An Integer Linear Programming Approach for Identifying Instruction-Set Extensions

Kubilay Atasu [*]
Department of Computer Engineering
Bogazici University, Turkey
atasu@boun.edu.tr

Günhan Dündar
Department of Electrical and Electronics Engineering
Bogazici University, Turkey
dundar@boun.edu.tr

Can Özturan
Department of Computer Engineering
Bogazici University, Turkey
ozturaca@boun.edu.tr

ABSTRACT

This paper presents an Integer Linear Programming (ILP) approach to the instruction-set extension identification problem. An algorithm that iteratively generates and solves a set of ILP problems in order to generate a set of templates is proposed. A selection algorithm that ranks the generated templates based on isomorphism testing and potential evaluation is described. A Trimaran based framework is used to evaluate the quality of the instructions generated by the technique. Speed-up results of up to 7.5 are observed.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles

General Terms

Algorithms, Design, Performance

Keywords

ASIPs, Extensible Processors, Integer Linear Programming

1. INTRODUCTION

One of the most important differences between embedded computing and general purpose computing is the customization opportunities available in embedded computing. Embedded systems are dedicated to an application domain. Components of an embedded system can be specialized in order to exploit the characteristics of the given application.

Very often, general-purpose processors fail to satisfy the strict performance, power, and area requirements of embedded applications. Use of Application-Specific Integrated-Circuits (ASIC), on the other hand, results in a loss of flexibility that could be provided by a general-purpose processor. Application-specific instruction-set processors (ASIP) are a good compromise between general-purpose processors and ASICs. A base processor is augmented with application specific functional units that implement the application-specific instruction-set extensions. While the datapath is extended with new functional units, a pre-verified, pre-optimized base processor design and instruction-set architecture (ISA) is reused. Control intensive parts of an application can be implemented in software using the base processor instructions, and time-critical, computation intensive parts can be migrated to the specialized datapath as special instructions. Several commercial examples of extensible processors exist, such as Tensilica Xtensa [1][2], MIPS Pro Series [3], ARC ARCtangent [4], and ARM OptimoDE [5].

Identifying the instruction-set extensions is a HW/SW partitioning problem that must respect the architectural constraints of the base processor. The way the specialized functional units read their inputs and write their outputs determines the communication cost of the HW/SW interface. Very often, specialized units can read several input operands from the register file and can write a single output operand to the result bus with no extra cost. Additional inputs and outputs require additional data transfers between the register file and specialized units, and increase the communication cost. Alternatively, additional read and write ports may be added to the register file, which results in increased area and register read and write delays.

Applications described in high-level languages are often converted to control dataflow graphs (CDFG) using compiler infrastructures. The basic blocks of the application form the nodes of the CDFG, and the edges of the CDFG represent the control flow across basic blocks. The aim is to identify clusters of primitive operations satisfying given input and output constraints, having possibly multiple instances within the basic blocks of the CDFG that collectively maximize some metric. In this work, we present an Integer Linear Programming (ILP) based methodology for identifying maximal speed-up instruction-set extensions under input and output constraints given a high level description of an application.

2. RELATED WORK

Automatic identification of instruction set extensions has received considerable attention from various research groups with interest in computer architecture, reconfigurable computing or HW/SW co-design.

[*]This author is supported by TUBITAK under National Ph.D. Scholarship Program.

A description of the HW/SW partitioning problem for ASIP design in the form of a formal combinatorial optimization problem is given in [6]. This work assumes that instruction-set extension candidates, and the matching of these instructions with the dataflow graph nodes are previously defined. The problem is formulated as finding minimum area solution that satisfies given latency and power consumption constraints. A branch-and-bound method is proposed for the solution of the problem. The work in [7, 8, 9] are similar in that they rely on incremental clustering of related dataflow graph nodes using heuristic approaches with the aim of identifying frequently occurring patterns. In [10], a simulated annealing based algorithm is employed to generate clusters based on schedule time and resource usage of dataflow graph nodes. The work of Choi et al. [11] formulates the instruction-set extension identification problem as a modified subset-sum problem. In order to overcome the computational explosion, this work puts a limit in the latency of the instruction-set extensions.

In [12], MaxMISO algorithm is introduced, which identifies all maximal-input single output instruction-set extension candidates within an application in linear run-time. The main limitation of this algorithm is that the number of outputs is restricted to to one, and the number of inputs is not parameterizable. In [13], a cone is defined as a subgraph consisting of a given node and a set of its connected predecessors. A dynamic programming based algorithm is proposed for the identification of single output cones as instruction-set extension candidates. The algorithm has exponential worst case time complexity, but it is very fast in practice. The number of inputs is parameterizable, but the number of outputs is restricted to one.

In [14] a complete tool chain based on Trimaran [19] is described. Candidate generation is based on a heuristic guide function and compilation with instruction-set extensions is formulated as a subgraph isomorphism problem. Machine description is updated automatically to support the instruction-set extensions. Extensive simulation results are presented.

In [15], an exact enumerative algorithm based on constraint propagation is proposed, that could handle any input and output combination without imposing any constraint on latency, or connectivity of instruction-set extensions. The algorithm is observed to handle dataflow graphs with up to 100 operations reasonably fast, but its performance for larger dataflow graphs is often unacceptable. Another limitation of this work is that there is no notion of instruction reuse, an instruction-set extension has only a single instance in the code.

In [16], the notion of cones used in [13] is extended to include backward cones. At each node the set of upward and downward cones are computed and saved. Later, the backward and forward cones are combined to enumerate all possible connected instruction-set extension candidates satisfying input and output constraints. The algorithm is observed to be faster than the algorithm of [15]. However, it is not able to identify parallel, disconnected components of the dataflow graph as part of a single instruction-set extension.

ILP based solutions have long been used in the HW/SW co-design area. Niemann et al. present a very elaborated example of such a work in [17]. However, the problem of instruction-set extension identification under input and output constraints has not yet been formulated as an ILP problem in the literature. Today's increasingly advanced ILP solvers such as CPLEX [20] are often able to solve problems with a few thousands of integer variables and tens of thousands of constraints efficiently. CPLEX Mixed Integer Optimizer incorporates state of the art techniques, such as cutting plane algorithms, heuristics, reduction algorithms, and a variety of branching techniques. To take advantage of this widely used sophisticated package, we formulate the instruction-set extension identification problem as an ILP.

The paper is organized as follows: Section 3 describes an ILP model for identifying a single template given a basic block. An algorithm that iteratively generates and solves a set of ILP problems to generate a set of templates from a basic block is described in Section 4. The set of all templates generated from all basic blocks are passed to the template selection algorithm of Section 5, which groups together isomorphic templates and ranks them based on their potential. The experimental setup and the results are presented in Sections 6 and 7.

3. ILP MODEL FOR IDENTIFICATION

The dataflow within a basic block is represented by a directed acyclic graph $G(V,E)$, where the nodes V represent operations and the edges E represent data dependencies between operations. Each dataflow graph G is associated with an extended dataflow graph $G^{ext}\left(V^{ext},E^{ext}\right)$ where $V^{ext}=V\cup V^{in}$, and $E^{ext}=E\cup E^{in}$. The additional nodes V^{in} represent input variables of the basic block. The additional edges E^{in} connect nodes V^{in} to V. The node set $V^{out}\subseteq V$ represent the operations generating the output variables of the basic block.

An instruction template T is an induced *subgraph* of G. A template T is *convex* if there exists no path from a node $u\in T$ to another node $v\in T$ which involves a node $w\notin T$. Our aim is to identify templates having less than or equal to N_{in} inputs, having less than or equal to N_{out} outputs, satisfying the convexity constraint, that maximise a certain metric. Such a metric should be a measure of speedup achievable when the template is executed as a single instruction in a specialised datapath. Fixing the values N_{in} and N_{out}, means a fixed communication cost. The convexity constraint is imposed on the templates to ensure that the compiler can generate a feasible schedule after the introduction of specialised instructions.

The definition of the indices we use in our formulations is as follows:

$$
\begin{aligned}
I_1 &= \{1..n_1\} && \textit{indices for nodes } v_{1i}\in V\\
I_2 &= \{1..n_2\} && \textit{indices for nodes } v_{2i}\in V^{in}\\
I_3 &= \{1..n_3\} && \textit{indices for nodes } v_{3i}\in V^{out}\\
I_4 &= \{1..n_4\} && \textit{indices for nodes } v_{4i}\in V/V^{out}
\end{aligned}
$$

We associate with each dataflow graph node a binary decision variable x_i that represents whether the node is contained in the template or not. We use x_i' to denote the complement of x_i. If $x_i=1$, the node is contained in the template. Otherwise, $x_i'=1$ and the node is not contained in the template. The formal definition is as follows:

$$x_i, x_i' \in \{0,1\} \qquad x_i' = 1 - x_i \qquad i\in I_1$$

To be able to formulate the problem, the list of successor nodes and the list of predecessor nodes of nodes in V as well as the list of successor nodes of nodes in V^{in} must be known. This information is formally defined as follows:

$$Succ(i \in I_1) = \{j \in I_1 \mid \exists e \in E : e = (v_{1i}, v_{1j})\}$$
$$Pred(i \in I_1) = \{j \in I_1 \mid \exists e \in E : e = (v_{1j}, v_{1i})\}$$
$$Succ(i \in I_2) = \left\{j \in I_1 \mid \exists e \in E^{in} : e = (v_{2i}, v_{1j})\right\}$$

For the sake of simplicity, we demonstrate our formulations making use of and/or type operations. Such operations can be easily converted to linear form by introducing additional variables and constraints.

3.1 Input Port Constraint

A dataflow graph node is an input node for the template T if it is not contained in T, and it has at least one successor contained in T. Similarly, an input node for the basic block is an input node for T if it has at least one successor contained in T.

$$\sum_{i \in I_1} \left(x'_i \wedge \left(\bigvee_{j \in Succ(i)} x_j \right) \right) + \sum_{i \in I_2} \left(\bigvee_{j \in Succ(i)} x_j \right) \leq N_{\text{in}} \tag{1}$$

3.2 Output Port Constraint

A dataflow graph node is an output node for the template T if it is contained in T and it has at least one successor not contained in T. A node in T is automatically an output node if it is an output node for the basic block.

$$\sum_{i \in I_3} x_i + \sum_{i \in I_4} \left(x_i \wedge \left(\bigvee_{j \in Succ(i)} x'_j \right) \right) \leq N_{\text{out}} \tag{2}$$

3.3 Convexity Constraint

For each dataflow graph node we introduce two new decision variables A_i and D_i. $A_i = 1$ if the node has at least one ancestor contained in the template, $A_i = 0$ otherwise. Similarly, $D_i = 1$ if the node has at least one descendant contained in the template, $D_i = 0$ otherwise. More formally,

$$A_i, D_i \in \{0, 1\} \qquad i \in I_1$$

$$A_i = \begin{cases} 0 & if Pred(i) = \emptyset \\ \left(\bigvee_{j \in Pred(i)} (x_j \vee A_j) \right) & otherwise \end{cases} \tag{3}$$

$$D_i = \begin{cases} 0 & if Succ(i) = \emptyset \\ \left(\bigvee_{j \in Succ(i)} (x_j \vee D_j) \right) & otherwise \end{cases} \tag{4}$$

To preserve the convexity, there should be no dataflow graph node that is not contained in T, having both an ancestor and a descendant contained in T:

$$x'_i \wedge A_i \wedge D_i = 0 \qquad i \in I_1 \tag{5}$$

3.4 Objective

The objective function is an estimation of reduction in the schedule length of the basic block when the template T is implemented as a special instruction. The software latency of the special instruction is estimated by quantizing its critical path length. To be able to calculate the objective, we associate with each dataflow graph node pre-computed software and hardware latencies s_i and h_i, where s_i is integer and h_i is real. In addition, we associate with each dataflow graph node a real decision variable l_i that represents the completion time of the operation corresponding to that node when the template T is executed in hardware. The largest such value gives us the critical path length L of T. The objective is the difference between the sum of the software latencies of the nodes contained in T and the ceiling of the critical path length of T. The nonlinearity introduced by the ceiling function can simply be avoided by defining L as an integer decision variable. Formally:

$$max \sum_{i \in I_1} (s_i x_i) - \lceil L \rceil \tag{6}$$
$$l_i = x_i h_i \qquad if\ Pred(i) = \emptyset \tag{7}$$
$$l_j \geq l_i + x_j h_j \qquad j \in Succ(i), i \in I_1 \tag{8}$$
$$L \geq l_i \qquad if\ Succ(i) = \emptyset \tag{9}$$

4. TEMPLATE GENERATION

Our template generation algorithm iteratively solves a set of ILP problems to generate a set of templates. For a given basic block, the first template is identified by solving the ILP problem as defined in Section 3. After the identification of the first template, the dataflow graph nodes contained in the template are collapsed to a single graph node, and adjacency information is updated. The updated graph is the input of the second iteration with the restriction that the graph node representing the previously identified template cannot be part of new templates. By imposing this restriction, we limit the number of templates to a linear function of the dataflow graph size. The process is continued until the solution of the ILP problem does not return a positive objective value. The same procedure is applied for all basic blocks. A more formal description is given in Figure 1.

```
PROCEDURE TEMPLATE_GENERATION
S : The set of templates to be generated
$I_B = \{1..n_b\}$ : Indices for basic blocks
$G_i(V_i, E_i)$ : Dataflow graph of basic block i
$G_i^{ext}(V_i^{ext}, E_i^{ext})$ : Extended dataflow graph of basic block i
$V_i^{out}$ : Output nodes of basic block i
BEGIN
  $S = \emptyset$
  FOR $i \in I_B$
    Generate $G_i$, $G_i^{ext}$, and $V_i^{out}$
    DO
      Generate ILP problem for $G_i$, $G_i^{ext}$, and $V_i^{out}$
      Identify template T solving the ILP problem
      IF (objective $\geq$ 0 )
        $S = S \cup \{T\}$
        Collapse T into a single graph node
        Disable inclusion of the node within new templates
        Update $G_i$, $G_i^{ext}$, and $V_i^{out}$
    WHILE (objective $\geq$ 0 )
END
```

Figure 1: Template Generation Algorithm

5. TEMPLATE SELECTION

Our template selection algorithm first applies pairwise isomorphism checks on the set of templates generated by the algorithm of Figure 1. The isomorphism checks are done using the nauty package [18]. The nauty package is known as one of the fastest isomorphism checking tools. It makes use of a backtracking algorithm that produces automorphism groups of a given graph, as well as the canonically labelled isomorph of the graph. To understand whether two graphs are isomorphic, it is enough to compare their canonically labelled isomorphs.

A set of isomorphic templates defines an isomorphism class. Once the isomorphism classes are determined, potential evaluation starts. The potential of a template is the value of the objective function described in Section 3.4 multiplied by the frequency of execution of the basic block that contains the template. The potential of an isomorphism class is the sum of the potentials of the individual templates contained in that class. The classes are then ranked according to their potentials. We choose the first N_k classes, where N_k is a user defined parameter.

A chosen isomorphism class defines an instruction-set extension that can implement the whole set of templates included in that class. Each template corresponds to a piece of code segment of the application. These code segments are later replaced with the new instruction.

6. EXPERIMENTAL SETUP

We use the Trimaran [19] framework to generate the control/dataflow information, and to achieve basic block level profiling of a given application. Specifically, we work with Elcor, the back-end of Trimaran. Elcor is associated with a parameterizable machine description for which it generates machine code. Once a subset of opcodes are reserved for instruction-set extensions, it is possible to introduce them in the code specifying only the opcode id. Elcor does not need to be internally aware of the structure of the instruction-set extensions, but it can obtain this information by making calls to a machine description database.

We read the Elcor intermediate representation after applying classical compiler optimizations, and before scheduling and register allocation are done. At this point we apply our algorithms to identify the instruction-set extensions. We use the CPLEX Mixed Integer Optimizer [20] within our algorithms to solve the ILP problems generated throughout their execution.

We use a single-issue machine as the baseline. We search for instruction-set extension candidates within basic block boundaries. We disable inclusion of memory access, and branch operations within instruction-set extensions. We calculate hardware latencies of various operations using a circuit complexity estimation methodology described in [21], and we normalize the latencies based on the latency of a 32-bit carry propagate adder. We use software latencies similar to latency specifications of a popular embedded processor.

Once the instruction-set extensions are identified, we configure the baseline machine to support the new instructions, and we replace the associated code segments with the new instructions. After that, we apply standard Trimaran scheduling and register allocation passes on the new code with instruction-set extensions. We calculate the cycles spent in a basic block by multiplying its schedule length by its frequency of execution. We calculate the total cycles spent in an application as the sum of the cycles spent in its basic blocks. We assume no change in the processor cycle time due to the introduction of instruction-set extensions.

7. RESULTS

We chose Advanced Encryption Standard to demonstrate the effectiveness of our algorithms. The core of the AES encryption is the Round Transformation (see Figure 2), which operates on a 16-byte State. The State can be considered as a two dimensional array of bytes having four rows, and four columns. The columns are often stored in four 32-bit registers, and are inputs and outputs of the round transformation. First, a nonlinear byte substitution is applied on each of the State bytes by making table lookups from S-Boxes stored in the memory. Next, the rows of the State Array are rotated over different offsets. After that, a linear transformation called MixColumn Transformation is applied on each column. The final operation of the Round Transformation is an XOR with Round Key. The output of a round transformation, becomes the input of the next round transformation. Very often, several round transformations are unrolled within a loop, resulting in very large basic blocks consisting of several hundreds of operations.

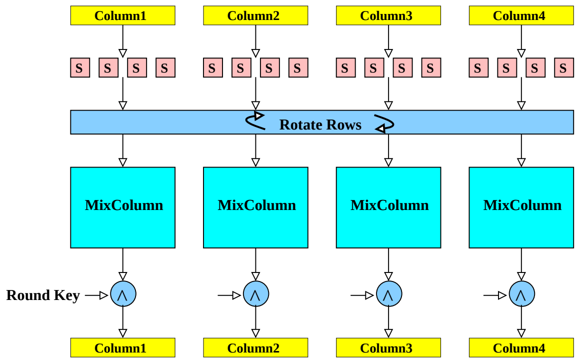

Figure 2: The Round Transformation

The most compute-intensive part of AES encryption is the MixColumn Transformation. The MixColumn Transformation is a single input, single output transformation consisting of around 20 simple bitwise operations with many constant coefficients. Its critical path is smaller than the latency of a 32-bit adder. It would be the most likely choice for a manual designer as a special instruction. The dataflow of operations implementing the transformation is depicted in Fig. 3.

We used a highly optimized 32-bit implementation of AES described in [22]. As shown in Figure 4, two Round Transformations are unrolled within a loop, resulting in the largest basic block of the application consisting of around 350 operations. A second basic block consists of a single Round Transformation followed by the Final Round Transformation, which does not incorporate MixColumn Transformations. The code also includes an initialization stage, where State is read from memory and reorganized for fast processing, and a finalization stage, where State is written back to memory in its original format.

Given an input port constraint of 1, and an output port constraint of 1, our template generation algorithm successfully finds the 12 instances of the MixColumn Transformations within the two basic blocks of the ENCRYPTBLOCK function, and our isomorphism checking algorithm successfully combines the 12 templates within a single isomorphism class as the most promising instruction set extension of the application. Given an input port constraint of 2, and an out-

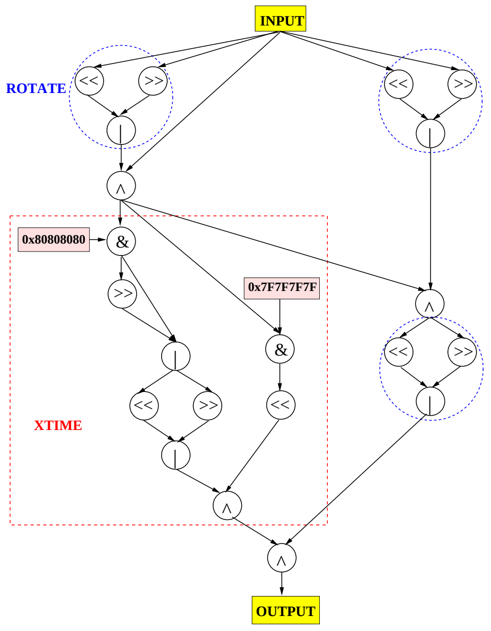

Figure 3: The MixColumn Transformation

```
FUNCTION ENCRYPTBLOCK
BEGIN
    Initialize(State)
    FOR i in {1..num_rounds/2}
    BEGIN
        Round(State, RoundKey++) ;
        Round(State, RoundKey++) ;
    END
    Round(State, RoundKey++) ;
    FinalRound(State, RoundKey) ;
    Finalize(State)
END
```

Figure 4: Implementation of AES Encryption

put port constraint of 2, our tool successfully identifies the two parallel MixColumn Transformations within a Round Transformation and matches it with five other isomorphic templates. Given an input port constraint of 4, and an output port constraint of 4, our tool successfully identifies the four parallel MixColumn Transformations within a Round Transformation. All three instances of the 4-input 4-output instruction set extension are matched in the code.

The AES decryption is very similar to AES encryption. Basically, Inverse MixColumn Transformations replace MixColumn Transformations in the Round Transformation. The Inverse MixColumn Transformations are again single-input, single-output transformations. However, they are more complex compared to MixColumn Transformations comprising around 40 bitwise operations. Again, two Round Transformations are unrolled within a loop, resulting in the largest basic block of the application consisting of around 550 operations. Again, our tool successfully identifies single or multiple parallel Inverse MixColumn Transformations as the most promising instruction-set extension candidates within the application basic blocks.

The number of instances matched in the code for the best 8 instruction-set extension candidates under different input and output constraints for AES encryption and AES decryption benchmarks are presented in Table 1. The degree of instruction reuse is high. Up to 18 instances of a single instruction are found for both benchmarks. The speed-up results obtained for the two benchmarks under different input and output constrains up to 8 instruction-set extensions are presented in Figures 5, and 6. The x-axis represents a set of constraints consisting of (N_{in},N_{out},N_k) tuples, and the y-axis represents the corresponding speed-up values in the figures. The figures show that a speed-up of more than 4.5 can be reached for AES encryption. In the case of decryption larger logic blocks are mapped from software to hardware, and speed-up results of up to 7.5 can be reached. Only a limited amount of resources are needed in all the cases.

Table 1: No. Instances of ISEs

AES Encryption								
N_{in},N_{out}	1st	2nd	3rd	4th	5th	6th	7th	8th
1,1	12	-	-	-	-	-	-	-
2,2	6	16	18	8	2	6	1	1
4,4	3	7	4	6	2	1	1	1

AES Decryption								
N_{in},N_{out}	1st	2nd	3rd	4th	5th	6th	7th	8th
1,1	12	-	-	-	-	-	-	-
2,2	6	13	18	7	6	1	1	3
4,4	3	7	4	6	2	2	2	1

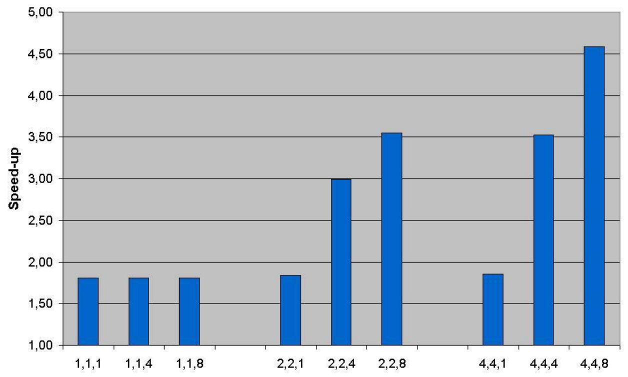

Figure 5: Speed-up Results for AES Encryption

The performance of our algorithms is quite notable. The identification algorithm of [15] fails to generate a result for the AES benchmark in several days. Under the same constraints and under the same objective function our ILP-based template identification algorithm finds the optimal solutions in only a few seconds. Figure 7 depicts the time taken by ILP solver to solve the set of identification problems generated throughout the execution of the template generation algorithm of Figure 1 on AES encryption and decryption benchmarks. The largest identification time we observed on a Pentium 4, 3.0 GHz machine with 1GB memory was around 25 seconds. The largest execution time we observed for the overall algorithm was only about 2 minutes.

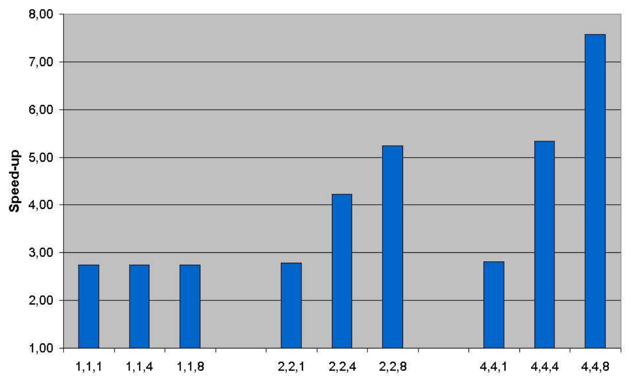

Figure 6: Speed-up Results for AES Decryption

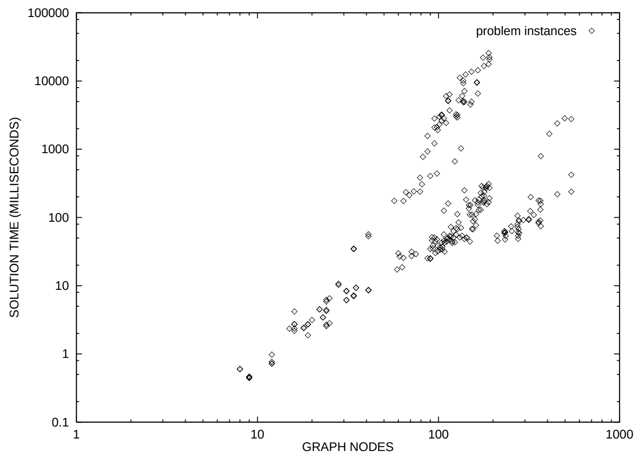

Figure 7: Performance of ILP Based Identification

8. CONCLUSIONS

We proposed an ILP based solution to the instruction-set identification problem. We start with a high-level description of a given application and generate a set of templates using the ILP based method. A template selection methodology based on isomorphism testing and a potential evaluation function results in combined instruction-set extension identification and code generation. Our algorithms can optimally identify multiple-input multiple output instruction-set extension candidates without imposing any constraint on the latency or connectivity. Our algorithms can handle benchmarks with very large basic blocks on which state of the art algorithms fail to reach a solution. The success of our approach is demonstrated on AES where MixColumn and Inverse MixColumn Transformations are successfully identified, and speed-up results of up to 7.5 are reached.

9. REFERENCES

[1] D. Goodwin, D. Petkov. Automatic Generation of Application Specific Processors. In *CASES 2003*, pages 137–147, San Jose, CA, Nov. 2003.

[2] Tensilica, http://www.tensilica.com

[3] MIPS, http://www.mips.com

[4] ARC, http://www.arc.com

[5] ARM, http://www.arm.com

[6] N.N. Binh, M. Imai, A. Shiomi, N. Hikichi. A Hardware/Software Partitioning Algorithm for Designing Pipelined ASIPs with Least Gate Counts. In *33rd DAC*, pages 527–532, Las Vegas, Nevada, 1996.

[7] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man. Instruction set definition and instruction selection for ASIPs. In *Proceedings of the 7th International Symposium on High-Level Synthesis*, pages 11–16, Apr. 1994.

[8] M. Arnold and H. Corporaal. Designing domain specific processors. In *Proceedings of the 9th International Workshop on Hardware/Software Codesign*, pages 61–66, Copenhagen, Apr. 2001.

[9] P. Brisk, A. Kaplan, R. Kastner, M. Sarrafzadeh. Instruction Generation and Regularity Extraction For Reconfigurable Processors In *CASES 2002*, pages 262–269, Grenoble, France, 2002

[10] I.-J. Huang and A. M. Despain. Synthesis of application specific instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-14(6):663–75, Jun. 1995.

[11] H. Choi, J.-S. Kim, C.-W. Yoon, I.-C. Park, S. H. Hwang, and C.-M. Kyung. Synthesis of application specific instructions for embedded DSP software. *IEEE Transactions on Computers*, C-48(6):603–14, Jun. 1999.

[12] C. Alippi, W. Fornaciari, L. Pozzi, and M. Sami. A DAG based design approach for reconfigurable VLIW processors. In *DATE 1999*, pages 778–79, Mar. 1999.

[13] J. Cong, Y. Fan, G. Han, Z. Zhang. Application-Specific Instruction Generation for Configurable Processor Architectures. In *FPGA 2004*, pages 183–189, Monterey, CA, Feb. 2004.

[14] N. Clark, H. Zhong, S. Mahlke. Processor Acceleration Through Automated Instruction Set Customization. In *36th MICRO*, pages 184–88, San Diego, CA, Dec. 2003.

[15] K. Atasu, L. Pozzi, P. Ienne. Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints. In *40th DAC*, Anaheim, California, Jun. 2003.

[16] P. Yu, T. Mitra. Scalable Custom Instructions Identification for Instruction-Set Extensible Processors. In *CASES 2004*, Washington, DC, Sep. 2004.

[17] R. Niemann, P. Marwedel. An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming. *Design Automation for Embedded Systems*, Vol. 2, No. 2, pages 165–193, Mar. 1997

[18] Nauty Package. http://cs.anu.edu.au/people/bdm/nauty.

[19] Trimaran: An Infrastructure for Research in Instruction Level Parallelism. http://www.trimaran.org.

[20] ILOG CPLEX :High-Performance Software for Mathematical Programming and Optimization. http://www.ilog.com/products/cplex/

[21] R. Zimmermann. Computer Arithmetic: Principles, Architectures, and VLSI Design, Lecture notes, Integrated Systems Laboratory, ETH Zürich, 1997

[22] K. Atasu, M. Macchetti, L. Breveglieri. Efficient AES implementations for ARM Based Platforms. In *ACM SAC 2004*, Nicosia, Cyprus, Mar. 2004.