Selecting Profitable Custom Instructions for Reconfigurable Processors

Tao Li\textsuperscript{a}, Wu Jigang\textsuperscript{b}, Siew-Kei Lam\textsuperscript{b}, Thambipillai Srikanthan\textsuperscript{b}, Xicheng Lu\textsuperscript{a}

\textsuperscript{a}School of Computer, National University of Defense Technology, Hunan, China 410073.
\textsuperscript{b}School of Computer Engineering, Nanyang Technological University, Singapore 639798.

Abstract

Custom-instruction selection is an essential phase in instruction set extension for reconfigurable processors. It determines the most profitable custom instruction candidates for implementing in the reconfigurable fabric of a reconfigurable processor. In this paper, a practical computing model is proposed for the custom-instruction selection problem that takes into account the area constraint of the reconfigurable fabric. Based on the new computing model, two heuristic algorithms and an exact algorithm are proposed. The first heuristic algorithm, denoted as HEA, dynamically assigns priorities to the custom instruction candidates and incorporates efficient strategies to select custom instructions with the highest priority. The second heuristic algorithm, denoted as TSA, employs an efficient tabu search algorithm to refine the results of HEA to near optimal ones. Also, a branch-and-bound algorithm (BnB) is proposed to produce exact solutions for relatively small-sized problems or problems with stringent area-constraints. Experimental results show that HEA can produce more specific approximate solutions with a difference of only about 3\% when compared to the optimal solutions produced by BnB. This difference is further reduced to about 0.6\% by TSA. In addition, for large-sized problems where the exact algorithm becomes prohibitive, HEA and TSA can still produce solutions within reasonable time.

\textit{Key words:}
Custom instruction, selection algorithm, reconfigurable processors, heuristic, branch-and-bound.

\textsuperscript{*}Corresponding author. Tel.: +86 134 8765 8531
\textit{Email address: taoli.nudt@gmail.com}
1. Introduction

In recent years, embedded applications in cell phones, personal digital assistants, network routers, etc., have placed stringent demands on the performance and flexibility of the underlying computing devices. Reconfigurable processors have emerged to offer a favorable tradeoff between efficiency and flexibility, while providing for short design turnaround time. Reconfigurable processors can be considered as a hybrid between general-purpose embedded processors and reconfigurable devices (e.g., FPGAs). By integrating reconfigurable function units (RFUs), the core processor can be extended to support the execution of custom instructions, where each custom instruction features a cluster of basic instructions. A reconfigurable processor system may be optimally reconfigured for different applications, meeting the application constraints within the same chip [1]. Moreover, the software tool chain for reconfigurable processors only needs to be augmented rather than redeveloped, as most software codes of the applications still use the core instruction set. Thus, reconfigurable processors have been widely studied in the academia [2, 3], and there are a number of commercial reconfigurable processors today, e.g. Stretch S5000 [4] and NIOS processors [5].

As manually generating custom instructions for reconfigurable processors is a time-consuming and error-prone process, automation approaches are required to efficiently explore the potential gains of custom instructions under various micro-architectural constraints. Automatic custom-instruction generation generally includes three phases of custom-instruction identification, isomorphism detection and custom-instruction selection (covering) [6]. During the identification of custom instructions, a set of custom-instruction candidates is generated such that all candidates adhere to the given micro-architectural constraints. To enable hardware reuse, isomorphism detection checks isomorphism among those identified candidates. In the selection phase, a subset of the candidates are selected as custom instructions for implementation in RFUs. Although custom-instruction identification and isomorphism detection are also challenging problems, this paper focuses on the problem of custom-instruction selection.

Custom-instruction selection for reconfigurable processors determines a set of the most profitable custom instructions while ensuring those custom instructions can fit into the hardware area of RFUs. Several practical factors and constraints should be taken into consideration during the selection process. Firstly, the selection process should be recurrence-aware, as more gains can be derived by the hardware reuse of isomorphic custom-instruction instances. Secondly, all the selected custom instructions should meet the area requirement brought by the re-
configurable fabric of the reconfigurable processor. Thirdly, overlapping is not allowed among the selected custom-instruction instances. Although duplicating the same operations across multiple custom instructions makes overlapping instances feasible for selection, this can lead to increased power consumption [6]. Moreover, two custom-instruction instances generating a cycle in the data flow graph are not allowed to be selected, as they are not able to be scheduled in the instruction stream due to the data-dependency hazard.

By taking into account all the above-mentioned factors and constraints, we proposed a new computing model in this paper for the custom-instruction selection problem of reconfigurable processors. Based on the new computing model, a novel heuristic algorithm is presented to rapidly generate approximate solutions for large problems. In addition, we customized a tabu search algorithm to refine the approximate solutions to near-optimal ones. An exact algorithm based on a branch-and-bound method is proposed for optimally solving small-sized problems. Experimental results show that the proposed three algorithms are capable of providing high-quality solutions for problems with different sizes.

The rest of the paper is organized as follows. Related work in the area of custom-instruction selection is discussed in Section 2. The motivation of our work and the main contributions are presented in Section 3. The custom-instruction selection problem is formulated in Section 4. The proposed heuristic algorithm, tabu search algorithm and an exact algorithm are described in Section 5. The experimental setup is introduced in Section 6, and this is followed by the experimental results in Section 7. We conclude this paper in Section 8.

2. Related Work

Custom-instruction selection for reconfigurable processors is a computationally complex problem, as many different factors and constraints, such as area budget or acyclicity constraint, have influences over the selection process. Many previously reported work considered only a subset of the factors or ignore some of the constraints, such that the selection problem was simplified to a manageable one.

Some earlier work restricted the selection to simpler custom instructions. For example, by allowing operation duplication, work in [7] formulated the selection problem under area constraint as a 0-1 knapsack problem and proposed an exact dynamic programming algorithm for solving the problem. However, only custom instructions with multiple inputs and single output (MISO) were considered, while
it has been later shown that relaxing the constraint to multiple-output can further improve the speedup [8].

Some literatures adopt loose practical constraints for simplicity. The exact algorithm proposed in [9] simplified the selection problem by assuming that there are no overlapping and isomorphism among the custom instruction candidates. The work in [10, 11, 12] has not taken recurrence into account when evaluating the gain of a custom-instruction candidate. However, even large custom instructions consisting of 30 to 40 nodes have been shown to have multiple occurrences as stated in [6].

In [13] and [14], the selection problem was modeled as an extended subset-sum problem and a unate covering problem respectively. As the aim of their work is to minimize the number of new custom instructions used, they cannot employ an arbitrary evaluation measure for the custom-instruction’s gain. Classic code generation techniques, such as proposed in [15] and [16], can be utilized for solving the custom-instruction selection problem under overlapping and acyclicity constraints. However, these techniques are not applicable when area constraint is taken into account.

Data-path merging techniques [17, 18] can exploit resource sharing across custom instructions for area-efficient synthesis. However, the techniques usually require a lot of implementation effort and make custom-instruction scheduling in instruction stream more complicated. Nevertheless, it is worth mentioning that data-path merging techniques can be a useful supplement to the work in this paper.

To date, work in [6] considered recurrence, overlapping and acyclicity constraints together, which is most similar to this paper with respect to the computing model. Moreover, both proposed techniques can be generalized to support an arbitrary evaluation measure for the custom-instruction gain. However, there are several significant differences between our work and [6]. Firstly, our paper addresses area constraint while they considered the number of custom-instructions as a constraint. As our algorithms are targeted towards reconfigurable processors, area budget is an important and practical constraint due to the fixed-sized reconfigurable fabrics. Our algorithms are more generic and can be easily tuned for solving the problem in [6]. Moreover, the check for cyclic data-dependency relationship in [6], is incomplete as it is not performed for non-isomorphic custom-instruction candidates. This may render the utilization of custom instructions intractable.
3. Motivation and Contributions

Most of the reported work on custom-instruction selection were specially targeted towards ASIC-style extensible processors (e.g., Xtensa processor [19]), where the custom instructions are fixed after the custom function units (CFUs) have been fabricated. The CFUs can only be reused and cannot be re-customized for different applications. Therefore, these custom-instruction selection algorithms mainly consider the structure and frequency of custom-instruction occurrences for versatility. Additional properties of the custom instruction that are specific to its location in the application, are often not considered.

In recent years, the increasing prevalence of reconfigurable processors issues new challenges to automatic custom-instruction generation. Reconfigurable hardware, such as FPGAs, is more flexible and this enables the CFUs to be re-customized for different applications. Therefore, specific information with regard to the application to be accelerated can be exploited to obtain more profitable custom instructions. However, the RFUs in the reconfigurable processors usually have fixed size and it may not accommodate all the identified custom-instruction candidates. Therefore, area budget constraint should be taken into consideration during the custom-instruction selection process for reconfigurable processors.

In this paper, the custom-instruction selection techniques proposed are based on instances rather than patterns, where a instance contains information that is specific to its location in the application (e.g. position and data-dependency relationship). On the other hand, a pattern is a set of isomorphic instances that only consists of structure and frequency information. The problem of custom-instruction selection based on instances is more challenging as the problem size is larger.

The main contributions of this paper are as follows.

(1) A more practical computing model is proposed for the problem of custom-instruction selection to cater for real applications. In addition to the hardware area constraint, two practical constraints, i) overlapping constraint and ii) acyclicity constraint, are taken into consideration in the new computing model. The overlapping constraint takes into account the position information of the custom instruction, and acyclicity constraint considers the data-dependency relationship.

(2) A formal description is introduced for the custom-instruction selection problem based on instances. A novel heuristic algorithm is presented to rapidly generate high quality approximate solutions. An efficient tabu search algorithm is customized to further refine the approximate solutions to near-optimal ones. An exact algorithm based on a branch-and-bound method is proposed for obtaining
optimal solutions for small-sized problems.

(3) A thorough experimentation based on the proposed techniques is carried out. Experimental results show that the difference between the approximate solutions of our heuristic algorithm and the optimal ones is only about 3%, and this is further reduced by the tabu search to about 0.6% on average.

4. Problem Statement

A dataflow graph (DFG), $G(V, E)$, is a directed acyclic graph of a region (i.e., basic block, superblock or hyperblock) in an application. $V$ is the set of nodes which represent primitive operations or instructions, and $E$ is the set of the edges reflecting the data dependencies among these operations.

An instance $S$ is an induced sub-graph of $G$. There are $2^{|V|}$ possible instances in $G$, where $|V|$ is the number of nodes (operations). Due to the micro-architectural constraints, such as register-port constraints, only some of the instances are valid for custom-instruction selection. These valid instances can be found in the identification phase of custom-instruction generation.

Let $S_1, S_2, \ldots, S_n$ be the valid instances found in the identification phase. Some of them are isomorphic, for example, the instances $S_1$ and $S_2$ in Fig. 1. In order to enable hardware reuse, isomorphism detection must be carried out before custom-instruction selection. After isomorphism detection, these instances are classified into $m$ patterns, denoted as $P_1, P_2, \ldots, P_m$. Note that, isomorphic instances may overlap, such as the instances $S_1$ and $S_2$ of the pattern $P_1$ in Fig. 1.

![Figure 1: Isomorphic instances.](image_url)

Custom-instruction selection for reconfigurable processors chooses a set of valid instances that are subjected to one or more constraints, in order to maximize the performance improvement by implementing them in RFUs. In this paper,
the performance improvement is estimated by the saved execution cycles of the applications. The following notations are used to describe the custom-instruction selection problem.

- \( g_i \) denotes the gain (i.e., the saved execution cycles) obtained by implementing the instance \( S_i \) in hardware RFU as opposed to software, \( 1 \leq i \leq n \). All the instances in a pattern have the same gain as they are isomorphic.
- \( a_i \) denotes hardware area requirement of instance \( S_i \), \( 1 \leq i \leq n \).
- \( a'_j \) denotes hardware area requirement of pattern \( P_j \), \( 1 \leq j \leq m \). Note that, \( \forall S_i \in P_j, a'_j = a_i \).
- \( o_i \) denotes the number of instances that overlap with instance \( S_i \), \( 1 \leq i \leq n \).
- \( M_i \) denotes the set of the nodes covered by instance \( S_i \), \( 1 \leq i \leq n \).
- \( \text{Pred}_i \) denotes the predecessor node set of instance \( S_i \), \( 1 \leq i \leq n \).
- \( \text{Succ}_i \) denotes the successor node set of instance \( S_i \), \( 1 \leq i \leq n \).

We use \( x_i \) (\( x_i \in \{1, 0\} \)) to indicate whether the instance \( S_i \) is selected as a custom instruction. \( x_i = 1 \) (\( x_i = 0 \)) indicates that the instance is (not) selected. Let \( E(x_1, x_2, \ldots, x_n) \) be the corresponding gain of the current solution. The gain can be calculated as

\[
E(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} g_i x_i.
\]

Note that the recurrences of custom instructions have been addressed implicitly in the gain calculation of the selected custom instructions, as the calculation is based on individual instances instead of patterns.

Let \( \Gamma \) be the index set of the selected instances, i.e., \( \Gamma = \{i| x_i = 1, 1 \leq i \leq n\} \). Given the available hardware area \( A \) for implementing custom instructions, the practical constraints considered in the computing model for custom-instruction selection are as follows.

- **Area constraint**: To reduce the cost and the delay associated with long wires, the chip area used for reconfigurable fabric is usually relatively small and it may not be able to accommodate all the identified patterns. Thus, the total area of the selected patterns cannot exceed the given hardware
area limitation $A$, i.e., $\sum_{j=1}^{m} a_j' z_j \leq A$, where $z_j = 1 - \prod_{i \in P_j} (1 - x_i)$. $z_j = 1$ indicates that one or more instance of the pattern $P_j$ are selected, while $z_j = 0$ indicates that no instance in $P_j$ is selected.

- **Overlapping constraint**: Two custom instructions may contain the same nodes in a DFG, such as $S_2$ and $S_3$ in Fig. 1. Normally, this is not allowed as operation duplication makes the custom-instruction mapping rather complicated and a very limited gain can be achieved. To prevent the selection of instances that overlap, the intersection of the sets of the nodes among the selected instances must be empty, i.e., $\bigcap_{i \in \Gamma} M_i = \emptyset$.

- **Acyclicity constraint**: The two instances $S_1$ and $S_2$ in Fig. 2, provide data to each other and thus generate a cycle. Such instances cannot be implemented as custom instructions together. We employ the following formula

$$\forall i, j \in \Gamma, M_i \cap Pred_j = \emptyset \text{ or } M_i \cap Succ_j = \emptyset$$

to detect such cases. The correctness of the formula will be proven later in this paper.

![Diagram](image.png)

**Figure 2**: Cyclic instances.

The custom-instruction selection problem ($SP$) discussed in this paper can be formalized as:
\[
SP : \begin{cases}
\text{maximize} & \sum_{i=1}^{n} g_{i}x_{i} \\
\text{subject to} & \sum_{j=1}^{m} a_{j}^{'z_{j}} \leq A, \\
\quad & \bigcap_{i \in \Gamma} M_{i} = \emptyset, \\
\quad & M_{i} \cap \text{Pred}_{j} = \emptyset \text{ or } M_{i} \cap \text{Succ}_{j} = \emptyset, \\
\forall i, j \in \Gamma.
\end{cases}
\]

5. Proposed Algorithms

5.1. Heuristic Algorithm

It can be observed that the selection problem is closely related to the 0-1 knapsack problem. We are thus motivated to construct a heuristic algorithm for solving the problem \( SP \) by extending the greedy idea for the 0-1 knapsack problem.

It is noteworthy that our greedy strategy is based on the whole pattern (i.e., a set of isomorphic instances), instead of one instance. This is because once an instance is selected for hardware implementation, all the other promising instances of the same pattern can also be selected as these isomorphic instances can lead to higher gains without extra area requirement.

Now, we construct two priority ratios for the pattern \( P_{i} \). One is the ratio of potential-gain to area, and the other is the ratio of potential-gain to overlapping.

Let \( g_{i}' \) be the potential gain of the pattern \( P_{i} \). \( g_{i}' \) can be calculated as the sum of the gains of all the promising instances in \( P_{i} \). We define the ratio of potential-gain to area as

\[
\rho_{i}^{(a)} = \frac{g_{i}'}{a_{i}}
\]

to evaluate the priority of the pattern \( P_{i} \). Initially, all the instances in \( P_{i} \) are promising instances. If some of them overlap or form a cycle with the selected instances during the selection process, these promising instances become unpromising. Thus, the value of \( g_{i}' \) may decrease, resulting in the decrease of \( \rho_{i}^{(a)} \).

Similarly, we define the ratio of potential-gain to overlapping as

\[
\rho_{i}^{(o)} = \frac{g_{i}'}{a_{i}'+1},
\]

\(^{1}\)A promising instance is an instance that does not overlap or form a cycle with the already selected instances.
where \( o'_i \) is the approximate number of instances that overlapped with the pattern \( P_i \). Note that \( o'_i \) may be 0, as it is calculated by \( \sum_{j \in I_i} o_j \), where \( I_i \) is the index set of all the promising instances in \( P_i \). \( r'_i^{(a)} \) may also change during the selection process based on the same reason as \( r_i^{(a)} \).

In our algorithm, we utilize the two ratios to determine which pattern has the higher priority for selection as custom instruction. It is obvious that the pattern \( P_i \) has higher priority than \( P_j \) if both \( r_i^{(a)} \) and \( r_i^{(o)} \) of pattern \( P_i \) are larger than those of \( P_j \). However, when one ratio of \( P_i \) is larger than that of \( P_j \) and the other ratio of \( P_i \) is smaller than that of \( P_j \), it is difficult to determine which pattern is of higher priority. Thus, we define a function \( Pri(i, j, \alpha) \) to determine the priority for pattern selection according to the two priority ratios. The function returns the pattern with higher priority.

Given two patterns \( P_i \) and \( P_j \), let \( p = r_i^{(a)}/r_j^{(a)} \) and \( q = r_i^{(o)}/r_j^{(o)} \),

\[
Pri(i, j, \alpha) = \begin{cases} 
P_i, & p > 1 \text{ and } q > 1; \\
P_i, & p > 1, q \leq 1 \text{ and } \alpha p > (1 - \alpha)/q; \\
P_i, & p \leq 1, q > 1 \text{ and } (1 - \alpha)q > \alpha/p; \\
P_j, & \text{else.}
\end{cases}
\]

where \( \alpha \in [0, 1] \) is an adjustable parameter. The parameter is used to weigh the two ratios in the evaluation of the priorities of the patterns. Note that if \( \alpha \) equals to 0, the ratio \( r_i^{(a)} \) is not taken into consideration and the priority of pattern \( P_i \) is only determined by the ratio \( r_i^{(o)} \). On the other hand, if \( \alpha \) equals to 1, the priority of the pattern \( P_i \) only depends on the ratio \( r_i^{(a)} \).

The outline of the proposed algorithm is as follows. Initially, the two priority ratios \( r_i^{(a)} \) and \( r_i^{(o)} \) are calculated for the pattern \( P_i \), \( i = 1, 2, \ldots, m \). These patterns are then sorted utilizing the function \( Pri(i, j, \alpha) \). After that, the pattern with the highest priority is iteratively extracted from the pattern candidate set. If some of the instances in the pattern are not promising instances, they are removed from the pattern and the two ratios of the pattern are updated. Next, the pattern is reinserted into the pattern candidate set for the next iteration. Otherwise, if all the instances in the pattern are promising, these instances can be selected directly. The iterative selection process terminates when the pattern candidate set is empty or when there is no residual area for accommodating more patterns.

Details of the heuristic algorithm is shown below.

**Input:** \( a'_i \) – source data of the pattern \( P_i \), \( 1 \leq i \leq m \);  
\( M_i, Pred_i, Succ_i, g_i \) – source data of the instance \( S_i \), \( 1 \leq i \leq n \);
A – hardware area limitation.

**Output:** The heuristic solution \((x_1, x_2, \ldots, x_n)\).

**Algorithm HEA**

/* A heuristic algorithm for custom-instruction selection. */

**begin**

/* Initialization */

\(residual\_area := A; \text{ Heap } H := \{\};\)

1 for \(i := 1 \text{ to } n\) do

1.1 \(x_i := 0;\)

1.2 \(o_i := 0; /* the number of the overlapped instances with } S_i */\)

1.3 for \(j := 1 \text{ to } n\) do

\[\text{if } M_i \cap M_j \neq \emptyset /* S_i \text{ is overlapped with } S_j */\]

\(o_i := o_i + 1;\)

2 for \(i := 1 \text{ to } m\) do

2.1 \(g'_i := 0 \text{ and } o'_i := 0;\)

2.2 Sort the instances of } P_i \text{ in descending order in terms of the } o_i \text{ value};

/* Remove the instances violating the other instances of } P_i */

2.3 for each instance } S_j \text{ of } P_i \text{ do}

\[\text{if } S_j \text{ overlaps or forms a cycle with other instances in } P_i \text{ then}
\]

Remove } S_j \text{ from } P_i;

else \(g'_i := g'_i + g_j \text{ and } o'_i := o'_i + o_j;\)

/* initial the priority ratios of } P_i */

2.4 \(r^{(a)}_i := g'_i/a'_i \text{ and } r^{(o)}_i := g'_i/(o'_i + 1);\)

2.5 Insert the 3-tuple \(<P_i, r^{(a)}_i, r^{(o)}_i>\) into

Heap } H;

3 Build Heap } H;

/* Selection process begins here! */

4 while \((residual\_area > 0) \text{ and } (H \neq \{\})\) do

4.1 \(need\_reinsert := 0;\)

4.2 Extract the top element of } H: \(<P_t, r^{(a)}_t, r^{(o)}_t>\);

4.3 if \(a'_t \leq residual\_area\) then

4.3.1 for each instance } S_i \text{ in } P_t \text{ do}

\[\text{if } S_i \text{ is not a promising instance then } \}

Delete } S_i \text{ from } P_t;

\(need\_reinsert := 1; \}

11
4.3.2 if need_reinsert = 1 then {
    Update $r_i^{(a)}$ and $r_i^{(o)}$;
    Re-insert the updated 3-tuple <$P_t$, $r_i^{(a)}$, $r_i^{(o)}$> into $H$;
    Heapifying $H$;
}
else {
    residual_area := residual_area − $a'_t$;
    for each instance $S_i$ in $P_t$ do $x_i := 1$;
}

/* Output heuristic solution. */
5  Output ($x_1, x_2, \cdots, x_n$);
end.

Heap structure is utilized to manage and sort all the patterns in our heuristic algorithm. As mentioned before, the two ratios $r_i^{(a)}$ and $r_i^{(o)}$ may vary during the selection process. As shown in step 4.3, some instances may be removed from the pattern $P_t$, which is the top element of the heap $H$, due to their violation of the overlapping and acyclicity constraints. Thus, the values of $r_i^{(a)}$ and $r_i^{(o)}$ for the current “best” pattern $P_t$ need to be recalculated. This may cause the pattern $P_t$ to become “non-best”, and therefore the heap needs heapifying. The complexity of heapifying is only $O(\log m)$ compared to the complexity of common sorting algorithms which is $O(m \cdot \log m)$, where $m$ is the number of the elements (patterns) in the heap.

In the algorithm HEA, there are several techniques that can make the selection process faster and better. The first is sorting all the instances of each pattern in descending order in terms of $o_i$ as shown in step 2.2. Thus, among the isomorphic instances, the instances that overlap with more instances will be removed from the pattern earlier, as shown in steps 2.3 and 4.3. The remaining instance candidates are the ones that overlap with fewer instances, which provides more opportunity for obtaining higher gain in the selection process. In addition, promising instance check in step 4.3 is an important step in HEA, which consists of overlapping check and cyclicity check. To check whether the instance $S_i$ overlaps with the selected instances, a covered node set is maintained for recording the nodes covered by the already selected instances. Based on the intersection of the node set of $S_i$ and the covered node set, the overlapping violation of $S_i$ can be easily detected. In comparison, the cyclicity check is more complicated than the overlapping check. The predecessor and successor information of the instances is thus utilized to accelerate the checking process. Our strategy for the cyclicity check is based on the following theorem.

Theorem 1: Two instances $S_i$ and $S_j$ in a DFG are acyclic if and only if $M_i \cap$
Proof: We prove the theorem by contradiction. $S_i$ and $S_j$ are cyclic if and only if there exist at least two different paths, one path is from $S_i$ to $S_j$ and the other is from $S_j$ to $S_i$. This implies that $S_i$ has at least one node that is a predecessor of $S_j$ and it also has at least one node that is a successor of $S_j$. In other words, $S_i$ and $S_j$ are cyclic if and only if $M_i \cap \text{Pred}_j \neq \emptyset$ and $M_i \cap \text{Succ}_j \neq \emptyset$. This concludes the theorem.

For cyclicity check, the predecessor and successor information of instances is precalculated in $O(|V|^3)$, where $|V|$ is the number of the nodes in DFG. In addition, the intersection operations for each pair of instances needs $O(|V|)$ time in cyclicity check process according to Theorem 1. The number of selected instances is less than $|V|$ due to the overlapping constraint. Therefore, the worst-case complexity of cyclicity check for each instance is only $O(|V|^2)$. It is worthwhile to mention that the predecessor and successor information can be directly obtained without additional calculation in HEA, but this requires additional computation in the adopted identification algorithm of [20].

In conclusion, for the worst case, the preprocessing of predecessor and successor information needs $O(|V|^3)$, step 1 runs in $O(n^2 \cdot |V|)$, step 2 runs in $O(n \cdot \log n)$, step 3 runs in $O(m \cdot \log m)$ and step 4 runs in $O(n^2 \cdot |V|^2)$, where $n$ is the number of the instances and $m$ is the number of the patterns. Generally, $n < m$ and $n^2 > |V|$, and thus the time complexity of the algorithm HEA is bounded by $O(n^2 \cdot |V|^2)$.

5.2. Refining Heuristic Solution via Tabu Search

Tabu Search (TS) is one of the traditional heuristic-based algorithms that is used to search for the global optimal solution in NP-hard problems [21]. In this subsection, we describe how we have customized a TS algorithm, denoted as TSA, to refine the heuristic solution generated by HEA.

Generally, TS consists of five primary parameters: local search procedure, neighborhood structure, tabu conditions, aspiration criterion and stopping rule. In the searching process, TS keeps a list (tabu list) of the search moves during each iteration, in order to restrict the local search procedure in reusing those moves. A recency-based memory stores the recent search areas in order to avoid being trapped in the local area. The tabu status of a search move can be released at a certain time according to the size of the recency-based memory. A frequency-based memory is used to store the frequency of the searching in each area. We also applied it to diversify the searching. Also, an aspiration criterion is utilized so that if a tabu move generates a better solution than all the feasible solutions obtained so far, its tabu-status is ignored. Meanwhile, the corresponding tabu area
becomes feasible for searching again. The stopping rule of the tabu search may be a fixed number of iterations, a fixed amount of CPU time, a fixed number of consecutive iterations without an improvement in the best solution value, etc. 

A feasible solution

Randomly choose a bit for flipping

\[ \begin{array}{cccccccc}
0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
\end{array} \]

A neighbor solution

\[ \begin{array}{cccccccc}
0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\
\end{array} \]

Figure 3: Generating a neighbor of a feasible solution.

In our algorithm TSA, as shown in Fig. 3, we generate a neighbor of a given feasible solution by randomly changing (flipping) one bit in the solution, instead of more complex neighbor generation methods, such as changing two bits each time. The complex neighbor generation methods are not as efficient as the one we adopt, because feasible neighbor check becomes intractable due to the constraints in the problem \( SP \). During the neighbor generation process, if the generated neighbor violates the constraints in the problem \( SP \), it is neglected and we try to flip another bit until a feasible neighbor is obtained. It is noteworthy that for the problem \( SP \), a feasible neighbor can be definitely derived from a feasible non-zero solution, because a solution derived by removing one selected instance from an initial feasible solution still fulfills the constraints in \( SP \), i.e., a bit ‘1’ in the initial solution is flipped to bit ‘0’.

Throughout this paper, \( x \) indicates a vector in n-dimensional space \( \{0, 1\}^n \). Let \( x_{neib} \) be a neighbor of a local solution \( x_{local} \). We define a function \( dcost \) as:

\[
dcost(x_{neib}, x_{local}) = E(x_{local}) - E(x_{neib}).
\]

It is obvious that the smaller the value of \( dcost \) is, the better quality the neighbor \( x_{neib} \) has.

To better understand how TSA works on recency-based memory and frequency-based memory, we employ the example shown in Fig. 4 to illustrate the first 4 iterations. For simplicity, the neighborhood size is set to 2 and the tabu tenure (length of the tabu list) is set to 2, i.e., a flipped bit only can be reflipped after 2 iterations. The solution consists of 10 bits.
In iteration 1, the first neighbor is generated by flipping bit ‘1’, and the second neighbor is derived by flipping bit ‘2’. Then we calculate their \( dcost \)s (assuming they are \(-1\) and \(2\) respectively in this example). After that, we choose the neighbor 1 as \( x_{local} \) because it has smaller \( dcost \) value. At the same time, the frequency-based memory is updated, i.e., the value at position 1 in the table increases by 1. Also, the flipped bit ‘1’ is recorded in recency-based memory and it is put into the tabu list, which is initially empty, in first in first out (FIFO) manner.

In iteration 2, we generate two neighbors with the flipping bits ‘7’ and ‘3’. Here, the \( dcost \) values of the two neighbors are assumed to be 4 and 2 respectively. When all non-tabu neighbors have \( dcost \) larger than 0, an award rule is applied in order to encourage searching a new area. In this example, \(-Q\) is awarded to the neighbors as the bits ‘7’ and ‘3’ have never been flipped, where \( Q \) is a given positive value. The second neighbor is chosen as \( x_{local} \) after the award is applied. The corresponding memories are then updated.

In iteration 3, the two neighbors are shown as bits ‘6’ and ‘1’. Noting that bit
‘1’ is already in the tabu list, we do not choose the second neighbor as $x_{\text{local}}$ even if its $d_{\text{cost}}$ is smaller than that of the first neighbor. $x_{\text{local}}$ is set to the first neighbor with bit ‘6’ flipped. For the same reason, the first neighbor in iteration 4 is not selected as $x_{\text{local}}$. It is noteworthy that the second neighbor with bit ‘1’ flipped is selected in this iteration, as the bit ‘1’ has been released from the tabu list in this iteration according to the tabu tenure.

If a tabu-active feasible solution is better than the best-so-far solution, an aspiration criterion is applied, such that the tabu-status of the solution is ignored.

In the whole searching process, a neighbor $x_{\text{neib}}$ may be in and out of the tabu list multiple times. Assuming that the latest entrance of $x_{\text{neib}}$ is in iteration $\text{iter} \_\text{late}(x_{\text{neib}})$ and the current searching is in iteration $\text{iter} \_\text{curr}$, the tabu degree of $x_{\text{neib}}$, denoted as $T_{\text{deg}}(x_{\text{neib}})$ is defined as

$$T_{\text{deg}}(x_{\text{neib}}) = \text{iter} \_\text{late}(x_{\text{neib}}) + \text{tabu} \_\text{tenure} - \text{iter} \_\text{curr}.$$ 

Tabu degree is updated for each neighbor in each iteration. A non-negative value of tabu degree implies that the neighbor is tabu-active, while a negative one implies that the neighbor is not tabu-active. The outline of the algorithm TSA is as follows.

**Input:** $x_{\text{heur}}$ – The heuristic solution generated by the algorithm HEA;  
**Output:** $x_{\text{best-so-far}}$ – the best solution found by Tabu Search;  
**Algorithm** TSA  
/* Tabu Search Algorithm for the problem $SP$. */

Iter – fixed number of consecutive iterations that cannot get improvement in terms of gain. Iter is used to terminate the tabu search.  
$q$ – the size of neighborhood list. */

begin
1. $x_{\text{local}} := x_{\text{heur}}$; $x_{\text{best-so-far}} := x_{\text{heur}}$; $i := 0$;
2. while $i < \text{Iter}$ do begin
   2.1 Generate $q$ neighbors of $x_{\text{local}}$;
   2.2 Update the degrees and $d_{\text{cost}}$s of the $q$ neighbors;
   2.3 if all $q$ neighbors are tabu-active then
      $x_{\text{local}} :=$ the neighbor with the minimal tabu degree;
   else $x_{\text{local}} :=$ the neighbor with the minimal
2.4 if $E(x_{\text{local}}) > E(x_{\text{best-so-far}})$ then

\[ x_{\text{best-so-far}} := x_{\text{local}} \text{ and } i := 0; \]

else \( i := i + 1; \)

2.5 Update frequency-based memory;

2.6 Update recency-based memory; /*put an item into tabu list*/

end:

end.

It is worthwhile to point out that, TSA starts with the heuristic solution generated by HEA. The heuristic solution refined by TSA is usually better than the initial solution provided by HEA. This is because once TSA finds a better local solution, the best-so-far solution is updated according to step 2.4.

5.3. A Branch and Bound Algorithm

We now present an exact algorithm, denoted as BnB in this subsection, based on branch and bound to compute the optimal solution for the problem $SP$.

The input to BnB is the related source data of the identified instances and the area limitation. The output of this algorithm is the optimal solution $x_{\text{opt}}$ which maximizes the gain. Let $g''_i$ be the maximum gain that can be obtained by implementing the instance $S_i$ in RFU. $g''_i$ can be calculated using $c_i g_i$, where $c_i$ is the number of instances isomorphic to the instance $S_i$. Note that, $S_i$ is isomorphic to itself. Initially, we sort all the instances in descending order in terms of $g''_i a_i$. For instances with the same value of $g''_i a_i$, we sort them in ascending order in terms of $a_i$. Note that the instances of the same pattern have the same values of $g''_i$ and $a_i$. These isomorphic instances are arranged consecutively in the sorted list. After initialization, we sequentially consider each instance as they are ordered in the sorted list by invoking standard recursive branch-and-bound procedures. Each instance has two cases to be examined. One case considers the current instance, denoted as $S_{\text{index}}$, to be included into the set of the selected instances, and the other considers $S_{\text{index}}$ to be excluded from the set of selected instances. The two cases correspond to the two branches of the searching tree. For the former branch, $S_{\text{index}}$ is checked for area violation and whether it is a promising instance. If either check fails, the branch is pruned immediately. Otherwise, the upper bound gain that can be obtained through this branch is calculated. We backtrack if the upper bound does not exceed the best gain obtained so far because this branch is not able to lead to an improved solution. For the latter branch where $S_{\text{index}}$ is
excluded from the selected instances set, the area constraint check and the promising instance check will definitely pass. Therefore, it is not necessary to execute the checks in this case. However, the calculation for the upper bound gain is still required for pruning.

The greedy idea of Dantzig bound [22] is adopted to calculate a tight upper bound of gain, denoted as ceiling\_gain, for pruning. For the case that $S_{\text{index}}$ is selected, we repeatedly select instances $S_i$ (where $\text{index} < i \leq n$), as long as $S_i$ fits into the residual area or it does not occupy additional unused area. Note that due to hardware reuse, if one or more instances of a pattern have been selected, the other instances of the pattern need not occupy additional area after being selected. The first instance which cannot be selected due to the area constraint is called break instance, denoted as $S_b$. Thus, ceiling\_gain is calculated as follows.

$$
\text{ceiling\_gain} = \left\lceil E(x_{\text{cur}}) + \sum_{i=\text{index}+1}^{b-1} g_i + r_{\text{area}} \frac{c_b g_b}{d_b} \right\rceil,
$$

where $x_{\text{cur}}$ in this case represents the current solution after $S_{\text{index}}$ is selected, and $r_{\text{area}}$ is the residual area after selecting instances from $S_{\text{index}+1}$ to $S_b-1$.

On the other hand, for the case that $S_{\text{index}}$ is not selected, the main calculation process for ceiling\_gain is similar. The only difference is that the $g''_i$ value of the instance $S_i$ that is isomorphic with $S_{\text{index}}$ decreases for each $i$, because the instance $S_{\text{index}}$ is excluded from the selected instance set. Therefore, unless some isomorphic instances of $S_{\text{index}}$ have been selected before excluding $S_{\text{index}}$, the sorted list of the remaining instances need to be reordered before the calculation of ceiling\_gain. This is to guarantee the correctness of the algorithm BnB. Moreover, the lower bound of the gain in BnB is the best gain obtained so far, denoted as floor\_gain. Note that the initial lower bound of floor\_gain is set to the gain of the heuristic solution derived from HEA, in order to quickly prune unnecessary branches of the searching tree.

To better understand how BnB works, Fig. 5 shows an example, which consists of 5 instances (3 patterns), to illustrate the searching tree of the algorithm. For simplicity, the overlapping constraint and acyclicity constraint are not taken into consideration in this example. In Fig. 5, the searching tree is traversed in a depth first search manner. Each level of the branching tree evaluates the effects when the instance is selected and when the instance is not selected. Each box corresponds to one procedure call while each downgoing arrow determines one invocation of the branch-and-bound procedure. Each upgoing arrow illustrates the return from a procedure.
As shown in Fig. 5, after initialization, a procedure is invoked to examine the case that $S_1$ is selected. The current gain and the area requirement of the current solution $x_{cur}$ are calculated, as shown in the left box of the first level, i.e., $E(x_{cur}) = 3$ and $area(x_{cur}) = 3$. The upper bound of ceiling_gain is also calculated using the formulation that was introduced earlier, while the lower bound floor_gain is initialized to $E(x_{HEA})$, which is the gain derived by the HEA solution. Next, another recursive procedure is invoked (along the downgoing arrow 2) for the case that $S_2$ is also selected. The gain obtained for this case increases to 6, while the area requirement is still 3 due to hardware reuse between the two instances $S_1$ and $S_2$. The ceiling_gain is 15, while the floor_gain is still 10. The box on the bottom left shows the case that $S_1$, $S_2$ and $S_3$ are all selected for hardware implementation. For this case, the current gain obtained increases to 15, floor_gain is then updated to 15, because the current best gain $E(x_{best-so-far})$ is 15 and it is larger than the initial value of floor_gain. We then backtrack as the ceiling_gain does not exceed the floor_gain, which implies that the branch will not lead to a better solution.
For more details, we describe the branch and bound algorithm as follows.

**Input:** $M_i$, $Pred_i$, $Succ_i$, $g_i$ – source data of the instance $S_i$, $1 \leq i \leq n$;
$A$ – hardware area limitation.

**Output:** The optimal solution $x_{opt}$

Algorithm BnB

/* An exact algorithm for custom-instruction selection. */

**begin**

1 Sort all the instances in terms of $\frac{g_i}{a_i}$;
/* Initialize the best solution so far and the current solution examined. */

2 $x_{best-so-far} := 0$ and $x_{cur} := 0$; /* $\theta = (0,0,\ldots,0)$; */
/* Initialize the lower bound of gain. */

3 $floor\_gain := E(x_{heur})$;

4 Call $Select(1, true, x_{cur})$;

5 Call $Select(1, false, x_{cur})$;
/* Output optimal solution. */

6 $x_{opt} := x_{best-so-far}$;

**end**

**Procedure Select(index, selected, x_{cur})**

**begin**

/* Handle the case that current instance is selected. */

1 if $selected = true$ then

1.1 $x_{index} := 1$; /* $x_{cur} := (x_{1,cur}, x_{2,cur}, \ldots, x_{n,cur})$ */
/* Check the constraints and the bound. */

1.2 if $area(x_{cur}) > A$ then return;

1.3 if $S_{index}$ is not a promising instance then return;
/* Update best-so-far solution. */

1.4 if $E(x_{cur}) > E(x_{best-so-far})$ then {

$x_{best-so-far} := x_{cur}$;
$floor\_gain := E(x_{best-so-far})$;
}

1.5 if $Upb(x_{cur}, index, selected) \leq floor\_gain$ then return;
/* Handle the case that current instance is not selected. */

2 if $selected = false$ then

2.1 if $Upb(x_{cur}, index, selected) \leq floor\_gain$ then
return;
/* All instances have been examined. */
3 if index ≤ n then return;
/* Build two new branches. */
4 call Select(index + 1, true, x_cur);
5 call Select(index + 1, false, x_cur);
end

In the procedure Select(), step 1 examines the case that current instance \( S_{index} \) is included into the set of selected instances, while step 2 handles the case that \( S_{index} \) is excluded from the set of selected instances. Step 3 is used to terminate the search after all the instances has been examined, while step 4 and 5 are invoked to consider the next instance in the sorted list. Note that step 1.3 checks whether \( S_{index} \) is a promising instance, i.e., whether the instance \( S_{index} \) violates the overlapping constraint or acyclicity constraint. The methods employed for the check is the same as in step 4.3 of algorithm HEA. The value of floor\_gain may increase after \( x_{best-so-far} \) is updated in step 1.4. In step 1.5 and step 2.1, function Upb() is used to calculate the upper bound of the gain, i.e., ceiling\_gain.

6. Experimental Setup

Let \( A(S) \) indicate the hardware area required to implement the instance \( S \), and let \( G(S) \) indicate the gain of the instance \( S \). As mention before, \( G(S) \) is estimated using the total number of execution cycles saved by implementing the custom-instruction instance \( S \). It can be calculated by

\[
G(S) = L_{sw}^S - L_{hw}^S,
\]

where \( L_{sw}^S \) and \( L_{hw}^S \) are the latencies of \( S \) when implemented in software and in hardware respectively.

In our experiments, we employ a widely used method (e.g. in [12, 23, 24]) to estimate \( G(S) \). Let \( \tau_{sw} \) be the estimated latency of the execution stage in the processor pipeline. \( \tau_{hw} \) and \( \tau_{area} \) represent the latency and hardware area requirement to implement the corresponding operation, respectively. The standard software latencies of HPL-PD architecture [25] are used in our experiments. \( \tau_{hw} \) and \( \tau_{area} \) have been obtained by synthesizing arithmetic and logic operations on a 0.13\( \mu m \) CMOS process [26]. They are then normalized to a 32-bit multiply accumulator. Table 1 shows the normalized hardware latency (\( \tau_{hw} \)) and area requirement (\( \tau_{area} \)) of some arithmetic and logic operations.

21
Table 1: Examples of hardware timing and area models of some operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Precision</th>
<th>Relative Latency</th>
<th>Relative Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC</td>
<td>$32 \times 32 + 64$</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Adder</td>
<td>$32 + 32$</td>
<td>0.31</td>
<td>0.10</td>
</tr>
<tr>
<td>Multiplication</td>
<td>$32 \times 32$</td>
<td>0.96</td>
<td>0.98</td>
</tr>
<tr>
<td>Divider</td>
<td>$32/32$</td>
<td>7.45</td>
<td>2.48</td>
</tr>
<tr>
<td>Barrel shifter</td>
<td>32</td>
<td>0.23</td>
<td>0.12</td>
</tr>
<tr>
<td>Bitwise AND/OR</td>
<td>32</td>
<td>0.03</td>
<td>0.02</td>
</tr>
</tbody>
</table>

For simplicity, the core processor is assumed to be single-issued and pipelined. The estimated software execution time of an instance $S$ ($L_{sw}^S$) can be calculated as follows.

$$L_{sw}^S = \sum_{i \in S} (\tau_{sw})_i.$$  

In hardware, the latency of an instance $S$ depends on the nodes lying on the critical path of the instance, while the area requirement depends on all the operations of the instance. Therefore,

$$L_{hw}^S = \sum_{i \in C(S)} (\tau_{hw})_i, \quad A(S) = \sum_{i \in S} (\tau_{area})_i.$$  

Note that the proposed algorithms in this paper are independent of the evaluation models of $G(S)$ and $A(S)$, and hence they can cater to other evaluation models.

7. Experimental Results

The proposed algorithms HEA, TSA, and BnB were implemented in C language on an Intel Core2 2.66G CPU with 2GB memory. The pentium time stamp counter is used for measuring the runtime of the algorithms. To optimize the runtime, bit vectors were used to store instances and the related source data information, as they are ideal for performing operations (e.g., intersection) on sets.

As the well-known 80-20 rule asserts that 80% of execution time is consumed by about 20% of an application’s code, our experiments focus on the hottest regions (which exceed 10% of the total execution cycles of the application) in the applications. It is worth mentioning that the proposed algorithms can be easily tuned to select custom instructions beyond region boundaries.
Table 2: Characteristics of the benchmarks

<table>
<thead>
<tr>
<th>DFG</th>
<th>Benchmark</th>
<th>Procedure</th>
<th>Region No.</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bitcount</td>
<td>_bit_shifter_7,62</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>ADPCM dec.</td>
<td>_adpcm_decoder</td>
<td>8</td>
<td>48</td>
</tr>
<tr>
<td>3</td>
<td>sha</td>
<td>_sha_transform_3,8</td>
<td>8</td>
<td>70</td>
</tr>
<tr>
<td>4</td>
<td>ADPCM enc.</td>
<td>_adpcm_coder</td>
<td>4</td>
<td>178</td>
</tr>
<tr>
<td>5</td>
<td>h264 enc.</td>
<td>_SetupFastFullPelSearch</td>
<td>128</td>
<td>222</td>
</tr>
<tr>
<td>6</td>
<td>jpeg</td>
<td>_rgb_ycc_convert_7,9</td>
<td>8</td>
<td>229</td>
</tr>
<tr>
<td>7</td>
<td>rijndael enc</td>
<td>_encrypt</td>
<td>15035</td>
<td>196</td>
</tr>
<tr>
<td>8</td>
<td>blowfish</td>
<td>_BF_encrypt</td>
<td>2</td>
<td>380</td>
</tr>
</tbody>
</table>

Table 2 describes the sources of the DFGs used in our experiments, where the size shown in the rightmost column represents the number of nodes (i.e., operations) in the DFG. The DFGs are obtained from the MediaBench [27] and MiBench [28] benchmarks. These benchmarks were compiled and simulated by the Trimaran compiler infrastructure [29]. After profiling, the hottest regions are identified and selected as the input to our algorithms. Next, the identification phase [20] is carried out for enumerating the valid instances. The vflib2 graph-matching library [30] is then used for isomorphism detection among the enumerated instances.

The algorithm HEA is executed multiple times, in which the weight parameter $\alpha$ varies from 0 to 1 with an interval of 0.2. The heuristic solution is the best solution derived from these iterations. In most of the previously reported heuristic algorithms for the selection problem, such as [7, 8], only gain/cost ratio is considered in assigning priorities to the custom-instruction instances. However, in the proposed HEA, the overlapping information of the instances is also taken into consideration. To compare the two strategies for assigning priorities, we implemented the former, denoted as OLDHEA, by simply modifying the proposed algorithm HEA. To evaluate the improvements of HEA over OLDHEA, $imp$ is defined below.

$$imp = \frac{E(x_{HEA}) - E(x_{OLDHEA})}{E(x_{OLDHEA})} \times 100\%.$$  

Fig. 6 shows the averaged $imp$ of the benchmarks with different I/O constraints and different area limitations. We can observe that the strategy for assigning priorities in HEA outperforms OLDHEA for all the benchmarks, especially for the benchmark rijndael-large-enc. This is because in HEA, we utilize not only the gain/area information of the instances but also the overlapping information of the
instances to guide the selection process. The instances that are overlapped with fewer instances are assigned higher priorities. This provides more opportunity for obtaining higher gain, as more possible promising instances are left in the subsequent selection process after higher-priority instances are selected. We also observe that more improvements can be obtained on relatively large benchmarks, such as the four rightmost benchmarks shown in Fig. 6. This shows that HEA is suitable for solving large-sized problems.

![Figure 6: Averaged improvements in the gain of HEA over OLDHEA.](image)

In our experiments, we set two different initial solutions for TSA, in order to show that a high-quality initial solution is important for the performance of TSA. One is $0 = (0, 0, \cdots, 0)$, and the other is the solution of HEA. In this paper, TSA(zero) and TSA(heur) indicate the TSAs with the two different initial solutions, respectively. Without loss of generality, the neighborhood size is set to $n/2$, where $n$ is the number of instances. Tabu tenure is set to 10, i.e., a flipped bit stays in the tabu list for 10 iterations.

Fig. 7 shows the gains obtained by TSA(zero) and TSA(heur) with different values of $\text{Iter}$ (i.e., number of consecutive iterations without gain improvement) used for terminating the tabu search process. Initially, the difference between the gains obtained by TSA(heur) and TSA(zero) are relatively large due to the better initial solution used by TSA(heur). The gains of the two solutions converge with the increase of $\text{Iter}$. However, the gain of TSA(heur) is consistently larger than that of TSA(zero) for a fixed value of $\text{Iter}$.

On the other hand, TSA(zero) requires more iterations in the tabu search than TSA(heur) to obtain the same gain. For example, TSA(zero) requires $\text{Iter}$ of about 2250 while TSA(heur) only requires about 1800 to obtain a gain value of 25.
This is because tabu search needs more moves to improve the relatively inefficient solution of TSA(zero).

As shown in Fig. 7, the gains of both TSA(heur) and TSA(zero) increase significantly to about 25 when Iter increases to 2000. The heuristic solution is further refined by TSA(heur) in the subsequent iterations. However, after Iter increases to 3000, the current gain does not have any notable increase. This implies that from this point onwards, refining the current solution in TSA(heur) gradually becomes difficult although the current solution still has opportunities for improvement. Based on this analysis, the value of Iter is set to 3000 in our experiments.

Let \( x^* \) be the optimal solution produced by BnB and \( E(x^*) \) be the gain (solution value) of \( x^* \). In order to estimate the quality of an approximate solution \( x \), we define
\[
\varepsilon(x) = \frac{E(x^*) - E(x)}{E(x^*)} \times 100%.
\]

The experimental results are shown in Table 3 and Table 4, where the input DFGs are described in Table 2. Table 3 shows the quality of the approximate solutions (\( \varepsilon \)) and the runtimes (\( T \)) of the algorithms, for the cases where BnB can find optimal solutions. We also provide the solutions and the runtimes of the algorithms HEA and TSA in Table 4 for the cases where BnB is prohibitive. In the two tables, the solution values \( E(x) \) are rounded to 1 decimal place. TSA indicates TSA(heur) and \( n \) represents the number of instances under the given I/O constraints of a DFG. The cases that the optimal solution cannot be obtained in

![Figure 7: Gains obtained by TSA(heur) and TSA(zero) with different Iter values, averaged over 20 samples, DFG 3, and \( A = 50\% \cdot \sum_{i=1}^{m} a_i' \).](image-url)
Table 3: The solution quality $\varepsilon$ (%) and the algorithm runtime $T$ (s)

<table>
<thead>
<tr>
<th>DFG(I/O)</th>
<th>$A = 20% \cdot \sum a_i'$</th>
<th>$A = 50% \cdot \sum a_i'$</th>
<th>$A = 80% \cdot \sum a_i'$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BnB</td>
<td>HEA</td>
<td>TSA</td>
</tr>
<tr>
<td>1(4/1)</td>
<td>2</td>
<td>0.003</td>
<td>0.001</td>
</tr>
<tr>
<td>2(4/1)</td>
<td>3</td>
<td>0.004</td>
<td>0.002</td>
</tr>
<tr>
<td>3(4/1)</td>
<td>14</td>
<td>0.005</td>
<td>0.006</td>
</tr>
<tr>
<td>4(4/1)</td>
<td>10</td>
<td>0.003</td>
<td>0.008</td>
</tr>
<tr>
<td>5(4/1)</td>
<td>86</td>
<td>23.3</td>
<td>0.14</td>
</tr>
<tr>
<td>6(4/1)</td>
<td>85</td>
<td>20.6</td>
<td>0.13</td>
</tr>
<tr>
<td>7(4/1)</td>
<td>119</td>
<td>26.4</td>
<td>0.15</td>
</tr>
<tr>
<td>1(3/2)</td>
<td>8</td>
<td>2.7</td>
<td>0.005</td>
</tr>
<tr>
<td>2(3/2)</td>
<td>252</td>
<td>17.07</td>
<td>3814</td>
</tr>
<tr>
<td>3(3/2)</td>
<td>234</td>
<td>19.5</td>
<td>0.4</td>
</tr>
<tr>
<td>4(3/2)</td>
<td>8</td>
<td>2.66</td>
<td>0.004</td>
</tr>
<tr>
<td>2(4/2)</td>
<td>296</td>
<td>18.53</td>
<td>8391</td>
</tr>
</tbody>
</table>

acceptable time (within 20 hours) are indicated by ‘-’ in the table.

Table 3 shows that BnB can provide optimal solutions for problems with relatively small sizes (i.e., the number of instances ($n$) is small). For the relatively large problem (e.g., $n > 300$), as shown in Table 4, BnB can hardly obtain solution in acceptable time due to its inherent computing complexity. In addition to the problem size, the available hardware area impacts the performance of BnB.

For the example of DFG 2 with I/O constraint 4/2 (see the last row of Table 3), BnB can produce the optimal solution for $A = 20\% \cdot \sum a_i'$ although there are 296 selected instances in this case. When $A$ increases to $50\% \cdot \sum a_i'$, BnB is not able to generate solution in acceptable time. This means the runtime of the algorithm BnB is also governed by the given hardware area constraint. In particular, BnB is more effective when the area constraint is more stringent, although it may become prohibitive when the available hardware area increases.

However, HEA can generate high-quality approximate solutions for most problems with different sizes, as shown in Table 3. The average value of $\varepsilon$ is about 3%. Noting that, for very small problem, even a slight difference between the solution of HEA and the optimal solution may lead to a large $\varepsilon$ value. For example, the $\varepsilon$ value of DFG 3 with I/O constraint 4/1 is 15.8% for $A = 50\% \cdot \sum a_i'$ with a small problem size of 14. It is worthwhile to point out that, such cases can be efficiently
Table 4: The solution value $E(x)$ and the algorithm runtime $T$ (s)

<table>
<thead>
<tr>
<th>DFG(I/O)</th>
<th>$A = 20% \cdot \sum a'_i$</th>
<th>$A = 50% \cdot \sum a'_i$</th>
<th>$A = 80% \cdot \sum a'_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HEA</td>
<td>TSA</td>
<td>HEA</td>
</tr>
<tr>
<td>8(4/1)</td>
<td>$n$</td>
<td>$E(x_{HEA})$</td>
<td>$T$</td>
</tr>
<tr>
<td>4(3/2)</td>
<td>308</td>
<td>108.0</td>
<td>0.04</td>
</tr>
<tr>
<td>5(3/2)</td>
<td>704</td>
<td>39.9</td>
<td>0.3</td>
</tr>
<tr>
<td>6(3/2)</td>
<td>5728</td>
<td>82.3</td>
<td>1.7</td>
</tr>
<tr>
<td>7(3/2)</td>
<td>17651</td>
<td>114.9</td>
<td>18.2</td>
</tr>
<tr>
<td>8(3/2)</td>
<td>22446</td>
<td>79.6</td>
<td>27.8</td>
</tr>
<tr>
<td>3(4/2)</td>
<td>129191</td>
<td>242.9</td>
<td>2052.0</td>
</tr>
<tr>
<td>4(4/2)</td>
<td>477</td>
<td>22.5</td>
<td>0.03</td>
</tr>
<tr>
<td>5(4/2)</td>
<td>965</td>
<td>44.5</td>
<td>0.3</td>
</tr>
<tr>
<td>6(4/2)</td>
<td>11152</td>
<td>94.9</td>
<td>6.5</td>
</tr>
<tr>
<td>7(4/2)</td>
<td>23036</td>
<td>114.7</td>
<td>35.2</td>
</tr>
<tr>
<td>8(4/2)</td>
<td>26200</td>
<td>79.0</td>
<td>40.2</td>
</tr>
<tr>
<td></td>
<td>143082</td>
<td>242.9</td>
<td>2606.0</td>
</tr>
</tbody>
</table>

handled by TSA. Moreover, the solution quality of HEA is not heavily affected by the given area constraint $A$ in contrast to BnB, as the selection strategy in HEA is not directly related to the given area constraint.

From Table 3 and Table 4, it can be observed that TSA can further refine the approximate solutions of HEA to obtain near optimal solutions. It can achieve the optimal solutions for most cases in Table 3. The average value of $\varepsilon$ is only about 0.6%. When the size of the problem increases, the solution difference of TSA may increase. This is because the inherent search space for the optimal solution increases exponentially with the increasing problem size. However, the approximate solution can be further refined by increasing the number of iterations for problems with a certain size. Moreover, the solution quality of TSA is not heavily affected by the given area constraint for the same reason as that of HEA.

In terms of runtime, as shown in Table 3 and Table 4, HEA is clearly faster than BnB and TSA, especially for large-sized problems. For example, for the case of DFG 2 with I/O constraint 4/2, HEA produces an approximate solution within 0.1s for $A = 0\% \cdot \sum a'_i$, while TSA requires about 2.1s and BnB needs about 8391s. Hence, HEA is more suitable for the large-sized problems, compared to BnB and TSA. On the other hand, BnB is suitable for small-sized problems or for problems with stringent area constraint, and TSA is appropriate for medium-sized
problems.

Table 5 summarizes the worst-case complexities of the algorithms BnB, HEA and TSA. $n$ represents the number of instances. $|V|$ indicates the number of nodes in the DFG. $M$ and $q$ indicate the total number of iterations and the size of the neighborhood list in tabu search, respectively.

Table 5: Comparisons of BnB, HEA and TSA.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time complexity</th>
<th>Space complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>BnB</td>
<td>$O(2^n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>HEA</td>
<td>$O(n^2 \cdot</td>
<td>V</td>
</tr>
<tr>
<td>TSA</td>
<td>$O(M \cdot q \cdot</td>
<td>V</td>
</tr>
</tbody>
</table>

8. Conclusions

We have proposed two heuristic custom-instruction selection algorithms, namely HEA and TSA, and an exact algorithm for reconfigurable processors. To the best of our knowledge, the proposed recurrence-aware algorithms are the first to select custom instructions that are subjected to three practical constraints simultaneously, i.e. area, overlapping and acyclicity constraints. We have shown that the proposed heuristic algorithm HEA is able to generate higher-quality solutions than existing techniques. The proposed tabu search algorithm TSA is a good choice for medium-sized problems, as it can refine the HEA solutions to nearly optimal ones with small increase in run-time. In addition, TSA can provide optimal solutions for small-sized problems. We have also proposed an exact algorithm BnB that is based on the branch-and-bound method, which incorporates efficient pruning strategies for reducing the search space. The proposed BnB algorithm can effectively solve problems with manageable size, although it is still exponential in the worst case. The proposed algorithms in this paper have been successfully incorporated in our instruction set customization framework [31].

References


