Optimizing Instruction-set Extensible Processors under Data Bandwidth Constraints

Kubilay Atasu[*] Robert G. Dimond, Oskar Mencer, Wayne Luk
Department of Computing, Imperial College London
{atasu,rgd,oskar,wl}@doc.ic.ac.uk

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

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

Abstract

We present a methodology for generating optimized architectures for data bandwidth constrained extensible processors. We describe a scalable Integer Linear Programming (ILP) formulation, that extracts the most profitable set of instruction-set extensions given the available data bandwidth and transfer latency. Unlike previous approaches, we differentiate between number of inputs and outputs for instruction-set extensions and the number of register file ports. This differentiation makes our approach applicable to architectures that include architecturally visible state registers and dedicated data transfer channels. We support a comprehensive design space exploration to characterize the area/performance trade-offs for various applications. We evaluate our approach using actual ASIC implementations to demonstrate that our automatically customized processors meet timing within the target silicon area. For an embedded processor with only two register read ports and one register write port, we obtain up to 4.3× speed-up with extensions incurring only a 35% area overhead.

1 Introduction

Application-specific instruction-set processors (ASIPs) provide a compromise between custom designs and general-purpose processors. A base processor with a basic instruction set is augmented with custom functional units that implement application-specific instruction-set extensions. The control-flow within the application is directed by the base processor, whereas computation intensive regions are implemented as custom logic. A dedicated link between custom logic and the base processor provides an efficient communication interface. Re-using a pre-verified, pre-optimized base processor reduces the design complexity, and the time to market. Several commercial examples exist, such as Tensilica Xtensa, Altera NiosII, Xilinx MicroBlaze, ARC 600/700, and MIPS Pro Series.

In this work, we apply formal optimization techniques to generate instruction-set extensions from C code. We target architectures, such as Tensilica Xtensa, where the data bandwidth between the base processor and the custom logic is constrained by the available register file ports (see Figure 1). Our approach is also applicable to architectures where the data bandwidth is limited by dedicated data transfer channels, such as the Fast Simplex Link channels of Xilinx MicroBlaze processor. Given the available data bandwidth and transfer latencies, our approach identifies the most-profitable instruction-set extensions based on a scalable Integer Linear Programming (ILP) model. We explicitly consider the data transfer overhead when generating and evaluating instruction-set extensions. We demonstrate that our automatically customized processors meet timing within the target silicon area using ASIC synthesis results.

Our main contributions in this work are:

1. We provide an ILP model which replaces the input/output abstraction of the previous approaches [5, 6, 7, 10, 15] with the actual data bandwidth constraints and data transfer costs.

2. We integrate our technique into an optimizing compiler that generates custom ASIC processor implementations from C code.

3. We consider silicon cell area as a primary constraint, and we explore the impact of different area constraints on the number of execution cycles and cycle time.

[*]Kubilay Atasu is also affiliated with the Department of Computer Engineering, Bogazici University, Istanbul

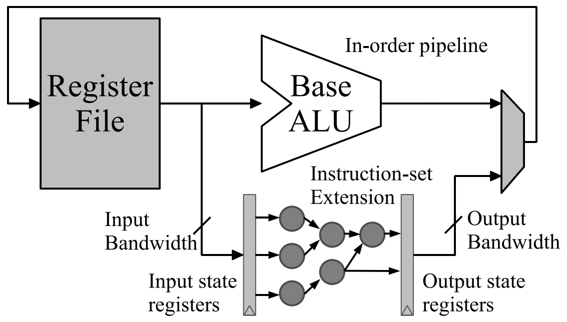

Figure 1. Datapath of the instruction-set extensible processor: data bandwidth may be limited by the register file ports or the dedicated data transfer channels

2 Related Work

The speed-up obtainable by instruction-set extensions is limited by the available data bandwidth between the base processor and custom logic. A multi-ported register file can increase the data bandwidth. However, additional read and write ports result in increased register file size, power consumption and cycle time. The Tensilica Xtensa [12] uses state registers to explicitly move additional input and output operands between the base processor and custom units. Clever binding of base processor registers to state registers at compile time reduces the number of data transfers. In addition, state register approach solves the problem of encoding many operands within a fixed length instruction word.

Shadow registers duplicate a subset of base processor registers [9] to increase the data bandwidth. The mapping between base processor registers and shadow registers can be fixed, or established at compile time. Contents of shadow registers can be read without a limitation on the bandwidth. Jayaseelan et al. [13] show that up to two additional input operands for instruction-set extensions can be supplied free of cost by exploiting the forwarding paths of the processor. Pozzi et al. [14] reduce the data transfer overhead by overlapping execution cycles with data transfers cycles for pipelined multi-cycle instruction-set extensions.

Automatic identification of instruction set extensions from high level application descriptions has received considerable attention in the recent years. In [8], related dataflow graph (DFG) nodes are heuristically clustered as sequential or parallel templates. In [6], input and output constraints are imposed on the subgraphs to reduce the exponential search space. Application of a constraint propagation technique results in an efficient enumerative algorithm. However, the applicability of the approach is limited to DFGs with around 100 nodes. Search space can be further reduced by imposing additional constraints such as single output [10], or connectivity [15] constraints on the subgraphs. In [5], Atasu et al. formulate the problem of identifying instruction-set extensions under input and output constraints as an ILP. Biswas et al. propose an extension to the Kernighan-Lin heuristic, again based on input and output constraints in [7].

In previous work [6, 7, 10, 15], optimality is limited by either an approximate search algorithm or some artificial constraints (such as input/output constraints) that make subgraph enumeration tractable. In this work, we extend the ILP formulation of [5], replacing the input/output constraints with the actual data bandwidth constraints and data transfer costs. The instruction-set extensions we generate may have an unlimited number of inputs and outputs. A baseline machine with architecturally visible state registers makes our approach feasible. We integrate the data bandwidth information directly into the optimization process, and we explicitly account for the cost of the data transfers between the core register file and custom state registers as part of the optimization.

The approaches described in [13] and [9] are complementary to ours, since our formulation can take advantage of the increased data bandwidth. The approach of Pozzi et al. [14] can be combined with ours to further optimize the performance of multi-cycle instruction-set extensions.

3 The Compilation Flow

We use the Trimaran [4] 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. We read Elcor intermediate representation after applying classical compiler optimizations. Immediately prior to register allocation, we apply our algorithms to identify the instruction-set extensions. We use the industry standard CPLEX Mixed Integer Optimizer [1] within our algorithms to solve our ILP problems.

An instruction-set extension template is a dataflow subgraph that can potentially be replaced by an instruction-set extension. We generate a set of instruction-set extension templates based on an ILP formulation described in Section 5. Next, we group structurally equivalent templates within isomorphism classes as instruction-set extension candidates. We generate the behavioral hardware descriptions of instruction-set extension candidates in VHDL, and we produce area estimates using Synopsys Design Compiler. After that, we select a a subset of the instruction-set extension candidates under a set of area constraints based on a Knapsack model.

Once the most profitable instruction-set extension candidates are selected under area constraints, we automatically generate a high level machine description (MDES) supporting the selected instructions. Next, for each selected instruction, we replace the matching code segments with an opcode representing the new instruction. After that, we apply standard Trimaran scheduling and register allocation passes on the code with the new instructions. Finally,

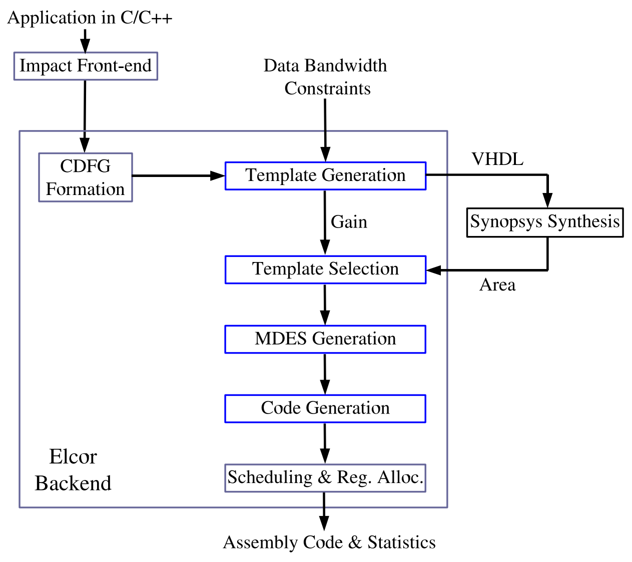

Figure 2. The compilation flow: we integrate our algorithms into the Trimaran framework [4]. Starting with C code, we automatically generate customized machine descriptions and assembly code.

we generate the assembly code, and collect the scheduling statistics. Figure 2 depicts our tool chain structure.

4 Template Generation and Selection

Our template generation algorithm iteratively solves a set of ILP problems in order to generate a set of templates. For a given application basic block, the first template is identified by solving the ILP problem as defined in Section 5. After the identification of the first template, the dataflow graph nodes contained in the template are collapsed into a single node, and the same procedure is applied for the rest of the graph. The process is continued until no more profitable templates are found. Template generation algorithm is applied on all application basic blocks, and a unified set of instruction-set extensions templates are generated.

After template generation is done, we calculate the isomorphism classes using the nauty package [3]. We assume that the set of generated templates $\mathcal{T}$ is partitioned into N_G distinct isomorphism classes:

$$\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2 \cup ... \cup \mathcal{T}_{N_G} \qquad (1)$$

The weight $W(T)$ of the template T is defined as the value of the objective function $Z(T)$ described in Section 5.3 multiplied by the frequency of execution $F(T)$ of the template, which estimates the reduction in the schedule length of the application by replacing the template with an instruction-set extension candidate. More formally:

$$W(T) = Z(T) * F(T) \qquad (2)$$

The weight of an isomorphism class is defined as the sum of the weights of all the templates within that class, which estimates the reduction in the schedule length of the application by replacing all the templates with an instruction-set extension candidate representing the isomorphism class.

$$W(\mathcal{T}_i) = \sum_{T \in \mathcal{T}_i} W(T), \qquad i \in \{1..N_G\} \qquad (3)$$

At this point, we generate behavioral descriptions of the instruction set extension candidates, and produce area estimates using high level synthesis. As a result, we associate with each instruction set extension candidate $\mathcal{T}_i$ an area estimate $A(\mathcal{T}_i)$. We formulate the selection of most profitable instruction-set extension candidates under area constraint A_{MAX} as a Knapsack problem, and solve it using ILP:

$$\begin{aligned} max \quad & \sum_{i \in \{1..N_G\}} W(\mathcal{T}_i) y_i \\ s.t. \quad & \sum_{i \in \{1..N_G\}} A(\mathcal{T}_i) y_i \le A_{MAX} \\ & y_i \in \{0,1\}, \qquad i \in 1..N_G \end{aligned} \qquad (4)$$

where the binary decision variable y_i represents whether candidate i is selected ($y_i = 1$) or not ($y_i = 0$).

5 ILP Model for Template Identification

We represent a basic block using a directed acyclic graph $G\left(V \cup V^{in}, E \cup E^{in}\right)$ where nodes V represent operations, edges E represent register flow dependencies between operations, nodes V^{in} represent input variables of the basic block, and edges E^{in} connect input variables V^{in} to consumer operations in V. Nodes $V^{out} \subseteq V$ represents operations generating output variables of the basic block.

An instruction-set extension template T is an induced *subgraph* of G. We associate with each dataflow graph node a binary decision variable x_i that represents whether the node is contained in the template ($x_i = 1$) or not ($x_i = 0$). We use x_i' to denote the complement of x_i ($x_i' = 1 - x_i$). A template T is *convex* if there exists no path in G from a node $u \in T$ to another node $v \in T$ which involves a node $w \notin T$. The convexity constraint is imposed on the templates to ensure that no cyclic dependencies are introduced in G, and a feasible schedule can be generated.

We associate with every graph node v_i a software latency s_i, and a hardware latency h_i, where s_i is integer and h_i is real. We normalize hardware latencies based on the latency of a 32-bit adder. We estimate the execution latency of a template T on the processor pipeline as an instruction-set extension by quantizing its critical path length L.

We assume RF_{in} read ports, and RF_{out} write ports supported by the core register file. If the number of inputs for a template is larger than RF_{in}, we assume additional data transfers from the core register file to custom state registers. If the number of outputs for a template is larger than $RF_{\rm out}$, we assume additional data transfers from custom state registers to the core register file. We assume a fixed cost of c_1 cycles for transferring additional $RF_{\rm in}$ inputs, and a fixed cost of c_2 cycles for transferring additional $RF_{\rm out}$ outputs.

We use the following indices in our formulations:

$$\begin{aligned} I_1 &: \quad \textit{indices for nodes } v_i^{in} \in V^{in} \\ I_2 &: \quad \textit{indices for nodes } v_i \in V \\ I_3 &: \quad \textit{indices for nodes } v_i \in V^{out} \\ I_4 &: \quad \textit{indices for nodes } v_i \in V/V^{out} \end{aligned}$$

5.1 Calculation of input data transfers

We introduce an integer decision variable $N_{\rm in}$ to compute the number of inputs for a template. An input operand $v_i^{in} \in V^{in}$ of the basic block is an input of the template T if it has at least one immediate successor in T. A node $v_i \in V$ generates an input operand of T if it is not in T, and it has at least one immediate successor in T.

$$N_{\rm in} = \sum_{i \in I_1} \left(\vee_{j \in Succ(i)} x_j \right) + \sum_{i \in I_2} \left(x'_i \wedge \left(\vee_{j \in Succ(i)} x_j \right) \right) \tag{5}$$

We calculate the number of additional data transfers from the core register file to the custom logic as $D_{\rm in}$:

$$D_{\rm in} \geq N_{\rm in}/RF_{\rm in} - 1, \qquad D_{\rm in} \in Z^+ \cup \{0\} \tag{6}$$

5.2 Calculation of output data transfers

We introduce an integer decision variable $N_{\rm out}$ to compute the number of outputs for a template. A node $v_i \in V^{out}$, generating an output operand of the basic block, generates an output operand of the template T if it is in T. A node $v_i \in V/V^{out}$ generates an output operand of T if it is in T, and it has at least one immediate successor not in T.

$$N_{\rm out} = \sum_{i \in I_3} x_i + \sum_{i \in I_4} \left(x_i \wedge \left(\vee_{j \in Succ(i)} x'_j \right) \right) \tag{7}$$

We calculate the number of additional data transfers from the custom logic to the core register file as $D_{\rm out}$:

$$D_{\rm out} \geq N_{\rm out}/RF_{\rm out} - 1, \qquad D_{\rm out} \in Z^+ \cup \{0\} \tag{8}$$

5.3 Objective

Our objective is to maximize the decrease in the schedule length by moving template T from software to the custom logic. We estimate the software cost of T as the sum of the software latencies of the instructions contained in T. We estimate the cost of moving T to a custom datapath as the sum of its estimated hardware execution latency L, and the number of cycles required to transfer its input and output operands. The objective function is defined as follows:

$$Z(T) = max \sum_{i \in I_2} (s_i x_i) - (L + c_1 D_{\rm in} + c_2 D_{\rm out}) \tag{9}$$

6 Experimental Setup and Results

We evaluate our technique using Trimaran scheduling statistics to estimate cycle counts, and hardware synthesis for exact timing and area information. We use our own in-order extensible processor [11] that implements the MIPS integer instruction set and supports up to 512 instruction-set extensions. Our core register file supports two read ports and a single write port. We generate state registers for each instruction extension operand and hardware move instructions that provide single cycle latency transfers between register file and custom units ($c_1 = c_2 = 1$).

We apply our algorithms on four encryption benchmarks with very large basic blocks to demonstrate the scalability of our approach: optimized 32-bit implementations of AES (Advanced Encryption Standard) encryption and decryption, DES (Data Encryption Standard), and SHA (Secure Hash Algorithm) from the Mibench suite [2]. DES and SHA are fully unrolled, resulting in basic blocks with more than a thousand instructions.

In Figures 3 and 4 we analyze the effect of different input and output constraints on the speed-up potentials of instruction-set extensions assuming a register file with 2 read ports and 1 write port. We scale the initial cycle count down to 100, and we plot the percent decrease in the cycle count for a range of area constraints (4 to 32 ripple carry adders). Relaxation of the input/output constraints results in coarser grain instruction-set extensions (i.e., larger dataflow subgraphs). Such extensions often offer higher speed-up at the expense of higher area. Figure 3 shows that imposing an input constraint of 2 and an output constraint of 1 (i.e., (2,1)) on the extensions, the cycle count for AES decryption is reduced to 29% at an area cost of 4 adders. On the other hand, 4-input 1-output extensions decrease the cycle count down to 23% at an area cost of 8 adders. Relaxing the input/output constraints completely (i.e., (∞,∞)) results in a slight reduction only, at an area cost of 32 adders. Figure 4 shows that 2-input 1-output extensions reduce the cycle count for DES down to 67%. 4-input 4-output extensions can exploit more parallelism, and the cycle count decreases to 56%. The best speed-up for DES is achieved when the input/output constraints are completely relaxed, where the cycle count is reduced to 52% at an area cost of 20 adders. This solution incorporates an 11-input 9-output extension, which is reused 12 times in the application.

In Figure 5, we assume a register file with 2 read ports and 1 write port and an area constraint of 24 adders. The

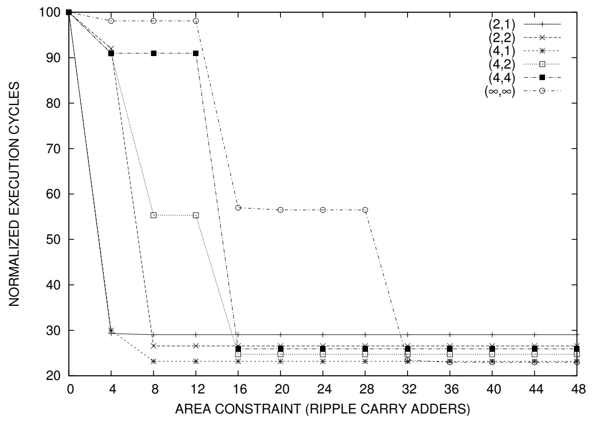

Figure 3. AES decryption: reduction in execution cycles. Register file support 2 read ports and 1 write port. (n,k) represents n-input k-output extensions. (∞,∞) represents no constraint on inputs or outputs.

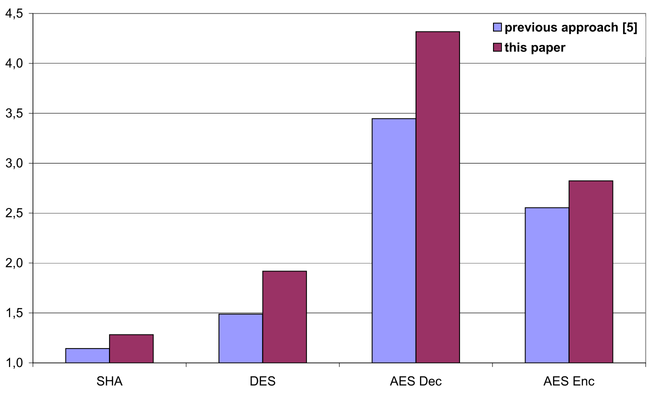

Figure 5. Speed-up improvement using a register file with 2 read ports and 1 write port. Previous approach [5] limits the number of inputs and outputs to the available register file ports.

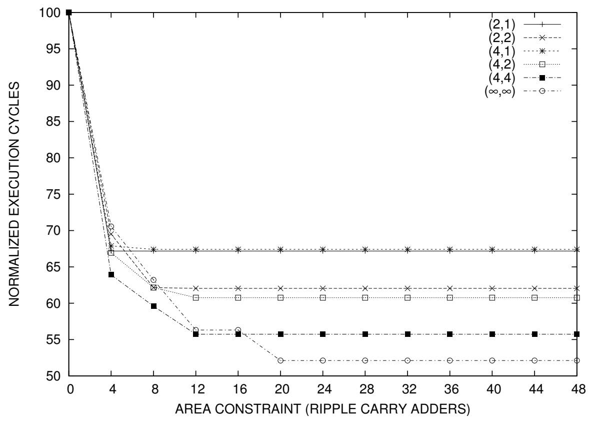

Figure 4. DES: reduction in execution cycles. Register file supports 2 read ports and 1 write port. (n,k) represents n-input k-output extensions. (∞,∞) represents no constraint on inputs or outputs.

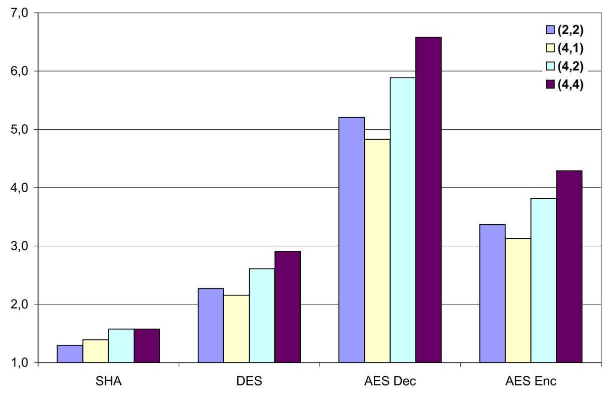

Figure 6. Speed-up improves with additional register file ports: (n,k) represents n read and k write ports.

previous approach [5] limits the number of inputs and outputs to the available register file ports. In contrast, the extensions we generate may have an unlimited number of inputs and outputs. Avoiding the previous limitation, we improve the speed-up from 1.1× to 1.3× for SHA, from 1.5× to 1.9× for DES, from 3.4× to 4.3× for AES Decryption, and from 2.6× to 2.8× for AES encryption.

In Figure 6, we study the improvement in speed-up using additional register file ports for an area constraint of 36 adders. A register file with 4 read and 2 write ports improves the speed-up to 1.6× for SHA, 2.6× for DES, 5.9× for AES decryption, and 3.8× for AES encryption. Up to 6.6× speed-up is reachable given 4 read and 4 write ports.

Assuming a register file with 2 read ports and 1 write port, we automatically generate a CPU core implementing the extensions selected for each area constraint. We obtain realistic timing and area results by synthesizing each core to UMC's 130nm standard cell library using Synopsys Design Compiler and Cadence SoC Encounter for routing and layout. The highest performance AES decryption processor we generate incurs only a 35% increase over the area of the unextended processor while offering a speed-up of 4.3×.

Figure 7 summarizes timing results for each generated processor (179 in total). The volume of designs prohibits manual optimization, hence we report the worst case negative slack with a 200MHz constraint for the tool vendor's recommended fully automated flow. Our technique pipelines multi-cycle instruction-set extensions to avoid decreasing the processor clock rate. Figure 7 shows that 48% of the customized designs meet timing in the first pass. A

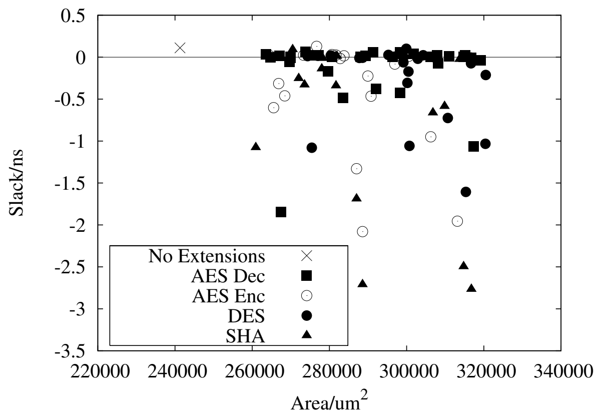

Figure 7. Standard cell ASIC area and the worst case negative timing slack with a 200MHz constraint on the clock rate.

further 31% marginally fail to meet timing (<1ns negative slack), and the remainder miss by a greater margin.

The most time consuming part of our algorithms is the template generation algorithm that iteratively solves ILP problems. Table 1 describes the ILP statistics associated with the first iteration of the template generation algorithm on the largest basic block of each benchmark given a constraint of (4,4) on the inputs and outputs. The solution time is generally only a few seconds. However, it may exceed one hour as it happens for SHA. We observed an overall runtime of 13 seconds for AES encryption, 20 seconds for AES decryption, 2.5 minutes for DES, and about 21.5 hours for SHA. We obtained optimal ILP results in all cases.

Benchmark	BB size	Vars	Constrs	Time
AES enc.	317	1403	4124	0.8
AES dec.	501	2483	7404	2.9
DES	822	3417	9760	10.4
SHA	1155	5899	18524	5116

Table 1. Size of the largest basic block (BB), number of integer variables (Vars), number of linear constraints (Constrs), and the solution time in seconds.

7 Conclusions

This work describes a comprehensive design flow exploration to identify the optimal instruction-set extensions given a high level application description. Our approach is based on a scalable ILP model that integrates the data bandwidth information and the data transfer costs into the instruction-set extension identification process. We evaluate our approach using actual ASIC implementations to demonstrate that our automatically customized processors meet timing within the target silicon area. For an embedded processor with only two register read ports and one register write port, we obtain up to 4.3× speed-up with only a 35% area overhead. In addition, we explore the potential of increasing the number of register file ports, improving the performance more than 6.6×. We are extending our approach to enable instruction extensions to access memory hierarchy, and to support a wide range of applications involving speed, area and power consumption trade-offs.

Acknowledgment

Kubilay Atasu is partially supported by TUBITAK under National Ph.D. Scholarship Program. The support of Bogazici University Research Projects (grant number 05HA102D), and the support of UK EPSRC (grant number EP/C549481/1) for this work is gratefully acknowledged.

References

[1] Ilog cplex. http://www.ilog.com/products/cplex/.

[2] Mibench. http://www.eecs.umich.edu/mibench/.

[3] Nauty package. http://cs.anu.edu.au/people/bdm/nauty.

[4] Trimaran. http://www.trimaran.org.

[5] K. Atasu, G. Dündar, and C. Özturan. An integer linear programming approach for identifying instruction-set extensions. In *CODES+ISSS 2005*, Jersey City, NJ, Sept. 2005.

[6] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *40th DAC*, Anaheim, CA, June 2003.

[7] P. Biswas, S. Banerjee, N. Dutt, L. Pozzi, and P. Ienne. Isegen: Generation of high-quality instruction set extensions by iterative improvement. In *DATE 2005*, Mar. 2005.

[8] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *CASES 2002*, Grenoble, France, 2002.

[9] J. Cong, Y. Fan, G. Han, A. Jagannathan, G. Reinmann, and Z. Zhang. Instruction set extension with shadow registers for configurable processors. In *FPGA 2005*, Feb. 2005.

[10] J. Cong, Y. Fan, G. Han, and Z. Zhang. Application-specific instruction generation for configurable processor architectures. In *FPGA 2004*, Monterey, CA, Feb. 2004.

[11] R. G. Dimond, O. Mencer, and W. Luk. Combining instruction coding and scheduling to optimize energy in system-on-fpga. In *FCCM 2006*, Napa Valley, CA, Apr. 2006.

[12] D. Goodwin and D. Petkov. Automatic generation of application specific processors. In *CASES 2003*, pages 137–147, San Jose, CA, Nov. 2003.

[13] R. Jayaseelan, H. Liu, and T. Mitra. Exploiting forwarding to improve data bandwidth of instruction-set extensions. In *43rd DAC*, Anaheim, CA, July 2006.

[14] L. Pozzi and P. Ienne. Exploiting pipelining to relax register-file port constraints of instruction-set extensions. In *CASES 2005*, San Francisco, CA, Sept. 2005.

[15] P. Yu and T. Mitra. Scalable custom instructions identification for instruction-set extensible processors. In *CASES 2004*, Washington, DC, Sept. 2004.