Custom Instruction Synthesis Framework for Application Specific Instruction-Set Processor with HW

Amr Fathy, Tsuyoshi Isshiki, Dongju Li, Hiroaki Kunieda
(amr,isshiki,dongju,kunieda)@vlsi.ss.titech.ac.jp
Grad. School of science and engineering
Dept. of Communications and Integrated Systems
Tokyo Institute of Technology

Abstract—This paper presents a framework for searching and ranking application specific special instruction patterns and cost calculation using simulated HW modules.

A key feature of Application Specific Instruction Set Processor (ASIP) design is the ability of defining application specific special Instructions, which are assisting HW modules in the EX phase of the processor that handles patterns of instructions that if handled by HW can improve the performance of the processor.

This paper presents a complete framework for searching for Application specific special instruction patterns based on tree scan algorithm while tweaking it to fit real applications, this algorithm works on reducing the search space by eliminating invalid tree nodes hence reducing the need to process the tree leaves.

Ranking of the special instructions is done by calculating speedup gained by calculating the difference between SW cycles and HW cycles, HW cycles are calculated by synthesizing basic HW operations of the critical path of the special instruction, while SW cycles are calculated from program simulation.

Final results show a big improvement in run time over brute force algorithms while having the same outcome of special instructions.

Index Terms—Computational complexity, Pattern analysis, Computer aided analysis, Greedy algorithms

I. INTRODUCTION

The rapid increase in the usage of embedded system devices has put a lot of load on finding processing solutions that are more efficient in terms of speed and power consumption, while taking into consideration the time to market factor for designing the processing units.

While conventional GPPs offer a very flexible and very fast time to market solution, it lacks in terms of performance and power consumption. On the other hand ASIC provides a rather high performance and low power consumption but it lacks in terms of flexibility.

From there emerged the need to find a solution that is flexible enough to overcome the rigidity of ASICs while maintaining an acceptable level of performance and power consumption. Application Specific Instruction set Processors, or ASIPs, are tailored processors that are designed to fit the need of specific applications while having enough flexibility for changes or upgrades [1][2][3]

Fig. 1. Comparison between different processing solutions, Flexibility vs. performance

A key feature of ASIP design is the ability to add special instructions (SI) in order to further increase the performance of the processor, a special instruction is a set of instructions that are implemented as a separate HW functional unit in the processor, and every time this pattern of instructions appears in the SW it is translated to one instruction handled by the HW functional unit.

Adding this functional unit helps increase the performance of the processor by removing all slack time wasted between
execution of every instruction and by exploiting the parallelism between instructions to reduce execution time to its minimum, this also helps in reduction of the power consumption by the processor by decreasing the execution cycle count of the program.

![Diagram](image)

The most conventional way to search for special instruction is using brute force algorithm, scanning the whole search space and analyzing all possible patterns, but this solution has exponential time consumption.

\[
\text{num. of all possible patterns in } BB = 2^{\text{num. of instructions}}
\]

An alternative search strategy is proposed in [4] where the design space is decreased by means of analyzing constraints put on the special instruction by the user (Inputs/Outputs), or constraints put by the properties of the special Instruction (Convexity).

We used this theory to build upon to form our framework, where we used constraints and properties of the special instruction to decrease the search space to a much smaller space (in most benchmarks search space is less than 30% of the whole search space), giving a reasonable run time for the whole framework (as will be seen in results section IV).

In this paper we are going to introduce a framework for automatic search and identification of patterns of instructions in a program code that qualifies to be a special instruction, and then ranking these patterns according to the performance improvement they offer.

In the following sections we are going to explain in detail the base work that we used in order to build our framework in section II, and then explain in detail the tool flow of our tool starting from the program source code, then the formation of the ranked list of special instructions in section III, then plotting the results obtained from simulations of different benchmarks under our tool in section IV, then we will conclude and discuss the shortcoming of our tool where we plan for further improvements in section V.

II. RELATED WORK

Our main question was how to perform a complete search while keeping the search space as small as possible, K. Atasu et al. proposed in [4] a simple way to identify the border between valid and invalid search space based on three main restrictions identified as (1) number of input ports to the special instruction, (2) number of output ports, and (3) an essential property of the special instruction which is convexity (explained in section III.C)

This concept was our seed to implement our algorithm to use the user’s restriction for reducing the valid search space to its minimum by drawing the border line between valid and invalid search space and performing a complete scan in the valid search space.

III. FRAMEWORK

This paper introduces a framework for analyzing program source code and searching the full valid search space for candidates of special instructions, and finally ranking these instructions according to its performance improvement.

Figure 3 shows the different components of the framework and the tool flow starting from C-SRC files ending with the final report of candidates of special instructions along with performance report on each of them.

In the next sub-sections every part of the tool flow is explained in more details, sub-section A talks about the compiler framework used which is illustrated in Fig. 3 stage (1), in this compilation stage the data flow analysis (sub-section B) and special instruction search algorithm (sub-section C) takes place. Target machine compilation (Fig. 3 stage 2) and Simulator (Fig. 3 stage 3) are explained in subsection D. HW analysis and report (Fig. 3 stage 4) is explained in subsection E. Finally cost function ranking stage (Fig. 3 stage 5) is explained in sub-section F.

A. Basic Compiler Framework

To start off our work we needed a compiler framework to support our analysis, while we considered different frameworks like gnu gcc[5] which is the most famous in terms of features and stability, or llvm[6] which stood out as a fast and an easy compiler to use, we decided to use CoSy compiler[7] from ACE for its sturdy structure and flexibility in handling changes in its
IR and backend.

Our search and analysis algorithms were built in the middle-end part of the compiler, which is based on the CoSy CCMIR\(^8\) (IR in CoSy).

For now we used our target machine as the x86 architecture which is supported by CoSy, so all our work is done on the local machine, keeping in mind that CoSy gives high flexibility to change its back-end to support any kind of ISA which can be used by our framework to target any kind of backend architectures.

CoSy compiler can only support a single file C to Assembly operation; this could be handled by separate scripts that can manage different source files, libraries, and include files, then using the target machine binary utilities to convert the generated Assembly to machine language.

As shown in Fig. 3, stage (1) is the compiler stage, we need a single execution of CoSy compiler for every SRC file in the project; every compilation results in a single ASM file that is later on compiled by the target machine low level compiler and linker.

Every compilation stage performs the following:
- Building of Control Data Flow Graph (CDFG) for every basic block and calculating Input/Output dependencies of every basic block (section A)
- Special Instruction Search algorithm (section B)
- Simulator Information (section D).

B. Dependence Analysis

Data flow analysis is an essential part in any instruction pattern analysis; data flow information is used in:
- Checking the validity of instructions patterns as special instructions
- Checking the critical path in every instruction pattern to calculate the most optimum HW implementation of the SI in the processor (Fig. 2)

Our first thought was to create a full Dependence Flow Graph\(^9\) since it contains a detailed analysis of data graph while preserving CFG information within the dependence graph, it also handles inter-block dependencies within loops and branches by building Single Entry Single Exit (SESE) blocks, but our analysis had only the scope of a single basic block while the Dependence Flow Graph provided a more complete and more complicated form of analysis which wasn’t needed.

Our choice was to build a CDFG (Control Data Flow Graph), which is a basic block based analysis of dependence information while preserving CFG information within the graph.

After building the CDFG for every basic block, inputs and outputs are defined for every block by analyzing the procedure’s CFG and interface dependencies inside every block.

In Fig. 4, basic block BB_1 has 3 inputs and 1 output, data inputs are \(a, b, c\) while output is \(l\), variables \(j, k\) are written in the basic block but not used later in the function so output data edges of those variables are removed, also variable \(k\) has a WAW dependency, which means that it is overwritten in the basic block before reading it, so the edge holding the WAW dependency of variable \(k\) is removed from the inputs of the basic block.

C. Special Instruction Search

Special instruction search algorithm is the main part of the analysis, the main target of the search algorithm is to search all the valid search space while wasting minimum time in the invalid search space.

The idea behind the algorithm is inspired by using the user constraints and special instruction properties constraints to reduce the search space\(^4\).

Analyzing patterns start off as a normal brute force algorithm, it considers the basic block as a binary tree and for every instruction there are 2 possibilities, either take it in the pattern which is the right branch of the tree or leave it out which is the left branch.

Fig. 5. Shows an example tree for 2 instructions basic block, every node forms a separate pattern, and its encoding refers to the instructions that are in the pattern. For ex. Pattern (01) is the pattern containing instruction \{0\}, pattern (10) is the pattern containing instruction \{1\}, and finally pattern (11) is the pattern containing instructions \{0, 1\}.

The scanner uses a depth first search mechanism to go through the tree, analyzing every single node and checking its validity status, there are two main categories of rules applied on every pattern or tree node, these rules defines the validity status of the pattern and the path for the DFS algorithm.

First set of rules, which we call the tree rules, these rules determine if this pattern violates the SI constraints in a way that cannot be fixed in leaf nodes of the tree, for example if a pattern exceeds maximum number of instructions, then adding more instructions to this pattern will for sure violate the same rule, so these set of rules if violated then all the leaves of that nodes are not even considered by the scanner and the DFS algorithm will...
consider this node the end of the branch. These set of rules are:

- Maximum number of inputs, there is no way to decrease the number of inputs by adding more instructions in the pattern.
- Maximum number of instructions in a pattern.
- Pattern containing invalid instructions: such as memory instructions or function calls, or single or constant assignments.

Finally after the two tests are done, the DFS algorithm has to take a decision, either:

- Add the pattern to the set of valid patterns and continue its path through the tree (pattern is valid).
- Discard pattern and discard all leaf nodes, the pattern violates first set of rules (tree rules).
- Discard the pattern but keep the normal path of the DFS algorithm, the pattern violates second set of rules (node rules)

For every pattern that passes all the rules, a set of information is calculated:

- Input and output variables, and this doesn’t mean input and output data edges, since a single variable can have more than input or output edge.
- Number of SW cycles taken by these instructions if not converted to SI, and since we assume simple RISC architecture then every instruction takes only 1 cycle.
- Critical data path within the pattern, this is necessary for calculating number of cycles the SI needs in execution in HW.

D. Simulator

In the compilation stage (Fig. 3 stage 1), simulator information is added to the assembly code of the program, this ASM code when compiled by the machine bin utilities (Fig. 3 stage 2) will produce an executable that when executed will produce a dynamic profiling report of the analyzed benchmark, this report will contain the number of executions of every basic block. This number is then used later to calculate the performance gain of every basic block.

E. HW Analysis

One way to calculate cost function and performance improvement is to actually implement SI HW module and calculate exactly how many cycles it takes and then calculate the performance improvement. But this way is very hard to automate and consumes too much time in execution. Instead our solution was to create using Verilog the basic arithmetic and logical functional units that is found in any ALU, and simulate every functional unit using Synopsis Design Compiler calculating time consumed by every functional unit. In our synthesis The HW library used is TSMC [11] 90n core library with a clock cycle of 10ns.

Every HW functional unit also considers the types of variables used by every operation, for example an addition of type short doesn’t require a 32 bit adder since it can be implemented using 16 bit integer which takes 50% of the time (add32 takes 3.12ns while add16 takes 1.57ns); that is why all operations are synthesized in its all forms (8, 16, 32, and 64 bits) providing more potential to decrease the time taken of every single instruction hence decreasing the time taken by the special instruction and increasing the performance.

The HW synthesis final report contains the time taken by every operation and its area overhead in the processor. Figure 8 shows a part of the HW analysis report file; this file is parsed later by the cost function calculation part to calculate cost.
function of the every scanned pattern.

<table>
<thead>
<tr>
<th># Mod</th>
<th>CLK_PERIOD</th>
<th>Time</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>add8</td>
<td>1000/100</td>
<td>0.8</td>
<td>259.6608</td>
</tr>
<tr>
<td>add16</td>
<td>1000/100</td>
<td>1.57</td>
<td>508.032</td>
</tr>
<tr>
<td>add32</td>
<td>1000/100</td>
<td>3.12</td>
<td>1004.259</td>
</tr>
<tr>
<td>add64</td>
<td>1000/100</td>
<td>6.22</td>
<td>278.712</td>
</tr>
<tr>
<td>sub8</td>
<td>1000/100</td>
<td>0.83</td>
<td>544.0176</td>
</tr>
<tr>
<td>sub16</td>
<td>1000/100</td>
<td>1.6</td>
<td>1074.629</td>
</tr>
<tr>
<td>sub32</td>
<td>1000/100</td>
<td>3.16</td>
<td>2114.683</td>
</tr>
<tr>
<td>sub64</td>
<td>1000/100</td>
<td>6.24</td>
<td>52419.73</td>
</tr>
<tr>
<td>mul8</td>
<td>1000/100</td>
<td>1.7</td>
<td>1305.36</td>
</tr>
<tr>
<td>mul16</td>
<td>1000/100</td>
<td>3.88</td>
<td>4586.4</td>
</tr>
<tr>
<td>mul32</td>
<td>1000/100</td>
<td>6.93</td>
<td>13914.43</td>
</tr>
<tr>
<td>mul64</td>
<td>1000/100</td>
<td>10</td>
<td>52419.73</td>
</tr>
</tbody>
</table>

Fig. 8. Part of the HW synthesis file

F. Cost Function

The main goal of cost function calculation is to rank patterns scanned by the special instruction search algorithm according to its performance improvement. In Fig. 3 the cost function calculation which is shown in stage 5 takes inputs from 3 other stages:

- Special Instruction analysis stage, and from this stage it takes all scanned patterns validated by the special instruction search algorithm
- Simulator stage, and from this stage it takes the dynamic profiling information indicating number of executions each instruction has made.
- HW analysis stage, and from this stage it takes time and area of every operation, in order to calculate HW and SW cycles of every operation (HW cycles are the number of cycles taken when SI is implemented in HW, while SW cycles are the number of cycles it currently takes in SW).

With these information the cost function is calculated by calculating the total number of saved cycles when implementing this pattern as a SI

saved cycles = SW cycles – HW cycles
total saved cycles = saved cycles * num. executions

Also it calculates the area overhead of the SI, but this area doesn’t include the area of the Input & Output buffers.

IV. EXPERIMENTAL RESULTS

In this section we will discuss the results of our simulations; in order to validate the stability and performance of our framework we tried to use several different benchmarks, like image processing, video processing modulation, encryption, filters and other benchmarks.

Experimental results are divided into two parts:

- Improvement in search space and hence run time of the search algorithm
- Simulated performance improvement achieved by using SI framework.

We achieved a huge improvement in search space, as seen in Fig. 9 we can see that the valid search space identified by our constraints is at maximum 30% of the total search space, this dropped down to a very small fraction in case of benchmarks like JPEG or H264 decoder.

As for run time, all benchmarks finished in a reasonable run time, Fig. 10 shows run time for every benchmark, while run time was in the range of few seconds for most benchmarks, JPEG and H264 benchmarks finished in a little longer time (26 and 69 sec respectively).

On the other hand getting run time for Exhaustive search using our framework was very hard for many of the benchmarks since it can take hours or even days to search the full search space, Fig. 11 plots the increase in run time when we relaxed the restrictions on benchmarks like AES or JPEG, as we relax the restrictions we move more towards extending the search space, and removing all restrictions would result in a full exhaustive search which according to this plot requires exponential time.

Second part of the experimental results is the performance improvements, since SI search algorithm searches the whole valid search space it is guaranteed to scan the whole search space for the best candidates that can be achieved using the
Fig. 12 shows the ratio between cycles before and after converting the single top candidate pattern in selected functions to HW SI, these results are found by simulation, as we can see some functions like FIR filter and gamma correction gained a good performance boost after converting the top candidate pattern to HW, while other functions like adpcm decoder didn’t show big improvement.

Changing constraints would result in different results, relaxing would definitely improve the performance gain but will also increase HW requirements for the SI. So finally it is all about tradeoff between performance requirements and HW resources.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>area(μm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm_dec</td>
<td>2984.69</td>
</tr>
<tr>
<td>adpcm_enc</td>
<td>2454.08</td>
</tr>
<tr>
<td>aes</td>
<td>31847.96</td>
</tr>
<tr>
<td>fir</td>
<td>17933.53</td>
</tr>
<tr>
<td>imageProc</td>
<td>42748.07</td>
</tr>
<tr>
<td>JPEG</td>
<td>48227.76</td>
</tr>
</tbody>
</table>

Fig. 13 shows a table containing area overhead after adding the top candidate SI to the silicon chip, to elaborate the overhead we used the TCT processor as our reference processor, the TCT processor uses the TSMC HW libraries with 90μm technology, TCT-PE is the basic PE which doesn’t contain memory units, comm. modules, or interconnects had an area of 0.096m², Fig. 14 shows the percentage of the added SI area compared to the TCT-PE area.

Fig. 14. percentage of area overhead of top candidate SI compared to TCT-PE

V. CONCLUSION

The constant growth in the need for custom designed processors has increased the need for ASIP CAD tools, one of the basic features of ASIPs is the ability to insert custom HW modules called special instructions in the EX phase of the pipeline in order to increase the performance of the processor.

In this paper we propose a framework for automatic search and ranking of candidate patterns of special instructions in benchmarks, fully automating the flow that starts from source code of the program ending with a full report containing the top candidate patterns for special instructions along with simulated performance improvement and statistics on the tool operations.

Our implementation includes several levels of computation, starting from data dependence analysis in the form of CDFG as a base for the search operation, then special instruction search algorithm guarded by user defined restrictions, scanning all the valid search space of the code, apart from that HW analysis on basic functional unit is done, and a report is generated in order to simulate improvement in performance for every candidate pattern, a simulator is integrated with the compilation phase enabling the executable file from compilation to generate dynamic profiling report that will be used later to calculate the improve in performance, and finally cost function calculation that uses the outputs from all stages to identify and rank the top candidates of instructions which can give the highest performance improvement, on top of all that logging functions, timers and statistics collector are implemented in parallel of all stages to collect information on the flow of the framework.

Preliminary results are encouraging, giving a big improvement in search space while resulting in good performance improvements in several cases, not all cases showed improvement in performance but that can be altered by changing design restrictions enabling more freedom for the SI search algorithm to explore more patterns of the code, keeping in mind that this can result in an exponential increase in run time of the tool and an increase in HW requirements of the SI.

Further improvements can be added in the future. For now relaxing any of the restrictions result in an exponential increase in the search space, so we plan to try other heuristic search algorithms to keep the search space in a linear shape rather than exponential. We are also planning to use the fact that loops usually offers big performance boosting by integrating loop analysis and SI HW loop representation in our framework. For now all configurations are managed using txt configuration files and shell command lines, so we are planning to integrate an eclipse based GUI to manage the full flow.

REFERENCES


