous experiments that the subgraph enumeration step may produce tens of thousands of subgraphs for both enumeration of convex subgraphs under I/O constraints and enumeration of maximal convex subgraphs.

Subgraph selection is the process of selecting a subset of profitable subgraphs from the set of enumerated subgraphs as custom instructions that will be implemented in custom function units. As the number of candidate subgraphs can be very large, the subgraph selection problem is generally considered as a computationally difficult problem [1]. Previous work on subgraph selection problem can be grouped into two categories:

Optimal algorithms: Some of previous work try to find exact solution for the subgraph selection problem. For example, the authors of [11] propose a branch-and-bound algorithm for selecting the optimal subset of custom instructions. Cong et al. solve this problem by using dynamic programming [12]. Some other researchers address the subgraph selection problem using integer linear programming (ILP) or linear programming (LP) [13]–[15]. These methods treat the selection problem as maximizing or minimizing a linear objective function, while each subgraph is represented as a variable, which has a boolean value. The area constraint or non-overlapping constraint is expressed as linear inequality. These formulations are provided to an ILP or LP solver as input. Then, the solver may produce a solution. Similar to LP method, Martin et al. try to deal with the selection problem by using constraint programming method [16], [17]. However, due to the complexity of the subgraph selection problem, these methods may fail to produce a solution when the size of the application graph becomes large.

Although the subgraph selection problem is widely considered as a computationally difficult problem, it still lacks of an exact upper bound on the number of feasible solutions. In this article, we first formulate the subgraph selection problem under non-overlapping constraint as a maximum clique problem. Then, with this formulation, the upper bound on the number of feasible solutions is formally proved to be $3^{n/3}$, where n is the number of subgraphs.

Heuristic algorithms: Since optimal algorithms are usually intractable when the problem size is large, it is necessary to solve the problem with heuristic algorithms. Kastner et al. heuristically solve the problem by contracting the most frequently occurring edges [18]. Wolinski and Kunchcinski propose a method that selects candidates based on the occurrence of specific nodes [19]. A method attempting to preferentially select the subgraphs along the longest path of a given application graph has been proposed by Clark et al [20].

In recent years, due to good scalability and trade-offs between search time and quality of the solutions, many meta-heuristic algorithms have been proposed to solve the subgraph selection problem. As an example, a genetic algorithm (GA) is introduced to overcome the intractability of the subgraph selection problem [3]. The experiments show that the genetic algorithm may produce near-optimal solutions for problems with different sizes. In [21], a tabu search algorithm (TS) is adapted to solve the subgraph selection problem. The authors report that the tabu search algorithm can provide optimal solutions for medium-sized problems. It can still produce solutions when optimal algorithms fail

to produce solutions for large-sized problems. Other meta-heuristic algorithms like particle swarm optimization (PSO) [22], ant colony optimization (ACO) [23], [24] and simulated annealing algorithms (SA) [25] have been also introduced or adapted to address the subgraph selection problem or similar problems. However, the comparisons between these popular meta-heuristic algorithms are still missing in the existing literature. Thus, the main objective of this paper is to adapt the SA, TS, GA, PSO and ACO algorithms to solve the subgraph selection problem, and compare these algorithms in terms of runtime and quality of the results.

III. Problem Formulation

The definition of the subgraph selection problem discussed in this paper is based on the following notations.

Each basic block in an application can be represented as a directed acyclic graph $G = (V, E)$, where $V = \{v_1, v_2, ..., v_m\}$ represents the primitive instructions and E indicates the data dependency between instructions. A custom instruction can be graphically expressed as a subgraph $S_i = \{V_i, E_i\}$ of G ($V_i \subseteq V$ and $E_i \subseteq E$).

Similar to [26], every node v_i in S_i is associated with a software latency s_i. The accumulated software latency L_i of each subgraph can be calculated as follows:

$$L_i = \sum_{v_i \in S_i} s_i \tag{1}$$

Each subgraph candidate is associated with a performance gain P_i. The performance gain represents the number of clock cycles saved by executing the set of primitive instructions in a custom function unit instead of executing them in the base processor. The calculation for the performance gain of each subgraph is as follows:

$$P_i = L_i - (HW_i + E_i) \tag{2}$$

where HW_i indicates the latency of executing these primitive instructions in the corresponding custom function unit, and E_i represents the extra time required to transfer the input and output operands of the custom instruction from and to the core register files.

Definition 1. Non-overlapping constraint: The subgraphs enumerated by the subgraph enumeration step may have some nodes in common (overlapping). Although allowing overlapping between selected subgraphs may bring better performance improvement, it may make the code regeneration intractable and increase unnecessary power consumption. Similar to most existing work [21], [26], [27], we apply the non-overlapping constraint for the selection problem in this paper.

Definition 2. Acyclicity constraint: Two enumerated subgraphs may provide data to each other. If these two subgraphs are selected together, then a cycle is formed. This makes it impossible to implement both subgraphs as custom instructions together. Hence, cyclicity check should be performed to guarantee the appropriate implementation. In this work, we use an efficient method proposed in [21] to perform cyclicity check.

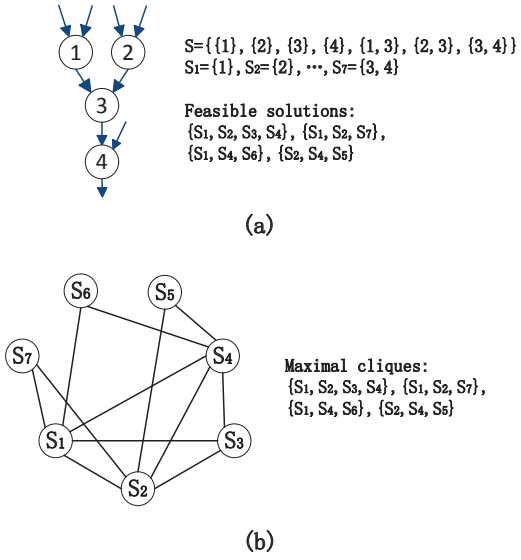

Figure 2. Example of building a compatibility graph from a given set of subgraphs. (a) original application graph, enumerated subgraphs and covering solutions. (b) corresponding compatibility graph and its maximal cliques

Problem $\mathcal{P}$. Given a DFG $G = (V, E)$ and a set of subgraphs $S = \{S_1, S_2, ..., S_n\}$, find a subset of subgraphs from S that maximizes the sum of performance gain while satisfying the non-overlapping constraint and the acyclicity constraint.

Mathematically, the subgraph selection problem can be described as the following integer linear formulation:

$$\mathcal{P} = \left\{ \begin{array}{l} maximize \quad \sum_{i=1}^{n} P_i \cdot x_i \\ subject \ \ to \ \ \sum_{j=1}^{n} a_{ij} \cdot x_j = 1, i = 1, \cdots, m \\ S_i \cap Pred(S_j) = \emptyset \ or \ S_i \cap Succ(S_j) = \emptyset, \forall x_i, x_j = 1 \\ x_i \in \{0, 1\}, \ i = 1, 2, \cdots, n \end{array} \right\} \tag{3}$$

where $a_{ij} = 1$ if the node v_i belongs to subgraph S_j, $a_{ij} = 0$ otherwise. $x_i = 1$ if the subgraph S_i is selected, $x_i = 0$ otherwise. $\sum_{i=1}^{n} P_i \cdot x_i$ indicates the objective of minimising the execution time of the application code (maximizing the performance gain). $\sum_{j=1}^{n} a_{ij} x_j = 1$ guarantees each node is only covered by one selected subgraph such that the non-overlapping constraint is satisfied. $Pred(S_j)$ and $Succ(S_j)$ represent the predecessors and the successors of subgraph S_j respectively. The acyclicity constraint is guaranteed by $S_i \cap Pred(S_j) = \emptyset \ \ or \ \ S_i \cap Succ(S_j) = \emptyset$.

Definition 3. A feasible solution to the subgraph selection problem is a solution which satisfies all its constraints. An optimal solution to the subgraph selection problem is a feasible solution which maximizes its objective function.

Definition 4. Two subgraphs in a feasible solution are said to be compatible if and only if two subgraphs satisfying both non-overlapping constraint and acyclicity constraint.

Definition 5. The compatibility graph $\mathcal{C}(\mathcal{S})$ of a set of subgraphs $S = \{S_1, S_2, ..., S_n\}$ is an undirected graph whose nodes correspond to the subgraphs in S, and an edge is placed between two nodes corresponding to subgraphs S_i and S_j if and only if S_i and S_j are compatible. It is clear that any two nodes have no edge between them if they overlap ($S_i \cap S_j \neq \emptyset$) or if there exists a cycle between them. Fig. 2 shows an example of building the compatibility graph for a given set of subgraphs.

Theorem 1: Each feasible solution corresponds to a maximal clique in the compatibility graph and vice versa.

Proof: Assume that a feasible solution is composed of subgraphs $S_1, \cdots, S_p$. Since the solution is feasible, any two subgraphs S_i and S_j in the solution are compatible. Then there must be an edge between S_i and S_j in the compatibility graph, so $S_1, \cdots, S_p$ is a clique.

On the other hand, suppose that $S_1, \cdots, S_p$ is a maximal clique in the compatibility graph, and that the corresponding solution cannot fully cover the original application graph G. Assume a node v of G is not covered by any of the subgraphs in the maximal clique. Then a subgraph S_k containing only the node v is not in the maximal clique. As the subgraph S_k is connected to each subgraph in $S_1, \cdots, S_p$, thus, $S_1, \cdots, S_p, S_k$ is a clique, contradicting the assumption that the former clique is maximal. Therefore, a maximal clique in the compatibility graph corresponds to a feasible solution. ■

Theorem 2: There exists an upper bound of $3^{n/3}$ on the number of feasible solutions for the subgraph selection problem $\mathcal{P}$.

Proof: By applying Theorem 1, the number of feasible solutions for $\mathcal{P}$ should be the same as the number of maximal cliques in the compatibility graph. As we known that the number of maximal cliques in a graph with n nodes is bounded by $3^{n/3}$ [28], thus, the number of feasible solutions for $\mathcal{P}$ is bounded by $3^{n/3}$. ■

With Theorem 1, we known that each feasible solution corresponds to a maximal clique in the compatibility graph. It is not difficult to understand that an optimal solution of the problem $\mathcal{P}$ corresponds to a maximum clique among all the maximal cliques. Therefore, the problem $\mathcal{P}$ can be also viewed as a typical weighted maximum clique problem.

IV. Heuristic Optimization Algorithms

This section presents how the five heuristic algorithms are adapted to solve the problem $\mathcal{P}$. An adaptive strategy implemented to search local optimum solution for PSO and ACO is also explained.

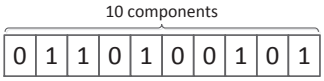

Figure 3. A solution with 10 components, five of which are selected in the solution

Algorithm 1 Pseudo-code for SA algorithm

Input: G - the application graph; S - the set of subgraphs; P_i - performance gain of each subgraph
Output: b - a best solution found

```
initialize parameters;
greedily generate an initial solution s
while termination condition is not true do
    for k:=1 to N do
        generate a neighbor solution ns of s
        if P(ns) <P(b) then
            s = ns with probability exp[(P(ns) − P(b))/t]
        else
            s = ns
        end if
    end for
    if P(s)>P(b) then
        b = s
    end if
    update temperature t = γ · t;
end while
```

In these algorithms, a solution is represented as a n-components vector (see Fig.3). n denotes the number of candidate subgraphs. A component corresponds to a subgraph. In a solution, 1 indicates that the corresponding subgraph is selected, while 0 means that the subgraph is not selected.

A. Simulated Annealing

Simulated annealing (SA) was first introduced by Kirkpatrick to solve combinatorial optimization problems [29]. It is an analogy to the process of physical annealing in solids. A crystalline solid is heated to a temperature, then it is allowed to be cooled slowly until the material freezes into a stable state-minimum lattice energy state that is free of crystal defects. Simulated annealing for combinatorial optimization problems mimics the thermodynamic behavior in physical annealing.

SA starts from an initial solution. The solution representation is presented in Fig. 3 as an example. In the adapted algorithm, the solution generated by a simple greedy algorithm is provided as the initial solution. Based on the given initial solution, the algorithm iteratively improves the quality of the solution by selecting the best solution from a number of neighbors.

Neighbor generation: The neighbor of the current solution is generated by randomly flipping the value of a component in current solution. Fig. 4 shows an example of generating a neighbor. Note that, as the problem $\mathcal{P}$ is a highly constrained

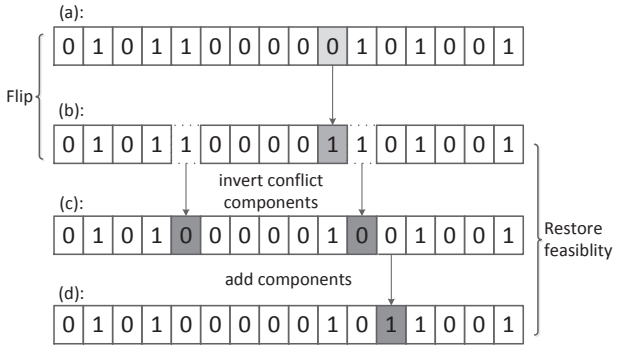

Figure 4. Generating a neighbor solution of a feasible solution. (a) a feasible solution. (b) an infeasible neighbor solution, the 5th and 11th components overlap with the 10th component. (c) invert the two conflict components. (d) add components to form a feasible solution

problem, flipping a component may lead to an infeasible solution. Thus, a feasibility restoring operation should be carried out. For example, if the value of a component c is flipped from 0 to 1, the non-overlapping constraint or acyclicity constraint is violated. Therefore, it is necessary to restore the feasibility after the flipping. In this work, all the selected components overlapping with c should be flipped from 1 to 0. In this case, the nodes covered by these components become uncovered. Then, we greedily add a component that covers only the uncovered nodes and has the best performance gain among the candidates until the solution is feasible.

At each iteration, the newly generated neighbors are compared to the current best solution. Better solutions are always accepted, while some worse solutions are also able to be accepted in order to escape local optima. The probability of accepting solutions including non-improving ones is defined as follows:

$$L(ns) = \left\{ \begin{array}{ll} exp[(P(ns) - P(b))/t] & if\ P(ns) < P(b) \\ 1, & if\ P(ns) \geq P(b) \end{array} \right\} \quad (4)$$

where $P(ns)$ represents the performance gain of the solution ns, $P(b)$ is the performance gain of the current best solution, and t is the current temperature. The current temperature is calculated as follows:

$$t = \gamma \cdot t \quad (5)$$

where γ is a temperature control parameter for cooling.

In the algorithm (see Algorithm 1), a set of parameters should be specified, i.e. initial temperature T_0, final temperature T_f, temperature cooling rate γ and the number of neighbors N in each iteration. The algorithm terminates when the temperature reaches the final given temperature or when there is no improvement over the last M iterations.

Algorithm 2 Pseudo-code for TS algorithm

Input: G - the application graph; S - the set of subgraphs; P_i - performance gain of each subgraph
Output: b - a best solution found

```
initialize parameters;
greedily generate an initial solution s
while termination condition is not true do
    generate q neighbor solutions of current solution s
    calculate the degrees of q neighbors
    if all q neighbors are tabu-active then
        s = the neighbor with minimal tabu degree
    else
        s = the neighbor with the best performance gain
    end if
    if P(s) > P(b) then
        b = s
    end if
    update frequency-based memory and recency-based memory respectively;
end while
```

B. Tabu Search Algorithm

Tabu search (TS) was initially introduced by Glover for solving mathematical optimization problems [30]. It has been showed to be an effective and efficient method for many combinatorial optimization problems. Similar to simulated annealing, TS is also a neighbourhood search based method. TS keeps track of searching history to restrict the next moves, while SA allows random moves. TS begins with an initial solution and iteratively moves to a next solution selected from a set of neighbourhood solutions. In the whole searching process, a *recency-based memory* (tabu list) that records recent moves is used to prevent cycling when moving from local optima. A *frequency-based memory* that stores the frequency of visiting each component is also used to diversify the searching.

In the adapted TS algorithm (see Algorithm 2), the method for generating a neighbor of a given solution is the same as that described in SA. Flipping the value of a component is considered as a move operation. In the tabu list, a list of recent moves are recorded in FIFO order. The length of the tabu list is called as *tabu tenure*. Values of tabu tenure between 7 and 20 appear work well for a variety of problem classes [31]. The frequency-based memory is used to penalize moves with high frequency and encourage moves with low frequency. Fig. 5 shows an example of the recency-based memory and the frequency-based memory.

At each iteration, a set of neighbors of the current solution are generated. The algorithm picks the best non-tabu neighbor from the new set of neighbors and puts the corresponding move on the top of the tabu list. Each neighbor is assigned with a degree that indicates its entry history in the tabu list. The definition of the degree can be found in [21]. However, a tabu neighbor that is better than any visited solution so far may be yielded sometimes. In such case,

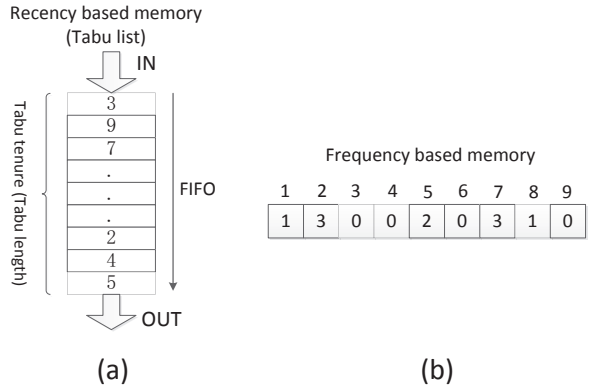

Figure 5. (a) An example of recency based memory with tabu tenure = 9; (b) An example of frequency based memory recording the selection frequency of each component (9 components in total in this example)

Algorithm 3 Pseudo-code for GA

Input: G - the application graph; S - the set of subgraphs; P_i - performance gain of each subgraph
Output: b - a best solution found
1: initialize parameters;
2: randomly generate an initial population;
3: calculate the fitness of each individual in the population;
4: **while** termination condition is not true **do**
5: draw $2T$ individuals from the population forming two pools
6: produce children by crossover or mutation
7: evaluate the fitness of the generated children
8: replace a fraction of least-fit ancestors by the children
9: **end while**

the tabu neighbor is still selected by overruling the tabu restriction. This is called as *aspiration criterion*. In some other cases, if all generated neighbors are tabu, the oldest one that has the smallest degree in the tabu list is selected. The algorithm terminates when the iteration counter reaches the given maximum iteration number N or when there is no improvement over the last M iterations.

C. Genetic Algorithm

Genetic algorithm (GA) was first introduced by Holland [32] in the 1970s. GA is one of the most popular evolutionary computation techniques. It is inspired from the genetic inheritance in the evolutionary process of nature world. Genetic algorithm begins with an initial population. The solutions are evolved generation by generation through inheriting the good characteristics from highly adapted ancestors, while the less adapted ancestors will be eliminated and replaced by the newly generated descendants.

In adapted GA, the encoding of a solution is similar to the representation in Fig 3. The components involved in the representation are called as *genes*. In order to keep diversity of initial solutions, all the initial solutions are randomly generated. Each individual of the population is evaluated according to a fitness function. In this work, the fitness function is defined as the performance gain (see formula 2).

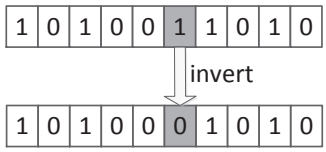

Figure 6. An example of mutation operator

At each generation, a number of solutions are selected from the current population to reproduce (crossover) new solutions. In general, fitter solutions are more likely to be selected to reproduce new solutions. In this work, a tournament selection method is adapted. The method first creates two pools, each of which contains T solutions randomly drawn from the existing population. The best solution in each pool is selected as a parent to produce new solutions using the crossover operator.

Crossover Operator is a key operation to inherit the good characteristics from parents. In this work, a guided fusion crossover operator is utilized. The fusion crossover operator enables the parent solutions to contribute the gene level rather than segment level. Each gene (subgraph) in the child solution is created by copying the corresponding gene from one or other parent with a random number generator [0,1]. Let f_{p_1} and f_{p_2} be the finesses of the parents p_1 and p_2 respectively. The operator creates gene by gene to form a child solution. The genes which are identical in parents are directly copied to the child solution. Otherwise, if the random number is between [0, $f_{p_1}/(f_{p_1}+f_{p_2})$], then the gene in p_1 is copied to the child solution. If the random number is between ($f_{p_1}/(f_{p_1}+f_{p_2})$,1], the gene in p_2 is copied to the child solution. The rule for constructing each gene in the child can be formulated as follows (this rule is applied only when the corresponding genes in parents are not identical):

$$child(i)=\left\{ \begin{array}{ll} parent1(i) & if\ rand() \ \leq \ \frac{f_{p_1}}{f_{p_1}+f_{p_2}} \\ parent2(i), & otherwise \end{array} \right\} \quad (6)$$

Mutation Operator diversifies the solutions by introducing new search space. The mutation operator generally flips the value of randomly chosen genes in the solution. An example of mutation operator is shown in Fig. 6. The genes are chosen according to a user-defined mutation rate δ. For example, assume there are 10 genes in a solution, $\delta = 0.2$, then two $(0.2{*}10)$ randomly chosen genes should be selected to invert.

The genetic algorithms have a tendency to converge. When the population converges to a set of homogeneous solutions, the solutions may fall into local optimum. In order to escape from local optimum and keep the diversity of the

Algorithm 4 Pseudo-code for PSO algorithm

Input: G - the application graph; S - the set of subgraphs; P_i - performance gain of each subgraph
Output: b - a best solution found
1: initialize parameters;
2: randomly generate an initial swarm;
3: calculate the velocity of each particle in the swarm;
4: **while** termination condition is not true **do**
5: **for** k:=1 to N **do**
6: update the components in particle k using equations (8) and (10)
7: carry the solution to its local optimum (optional)
8: update the local best particle of k
9: **end for**
10: update the global best particle
11: **end while**

population, a random immigrant mechanism is used in the adapted algorithm. The mechanism replaces a fraction of less fit solutions in the population by randomly generated solutions. The termination condition of this algorithm is the same as the ones set for TS.

D. Particle Swarm Optimization

Particle swarm optimization (PSO) is another popular evolutionary computation technique that was firstly proposed by Kennedy and Eberhart [33]. PSO simulates the behaviors of a group of birds when searching for food in an area. In the scenario of searching food, only the bird who is nearest to the location of food knows where to find food. In order to find food, the other birds may follow the bird which is nearest to the food.

As the target problem $\mathcal{P}$ is a discrete optimization problem, the binary version of the particle swarm algorithm is adapted. In the adapted PSO, a *swarm* that is composed of a group of random *particles* is initialized. A particle represents a solution for the problem $\mathcal{P}$. The PSO algorithm iteratively updates the particles in the swarm. As described in Fig. 3, a particle can be expressed as an n-dimensional vector $X_k = (x_{k1}, x_{k2}, ..., x_{kn})$. The component x_{kd} has a discrete value "1" or "0". Each component in the particle is associate with a velocity v_{kd}, which determines the probability to set the component to 1 in the next solution construction. The velocity is calculated as the following formula:

$$v_{kd} = v_{kd} + c_1 \cdot (b_{kd} - x_{kd}) + c_2 \cdot (b_{gd} - x_{kd}) \quad (7)$$

where c_1 and c_2 are two positive learning factors. In PSO, each particle keeps track of its best solution it has achieved so far, this best solution is represented as $B_k = (b_{k1}, b_{k2}, ..., b_{kn})$. The best solution so far achieved among all the particles in the population is also recorded. It is represented as $B_g = (b_{g1}, b_{g2}, ..., b_{gn})$.

Since v_{kd} is used to determine the value of d component for k particle, the probability should be constrained to the interval [0.0,1.0]. Thus, v_{kd} is transformed into probability using the sigmoid function as follows:

$$sig(v_{kd}) = \frac{1}{1 + exp(-v_{kd})} \quad (8)$$

where $sig(v_{kd})$ indicates the probability of component x_{kd} taking value 1. Then, the rule deciding the value of component x_{kd} is defined as follows:

$$x_{kd}=\left\{ \begin{array}{ll} 1 & if\ rand() \ \leq \ sig(v_{kd}) \\ 0, & otherwise \end{array} \right\} \quad (9)$$

where $rand()$ is a random number generated from [0.0, 1.0]. In the algorithm (see Algorithm 4), few parameters need to be specified. M is the number of particles in the swarm. c_1 and c_2 indicate the influence of the local best solution and global best solution respectively. The termination condition for this algorithm is similar to the ones set in the tabu search algorithm.

1) Adaptive local optimum search: Inspired from local optimum based search methods (tabu search and simulated annealing), an adaptive local optimum search is performed in our proposed PSO algorithm and the latter ACO algorithm after constructing a feasible solution. If a neighbor solution with a better quality is found, the original solution will be replaced by the better solution. In this paper, a neighbor of a given solution is obtained from flipping the value of r components in the given solution at random. The value of r is adapted according to the following formula:

$$r=\left\{ \begin{array}{ll} (1 - \frac{P(i)}{P(b)}) \cdot l & if\ P(i) \ < \ P(b)) \\ 1, & otherwise \end{array} \right\} \quad (10)$$

where $P(i)$ is the performance gain of the current solution, $P(b)$ represents the performance gain of the best solution obtained so far and l is the number of components with value "1" in the current solution.

E. Ant Colony Optimization

Ant colony optimization (ACO) was initially proposed by Colorni, Dorigo and Maniezzo [34]. The idea of ACO was extracted from biological studies about ants: in the natural world, the ants initially wander randomly to find food. A chemical substance called pheromone is laying down along the paths where the ants traversed. Ants use pheromone to communicate with each other. As time goes on, all the pheromone trails start to evaporate. In this case, the shorter paths may have higher density of pheromone such that more ants will be attracted to follow the shorter paths. Finally, the other ants will be more likely to follow the shortest path to find food.

In ACO, we initially create a group of artificial ants. Each artificial ant constructs a solution from scratch. It iteratively adds solution components to complete a solution. At each

Algorithm 5 Pseudo-code for ACO algorithm

Input: G - the application graph; S - the set of subgraphs; P_i - performance gain of each subgraph
Output: b - a best solution found

```
initialize parameters;
while termination condition is not true do
    for k:=1 to N do
        while solution not completed do
            compute η(c), P_k(c)
            select the component to add, with probability P_k(c)
        end while
        carry the solution to its local optimum (optional)
    end for
    update pheromone trails τ_i(c) = ρ · τ_{i-1}(c) + Σ_{k=1}^{N} △τ_k(c);
end while
```

iteration, the ant selects a solution component according to a probabilistic state transition rule. The probabilistic state transition rule gives the probability that a component c is selected by ant k:

$$P_k(c)=\left\{ \begin{array}{ll} \frac{\alpha\cdot\tau(c)+\beta\cdot\eta(c)}{\sum_{v\in U_k(c)}(\alpha\cdot\tau(v)+\beta\cdot\eta(v))} & if\ c\in U_k(c) \\ 0, & otherwise \end{array} \right\} \quad (11)$$

where $\tau(c)$ is the pheromone trail on component c, $\eta(c)$ is a local heuristic information for rewarding the component leading to good performance gain. α and β are parameters that allow a user to control the relative importance of the pheromone trail versus performance gain. $U_k(c)$ is the set of components that remain to be visited by ant k positioned on component c.

The heuristic information $\eta(c)$ is implemented as follows:

$$\eta(c) = \frac{P_c}{ms} \quad (12)$$

where P_c is the performance gain obtained by component c, ms is the maximum performance gain obtained by a component among all the components.

After ant k completes a solution, a global update should be performed to calculate the pheromone trail taking into account evaporation and increment:

$$\tau_i(c) = \rho\cdot\tau_{i-1}(c) + \sum_{k=1}^{N}\triangle\tau_k(c) \quad (13)$$

where $0<\rho<1$ and ρ is a coefficient which represents the extent the pheromone retained on the component c. N is the number of ants. $\triangle\tau_k(c)$ is the pheromone ant k deposits on the component c. $\triangle\tau_k(c)$ is calculated as follows:

$$\triangle\tau_k(c)=\left\{ \begin{array}{ll} 1-\frac{1}{P(k)+1} & if\ c\ is\ selected\ by\ ant\ k \\ 0, & otherwise \end{array} \right\} \quad (14)$$

Table I
CHARACTERISTICS OF THE BENCHMARKS

Benchmark (size)	Input	Output	Number of Subgraphs
JPEG (349/140/160)	3	1	248
	4	1	324
	6	1	452
	3	2	298
	4	2	654
	5	2	819
	6	2	1195
SUSAN (345/174/224)	3	1	236
	4	1	396
	6	1	822
	3	2	236
	4	2	683
	5	2	926
	6	2	1492
DES3 (157/58/58)	3	1	92
	4	1	110
	6	1	128
	3	2	114
	4	2	159
	5	2	211
	6	2	254
GSM (946/290/290)	3	1	513
	4	1	736
	6	1	1293
	3	2	768
	4	2	1034
	5	2	1590
	6	2	2102

where $P(k)$ is the total performance gain of the solution completed by ant k. This calculation implies that ant k deposits more pheromone on component c for the solution with higher performance gain.

The pseudo code of the ACO algorithm is shown in Algorithm 5. In this work, the termination criterion is set as the number of iterations incurred without any improvement in the quality of the solution or the maximum iteration number like TS algorithm.

V. EXPERIMENTAL RESULTS

In the experiments, a front-end compiler named GECOS [35] is used in our design flow to transform the source code to DFGs. After transformation, the subgraph enumeration step presented in [5] is carried out for enumerating all the subgraphs satisfying I/O constraints (maximum number of I/O) as custom instruction candidates. Then, the heuristic algorithms discussed in the previous section are applied to select the subgraphs. All the heuristic algorithms are implemented in Java and run on a Intel i3-3240 with (3.4 GHz) with 3.4 GB RAM.

We compare the five standard heuristic algorithms in terms of search time and quality of the results when considering the same amount of solutions. Then, we compare the ACO algorithm and the PSO algorithm with the proposed adaptive local optimum search with the standard ACO algorithm and the standard PSO algorithm respectively.

A set of real-life benchmarks that are rich in computing operations has been selected from MiBench [36] and MediaBench [37]. Table 1 describes the characteristics of the

Table III
COMPARISON OF THE QUALITY OF RESULTS AND SEARCH TIME FOR THE FIVE STANDARD HEURISTICS

Benchmark	I/O	BnB		SA		TS		GA		PSO		ACO	
		Gain	Time	Gain	Time	Gain	Time	Gain	Time	Gain	Time	Gain	Time
JPEG	3,1	51	143	51	0.5	50	0.9	51	13.4	51	11.1	47	135
	4,1	67	195	67	0.8	61	1.3	67	13.8	67	13.5	67	178
	6,1	67	298	67	1.5	67	2.2	67	16.9	67	13.8	67	209
	3,2	58	169	58	0.8	51	1.1	58	14.9	58	13.9	46	157
	4,2	-	-	76	1.5	70	2.2	75	15.6	74	14.1	75	190
	5,2	-	-	80	2.7	78	3.5	80	24.0	74	15.5	76	236
	6,2	-	-	81	4.0	81	4.9	81	29.3	81	17.9	77	283
SUSAN	3,1	28	193	26	0.4	25	0.6	26	17.5	25	20.7	24	171
	4,1	54	298	53	0.9	52	1.3	53	20.8	50	21.9	49	246
	6,1	-	-	76	3.6	74	4.1	76	29.2	65	22.2	69	438
	3,2	27	193	26	0.5	27	0.6	26	17.6	25	20.8	23	175
	4,2	-	-	53	0.8	52	1.4	53	21.7	50	21.9	48	268
	5,2	-	-	72	1.7	70	2.6	72	24.6	69	22.7	70	323
	6,2	-	-	76	3.7	74	4.6	76	30.9	73	23.2	74	474
DES3	3,1	24	69.4	24	0.4	23	0.7	24	7.1	24	5.0	24	24.8
	4,1	28	72.3	28	0.6	26	1.0	28	8.2	28	5.3	27	28.1
	6,1	30	78.8	30	0.9	29	1.3	30	9.1	30	5.6	30	31.2
	3,2	30	74.4	30	0.5	29	0.9	30	7.9	30	5.3	30	25.1
	4,2	32	84.2	32	0.9	27	1.3	32	8.7	32	5.7	32	37.2
	5,2	36	122	36	1.5	33	2.1	36	10.0	36	5.8	35	43.0
	6,2	36	137	36	2.2	35	2.9	36	12.6	35	6.0	35	49.8
GSM	3,1	117	2101	117	0.6	109	1.1	117	25.7	116	18.0	117	427
	4,1	-	-	157	1.0	143	1.7	157	27.4	157	19.7	157	655
	6,1	-	-	189	2.7	182	4.0	189	33.1	189	22.3	189	716
	3,2	-	-	133	0.5	122	1.1	133	20.7	131	18.6	133	521
	4,2	-	-	173	0.9	157	1.8	173	28.1	172	19.4	173	666
	5,2	-	-	174	1.6	169	2.6	174	34.0	174	22.1	174	923
	6,2	-	-	205	2.8	198	4.0	205	36.0	205	23.2	205	1066

Table II
HARDWARE TIMING OF THE OPERATIONS IMPLEMENTED IN THE CUSTOM FUNCTION UNITS

Operation	Input Data-Width	Normalized Latency
MAC	32x32+64	1.00
Adder	32+32	0.31
Multiplier	32x32	0.96
Divider	32/32	7.45
Barrel shifter	32	0.23
Bitwise AND/OR	32	0.03

selected benchmarks. The first column indicates the benchmark used, the number of nodes and the number of valid nodes (the operations except memory and branch operations) in the DFG of a computationally intensive basic block extracted from each benchmark. It also indicates the number of clock cycles required when executing the DFG on the base processor. The columns $Input$ and $Output$ represent the I/O constraints imposed in the subgraph enumeration step (maximum number of inputs and outputs). The number of enumerated subgraphs under different I/O constraints is given in the column $Number\ of\ Subgraphs$. Larger subgraphs may be selected when the I/O constraints are relaxed. Notice that only convex subgraphs are considered.

Table 2 shows the hardware latency of arithmetic and logic operators by synthesizing on a 0.13 μm CMOS process and normalizing to the delay of a 32-bit multiply accumulator (MAC). HW_i (see equation (2)) estimates the hardware latency of a custom instruction using the latency information of Table 2. In the experiments, we assume the core processor is single-issued and pipelined.

In the first part, we compare the results obtained by the five standard heuristic algorithms in terms of the search time and the quality of the solutions when evaluating the same amount of solutions. For a fair comparison, the parameters were set such that the number of solutions considered is the same whatever the algorithm. Furthermore, the parameters of each heuristic algorithms have been carefully selected as follows such that a good quality of results can be obtained for most of tests. The parameter values of each heuristic algorithm have been obtained empirically by comparing the quality of results of each algorithm with different hand-tuned parameter values.

1) SA: In the case of SA, initial temperature $= 3000$, the final temperature $= 50$, constant rate for cooling $= 0.98$, number of neighbors in each iteration $= 1000$ and non-improvement threshold $= 20$.

2) TS: For TS, the parameters are set as follows: neighborhood size $= 2000$, tabu list length $= 10$, maximum iteration number $= 100$ and non-improvement threshold $= 20$.

3) GA: The set of parameters in GA are set as follows: size of population $= 2000$, size of tournament 40, crossover rate $= 0.95$, mutation rate $=$ 1/number of genes, replacement rate $= 10\%$, maximum number of generations $= 100$ and non-improvement threshold $= 20$.

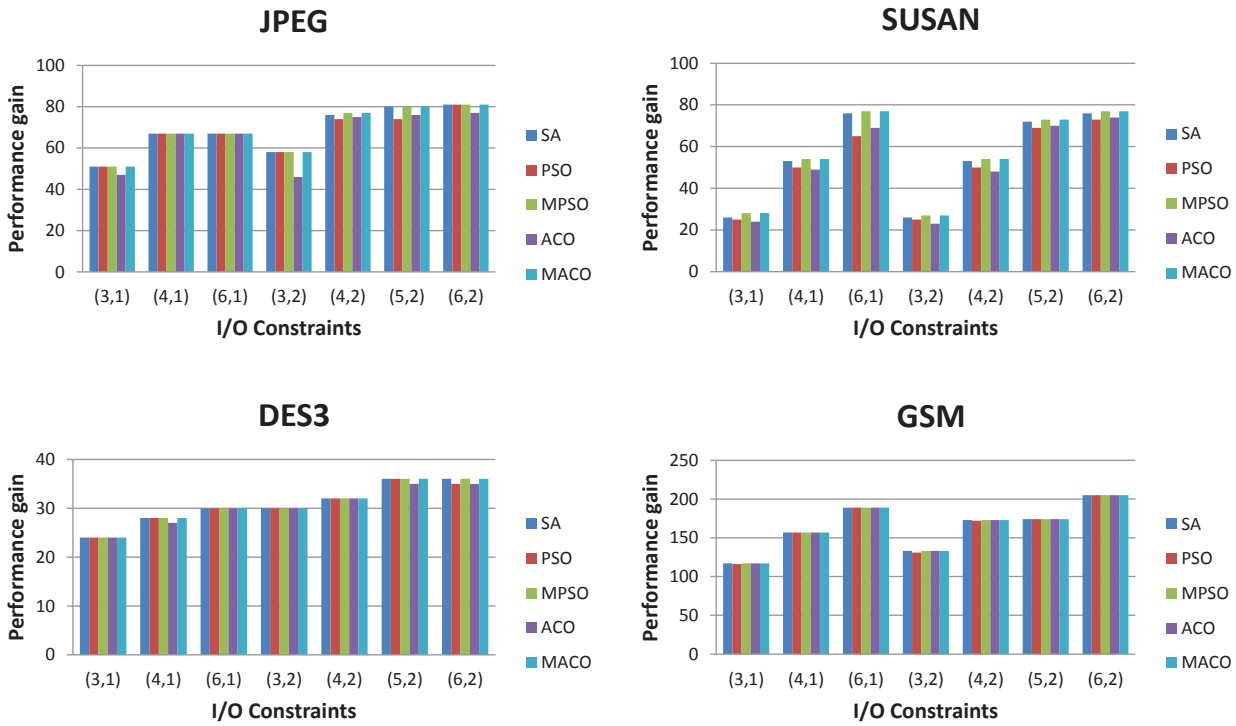

Figure 7. Comparison of the quality of the solutions obtained by SA, ACO, PSO, MACO and MPSO

4) PSO: In PSO, number of particles = 2000, c_1 =2, c_2 =2, maximum number of iterations = 100 and non-improvement threshold = 20.

5) ACO: In the case of ACO, $\alpha = 4$, $\beta = 1$, $\rho = 0.5$, the number of ants = 2000, maximum iteration number = 100 and non-improvement threshold = 20.

In these parameters, non-improvement threshold = 20 means that the algorithms terminate when there is no improvement over the last 20 iterations. It can be observed that, among the parameters setting for the five adapted heuristic algorithms, TS algorithm has the smallest number of parameters to set (four parameters), while GA algorithm has the largest number of parameters to set (seven parameters). The maximum number of solutions evaluated by each algorithm is 200 000 (2000*100). Furthermore, an exact algorithm (BnB) [21] that can give optimal solutions is also implemented to evaluate the heuristic algorithms.

Table 3 reports the quality of the solutions (gain is the number of saved clock cycles when compared to the execution on the base processor) and search time (in seconds) of the five standard heuristic algorithms over 10 runs and the BnB algorithm. These algorithms are evaluated with the benchmarks under different I/O constraints. In the experiments, we observed that the heuristic algorithms usually terminate when there is no improvement over the last 20 iterations for most of tests. The results show that the BnB algorithm can only produce result for problems with small size (the number of subgraphs is less than 600). For the problems with large size, the BnB algorithm fails to provide an optimal solution in acceptable time (within 10 h). From the results, it is clear that the SA algorithm and the GA algorithm achieve similar results that are better than the other three heuristic algorithms. It can be verified that the results generated by the SA algorithm and the GA algorithm are optimal or near-optimal when the problem size is small. The PSO algorithm slightly outperforms the standard ACO algorithm (on average 0.7%). The standard ACO algorithm outperforms the TS algorithm on average 3.1%.

Comparing the search time (in seconds), it can be observed that the two hill-climbing methods (TS and SA) consumes the least of time when performing 200 000 evaluations. The search time of the PSO algorithm is slightly less than that of the GA algorithm. The ACO algorithm requires the largest search time. Based on the reported quality of the results and search time, it can be seen that the SA algorithm can produce the best results, while taking the least amount of computation time.

In the second part, the PSO algorithm and the ACO algorithm with adaptive local optimum search (MPSO and MACO respectively) are compared to the standard PSO algorithm and ACO algorithm respectively. As the standard SA algorithm achieves the best results with the least search time among all the five standard heuristics in the first part, it is used to evaluate the MPSO algorithm and the MACO algorithm.

Fig 7. shows the quality of the results obtained by the five

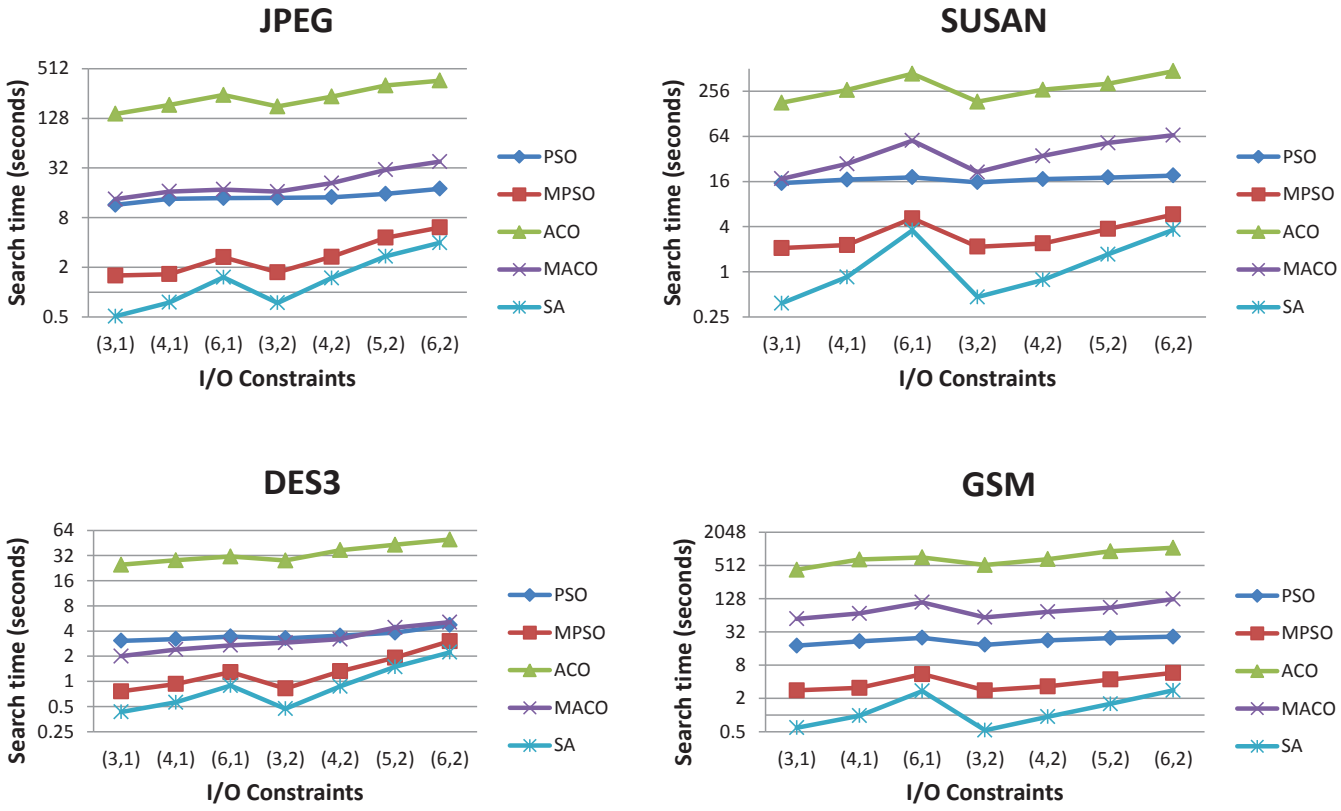

Figure 8. Comparison of the search time consumed by SA, ACO, PSO, MACO and MPSO

algorithms in 10 runs [1]. Based on the results, we can see that the MPSO algorithm and the MACO algorithm outperform the standard PSO algorithm and the standard ACO algorithm on average 2.1% and 2.9% respectively. Furthermore, the MPSO algorithm and MACO algorithm achieve the same results that are slightly better than the results obtained by the SA algorithm. In particular, the SA algorithm can not reach the optimal results for the benchmark SUSAN when the I/O is (3,1),(4,1) or (3,2), while the MPSO algorithm and MACO algorithm can produce the optimal results (the same results as BnB).

Fig 8. compares SA, PSO, MPSO, ACO and MACO algorithms in terms of average search time. The maximum number of solutions evaluated by each algorithm is 200 000. The results are obtained from 10 runs. From the results, we can see that the SA algorithm is the fastest among the five algorithms. It can be also observed that the computation time of the MPSO algorithm and the MACO algorithm are much more less than that of the standard PSO algorithm and the standard ACO algorithm. The MACO algorithm (or MPSO) with adaptive local optimum search builds 20 000 solutions using the standard ACO (or PSO) optimization and the other 180 000 solutions using proposed local optimum search. As the optimum search method produce a neighbor solution much faster than the standard ACO or PSO optimization, the overall search time is significantly reduced by the MACO algorithm. On average, the MACO algorithm is 8.7 times faster than the standard ACO algorithm. The MPSO algorithm is 3.9 times faster than the standard PSO algorithm. Combining the quality of the results shown in Fig 6. and the search time presented in Fig 7., we can conclude that the the MPSO algorithm and the MACO algorithm with adaptive local optimum search can find better results in shorter time than the standard PSO algorithm and the standard ACO algorithm.

[1]the standard PSO algorithm, the standard ACO algorithm, the MPSO algorithm, the MACO algorithm and the standard SA algorithm.

VI. Extra Comments

When the given hardware area is limited, it is necessary to take into account the area constraint. Although the area constraint is not considered in these experiments, the heuristic algorithms presented in this paper are able to take it into account with few modifications. In the case of considering area constraint, a set of patterns should be generated. The set of patterns is generally collected using a graph isomorphism algorithm. Given two subgraphs a and b, if a is isomorphic to b, a pattern T_i is created, and the subgraphs a and b are recorded in the pattern T_i as instances. If two or more instances of a pattern are selected, they may share the same

hardware implementation of their corresponding pattern (this is the case of reusing hardware). We can use the following inequality to express the area constraint:

$$\sum_{i=1}^{k} c_i y_i \leq A \tag{15}$$

where $y_i = 1$ indicates that one or more instances of the pattern T_i are selected, while $y_i = 0$ indicates that no instance of T_i is selected. c_i is the area cost of the hardware implementation of a pattern T_i. A is a given maximum area constraint. In the heuristic algorithms, the area cost of a solution should be calculated and the area constraint can be used to guide the construction of solutions. It is noteworthy that the upper bound on the number of feasible solutions should be less than $3^{n/3}$ when taking into account the area constraint.

For each of the experimented heuristic algorithms, modified versions that may improve quality of results or require less runtime can be found in the literature. However, in this paper the main objective is to compare the standard metaheuristic algorithms when applied to the custom instruction selection problem. Implementing modified versions of these heuristic algorithms is out of the scope of this paper. People who are interested can find details for example in [21]–[23], [38].

In this paper, we have proved the upper bound on the number of feasible solutions with respect to non-overlapping constraint and acyclicity constraint, however, allowing overlapping between selected subgraphs may bring more performance improvement. In this scenario, it is interesting to know the upper bound on the number of feasible solutions when overlapping is allowed. It is also necessary to compare the difference on the performance improvement and the search time between allowing overlapping and disallowing overlapping during subgraph selection. These can be part of our future work.

VII. Conclusions

In this paper, the upper bound on the number of feasible solutions for the subgraph selection problem has been given and formally proved. We have introduced five popular heuristic algorithms, namely simulated annealing, tabu search, genetic algorithm, particle swarm optimization algorithm and ant colony optimization algorithm, for solving the subgraph selection problem. Extensive experiments with real-life benchmarks have been carried out to evaluate the advantages and disadvantages of the five heuristic algorithms in terms of runtime performance and quality of the results. Furthermore, we have also implemented an adaptive local optimum search strategy for particle swarm optimization algorithm and ant colony optimization algorithm, which can further improve the quality of the solutions. Future work will deal with the inclusion of area constraint and allowing overlapping in the heuristic algorithms.

VIII. Acknowledgement

We are grateful to the anonymous referees for valuable suggestions and comments which helped us to improve the paper. The authors would like to thank the various grants from the National Natural Science Foundation of China (No. 61404069 and No. 61172144) and the Scientific Research Foundation for Ph.D. of Liaoning Province (No. 20141140).

References

[1] C. Galuzzi, K. Bertels. The Instruction-Set Extension Problem: A Survey. ARC 2008, pp. 209-220.

[2] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. DAC 2003, pp. 256-261.

[3] L. Pozzi, K. Atasu, and P. Ienne. Exact and approximate algorithms for the extension of embedded processor instruction sets. IEEE Trans. Comput.Aided Design Integr. Circuits Syst. 25(7), 2006, pp. 1209-1229.

[4] C. Xiao, E. Casseau. An efficient algorithm for custom instruction enumeration. GLSVLSI 2011, pp. 187-192.

[5] C. Xiao, E. Casseau. Exact custom instruction enumeration for extensible processors. Integration, the VLSI Journal 45 (3), 2012, pp. 263-270.

[6] X. Chen, D.L. Maskell, Y. Sun. Fast identification of custom instructions for extensible processors. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 26(2), 2007, pp. 359-368.

[7] P. Bonzini, L. Pozzi. Polynomial-time subgraph enumeration for automated instruction set extension. DATE 2007, pp. 1331-1336.

[8] T. Li, Z. Sun, W. Jigang. Fast enumeration of maximal valid subgraphs for custom-instruction identification. CASES 2009, pp. 29-36.

[9] A.K. Verma, P. Brisk, P. Ienne. Fast, nearly optimal ISE identification with I/O serialization through maximal clique enumeration. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 29(3), 2010, pp. 341-354.

[10] J. Reddington, K. Atasu. Complexity of computing convex subgraphs in custom instruction synthesis. IEEE Transactions on Very Large Scale Integration (VLSI) Systems. 20(12), 2012, pp. 2337-2341.

[11] N. Clark, A. Hormati, S. Mahlke, and S. Yehia. Scalable subgraph mapping for acyclic computation accelerators. CASES 2006, pp. 147-157.

[12] J. Cong et al. Application-specific instruction generation for configurable processor architectures. FPGA 2004, pp. 183-189.

[13] T. Mitra, Y. Pan. Satisfying real-time constraints with custom instructions. CODES+ISSS 2005, pp. 166-171.

[14] C. Galuzzi, E.M. Panainte, Y. Yankova, K. Bertels, S. Vassiliadis. Automatic selection of application-specific instruction-set extensions. CODES+ISSS 2006, pp.160-165.

[15] K. Atasu, G. Dundar, C. Ozturan. An integer linear programming approach for identifying instruction-set extensions. CODES+ISSS 2005, pp.172-177.

[16] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch, F. Charot: Constraint Programming Approach to Reconfigurable Processor Extension Generation and Application Compilation. TRETS 5 (2), 2012.

[17] M. A. Arslan, K. Kuchcinski. Instruction Selection and Scheduling for DSP Kernels on Custom Architectures. DSD 2013, pp. 821-828.

[18] R. Kastner, A. Kaplan, S. Ogrenci Memik, and E. Bozorgzadeh. Instruction generation for hybrid reconfigurable systems. ACM Trans. Des. Autom. Electron. Syst. 7(4), 2002, pp. 605-627.

[19] C. Wolinski, K. Kuchcinski. Automatic Selection of Application-Specific Reconfigurable Processor Extensions. DATE 2008, pp. 1214-1219.

[20] N. Clark, M. Kudlur, P. Hyunchul, S. Mahlke, K. Flautner. Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization. MICRO 2004, pp. 30-40.

[21] T. Li, J. Wu, S. Lam, T. Srikanthan, and X. Lu. Selecting profitable custom instructions for reconfigurable processors. J. Syst. Archit. 56(8), 2010, pp. 340-351.

[22] M. Kamal, K.N. Amiri, A. Kamran A. Dual-purpose custom instruction identification algorithm based on particle swarm optimization. ASAP 2010, pp. 159-166.

[23] F. Ferrandi, P.L. Lanzi, C. Pilato C. Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 29(6), 2010, pp.911-924.

[24] I.W. Wu, Z.Y. Chen, J.J. Shann J J. Instruction set extension exploration in multiple-issue architecture. DATE 2008, pp. 764-769.

[25] H. Lin. Multi-objective Application-specific Instruction set Processor Design: Towards High Performance, Energy-efficient, and Secure Embedded Systems (2011). Doctoral Dissertations.

[26] K. Atasu, W. Luk, O. Mencer, C. Ozturan, G.Dundar. FISH: Fast instruction synthesis for custom processors. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 20(1), 2012, pp. 52-65.

[27] Y. Guo, G.J.M. Smit, H. Broersma. A graph covering algorithm for a coarse grain reconfigurable system. LCTES 2003, pp. 199-208.

[28] J.W. Moon, L. Moser. On cliques in graphs. Israel journal of Mathematics, 1965, 3(1): pp. 23-28.

[29] S., Kirkpatrick, C.D. Gelatt, M. P. Vecchi. Optimization by simmulated annealing. Science. 220(4598), 1983, pp. 671-680. APA

[30] F. Glover. Tabu search: A tutorial. Interfaces, 1990, 20(4): pp. 74-94.

[31] T. Wiangtong, P. Cheung, and W. Luk. Comparing three heuristic search methods for functional partitioning in hardware-software codesign. Des. Autom. Embed. Syst. 6(4), 2002, pp.425-449.

[32] John H. Holland. Adaptation in natural and artificial systems. Univ. of Mochigan Press, 1975.

[33] J. Kennedy, R. Eberhart. Particle swarm optimization. IEEE International Conference on Neural Networks 1995, pp.1942-1948.

[34] M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: optimization by a colony of cooperating agents. Trans. Sys. Man. Cyber. Part B 26, 1996, pp. 29-41.

[35] GeCoS: Generic compiler suite - http://gecos.gforge.inria.fr/

[36] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M.Austin, T. Mudge, and R. B. Brown. Mibench: A free,commercially representative embedded benchmark suite. WWC 2001, pp. 3-14.

[37] C. Lee, M. Potkonjak, and W. H. Mangione-smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. MICRO 1997, pp 330-335.

[38] D. Pham, D. Karaboga. Intelligent optimisation techniques: genetic algorithms, tabu search, simulated annealing and neural networks. Springer Science & Business Media, 2012.