Fast Custom Instruction Identification by Convex Subgraph Enumeration

Kubilay Atasu, Oskar Mencer, Wayne Luk
Department of Computing, Imperial College London
{atasu,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

Automatic generation of custom instruction processors from high-level application descriptions enables fast design space exploration, while offering very favorable performance and silicon area combinations. This work introduces a novel method for adapting the instruction set to match an application captured in a high-level language. A simplified model is used to find the optimal instructions via enumeration of maximal convex subgraphs of application data flow graphs (DFGs). Our experiments involving a set of multimedia and cryptography benchmarks show that an order of magnitude performance improvement can be achieved using only a limited amount of hardware resources. In most cases, our algorithm takes less than a second to execute.

1 Introduction

Combining programmability and efficiency, custom instruction processors are emerging as basic building blocks in the design of complex systems-on-chip. Typically, a base processor is extended with custom functional units that implement application-specific instructions. A dedicated link between custom functional units and the base processor provides an efficient communication interface. Re-using a preverified, pre-optimized base processor reduces design complexity and time to market. Commercial examples include Tensilica Xtensa, ARC 700, Altera Nios II, MIPS Pro Series, Xilinx MicroBlaze, and Stretch S6000.

Techniques for the automated synthesis of custom instructions from high level application descriptions have received considerable attention in the recent years. The typical approach limited the maximum number of input and output operands of custom instructions can have to the available register file ports [6, 9, 16]. Although these constraints can be prohibitive on some architectures, most existing customizable processors (such as Tensilica Xtensa) allow custom instructions to have more input and out operands than the available register file ports through custom state registers that can temporarily hold some of the operands. In fact, recent work shows that input/output constraints deteriorate the solution quality on architectures where there is no explicit limit on the number of custom instruction operands [4, 12, 13, 15]. Thus, there is a need for new algorithms that can efficiently explore custom instruction candidates within application DFGs without imposing a limit on the number of input and output operands.

In this work, we develop an efficient subgraph enumeration approach for the automated identification of custom instructions. Similar to the work of Pothineni [12] and the work of Verma [15], we enumerate only convex subgraphs of application DFGs that are maximal, imposing no constraints on the number of input and output operands for custom instructions. Our main contributions in this work are:

1. a tight upper bound on the number of maximal convex subgraphs within a given DFG (Section 3, 4);
2. a novel maximal convex subgraph enumeration algorithm for custom instruction synthesis (Section 5, 6);
3. demonstration of the scalability of our algorithm on a set of benchmarks, which can achieve an order of magnitude speed-up with respect to a single issue base processor (Section 7).

2 Related Work

Most of the early work and some of the recent work [8, 14] in automated instruction-set customization relied on heuristic clustering of related DFG nodes. Gradually, the attention shifted towards closer to optimal solutions, such as subgraph enumeration [6, 7, 9, 16] and integer linear programming (ILP) [5, 11]. All of these approaches assumed

constraints on the number of input and output operands for custom instructions. In particular, subgraph enumeration based approaches explicitly make use of the input/output constraints to prune the search space and reduce the exponential computational complexity. It was recently shown that ILP based techniques [4] scaled well with the relaxation of the input/output constraints. However, subgraph enumeration based approaches become intractable, as the computational complexity grows exponentially with the relaxation of the input/output constraints. In fact, recent work [7] showed that the worst case time complexity of enumerating subgraphs having N_{in} input and N_{out} output operands in a DFG with N nodes is $O(N^{N_{in}+N_{out}+1})$.

Pothineni et al. [12] targeted the maximal convex subgraph enumeration problem. Given a DFG, Pothineni et al. first define an incompatibility graph, where the edges represent pairwise incompatibilities between DFG nodes. Pothineni et al. define the ancestors and the descendants of an invalid node as incompatible. A node clustering step identifies groupwise incompatibilities and reduces the size of the incompatibility graph. The incompatibility graph representation allowed Pothineni et al. to formulate the maximal convex subgraph enumeration problem as a maximal independent set enumeration problem. Pothineni et al. indicate that the complexity of enumeration is $O(2^{N_C})$, where N_C represents the number of nodes in the incompatibility graph.

Verma et al. [15] used maximal clique enumeration instead of maximal independent set enumeration. However, the two problems can be directly transformed into each other (see for example, Garey and Johnson [10, p.54]). Therefore, the approach of Verma et al. [15] and the approach of Pothineni et al. [12] are essentially the same.

3 Problem Formulation

We assume that the source program is converted into an intermediate representation (IR), where every statement in the IR is a branch, or an assignment with at most two source operands and one destination operand. We represent an application basic block as a DFG $G(V_b, E_b)$ where the nodes V_b represent statements within the basic block, and the edges E_b represent flow dependencies between nodes.

The subset $V_b^f \subseteq V_b$ represent forbidden statements in G that cannot be included in custom instructions, either because of the limitations of the custom processor architecture, or because of the limitations of the custom datapath, or by the choice of the designer.

A custom instruction candidate is a *subgraph* of G induced by a set of nodes $V_s \subseteq V_b/V_b^f$. A subgraph S is *convex* if there exists no path in G from a node $u \in V_s$ to another node $w \in V_s$ which involves a node $v \notin V_s$. The convexity constraint is imposed on the subgraphs to ensure that no cyclic dependency is introduced in G and that a feasible schedule can be achieved for the instruction stream.

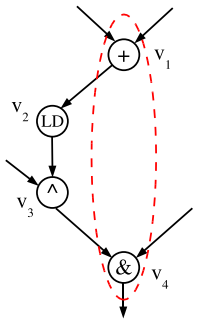

Figure 1. A subgraph that is not convex.

Figure 1 depicts an example subgraph that is not convex.

We associate with every graph node $v_i \in V_b$ a binary variable x_i that represents whether the node is included in the subgraph ($x_i = 1 \Leftrightarrow v_i \in V_s$) or not ($x_i = 0 \Leftrightarrow v_i \notin V_s$). We use x_i' to denote the complement of x_i ($x_i' = 1 - x_i$). For $v_i \in V_b^f$ we set $x_i = 0$. In this way, it is possible to encode all $2^{|V_b/V_b^f|}$ valid subgraphs in G.

In this work, we assume a generic and simple optimization model for custom instruction identification. We represent our problem using the following indices:

$$I_1: \quad \textit{indices for nodes } v_i \in V_b^f$$
$$I_2: \quad \textit{indices for nodes } v_i \in V_b/V_b^f$$

We associate with every graph node $v_i \in V_b$ a software latency $s_i \in Z^+$, which gives the time in clock cycles that it takes to execute v_i on the pipeline of the base processor. The objective of optimization is to identify the subgraph S with the maximum accumulated software latency:

$$\max \sum_{i \in I_2} (s_i x_i). \tag{1}$$

4 An Upper Bound on the Search Space Size

Definition 1. A convex subgraph S is maximal if it cannot be grown further by including additional nodes from V_b/V_s.

In this Section, we are going to show that the number of maximal convex subgraphs in G is bounded by $2^{|V_b^f|}$.

Remark 1. A subgraph S is convex if and only if there exists no node in V_b/V_s having both an ancestor and a descendant in V_s.

For each node $v_i \in V_b$ we introduce two binary variables: a_i represents whether v_i has an ancestor in S ($a_i = 1$) or not ($a_i = 0$), and d_i represents whether v_i has a descendant in S ($d_i = 1$) or not ($d_i = 0$). Based on Remark 1, the convexity property can be formulated as follows:

$$a_i \wedge d_i = 0, \qquad i \in I_1, \tag{2}$$
$$x_i' \wedge a_i \wedge d_i = 0, \qquad i \in I_2. \tag{3}$$

Theorem 1. *A maximal subgraph that satisfies Equation (2), satisfies also Equation (3).*

Proof. Assume that Equation (2) holds for subgraph S, i.e., no node in V_b^f has both an ancestor and a descendant in V_s. Assume also that S is maximal, i.e., no additional node can be included in V_s without violating Equation (2). We are going to show that S satisfies Equation (3), as well.

Suppose that a node $v_i \in V_b/(V_s \cup V_b^f)$ violates Equation (3), i.e., v_i has both an ancestor and a descendant in V_s. We are going to show that including v_i in V_s does not violate Equation (2).

In a convex solution, there exist three possible choices for each $v^f \in V_b^f$:

- v^f *has ancestors, but no descendants in* V_s. In this case, we know that v_i cannot be a descendant of v^f. If v_i was a descendant of v^f, v^f would have descendants in V_s, since v_i has descendants in V_s. Because v_i is not a descendant of v^f, including v_i in V_s does not violate Equation (2).
- v^f *has descendants, but no ancestors in* V_s. In this case, we know that v_i cannot be an ancestor of v^f. If v_i was an ancestor of v^f, v^f would have ancestors in V_s, since v_i has ancestors in V_s. Because v_i is not an ancestor of v^f, including v_i in V_s does not violate Equation (2).
- v^f *has neither ancestors nor descendants in* V_s. In this case, we know that v_i is neither an ancestor nor a descendant of v^f. Otherwise v^f would have ancestors or descendants in V_s. As a result, including v_i in V_s does not violate Equation (2).

We have shown that if there exists a $v_i \in V_b/(V_s \cup V_b^f)$ that violates Equation (3), we can safely include it in V_s without violating Equation (2). However, this contradicts with the maximality of S.

Therefore, a $v_i \in V_b/V_b^f$ that violates Equation (3) cannot exist in the maximal S satisfying Equation (2). □

Corollary 1. *The maximal subgraph S that satisfies Equation (2) is a maximal convex subgraph.*

We note that there exists only three valid a_j, d_j choices for a $v_j \in V_b^f$: (1) $a_j = 1, d_j = 0$; (2) $a_j = 0, d_j = 1$; (3) $a_j = 0, d_j = 0$. The third choice can be disregarded, since the maximal S would include as many nodes as possible and only the first and the second choices can improve the size of S. We note that the case where a node $v_j \in V_b^f$ has neither an ancestor nor a descendant in S can still occur (a) if we choose $a_j = 1, d_j = 0$ and none of the ancestors of v_j can be included in the solution either by the properties of the G or because all of the ancestors of v_j are prohibited by the choices made for the remaining forbidden nodes; (b) if we choose $a_j = 0, d_j = 1$ and none of the descendants of v_j can be included in the solution either by the properties of G or because all of the descendants of v_j are prohibited by the choices made for the remaining forbidden nodes.

Given a valid a_j, d_j combination for $j \in I_1$, the associated maximal convex subgraph can be found as follows:

- A node $v_i \in V_b/V_b^f$ cannot be included in S if it has an ancestor $v_j \in V_b^f$ for which an ancestor exists in S ($a_j = 1$). Otherwise, v_j would have both an ancestor and a descendant in S.
- A node $v_i \in V_b/V_b^f$ cannot be included in S if it does not have a descendant $v_j \in V_b^f$ for which a descendant exists in S ($d_j = 1$).
- All the remaining nodes in V_b/V_b^f can be safely included in S without violating Equation (2).

Corollary 2. *Every valid a_j, d_j combination for $v_j \in V_b^f$ defines a maximal convex subgraph.*

We introduce the following notation to represent the set of ancestors, and the set of descendants of the nodes in V_b/V_b^f that are in V_b^f:

$$Anc(i \in I_2) = \{j \in I_1 \mid \textit{There exists a path from } v_j \textit{ to } v_i\}$$
$$Desc(i \in I_2) = \{j \in I_1 \mid \textit{There exists a path from } v_i \textit{ to } v_j\}$$

Once a_j and d_j values are fixed for the nodes $v_j \in V_b^f$, whether a node $v_i \in V_b/V_b^f$ is part of the maximal subgraph S can be found as follows:

$$x_i = \begin{cases} 1 & \textit{if } Anc(i) = Desc(i) = \emptyset \\ \left(\bigwedge_{j \in Anc(i)} a'_j\right) & \textit{if } Desc(i) = \emptyset \\ \left(\bigwedge_{j \in Desc(i)} d'_j\right) & \textit{if } Anc(i) = \emptyset \\ \left(\bigwedge_{j \in Anc(i)} a'_j\right) \wedge \left(\bigwedge_{j \in Desc(i)} d'_j\right) & \textit{Otherwise} \end{cases} \quad (4)$$

For each $v_j \in V_b^f$, it is sufficient to evaluate two possible choices (i.e., $a_j = 1, d_j = 0$ or $a_j = 0, d_j = 1$). Each choice is associated with a single maximal solution S, which can be inferred directly using Equation (4).

Corollary 3. *There exists an upper bound of $2^{|V_b^f|}$ on the number of maximal convex subgraphs.*

Figure 2 depicts an example DFG. The nodes v_4 and v_5 are forbidden nodes. Because there exists only two forbidden nodes, there exists only $2^2 = 4$ possible choices we need to consider: (1) ancestors of v_4 and ancestors of v_5 can take part in the solution ($a_4 = 1, d_4 = 0$ and $a_5 = 1, d_5 = 0$); (2) ancestors of v_4 and descendants of v_5 can take part in the solution ($a_4 = 1, d_4 = 0$ and $a_5 = 0, d_5 = 1$); (3) descendants of v_4 and ancestors of v_5 can take part in the solution ($a_4 = 0, d_4 = 1$ and $a_5 = 1, d_5 = 0$); (4) descendants of v_4 and descendants of v_5 can take part in the solution ($a_4 = 0, d_4 = 1$ and

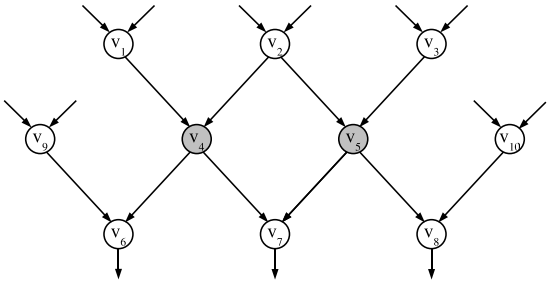

Figure 2. v_4 and v_5 are forbidden nodes.

Table 1. Solutions for the DFG of Figure 2

(a_4, d_4)	(a_5, d_5)	Solution
(1,0)	(1,0)	$\{v_1, v_2, v_3, v_9, v_{10}\}$
(1,0)	(0,1)	$\{v_1, v_8, v_9, v_{10}\}$
(0,1)	(1,0)	$\{v_3, v_6, v_9, v_{10}\}$
(0,1)	(0,1)	$\{v_6, v_7, v_8, v_9, v_{10}\}$

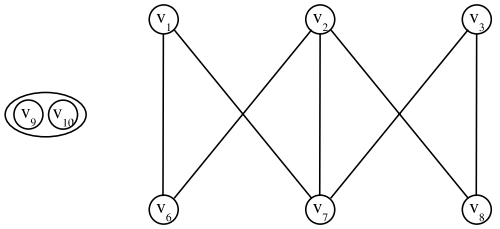

Figure 3. Pothineni's Incompatibility graph.

$a_5 = 0, d_5 = 1$). Table 1 shows the solutions associated with each of these these four choices.

Figure 3 shows the incompatibility graph generated by Pothineni's algorithm [12] for the DFG of Figure 2. The incompatibility graph contains seven nodes. According to Pothineni's work, the worst case complexity of maximal convex subgraph enumeration for this graph is 2^7. On the other hand, we have shown that it is possible to enumerate all maximal convex subgraphs in the DFG in only 2^2 steps.

5 A Novel Enumeration Algorithm

We have shown in Section 4 that there exists an upper bound of $2^{|V_b^f|}$ on the number of maximal convex subgraphs given a graph with $\left|V_b^f\right|$ forbidden nodes. Therefore, the time complexity of the maximal convex subgraph enumeration algorithms should not have an exponential factor higher than $2^{|V_b^f|}$. In this section, we describe a novel enumeration algorithm that further reduces the execution time.

Similar to the work of Pothineni et al. [12] and the work of Verma et al. [15], we firs apply a node clustering step that reduces the size of the DFG, and the number of forbidden nodes. In particular, if $v_i, v_j \in V_b/V_b^f$ and

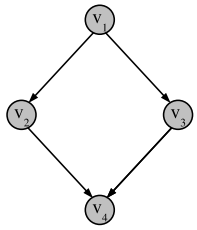

Figure 4. Connectivity of forbidden nodes.

$Anc(i) = Anc(j)$ and $Desc(i) = Desc(j)$, we can cluster the nodes v_i and v_j together, without affecting the result of enumeration, since Equation (4) guarantees that x_i and x_j will always be the same. Similarly, if two forbidden nodes $v_i, v_j \in V_b^f$ have the same set of ancestors and descendants that are in V_b/V_b^f, we can cluster the two as a single forbidden node without affecting the optimality.

In order to demonstrate further ways of reducing the complexity, we construct a new graph G' from G, where the nodes of G' represent the forbidden nodes of G. We introduce a directed edge between two forbidden nodes $v_i, v_j \in V_b^f$ in G' only if there is a path from v_i to v_j in the original graph G. Figure 4 illustrates a simple graph G'.

Assume that we set $d_1 = 0$ in Figure 4. This disables inclusion of any node that is a descendant of v_1 in the solution. Therefore, we can set $d_i = 0$ for all forbidden nodes that are descendants of v_1 (i.e., $d_2 = d_3 = d_4 = 0$). Thus, in practice, the number of possible d_i combinations for $v_i \in V_b^f$ (where $a_i = 1 - d_i$ always) is much smaller than $2^{|V_b^f|}$. We exploit this property in order to design a simple and efficient algorithm for maximal convex subgraph enumeration.

Figure 5 shows the pseudo-code of our algorithm. We first apply a node clustering step on G. Next we derive G' from the clustered graph. After that, we order the nodes of G' topologically, such that if there is a path from v_i to v_j in G', v_i is associated with a lower index value than v_j. Our algorithm generates combinations of d_i values. We note that each combination in turn, is associated with a maximal convex subgraph that can be derived using Equation (4).

6 Overall Approach

Once enumeration of all maximal convex subgraphs within a basic block is complete, we first pick the maximum convex subgraph as the most promising custom instruction candidate. After that we prune the maximal convex subgraphs (1) that overlap with the chosen subgraph, (2) that are cyclicly dependent with the chosen subgraph. Again we pick the largest one among the remaining maximal convex subgraphs, and we continue the same process until no more profitable maximal convex subgraphs can be found.

We apply the same procedure on all application basic blocks, and generate a unified set of subgraphs. After that, we group structurally equivalent subgraphs that can be im-

```
1: ALGORITHM: search(index, choice, graph)
2: curent_combination[index] = choice;
3: if index == num_nodes_in_graph-1 then
4:     store current_combination;
5:     return;
6: end if
7: if choice == 0 then
8:     ensure that all descendants of index in graph are disabled;
9: end if
10: index=index+1;
11: search(index, 0, graph);
12: if index is not disabled then
13:     search(index, 1, graph);
14: end if
15: if choice == 0 then
16:     ensure that the disabled descendants are again enabled;
17: end if
18: ALGORITHM: enumerate()
19: Apply node clustering on G;
20: Generate G' from the clustered graph;
21: Topologically sort the nodes of G';
22: search(0, 0, G');
23: search(0, 1, G');
```

Figure 5. Enumeration Algorithm.

plemented using the same hardware. We estimate the software execution latency $Z(S)$ of a subgraph S by scheduling the subgraph in software under base processor resource constraints. We obtain the hardware execution latency $H(S)$ of S through hardware synthesis using Synopsys Design Compiler. We estimate the communication latency $C(S)$ of S by calculating the number of cycles required to transfer its input and output operands under register file port constraints, in a similar way as described in [4]. Given the frequency of execution $F(S)$ of the subgraph S, we estimate the amount of reduction in the schedule length of the application by moving S from software to hardware as follows:

$$F(S) * (Z(S) - H(S) - C(S)). \quad (5)$$

Finally, we obtain the area costs of subgraphs using Synopsys synthesis, and we choose the most promising subgraphs under area constraints using a Knapsack model [8].

7 Experiments and Results

We integrated our algorithms into Trimaran compiler [3]. We marked the memory access and branch instructions as forbidden instructions. We applied our algorithms on eight benchmarks from multimedia and cryptography domains.

We carried out our experiments on an Intel Pentium 4 3.2-GHz workstation with 1-GB main memory, running Linux. We developed our algorithms in C/C++ and compiled with gcc-3.4.3 using -O2 optimization flag.

In Table 2, we compare the run-time of our enumeration algorithm with an ILP based approach [4], using both a commercial solver (CS) [1] and a public domain solver (PDS) [2]. The second and the third columns of Table 2 show the number of nodes and the number of forbidden nodes respectively, in the largest basic block of each benchmark. The run-time of our algorithm for identifying the maximum convex subgraph within the largest basic block is given in the fourth column. The remaining two columns show the respective ILP results using CS and PDS.

Table 2. Run-time comparison (in seconds).

Bench.	$\|V_b\|$	$\|V_b^f\|$	Our work	CS	PDS
AES	357	43	24s	0.18s	2.1s
DES	822	176	0.15s	11s	307s
SHA	1155	90	0.02s	0.20s	26s
mpeg2enc	520	208	0.008s	2.28s	643s
idea	96	8	0.0008s	0.03s	0.33s
djpeg	92	19	0.0003s	0.025s	0.14s
rawcaudio	54	8	0.0002s	0.03s	0.24s
rawdaudio	45	8	0.0002s	0.02s	0.14s

We observe that except for AES encryption, our approach is at least an order of magnitude faster than the ILP based approach. The advantage of using our technique is more evident when PDS is used instead of CS. In fact, CS [1] can automatically recognize constraint patterns that correspond to maximum independent set and maximum clique problems and includes efficient solvers targeted specifically for these problems. We observe that in the case of AES, ILP based approach is faster than ours even if PDS is used. This is also possible. We note that, ILP solvers can reduce the search space not only based on the constraints, but also based on the definition of the objective function.

The only run-time result provided by Verma et al. [15] is for AES, which is reported to be around 30 seconds. Pothineni et al. [12], on the other hand, report a run-time of two seconds for DES. Although we have not implemented these two techniques, our work appears to be faster for AES and DES. We note that other existing subgraph enumeration algorithms [6, 7, 16] do not scale well with the relaxation of input/output constraints, and fail to identify the maximum convex subgraphs in several hours given benchmarks with very large basic blocks, such as AES and DES.

Using Trimaran framework, we defined a single issue base processor that implements an instruction-set similar to MIPS IV instruction-set. We synthesized the custom instructions to UMC's 130nm standard cell library using Synopsys Design Compiler. We note that our custom instructions are pipelined in order not to increase the cycle time of the base processor, which we estimated to be around the critical path delay of a 32-bit carry propagate adder.

Figure 6 shows the speed-up results we obtain using custom instructions with respect to the base processor on our benchmarks. Assuming a base processor register file with 32 32-bit registers, we evaluate four register file read and write port combinations: (2,1), (2,2), (4,2), and (4,4). We

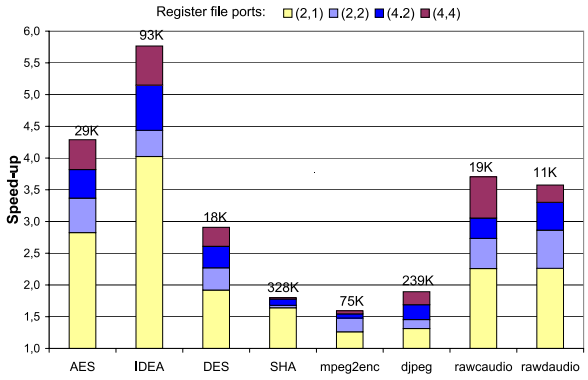

Figure 6. Additional ports improve speed-up.

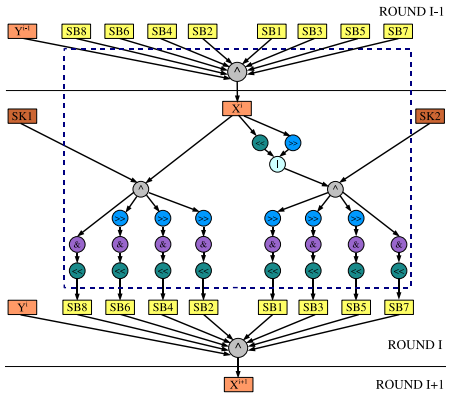

Figure 7. Custom instruction for DES.

observe that increasing the number of read and write ports supplied by the register file, decreases the communication overhead of the custom instructions, which improves the speed-up. Finally, above each column, we show the cell area for the associated custom datapath in terms of μm^2. We note that, we have observed marginal difference in our speed-up results compared with the ILP based approach [4].

Figure 7 shows the most promising custom instruction our algorithms identify from the DES C code. The software implementation fully unrolls DES round transformations within a single basic block, which consists of 822 base processor instructions. The custom instruction implements the combinational logic between the memory access layers of two consecutive DES rounds. We note that X and Y represent the DES encryption state. Eight of the inputs of the custom instruction are static look-up table entries (SBs), and two of the inputs (SK1,SK2) contain the DES round key. Accordingly, eight of the outputs contain the addresses of the look-up table entries that should be fetched for the next round. We observe that the size of the look-up tables is rather small (256 bytes only). We could avoid all the related main memory accesses and address calculations by embedding the eight look-up tables in local memories. Similarly, the DES scheduled key is only 32 bytes wide, and can be embedded in local memories, again eliminating a number of main memory accesses. Once these optimizations are done, the size of the core basic block of DES drops from 822 instructions into only 22 instructions, which incorporate only three different types of custom instructions, each one having only a single cycle execution latency. The result is about 30 times improvement in the performance of DES.

8 Summary

This paper provides novel theoretical and practical results for improving efficiency of automatically-generated custom instruction processors and their design. Current and future work includes extending our approach to cover additional application domains, and to support implementations targeting field-programmable devices.

References

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

[2] lpsolve. http://sourceforge.net/projects/lpsolve.

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

[4] K. Atasu et al. Optimizing instruction-set extensible processors under data bandwidth constraints. In *DATE*, pages 588–593, Nice, France, Apr. 2007.

[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. Bonzini and L. Pozzi. Polynomial-time subgraph enumeration for automated instruction set extension. In *DATE*, pages 1331–1336, Nice, France, Apr. 2007.

[8] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *MICRO*, San Diego, CA, Dec. 2003.

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

[10] R. M. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., New York, 1979.

[11] R. Leupers et al. A design flow for configurable embedded processors based on optimized instruction set extension synthesis. In *DATE 2006*, Munich, Germany, Mar. 2006.

[12] N. Pothineni, A. Kumar, and K. Paul. Application specific datapath with distributed I/O functional units. In *VLSI Design*, pages 551–558, Hyderabad, India, Jan. 2007.

[13] 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.

[14] F. Sun, S. Ravi, A. Raghunathan, and N. Jha. A scalable application-specific processor synthesis methodology. In *ICCAD*, pages 283–290, San Jose, CA, Nov. 2003.

[15] A. K. Verma, P. Brisk, and P. Ienne. Rethinking custom ise identification: A new processor-agnostic method. In *CASES*, pages 125–134, Salzburg, Austria, Sept. 2007.

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