Instruction Set Extension Exploration in Multiple-Issue Architecture

I-Wei Wu, Zhi-Yuan Chen, Jyh-Jiun Shann, and Chung-Ping Chung
Dept. of Computer Science
National Chiao Tung University
wuiw@cs.nctu.edu.tw, shiva.cs94g@nctu.edu.tw, jjshann@cs.nctu.edu.tw, cpchung@cs.nctu.edu.tw

Abstract

To satisfy high-performance computing demand in modern embedded devices, current embedded processor architectures provide designer with possibility either to define customized instruction set extension (ISE) or to increase instruction issue width. Previous studies have shown that deploying ISE in multiple-issue architecture can significantly improve performance. However, identifying ISE for multiple-issue architecture by using current ISE exploration algorithms will result in unnecessary waste of silicon area and limitation of performance improvement. This is because most algorithms overlook two important considerations: (1) only packing the operations lying on the critical path into ISE can improve performance; (2) the critical path usually changes after packing operations into an ISE. With these considerations, this paper presents an algorithm for ISE exploration based on list scheduling and Ant Colony Optimization (ACO), in which combines ISE exploration and the critical path identification (i.e. instruction scheduling). Results indicate that our approach outperforms the previous work in both performance improvement and area efficiency.

1. Introduction

Recently, next-generation digital entertainment and mobile communication devices are driving the need for high-performance processing solutions. In order to satisfy this demand, current embedded processor architectures, such as Tensilica Xtensa, ARC ARCTangent, MIPS CorExtend and Nios II, provide designer with possibility either to define customized instruction, or called instruction set extension (ISE) [1, 2, 3, 4, 5, and 6]. ISEs are realized by application specific functional units (ASFU) within the execution stage of the pipeline.

Previous studies [5, 7, and 8] have show ISE benefits for the multiple-issue architecture (e.g. VLIW), especially in performance improvement. The reason why the multiple-issue architecture can gain performance improvement from ISE(s) is that even the issue width and hardware resources are infinite, performance is still severely limited by data dependency. However, using ISE can reduce the execution time of operations having data dependency, but increasing issue width cannot achieve that.

Except for the approach proposed in [5], most ISE exploration algorithms only consider the legality of operation but not the criticality of operation. (A legal operation means that after encapsulating this operation into ISE X, ISE X still obeys all predefined constraints; the critical operation is an operation lying on the critical path.) Overlooking the criticality of operations will lead to unnecessary waste of silicon area because packing the operations along the non-critical path is unlikely to give the application performance improvement. On the other hand, the critical path usually changes after encapsulating operations into an ISE. That means that if the ISE exploration algorithm cannot become aware of the change of the critical path, then it will pack ineffectual operations into ISE (waste silicon area) and/or omit some valuable ISEs (limit performance improvement). Nevertheless, Clark’s work [5] has neglected this point.

Figure 1: Benefit of considering criticality

Figure 1 is an example to explain the benefit of considering the criticality of operation. In Fig.1, DFG stands for a data flow graph; Conventional (only considering the legality) and Our (our method) are the results of ISE exploration by using different approaches; each block represents one operation (i.e. instruction). In this example, assume that only normal operations can be packed into ISE and a 2-issue architecture is deployed; the number in parentheses of blocks is mobility value (mobility = ALAP (as-late-as-possible) − ASAP (as-soon-
as possible)). If only considering the legality, operation 5 and 9 will be packed into ISE. However, packing these two operations cannot have any performance improvement and furthermore wastes silicon resource.

This paper presents an ISE exploration algorithm which combines ISE exploration and instruction scheduling. Our approach is derived from list scheduling and Ant Colony Optimization (ACO) [9]. ISE exploration aims to determine each operation should use hardware or software implementation option. If an operation is encapsulated into ISE, it means that this operation deploys the hardware implementation option; on the contrary, if not encapsulated, this operation chooses software implementation option and is executed in the processor core. Instruction scheduling is used to identify which operation lies on the critical path.

The objective of proposed approach is to minimize the execution time of program(s) while consume less silicon area. Results reveal that our approach outperforms than previous works [2] and [5] in execution time reduction and area efficiency (reduced cycles per unit of area, cycles/μm²). This study has the following contributions:

1. Explain why current ISE exploration algorithms are unlikely to be employed in multiple-issue architecture.
2. A novel ISE exploration algorithm for multiple-issue architecture is presented to increase area efficiency and performance improvement.
3. We show the importance of taking the criticality of operation into consideration when exploring ISE in multiple-issue architecture.

The rest of this work is structured as follows. Section 2 studies the previous related work and background. Section 3 then introduces the proposed algorithm. Next, Section 4 presents the simulation results and discusses the results. Conclusions are finally drawn in Section 5.

2. Relative Works and Background

Pozzi [2] proposed an algorithm to examine all possible ISE candidates such that it can obtain an optimal solution. This maps the ISE search space, such as a basic block, to a binary tree, and then discards some portion of the tree that violates predefined constraints. Nevertheless, this algorithm is highly compute-intensive, so does not process a larger search space. To decrease the computing complexity, heuristic algorithms derived from genetic algorithm [2], Kernighan-Lin (KL) [3] and greedy-like algorithm [5] have been developed. An Integer Linear Programming formulation of the ISE exploration was presented in [6]: in this case, the enumeration of subgraphs is implicit in the formulation’s constraints, and the worst-case complexity is still exponential. Nevertheless, most algorithms [1, 2, 3, 4, 6, 7, and 8] only consider the legality of operations when exploring ISE.

Ant Colony Optimization algorithm [9] is inspired by the behavior of ants in finding paths from their colony to food. This algorithm has been extensively applied to solve many optimization problems. Many studies [10 and 11] have proven that ACO outperforms other meta-heuristics, such as genetic algorithm and simulated annealing, in many optimization problems. Moreover, these works also demonstrated that ACO is extremely close to optimal scheme. This is why we adopt ACO to solve ISE exploration rather than other algorithms.

3. Proposed Approach

Figure 1: ISE exploration flow

The input and output of ISE exploration algorithm are selected basic block(s) and ISE candidate(s), respectively. Before exploring ISE candidate, a selected basic block must be transformed to a data flow graph (DFG). DFG is represented by a directed acyclic graph $G(V,E)$ where $V$ denotes a set of vertices, and $E$ represents a set of directed edges. Every vertex $v \in V$ is an assembly instruction in basic block, and an assembly instruction is hereafter also
called as an operation or a node. Each edge \((u,v)\in E\) from operation \(u\) to operation \(v\) signifies that the execution of operation \(v\) needs the data generated by operation \(u\).

Figure 1 depicts the ISE exploration flow of the proposed approach. It takes the DFG as input, and outputs the ISE candidate(s). The proposed algorithm explores ISE candidate iteratively until no ISE candidate can be found in the input DFG. The algorithm, therefore, is performed for several rounds (a round comprises all steps in Fig. 1); except for last round, each round produces at least one ISE candidate. The kernel of each round (Step 2~9 in Fig. 1) would be executed repeatedly until convergence is achieved. Executing these steps within the kernel once is called one iteration. The kernel of algorithm can be divided into four parts: (1) instruction scheduling and implementation option selection (Step 2~6); (2) exploration result evaluation (Step 7); (3) merit value calculation (Step 8); (4) convergence determination (Step 9).

Taking DFG as input, Step 1 initializes the both trail and merit values for all implementation options. Noticeable, each operation has two implementation options, namely hardware implementation option and software implementation option. The meaning of trail is the similar to the pheromone used in the ACO algorithm, i.e. the number of valid chosen times of an implementation option in previous iterations. The valid chosen time is counted only when choosing this implementation option can reduce the execution time. Here, the trail value of hardware and software implementation option of operation \(x\) is denoted by \(trail_{x,\text{HW}}\) and \(trail_{x,\text{SW}}\), respectively. The merit value is defined as the benefit of one implementation option being selected, and it is obtained by using the merit function, which is described in detail later. The merit value of hardware and software implementation option of operation \(x\) is represented by \(merit_{x,\text{HW}}\) and \(merit_{x,\text{SW}}\), respectively.

Step 2 initializes the Ready-Matrix. Ready-Matrix is a data structure which is very similar with ready list used in list scheduling. Same with ready list, Ready-Matrix contains only those operations which have no predecessors, i.e. predecessors have already been scheduled. Figure 2 is an example of Ready-Matrix in which “*” means no this implementation option. The value in Ready-Matrix is chosen-probability \((cp)\) which depends on trail and merit values. Chosen-probability \((cp)\) of an operation is derived from:

\[
cp = \frac{\alpha \times \text{trail} + (1 - \alpha) \times \text{merit}}{\sum \alpha \times \text{trail} + (1 - \alpha) \times \text{merit}} \quad (1)
\]

where \(\alpha\) and \(\beta\) are utilized to determine the relative influence of trail and merit, and

\[
\sum cp = 1 \quad (2)
\]

### All implementation options in Ready-Matrix

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Operation 1</th>
<th>Operation 2</th>
<th>Operation 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>0.35</td>
<td>*</td>
<td>0.15</td>
</tr>
<tr>
<td>Software</td>
<td>0.15</td>
<td>0.1</td>
<td>0.25</td>
</tr>
</tbody>
</table>

**Figure 2: Example of Ready-Matrix**

With respect to chosen-probability \((cp)\), Step 3 probabilistically selects one implementation option from Ready-Matrix. After choosing one implementation option, the algorithm schedules the operation which owns this selected implementation option. Operation-Scheduling (Step 4) tries to schedule the selected operation in an earliest idle time slot. The main difference between scheduling the operation used hardware and software implementation options is that it is possible to schedule several operations using hardware implementation option in one cycle, but it is impossible to do that for operations using software implementation option. After Operation-Scheduling, the following two steps are performed to update Ready-Matrix: (1) remove the operation which owns the selected implementation option from Ready-Matrix; and (2) add the operation into Ready-Matrix if all dependencies for this operation have been resolved. The algorithm repeatedly executes step 3 to 6 until all operations are scheduled. After all operations are scheduled, the algorithm updates trail values according to execution time reduction, and then computes merit value of all implementation options of each operation in DFG based on the results of this iteration (e.g. which operation locate on the critical path and which operation choose hardware implementation option). Above steps (Step 2~8) are repeatedly performed until the end condition is fulfilled, i.e. until converge. The end condition has two: (1) for all operations in DFG, the chosen-probability \((cp)\) of one of implementation options exceeds P_END, which is a predefined threshold value and is very close to 100% (P_END is 99% in this paper); (2) after executing P_TIMES (P_TIMES is 1500 in this paper) number of iterations (Step 2~8), the performance improvement is still same. A larger P_END and P_TIMES have a higher opportunity of obtaining a better result, but typically take a longer time to converge. After convergence, the implementation option with larger chosen-probability \((cp)\) is called the taken implementation option. An ISE is a set of connected/reachable nodes (i.e. operations) all of which have taken hardware implementation option. Finally, the algorithm executes Make-Convex to let every ISE candidate comply with the convex constraint. But, if an ISE has conformed to the convex constraint, then the algorithm will skip this step. Make-Convex repeatedly divides the ISE candidate into smaller ones until all smaller ISE candidates can comply with convex constraint.
In following paragraph, we describe the several processes/ steps used in the proposed algorithm, including Trail Update, Hardware-Grouping and merit calculation (Merit Function).

**Trail Update**

Trail is updated according to the execution time reduction of each iteration. In the proposed algorithm, the execution time mainly depends on two factors: (1) selection of implementation option, and (2) the decision of execution order of operation. Hence, the trail values are update based on these two factors. The algorithm of trail update is displayed in figure 3. Here, $TET_{new}$ and $TET_{old}$ are the execution times of current and previous iteration, respectively; $\rho$ are positive constant values, and is called evaporating factor. The evaporating factor is very similar to the evaporation rate in ACO.

$$\text{If } (TET_{new} \leq TET_{old};$$
$$TET_{old} = TET_{new};$$
$$\text{For operation } x (x=1 \text{ to } k) \text{ in DFG}$$
$$\text{If (the hardware implementation option is selected) }$$
$$\quad trail_{x,SW} = trail_{x,SW} + \rho;$$
$$\quad trail_{x,HW} = trail_{x,HW} - \rho;$$
$$\text{Else}$$
$$\quad trail_{x,SW} = trail_{x,SW} - \rho;$$
$$\quad trail_{x,HW} = trail_{x,HW} + \rho;$$
$$\text{Else}$$
$$\text{For operation } x (x=1 \text{ to } k) \text{ in DFG}$$
$$\text{If (the hardware implementation option is selected) }$$
$$\quad trail_{x,SW} = trail_{x,SW} - \rho;$$
$$\quad trail_{x,HW} = trail_{x,HW} + \rho;$$
$$\text{Else}$$
$$\quad trail_{x,SW} = trail_{x,SW} + \rho;$$
$$\quad trail_{x,HW} = trail_{x,HW} - \rho;$$
$$\text{If (execution order of operation } x \text{ is earlier than previous iteration) }$$
$$\quad trail_{x,SW} = trail_{x,SW} - \rho;$$
$$\quad trail_{x,HW} = trail_{x,HW} + \rho;$$

**Figure 3: Algorithm of trail update**

**Hardware-Grouping**

Hardware-Grouping checks whether the operation $x$ can be encapsulated with its reachable nodes (i.e. operations) into a virtual ISE candidate; if yes, Hardware-Grouping then recursively groups operation $x$ with its reachable nodes, which have chosen hardware implementation option in current iteration, as a virtual ISE candidate, i.e. a virtual subgraph $vS_x$. The result of Hardware-Grouping of operation $x$ is denoted as $vS_x$. Note that Hardware-Grouping is performed if an operation has hardware implementation option, no matter which implementation option is chosen in the current iteration. Using $vS_x$, Hardware-Grouping can measure the execution time and silicon area of $vS_x$. Notably, the execution time of $vS_x$ is the execution time of critical path in $vS_x$, and the silicon area of $vS_x$ is the sum of silicon areas of $vS_x$.

**Figure 4** depicts one example of the Hardware-Grouping function. The table in Fig. 4 lists the delay and area of each implementation option of all operations, and specifies the chosen implementation option in the current iteration. For operation #3, Hardware-Grouping groups operation #1, #2 and #3 as a virtual ISE candidate, i.e. $vS_3=\{#1, #2, #3\}$. The execution time and silicon area of $vS_3$ are 0.08 cycle and 9788 $\mu m^2$, respectively. Because no researchable node selects hardware implementation option, the results of Hardware-Grouping of operation #1 and #2 only contain itself, i.e. $vS_1=\{#1\}$ and $vS_2=\{#2\}$.

**Figure 4: Examples of Hardware-Grouping**

**Merit Function**

The merit function mainly consists two parts: (1) compute the benefit of different implementation options in an operation (determine which implementation option is better choice); (2) calculate the scheduling priority of all operations in the DFG (compute the scheduling order of operations).

In the first part, the merit function consists of four cases: critical path verify (Case 1), virtual ISE size check (Case 2), constraints violation determine (case 3) and performance calculate (case 4). Figure 5 shows the merit function algorithm for first part.

As mentioned above, only packing the operation locating on the critical path can have benefit in execution time reduction. Hence, in case 1, the algorithm adjusts the merit value according to the criticality of operation. The criticality of operation is calculated by its mobility value (mobility = ALAP— ASAP). Then, in case 2, the algorithm determines whether or not size(vSx), the number of operations in vSx, is equal to 1; and (2) whether or not the execution time of hardware implementation option (i.e. ET(vSx,HW)) is equal to the execution time of the fastest software implementation option (i.e. ET(x,SW-FASTEST)). If both conditions are met, it means that vSx only has one operation and does not have performance improvement. Therefore, the algorithm multiplies the merit value of every hardware implementation option in operation x by a constant $\beta_{Size}$ ($0 < \beta_{Size} < 1$) to lower the chance of being chosen at following iterations. The calculation of the merit function is then terminated. If no, then goto case 3.

Case 3 verifies whether vSx violates input/output port and/or convex constraints. If yes, then the merit value of
the hardware implementation option is multiplied by constant $\beta_{\text{IO}}$ and/or $\beta_{\text{Convex}}$ (0 < $\beta_{\text{IO}}$ < 1 and 0 < $\beta_{\text{Convex}}$ < 1), reducing the opportunity for selecting the hardware implementation option, as in case 2. The calculation of the merit function is then terminated. If no, then enter case 4. Since operation $x$ may have chance to be packed in an ISE at the following iterations, the algorithm only multiplies the merit value of each hardware implementation option by a constant. If the algorithm does not allow the possibility of operation $x$ becoming an operation in an ISE candidate, the optimal solution may also be excluded. In case 4, the merit value of hardware implementation option in operation $x$ is computed according to the speedup that can be achieved by $v_{S_i}$.

**Case 1.** (Critical path)

If (operation $x$ locates on the critical path AND operation $x$ has hardware implementation option)

$$\text{merit}_{x,\text{HW}} = \text{merit}_{x,\text{SW}} \times \delta_{P_i};$$

**Case 2.** (No performance improvement)

If (size($v_{S_i}$) = 1 AND $ET(v_{S_i,\text{HW}}) = ET(x,\text{SW-FASTEST})$)

$$\text{merit}_{x,\text{HW}} = \text{merit}_{x,\text{SW}} \times \delta_{S_i}; \text{ return merit}_{x,\text{SW}};$$

**Case 3.** (Violate constraints)

Else if ($v_{S_i}$ violates in/out constraint)

$$\text{merit}_{x,\text{HW}} = \text{merit}_{x,\text{SW}} \times \delta_{S_i}; \text{ return merit}_{x,\text{SW}};$$

Else if ($v_{S_i}$ violates convex constraint)

$$\text{merit}_{x,\text{HW}} = \text{merit}_{x,\text{SW}} \times \delta_{\text{Convex}}; \text{ return merit}_{x,\text{SW}};$$

**Case 4.** (Conform to constraints, and has performance improvement)

Else

$$\text{merit}_{x,\text{HW}} = \text{merit}_{x,\text{SW}} \times \text{cycle} _{\text{saving}}_{x,\text{HW}}; \text{ return merit}_{x,\text{SW}};$$

**Figure 5: Algorithm of merit function (Part I)**

In second part, the merit function calculates the merit values according to the scheduling priority of operations. In list scheduling, each operation must be assigned a scheduling priority before scheduling. The scheduling priority can be derived from several ways. In this paper, the scheduling priority is calculated by the number of child nodes. After first part of merit calculation, hardware and software implementation options of each operation in the DFG multiply by its number of child nodes.

Noticeably, the merit values of all implementation option are reset to the initial values at each iteration. In order to reach a maximal speedup, we prefer to choose hardware implementation option for an operation. That means that the initial merit value of hardware implementation option is larger than that of software one.

4. Experimental Results

The MIPS R3000 instruction set was employed to evaluate the proposed ISE exploration algorithm and Pozzi’s and [2] Clark’s [5] works. All algorithms were implemented by us in C language. All benchmarks (including adpcm decode/encode, basicmath, epic decode/encode, fft, inverse-fft, g.721 decode/encode, jpeg decode/encode, mpeg2 decode/encode, sha, and susan) used for all algorithms were compiled with inlining and loop-unrolling by gcc 4.1.1.

For all algorithms, we assume that: (1) the CPU core executes in 300MHz; (2) the issue width are 2 and 4; (3) except for multiply (3 cycles) and divide (12 cycles) operations, the execution cycle of all instructions is one cycle. The settings of hardware implementation option (delay and area) were modeled by Verilog and synthesized in 0.13µm CMOS technology. Table 1 shows the settings of hardware implementation option in which the unit of Delay and Area are nanosecond (ns) and µm², respectively.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Delay</th>
<th>Area</th>
<th>Operator</th>
<th>Delay</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/Sub</td>
<td>0.5</td>
<td>5768</td>
<td>XOR</td>
<td>0.12</td>
<td>217</td>
</tr>
<tr>
<td>Multiply</td>
<td>2.49</td>
<td>24534</td>
<td>NOR</td>
<td>0.03</td>
<td>163</td>
</tr>
<tr>
<td>AND</td>
<td>0.09</td>
<td>217</td>
<td>Barrel</td>
<td>0.26</td>
<td>4773</td>
</tr>
<tr>
<td>OR</td>
<td>0.12</td>
<td>217</td>
<td>Shifter</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 1: Settings of hardware implementation option**

As mentioned before, the results of ISE exploration are only ISE candidates. However, without performing ISE merging and selection, these results cannot be viewed as the final ones. In this paper, therefore, we performed ISE merging and selection after ISE exploration. The selection criteria used in this paper was the number of reduced cycles of ISE. In addition, to reduce program execution time, the shape of replaced instruction pattern must be fully identical to that of ISE, i.e. the instruction pattern $P$ is replaced by ISE $A$, if $P$ is isomorphism to $A$.

The parameters adopted in this work and their meanings are listed below.

- $\alpha$: the relative influence of merit and trail. A larger $\alpha$ causes the selection of implementation option more random such that the variation in the exploration results becomes larger.
- $\rho$: the evaporating factor in trail update. A smaller $\rho$ makes the algorithm converge slowly, but has higher opportunity to obtain a better solution.
- $\beta_{CP}$, $\beta_{\text{Size}}$, $\beta_{\text{IO}}$, and $\beta_{\text{Convex}}$: the penalty for a non-criticality and/or illegal operation. Violating the input/output and/or the convex constraints will make an illegal ISE, but violating the remaining two cases can still generate a legal ISE. Hence, $\beta_{\text{IO}}$ and $\beta_{\text{Convex}}$ must be smaller than $\beta_{CP}$ and $\beta_{\text{Size}}$.

In this experiment, the initial merit value of the software and hardware implementation option was 100 and 200, respectively; the initial trail value of all implementation options were 0; the evaporating factor $\rho$ is 4; and the merit function had $\beta_{CP} = 0.9$, and $\beta_{\text{Size}} = 0.7$, $\beta_{\text{IO}} = 0.5$ and $\beta_{\text{Convex}} = 0.5$.

Table 2 shows comparative results among different approaches, these results are averaged from all benchmarks.
The value of execution time reduction is the ratio of execution cycle difference between with and without ISE in M-issues architecture, i.e., execution time reduction = (EC_{without ISE} - EC_{with ISE}) / EC_{without ISE} (EC is abbreviated from execution cycle); the area efficiency denotes the reduced cycles per unit of area (cycles/µm²). The first word of each label in the first column indicates which ISE exploration algorithm is adopted. “Our” represents our proposed algorithm; “Ge” and “Clark” denote the approach presented by Pozzi [2] and that of Clark [5], respectively. The symbols in parentheses of each label in the first column are the number of register file read/write ports in use, issue width. For instance, (4/2, 2IS) means that the register file has four read ports as well as two write ports, issue width is two.

According to the results shown in the table 2, our proposed algorithm exhibits better execution time reduction and area efficiency than previous works in most cases. However, in several cases, Clark’s work has better area efficiency than our approach. This because that the average size of ISE identified by Clark’s work is smaller than that of our approach such that more instruction patterns can be replaced by ISEs identified by Clark’s work. This problem can be solved by performing subgraph replacement, i.e., replacing instruction pattern P by ISE A, if $P$ is a subgraph of $A$ in this work, $P$ is replaced by $A$, only if $P$ is isomorphism to $A$. But, performing subgraph replacement is a highly time-consuming task.

5. Conclusion

In this paper, we have shown the importance of taking the criticality of operation into account when exploring ISE in multiple-issue architecture. Moreover, the simulation results demonstrate the superiority of the proposed algorithm in execution time reduction and area efficiency. Although only one scheduling priority function is deployed in this work, it will be an interesting if we study the effect of different scheduling priority functions. Furthermore, to further improve the area efficiency, hardware design space exploration of ASFU should be taken into account when exploring ISE. We are already working on this topic.

Reference


<table>
<thead>
<tr>
<th># of ISEs</th>
<th>Execution Time Reduction (%)</th>
<th>Area Efficiency (cycles/µm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Get(4/2, 2IS)</td>
<td>2.54</td>
<td>3.68</td>
</tr>
<tr>
<td>Clark (4/2, 2IS)</td>
<td>4.32</td>
<td>4.90</td>
</tr>
<tr>
<td>Our(4/2, 2IS)</td>
<td>6.17</td>
<td>7.53</td>
</tr>
<tr>
<td>Get(8/4, 4IS)</td>
<td>1.32</td>
<td>2.50</td>
</tr>
<tr>
<td>Clark (8/4, 4IS)</td>
<td>2.75</td>
<td>5.33</td>
</tr>
<tr>
<td>Our (8/4, 4IS)</td>
<td>4.67</td>
<td>5.32</td>
</tr>
<tr>
<td>Ge-Average</td>
<td>1.93</td>
<td>3.09</td>
</tr>
<tr>
<td>Clark-Average</td>
<td>3.54</td>
<td>5.12</td>
</tr>
<tr>
<td>Our-Average</td>
<td>5.42</td>
<td>6.43</td>
</tr>
</tbody>
</table>

Table 2: Execution Time Reduction and Area Efficiency for different number of ISEs