FISH: Fast Instruction SynthHesis for Custom Processors

Kubilay Atasu, *Member, IEEE*, Wayne Luk, *Fellow, IEEE*, Oskar Mencer, *Member, IEEE*, Can Özturan, and Günhan Dündar

***Abstract*—This paper presents Fast Instruction SynthHesis (FISH), a system that supports automatic generation of custom instruction processors from high-level application descriptions to enable fast design space exploration. FISH is based on novel methods for automatically adapting the instruction set to match an application in a high-level language such as C or C++. FISH identifies custom instruction candidates using two approaches: 1) by enumerating maximal convex subgraphs of application data flow graphs and 2) by integer linear programming (ILP). The experiments, involving ten multimedia and cryptography benchmarks, show that our contributed algorithms are the fastest among the state-of-the-art techniques. In most cases, enumeration takes only milliseconds to execute. The longest enumeration run-time observed is less than six seconds. ILP is usually slower than enumeration, but provides us with a complementary solution technique. Both enumeration and ILP allow the use of multiple different merit functions in the evaluation of data-flow subgraphs. The experiments demonstrate that, using only modest additional hardware resources, up to 30-fold performance improvement can be obtained with respect to a single-issue base processor.**

***Index Terms*—Custom processors, design automation, design optimization, graph theory, mathematical programming, subgraph enumeration, system-on-chip (SoC).**

I. Introduction

THE complexity of system-on-chip (SoC) devices is increasing continuously. The competitive end-market requires the design of more versatile systems with more hardware and software resources in shorter time. Today, a major problem in SoC design is the limited designer productivity in comparison with the growth in hardware complexity [1]. This phenomenon necessitates new approaches in the design of complex SoCs. First, the new generation of SoCs should be sufficiently programmable in order to amortize chip design costs. Second, there is a need for optimizations of the performance, area, and power efficiency of SoC designs for specific applications.

Manuscript received March 02, 2010; revised August 25, 2010, October 25, 2010; accepted October 25, 2010. Date of publication December 03, 2010; date of current version December 14, 2011. This work was supported in part by UK EPSRC and IBM Research.

K. Atasu is with IBM Research—Zurich, CH-8803 Ruschlikon, Switzerland (e-mail: kat@zurich.ibm.com).

O. Mencer and W. Luk are with the Department of Computing, Imperial College London, SW7 2BZ London, U.K. (e-mail: o.mencer@imperial.ac.uk; w.luk@imperial.ac.uk).

C. Özturan is with the Department of Computer Engineering, Boğaziçi University, 80815 Istanbul, Turkey (e-mail: ozturaca@boun.edu.tr).

G. Dündar is with the Department of Electrical and Electronics Engineering, Boğaziçi University, 80815 Istanbul, Turkey (e-mail: dundar@boun.edu.tr).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2010.2090543

Combining programmability and efficiency, custom instruction processors are emerging as key building blocks in the design of complex SoCs. The typical approach extends a base processor with custom functional units that implement application-specific instructions [2]–[4]. A dedicated link between the custom functional units and the base processor provides an efficient communication interface. Reusing a preverified, preoptimized base processor reduces design complexity and time to market. Commercial examples include Tensilica Xtensa, ARC 700, Altera Nios II, MIPS Pro Series, Xilinx MicroBlaze, Stretch S6000, and IBM's PowerPC® 405.

Modern custom instruction processors comprise parallel, deeply pipelined custom functional units including state registers, local memories, and wide data buses to local and global memories and provide support for flexible instruction encoding [5]. These features enable custom processors to reach a computational performance comparable to the performance of custom register transfer-level blocks. On the other hand, such advances necessitate improved design automation methodologies that can take advantage of the new capabilities.

Techniques for the automated synthesis of custom instructions from high-level application descriptions have received much attention in recent years. Typically, compiler infrastructures are used to extract the source-level data flow graphs (DFGs), and data flow subgraphs are evaluated as custom instruction candidates. This is followed by the synthesis of the customized hardware and software components (see Fig. 1). The traditional subgraph exploration approach restricts the maximum number of input and output operands that custom instructions can have to the available register file ports [6]–[13]. Although these constraints can be prohibitive on some customizable processors, most existing architectures—such as Tensilica Xtensa—allow custom instructions to have more input and output operands than the available register file ports, typically through custom state registers that can temporarily hold some of the operands. Recent work has shown that input/output constraints deteriorate solution quality for architectures in which there is no explicit limit on the number of custom instruction operands [14]–[18]. Thus, there is a need for new algorithms that can efficiently explore custom instruction candidates without imposing a limit on the number of input/output operands.

This paper presents effective techniques for automating the identification of custom instructions starting with application descriptions in C/C++. In this work, only convex subgraphs of application DFGs that are maximal are considered, and, while generating custom instruction candidates, no constraints are imposed on the number of input and output operands. Unlike

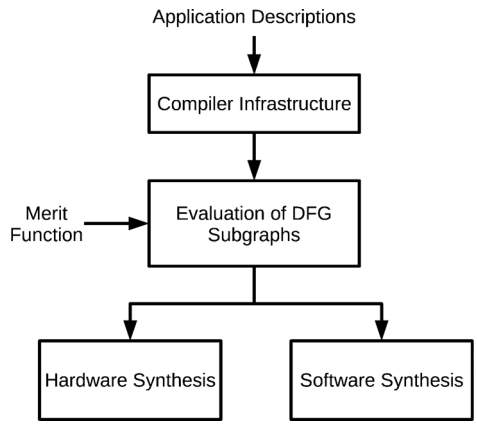

Fig. 1. Compiler infrastructures transform high-level application descriptions into source-level control and DFGs. Data flow subgraphs are evaluated as custom instruction candidates. A high-level merit function ranks the subgraphs based on estimations of hardware and software costs.

prior work, the techniques presented in this work do not rely on enumeration of maximal independent sets or maximal cliques. This work extends the work in [19], which describes a maximal convex subgraph enumeration algorithm with a proven upper bound on the size of the search space. The enumeration algorithm of [19] is explained in more depth, and additional search space reduction techniques are demonstrated. The enumeration approach makes it possible to integrate any merit function for ranking the subgraphs. In addition, a new ILP formulation provides a second way of finding the maximal convex subgraphs that optimize a linearly expressed merit function. The main contributions of this work are:

1) an upper bound on the number of maximal convex subgraphs within a given DFG (Section IV);
2) an integer linear programming formulation for evaluating maximal convex subgraphs (Section V);
3) a maximal convex subgraph enumeration algorithm for custom instruction synthesis that includes novel clustering and search space reduction techniques (Section VI);
4) evaluation of multiple different merit functions in a design space exploration framework (Sections VII, VIII);
5) illustration of the scalability of our algorithms on a set of benchmarks (Section IX), where we also demonstrate an order of magnitude performance improvement with respect to a single-issue base processor (Sections X and XI).

II. Related Work

Heuristic clustering algorithms have long been used in automated custom instruction synthesis [20]–[24]. However, the attention gradually shifts towards techniques for producing optimal solutions, such as subgraph enumeration [6]–[12] and ILP [13], [15], [18], [25], [26]. Given a DFG with N nodes, the complexity of enumerating subgraphs with N_{in} input and N_{out} output operands is shown to be $O(N^{N_{\text{in}}+N_{\text{out}}+1})$ [27], which grows exponentially with the number of inputs and outputs. A common property of the aforementioned enumeration techniques [6]–[12] is that explicit constraints are imposed on the number of input and output operands of the subgraphs, which are used to prune the search space and reduce the exponential computational complexity. A downside is that, as the constraints on the number of input and output operands are relaxed, enumeration becomes computationally expensive. On the other hand, the ILP model of [13] efficiently handles DFGs with more than a thousand nodes under any input and output constraint, including the option of removing these constraints completely [15], [18]. The reason is that the number of integer variables and the number of linear constraints used in ILP grow linearly with the number of nodes and the number of edges in the DFG.

In [14], a pipelining algorithm that serializes the register file accesses when the number of inputs and the number of outputs of the custom instructions exceed the available register file ports, is described. This technique relies on [6], [9] for generating custom instructions, which have limited scalability. In addition, pipelining is applied on the source-level DFGs, where it is nearly impossible to estimate the critical path accurately and, therefore, to find an optimal pipeline. However, this work empirically shows that achievable speed-up grows monotonically with the relaxation of the constraints on the number of input and output operands. Similar conclusions are made also by [15]–[18], which employ more scalable algorithms for generating custom instructions.

Pothineni *et al.* [16] were the first ones to target the maximal convex subgraph enumeration problem. Given a DFG, Pothineni *et al.* first define an incompatibility graph, where the edges represent pair-wise incompatibilities between the nodes. A node clustering step identifies group-wise incompatibilities and reduces the size of the incompatibility graph. The incompatibility graph representation transforms the maximal convex subgraph enumeration problem into 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. Although Pothineni *et al.* apply enumeration independently on the connected components of a DFG, the disconnected subgraphs within each connected component are enumerated too.

In [17], the maximal independent set enumeration problem of [16] is reformulated as a maximal clique enumeration problem, and enumeration is applied at once to the whole graph. Additionally, [17] describes a heuristic algorithm for pipelining the enumerated subgraphs and serializing register file accesses at the source level. [17] includes a formal proof, which shows that under certain assumptions speed-up potential of a subgraph grows monotonically with its size.

In [26], the ILP model of [13] is extended to include a resource-constrained scheduling model for serializing the register file accesses. However, the resulting model scales quadratically with the size of the DFGs versus the linear scaling of [13], [15], and [18]. As a result, the work of [26] fails to identify optimal solutions on large DFGs, although the ILP models of [13], [15], and [18] complete optimally in all practical cases.

In [28], Li *et al.* describe an algorithm that constructs a search tree similar to the one described in [19]. The enhancements proposed in [28] include: 1) an adaptive branch ordering mechanism; 2) migration of some of the computation from the leaves of the search tree to the branches, where simpler bitwise operations can be used; 3) an efficient mechanism to discard subgraphs that are not maximal or that are repeated.

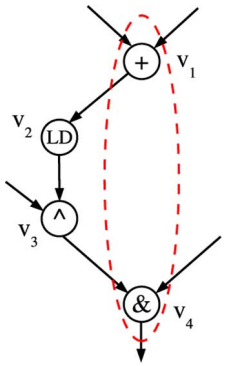

Fig. 2. Subgraph that is not convex. v_2 is a forbidden node.

Other related and complementary work includes: 1) use of local memories for improving the performance of custom instructions [25], [29]; 2) datapath merging [30] and combinational equivalence checking [31] techniques to improve resource sharing; 3) synthesis of custom instructions targeting field-programmable systems [32], [33], including support for run-time reconfiguration [34], [35]; and 4) transparent integration of custom instructions with a general-purpose processsor [36].

III. 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 operation. An application basic block is represented as a DFG $G(V, E)$, where the nodes V are statements within the basic block, and the edges E represent flow dependencies between nodes. The subset $V_f \subseteq V$ represents 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 due to the choice of the processor designer. Examples may include memory access, branch, division, and floating-point instructions.

Definition 1: A custom instruction candidate is a *subgraph* of G, where the nodes V_s of the subgraph are in $V - V_f$, and the edges of the subgraph are induced by the nodes V_s.

Definition 2: 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 $v \notin V_s$.

Corollary 1: A subgraph S is convex if and only if there exists no node in $V - V_s$ having both an ancestor and a descendant in 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. Fig. 2 depicts an example subgraph that is not convex.

Definition 3: A convex subgraph S is maximal if it cannot be grown further by including additional nodes from $V - V_s$.

Every graph node $v_i \in V$ is associated with a binary variable x_i, which indicates whether the node is included in the subgraph ($x_i = 1 \Leftrightarrow v_i \in V_s$) or not ($x_i = 0 \Leftrightarrow v_i \notin V_s$). The complement of x_i is denoted by x_i' ($x_i' = 1 - x_i$). For $v_i \in V_f$, x_i is simply set to be zero. In this way, up to $2^{|V - V_f|}$ subgraphs not containing forbidden nodes can be encoded. The following indexes are used in the rest of the text:

$$I: \quad \text{indices for nodes } v_i \in V - V_f$$
$$J: \quad \text{indices for nodes } v_j \in V_f.$$

The following sections use the introduced notation. Section IV shows that the number of maximal convex subgraphs is bounded by $2^{|V_f|}$ for a given DFG. Section V provides an ILP model, which evaluates all maximal convex subgraphs. Section VI describes an efficient algorithm for enumerating maximal convex subgraphs. A user defined merit function $M(S)$ ranks the subgraphs as part of enumeration or ILP. Enumeration can integrate any software function to compute $M(S)$. ILP accepts linearly expressed merit functions only.

IV. Upper Bound on the Search Space Size

Here, we prove that the number of maximal convex subgraphs in a given DFG $G(V, E)$ is at most $2^{|V_f|}$.

For each node $v \in V$ we introduce two binary variables: a indicates whether v has an ancestor in S ($a = 1$) or not ($a = 0$), and d indicates whether v has a descendant in S ($d = 1$) or not ($d = 0$). Based on Corollary 1, the convexity property can be formulated as follows:

$$x_i' \wedge a_i \wedge d_i = 0, \qquad i \in I \tag{1}$$
$$a_j \wedge d_j = 0, \qquad j \in J. \tag{2}$$

Theorem 1: A maximal subgraph that satisfies (2), satisfies also (1).

Proof: Assume that (2) holds for subgraph S, i.e., no node in V_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 (2). We are going to show that S also satisfies (1).

Suppose that a node $v_i \in V - (V_s \cup V_f)$ violates (1), i.e., v_i has both an ancestor and a descendant in V_s. First we prove that including v_i in V_s does not violate (2). We note that, in a convex solution, there exist three possible choices for each $v_j \in V_f$, which are given here.

1) v_j *has ancestors, but no descendants in* V_s. In this case, v_i cannot be a descendant of v_j. If v_i were a descendant of v_j, v_j would have descendants in V_s, since v_i has descendants in V_s. Because v_i is not a descendant of v_j, including v_i in V_s does not violate (2).
2) v_j *has descendants, but no ancestors in* V_s. In this case, v_i cannot be an ancestor of v_j. If v_i were an ancestor of v_j, v_j would have ancestors in V_s, since v_i has ancestors in V_s. Because v_i is not an ancestor of v_j, including v_i in V_s does not violate (2).
3) v_j *has neither ancestors nor descendants in* V_s. In this case, v_i is neither an ancestor nor a descendant of v_j. Otherwise v_j would have ancestors or descendants in V_s. Thus, including v_i in V_s does not violate (2).

We have shown that if there exists a $v_i \in V - (V_s \cup V_f)$ that violates (1), we can safely include it in V_s without violating (2). However, this contradicts the maximality of S. Thus, a $v_i \in V - V_f$ that violates (1) cannot exist for the maximal S satisfying (2). ■

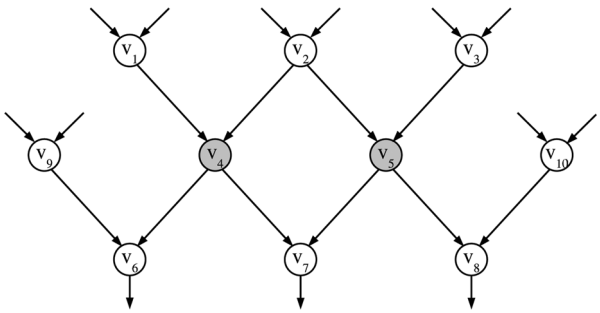

Fig. 3. v_4 and v_5 are forbidden nodes.

Based on Theorem 1, there exist only three valid a_j, d_j choices for a $v_j \in V_f$: 1) $a_j = 1, d_j = 0$; 2) $a_j = 0, d_j = 1$; and 3) $a_j = 0, d_j = 0$. Every maximal convex subgraph is associated with at least one valid a_j, d_j combination for $j \in J$.

Given a valid a_j, d_j combination for $j \in J$, an associated convex subgraph S that cannot be grown further by including additional nodes from $V - V_s$ can be found as follows.

- A node $v_i \in V - V_f$ cannot be included in S if it has an ancestor $v_j \in V_f$ for which $d_j = 0$.
- A node $v_i \in V - V_f$ cannot be included in S if it has a descendant $v_j \in V_f$ for which $a_j = 0$.
- All of the remaining nodes in $V - V_f$ can be safely included in S without violating (2).

To enumerate all maximal convex subgraphs, it is sufficient to evaluate two possible choices for each $j \in J$ (i.e., $a_j = 1, d_j = 0$ or $a_j = 0, d_j = 1$). The third choice, where $a_j = 0, d_j = 0$, can be disregarded because it does not contribute to finding a convex subgraph of maximal size. Each a_j, d_j combination for $j \in J$ is associated with a single subgraph S that can be computed using the procedure described above.

Theorem 2: There exists an upper bound of $2^{|V_f|}$ on the number of maximal convex subgraphs.

Proof: Mathematical induction based on the number of forbidden nodes in G. ∎

Fig. 3 depicts an example DFG where nodes v_4 and v_5 are the forbidden nodes. Because there are two forbidden nodes in the graph, there exist only $2^2 = 4$ possible combinations we need to evaluate: 1) ancestors of v_4 and ancestors of v_5 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 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 take part in the solution ($a_4 = 0, d_4 = 1$ and $a_5 = 1, d_5 = 0$); and 4) descendants of v_4 and descendants of v_5 take part in the solution ($a_4 = 0, d_4 = 1$ and $a_5 = 0, d_5 = 1$). Table I shows the solutions associated with each of these four choices. Note that nodes v_9 and v_{10} are included in the solutions associated with all possible combinations because they have neither ancestors nor descendants among the forbidden nodes.

Fig. 4 shows the incompatibility graph generated by Pothineni's algorithm [16] for the DFG of Fig. 3. 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.

TABLE I
SOLUTIONS FOR THE DFG OF FIG. 3

(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)	$\{v_6, v_7, v_8, v_9, v_{10}\}$

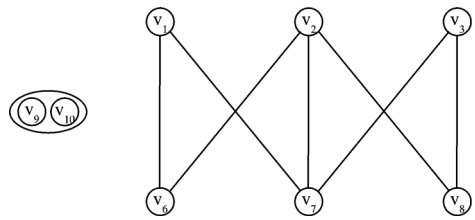

Fig. 4. Pothineni's incompatibility graph. The ancestors and the descendants of a forbidden node are defined as incompatible.

V. AN INTEGER LINEAR PROGRAMMING MODEL

This section describes an ILP model that addresses the optimization problem described in Section III. The objective of ILP is to maximize a linearly expressed merit function $M(S)$.

The following notation introduces the set of ancestors, and the set of descendants of the nodes in $V - V_f$ that are in V_f:

$$\begin{aligned}&\mathrm{Anc}(i \in I)\\ &\quad = \{j \in J \mid \text{There exists a path from } v_j \text{ to } v_i\}\\ &\mathrm{Desc}(i \in I)\\ &\quad = \{j \in J \mid \text{There exists a path from } v_i \text{ to } v_j\}\end{aligned}$$

Once a_j and d_j values are fixed for $j \in J$, the following formula computes whether a node $v_i \in V - V_f$ is part of the associated convex subgraph S of maximal size

$$x_i = \begin{cases} 1, & \text{if } \mathrm{Anc}(i) = \emptyset \wedge \mathrm{Desc}(i) = \emptyset \\ \left(\bigwedge_{j \in \mathrm{Anc}(i)} d_j\right), & \text{if } \mathrm{Anc}(i) \neq \emptyset \wedge \mathrm{Desc}(i) = \emptyset \\ \left(\bigwedge_{j \in \mathrm{Desc}(i)} a_j\right), & \text{if } \mathrm{Anc}(i) = \emptyset \wedge \mathrm{Desc}(i) \neq \emptyset \\ \left(\bigwedge_{j \in \mathrm{Anc}(i)} d_j\right) \wedge \left(\bigwedge_{j \in \mathrm{Desc}(i)} a_j\right), & \text{otherwise.} \end{cases} \quad (3)$$

According to (3), a node $v_i \in V - V_f$ can be included in the solution if it has no ancestors in V_f for which $d_j = 0$ and no descendants in V_f for which $a_j = 0$. Four different conditions are explicitly formulated: 1) v_i has no ancestors and no descendants in V_f; 2) v_i has ancestors but no descendants in V_f; 3) v_i has descendants but no ancestors in V_f; and 4) v_i has both ancestors and descendants in V_f.

Equation (3) generates convex subgraphs only. For each a_i, d_i combination, ILP computes the x_i values for all $i \in I$, which explicitly define a convex subgraph. Out of all such subgraphs, ILP picks the one that maximizes $M(S)$. An alternative model for the same problem can be found in [37, ch. 5].

VI. Novel Enumeration Algorithm

We have shown in Section IV that, given a graph with $|V_f|$ forbidden nodes, there exists an upper bound of $2^{|V_f|}$ on the number of maximal convex subgraphs. Therefore, the time complexity of the maximal convex subgraph enumeration algorithms should not have an exponential factor higher than $2^{|V_f|}$. Here, we describe a novel enumeration algorithm that significantly improves the run-time efficiency using additional search space reduction techniques. Our improvements over an exhaustive search can be categorized into two groups: 1) obtaining a more compact graph representation through preprocessing and clustering and 2) building a search tree and applying constraint propagation to prune the search space.

A. Graph Compaction

1) Basic Preprocessing: The forbidden nodes $v_i \in V_f$ not having any ancestors in $V - V_f$ and the forbidden nodes $v_i \in V_f$ not having any descendants in $V - V_f$ can be dropped from consideration because such nodes have no effect on the computation of the maximal convex subgraphs, as shown in (3). This basic optimization often eliminates a considerable number of forbidden nodes and significantly reduces the size of the search space.

2) Simple Clustering: Following the basic preprocessing step, similar to the approaches of Pothineni *et al.* [16] and Verma *et al.* [17], we apply a node-clustering step that reduces the size of the DFGs. In particular, if $v_i, v_j \in V - V_f$ and $\mathrm{Anc}(i) = \mathrm{Anc}(j)$ and $\mathrm{Desc}(i) = \mathrm{Desc}(j)$, the nodes v_i and v_j can be clustered without affecting the result of enumeration because (3) guarantees that x_i and x_j will always be equal. Similarly, if two forbidden nodes $v_i, v_j \in V_f$ have the same set of ancestors and descendants that are in $V - V_f$, (3) guarantees that the two forbidden nodes can be clustered into a single node without affecting the result of enumeration.

3) Enhanced Clustering: The simple clustering approach usually does not find any forbidden nodes that can be clustered. However, reducing the number of forbidden nodes is of utmost value as the complexity of enumeration is exponential in the number of forbidden nodes, as stated by Theorem 2. Assume that two forbidden nodes $v_i, v_j \in V_f$ have the same set of descendants that are in $V - V_f$. Setting $d_i = 0$ disables inclusion of those nodes that are descendants of v_i in the solution. However, these nodes are the descendants of v_j, too. We can set $d_j = 0$ whenever $d_i = 0$, and *vice versa* even if the ancestors of v_i and v_j in $V - V_f$ differ. As a result, we need to evaluate only two choices in such a case: $d_i = 0, d_j = 0$ and $d_i = 1, d_j = 1$. Thus, we can simply cluster the forbidden nodes having the same set of descendants that are in $V - V_f$. Similarly, it is also possible to cluster two forbidden nodes if they have the same set of ancestors that are in $V - V_f$.

In this work, we first apply the basic preprocessing step. Second, we cluster forbidden nodes having the same set of descendants that are in $V - V_f$. Third, we cluster the forbidden nodes having the same set of ancestors that are in $V - V_f$, if they have not already been clustered by the second step. Finally, using the graph computed by the third step, we cluster the nodes that are not forbidden, i.e., $v_i \in V - V_f$. The enhancement in the clustering of the forbidden nodes also enhances the clustering of the nodes that are not forbidden.

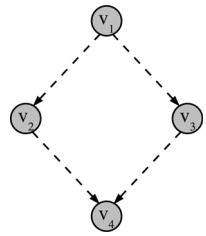

Fig. 5. Connectivity of forbidden nodes helps us reduce the search space.

The result is the clustered graph $G'(V', E')$, where $V'_f \in V'$ denotes the set of forbidden nodes. Reducing the number of forbidden nodes in the clustered graph (i.e., $|V'_f|$) immediately contributes to a reduction in the exponential complexity and reduces the time to compute (3) as well. On the other hand, reducing the number of nodes that are not forbidden in the clustered graph (i.e., $|V - V'_f|$) does not affect the exponential complexity, but again reduces the time to compute (3), which is in the inner loop of our enumeration algorithms.

Finally, we derive a tighter upper bound on the number of maximal convex subgraphs using the clustered graph G'.

Theorem 3: There exists an upper bound of $2^{|V'_f|}$ on the number of maximal convex subgraphs.

Proof: Mathematical induction based on the number of forbidden nodes in G'. ■

B. Building a Search Tree

To demonstrate further ways of reducing the complexity, we construct a new graph $G''(V'', E'')$ from the clustered graph $G'(V', E')$. G'' is a simplified form of G', and contains only the forbidden nodes of G'. The edges of G'' store the connectivity information between the forbidden nodes of G'. More formally, we have $V'' = V'_f$, and a directed edge is defined between two nodes v_i, v_j of G'' only if there is a path from v_i to v_j that does not go through another $v_k \in V'_f$ in G'. Such a simplification is introduced to clarify our search algorithm, which reduces the search space using only the connectivity information between the forbidden nodes.

Fig. 5 illustrates a simple graph G''. Setting $d_1 = 0$ disables inclusion of any node that is a descendant of v_1 in the solution. In such a case, we can simply set $d_i = 0$ for all forbidden nodes that are descendants of v_1 (i.e., $d_2 = d_3 = d_4 = 0$). In other words, we do not need to explore the combinations where $d_1 = 0$ and at least one of d_2, d_3, d_4 is not zero. In practice, the number of possible d_i combinations for $v_i \in V''$ (where $a_i = 1 - d_i$) is much smaller than $2^{|V''|}$. We exploit this property to design a simple and efficient algorithm for maximal convex subgraph enumeration.

Fig. 6 shows the pseudo-code of our algorithm. We first apply a node clustering step on G and obtain the clustered graph G'. Next, we derive G'' from G'. After that, we order the nodes of G'' topologically such that if there exists a path from v_i to v_j in G'', v_i is associated with a lower index value than v_j. Our enumeration algorithm uses this ordering while building a search tree, i.e., the nodes with lower indexes have their d_i

```
1: ALGORITHM: search(index, choice, graph, disabled)
2: local_disabled = disabled;
3: current_combination[index] = choice;
4: if index == size(graph)-1 then
5:     if M(current_combination) > M(best_combination) then
6:         best_combination = current_combination;
7:     end if
8:     return;
9: end if
10: if choice == 0 then
11:     local_disabled = local_disabled ∪ descendants(index);
12: end if
13: index=index+1;
14: search(index, 0, graph, local_disabled);
15: if index ∉ local_disabled then
16:     search(index, 1, graph, local_disabled);
17: end if
18: ALGORITHM: enumerate()
19: Apply preprocessing and clustering on G to generate G';
20: Derive G'' from G';
21: Topologically sort the nodes of G'';
22: current_combination = ∅;
23: best_combination = ∅;
24: search(0, 0, G'', ∅);
25: search(0, 1, G'', ∅);
26: return best_combination;
```

Fig. 6. Enumeration algorithm: the best solution is defined by M(S). "current_combination[index]" stores the value of d_{index}. The parameter "choice" represents the direction of the branch. Setting "choice" equal to zero disables the descendants of "index" in G''. The disabled nodes call the recursive search function only once by setting the "choice" argument equal to zero. The algorithm backtracks when all the nodes in G'' are evaluated.

values assigned earlier. The search tree is built using a recursive search function, where each node can make a zero or one branch. A zero branch on node i sets d_i to zero, and effectively disables setting the d values to one for the descendants of node i in G'', all of which appear later in the topological ordering. When the search algorithm backtracks and sets d_i to one, the disabled descendants must be enabled again. This feature can be implemented using a local array or a stack. The search algorithm produces combinations of d_i values. Each such combination is associated with a convex subgraph that can be derived using (3). In this work, we keep track of the best solution only, which maximizes a given metric function $M(S)$. However, our approach can easily be adapted to store all of the enumerated subgraphs too.

Fig. 7 shows the d_i combinations identified during the execution trace of our enumeration algorithm on the graph of Fig. 5. We observe that only six out of 16 possible combinations had to be enumerated. In fact, the worst case scenario of $2^{|V''|}$ can occur only if all forbidden nodes are disconnected from each other. This is illustrated in Fig. 3, where the two forbidden nodes result in 2^2 combinations. On the other hand, when the forbidden nodes in the G'' are simply cascaded, the number of combinations enumerated by our algorithm grows only linearly with the number of nodes. As a result, the enumeration complexity depends primarily on the number of nodes in G'' and their connectivity.

Our enumeration algorithm uses the topological ordering while building the search tree, i.e., the nodes that are higher in the topological ordering are handled first. The constraints of type $d_i = 0$ are propagated to the lower levels to reduce the search space. However, it is possible to use other orderings as well. As an example, it is also possible to invert the topological

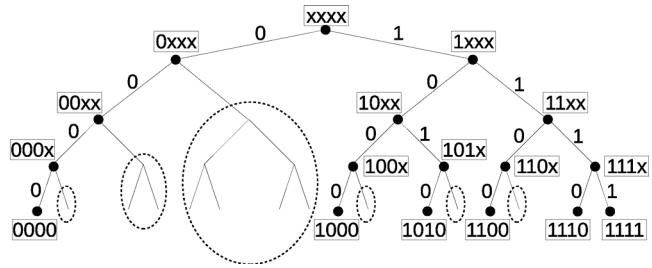

Fig. 7. Search tree is built by the enumeration algorithm for the G'' shown in Fig. 5. A branch at level i of the search tree assigns a zero or one value to d_i. The leaves of the search tree represent the enumerated d_i combinations. Only six out of 16 combinations are enumerated for this example. The dashed ellipses show the pruned regions of the search tree.

ordering and propagate the constraints of type $a_i = 0$. A similar approach is described in [28], where the nodes are ordered by estimating the remaining search space size associated with the selection of each node, and by allowing the constraints of both types, $a_i = 0$ and $d_i = 0$, to be propagated. A disadvantage of this approach is the additional processing time that is spent for the selection of the branch node before each call to the recursive search function.

While setting the d_i value for a $v_i \in V''$, our approach effectively divides $G'(V', E')$ into two parts: 1) nodes that are ancestors of v_i and nodes that are neither ancestors nor descendants of v_i and 2) nodes that are descendants of v_i and nodes that are neither ancestors nor descendants of v_i. A similar graph division operation is presented also in [28]. A main difference between this work and [28] is that our algorithm computes (3) at the leaves of the search tree, whereas [28] updates an intermediate graph data structure at each branch of the search tree using cheaper bitwise operations.

The algorithms described in [19] and, in this work, can enumerate convex subgraphs that are not maximal, and the same subgraph can be generated multiple times. However, the clustering techniques described in this work significantly reduce the amount of redundancy. On the other hand, [28] does not incorporate any clustering techniques. It is reported in [28] that the algorithm of [28] does not enumerate nonmaximal subgraphs. Moreover, [28] describes an efficient way of detecting and discarding repeated subgraphs. Note that such a step is not necessary for our work since our technique does not need to store all the enumerated subgraphs.

VII. Merit Functions for Evaluating Subgraphs

In this work, the merit function $M(S)$ is a parameter for design space exploration, and can be specified by a designer. Basically, $M(S)$ defines an ordering of the subgraphs. If the objective of optimization is to improve processor performance, $M(S)$ should provide an estimation of the reduction in the schedule length of the application by moving subgraph S from software to hardware. In a different setup, $M(S)$ can integrate area and power consumption estimations as well.

The software execution latency $SW(S)$ of a subgraph S can be estimated by statically scheduling S using the base processor instructions. The hardware execution latency $HW(S)$ can be obtained by the hardware synthesis of S and by pipelining the

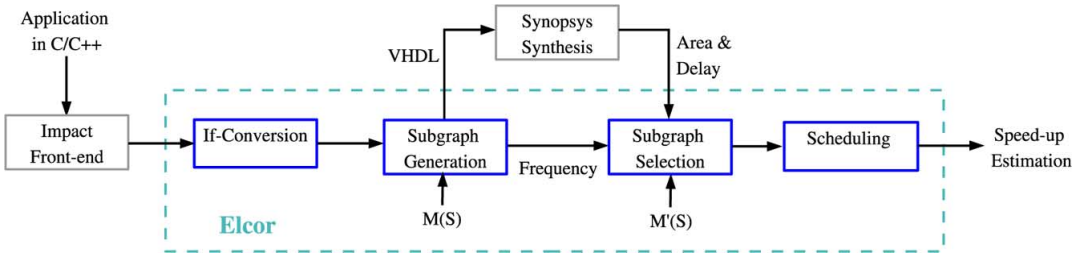

Fig. 8. We adapt the CHIPS [18] tool chain with a novel enumeration algorithm and a new ILP model for Subgraph Generation. Our tools are integrated into the Trimaran compiler infrastructure. Starting with C/C++ code, our tool chain automatically produces behavioral hardware descriptions of the generated subgraphs in VHDL and the scheduling statistics after replacing selected subgraphs with custom instructions. User defined merit functions $M(S)$ and $M'(S)$ can be integrated into subgraph generation and subgraph selection for ranking the subgraphs. We advocate the use of a simpler $M(S)$ in subgraph generation.

final circuit to achieve a target clock frequency. The communication latency $C(S)$ represents the additional cycles needed to transfer the input and output operands of S between the base processor and the custom logic. If the goal is to improve the performance of the processor, the objective of optimization can be formulated as maximizing the reduction in the schedule length by moving the source level data flow subgraph S from software to hardware as follows:

$$M(S) = SW(S) - HW(S) - C(S).$$

Estimating $SW(S)$, $HW(S)$, and $C(S)$ accurately for each subgraph S within the inner loop of any optimization algorithm can be a very time-consuming process. Accurate estimation of $SW(S)$ requires integration of instruction scheduling algorithms under base processor resource constraints. Accurate estimation of $HW(S)$ requires taking into account the effects of hardware optimizations, such as word-length optimization, arithmetic transformations, logic minimization, technology mapping, and register retiming. Similarly, accurate estimation of $C(S)$ requires evaluation of the effects of optimal data partitioning, including the use of local memories and custom state registers, and optimal pipelining after the synthesis of S.

In this work, to estimate $SW(S)$, a software latency $s_i \in Z^+$ is associated with every graph node $v_i \in V - V_f$, which gives the time in clock cycles that it takes to execute v_i on the pipeline of the base processor. To estimate $HW(S)$, every graph node $v_i \in V - V_f$ is associated with a hardware latency $h_i \in R$, which is estimated by synthesizing individual operators using Synopsys Design Compiler and normalizing to the delay of a 32-bit adder. Given the number of register file read and write ports, $C(S)$ is estimated by calculating the data transfer cycles needed to retrieve the input operands of S before its computation starts and the cycles needed to write back the results when the computation of S ends. In this work, three different merit functions are used for ranking subgraphs, given here.

1) $M_1(S)$ identifies S with the maximum accumulated software latency, and assumes that $HW(S)$ and $C(S)$ can later be optimized by a postprocessing step, which involves hardware optimizations to minimize the critical path length, and the use of local memories and custom state registers to minimize the communication overhead:

$$M_1(S) = \sum_{i \in I}(s_i x_i). \tag{4}$$

2) $M_2(S)$ optimizes the difference between maximum accumulated software latency and the sum of $HW(S)$ and $C(S)$, where $HW(S)$ is estimated by computing the critical path of the source level data flow graph S using h_i values and by applying an unconstrained scheduling. We note that $M_2(S)$ is equivalent to the objective function definition of [18]

$$M_2(S) = \sum_{i \in I}(s_i x_i) - HW(S) - C(S). \tag{5}$$

3) Similar to $M_2(S)$, $M_3(S)$ optimizes the difference between maximum accumulated software latency and the sum of $HW_S(S)$ and $C(S)$. However, $HW_S(S)$ is estimated by the actual synthesis of S using Synopsys Design Compiler. As a result, $M_3(S)$ provides a much more accurate estimation than $M_1(S)$ and $M_2(S)$:

$$M_3(S) = \sum_{i \in I}(s_i x_i) - HW_S(S) - C(S). \tag{6}$$

VIII. Overall Approach

The algorithms described in this work are integrated into the Trimaran compiler.[1] Fig. 8 illustrates the associated tool chain. Starting with an application specification in C/C++, an if-conversion pass selectively eliminates control dependencies within an application and transforms multiple basic blocks into a single basic block with predicated instructions. This extends the scope of our algorithms, and enables us to identify coarser grain custom instructions. Synopsys synthesis provides area and delay estimations for generated custom instructions and Trimaran scheduling statistics are used to estimate the number of execution cycles with and without custom instructions.

Given the DFG of a basic block, the subgraph generation algorithm picks the subgraph with the maximum $M(S)$ value as the most promising custom instruction candidate. This subgraph can be found either by ILP (Section V) or by enumeration (Section VI). Additionally, different $M(S)$ functions can be integrated into subgraph generation. After the computation of the first custom instruction candidate, the associated subgraph is collapsed into a single forbidden node in the DFG. Next, a new subgraph with the maximum $M(S)$ value among the remaining

[1]Trimaran. [Online]. Available: http://www.trimaran.org

TABLE II
EFFECT OF CLUSTERING ON THE NUMBER OF FORBIDDEN NODES AND ON THE NUMBER OF REMAINING NODES IN THE LARGEST BASIC BLOCK

Benchmark	Original		Basic Preprocessing		Partial Clustering		Simple Clustering		Enhanced Clustering	
	$\|V_f\|$	$\|V-V_f\|$	$\|V'_f\|$	$\|V'-V'_f\|$	$\|V'_f\|$	$\|V'-V'_f\|$	$\|V'_f\|$	$\|V'-V'_f\|$	$\|V'_f\|$	$\|V'-V'_f\|$
AES enc.	43	274	36	274	36	86	36	86	33	84
AES dec.	43	458	40	458	40	90	40	90	33	85
DES	176	646	128	646	128	274	128	274	96	242
SHA (full)	90	1065	0	1065	0	1	0	1	0	1
IDEA	8	88	0	88	0	1	0	1	0	1
djpeg	19	73	8	73	8	19	8	19	1	3
g721encode	32	99	6	99	6	11	6	11	2	6
g721decode	32	99	7	99	7	15	7	15	3	9
rawcaudio	8	46	2	46	2	6	2	6	2	6
rawdaudio	8	37	3	37	3	10	3	10	3	10

nodes is picked as the second custom instruction candidate. The process continues until no more maximal convex subgraphs composed of more than one node can be found. This process is similar to the iterative subgraph generation algorithm described in [18]. The approach of generating the most promising subgraph first helps in reducing the overall search space, without sacrificing source-code coverage. However, such a strategy precludes the possibility of including one DFG node in two or more custom instructions, which could provide benefits in multi-issue processors. The subgraph generation algorithm applies enumeration or ILP multiple times in each basic block. Note that the number of iterations carried out on a basic block is usually only a few. The DFG shrinks significantly after each iteration, which reduces the run-times of both enumeration and ILP considerably.

In this work, we have integrated only $M_1(S)$ and $M_2(S)$ into the subgraph generation algorithms as it is often impractical to include the hardware synthesis in the inner loop of an enumeration algorithm having an exponential worst-case complexity. On the other hand, the subgraph selection algorithm makes use of $M_3(S)$, which takes into account both the hardware synthesis results and the data transfer costs, and provides more accurate and realistic speed-up estimations.

The subgraph generation procedure is applied only to application basic blocks with positive execution frequency, and a unified set of subgraphs is generated. Next, structurally equivalent subgraphs that can be implemented using the same hardware are grouped. Given the frequency of execution $F(S)$ of the subgraph S, the amount of reduction in the schedule length of the application by moving S from software to hardware is estimated as $F(S) * M_3(S)$. Finally, a Knapsack model [22] is utilized to select the most promising set of subgraphs under area constraints. An important point is that the area and the delay coefficients used by the subgraph selection step are computed using actual Synopsys synthesis.

IX. RUN-TIME RESULTS

The memory access, branch, and division instructions are marked as forbidden instructions in our experiments. Our algorithms are applied to ten benchmarks from multimedia and cryptography domains [38], [39], including an optimized 32-bit implementation of Advanced Encryption Standard (AES) encryption and decryption [40], and a FIPS-compliant fully unrolled Data Encryption Standard (DES) implementation [41].

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

Table II shows the effect of preprocessing and clustering options on the number of forbidden nodes (i.e., $|V'_f|$) and on the number of nodes that are not forbidden (i.e., $|V' - V'_f|$) in the clustered graph (i.e., G'). The results for the largest basic block of each benchmark are shown. For an explanation of different options, see Section VI-A. Note that the Basic Preprocessing option does not implement any clustering and that all of the clustering options include the Basic Preprocessing. Additionally, the Partial Clustering option implements clustering of the nodes that are not forbidden (i.e., $|V' - V'_f|$ only). We observe that the Basic Preprocessing option usually results in a significant reduction in the number of forbidden nodes, and the Partial Clustering option usually results in a significant compaction in the number of nodes that are not forbidden. Note that [19] uses the Partial Clustering option, whereas [28] implements the Basic Preprocessing option only. Another observation is that Simple Clustering, in practice, results in no improvement with respect to Partial Clustering in terms of the number of forbidden nodes and of the number of nodes that are not forbidden. On the other hand, Enhanced Clustering can further optimize the number of forbidden nodes and the number of nodes that are not forbidden. Another important point to note is that, immediately after Partial Clustering, the largest basic blocks of SHA and IDEA benchmarks are reduced to only a single node that is not forbidden.

Table III presents the run-time results for different clustering and branch ordering options. "Enum-TO" uses the topological ordering described in Section VI.B while choosing the next node to branch, whereas "Enum-AO" uses the adaptive ordering proposed in [28], which tries to pick those nodes that are estimated to reduce the search space more strongly than the others. The setup time includes the time to do the basic preprocessing and the clustering, which increases while moving from Basic Preprocessing towards Enhanced Clustering. Note that "Enum-TO" and "Enum-AO" only include the search time, and exclude the set-up time. The Simple Clustering option is not included in the table because it does not result in any improvement in the search time with respect to the Partial Clustering option and has a setup time comparable to that of the Enhanced Clustering option. We observe that the Partial Clustering option significantly decreases the search time with respect to Basic Preprocessing, and the Enhanced Clustering option further reduces the

TABLE III
RUN-TIME TO COMPUTE THE SUBGRAPH THAT MAXIMIZES $M_1(S)$ IN THE LARGEST BASIC BLOCK (IN SECONDS)

Benchmark	Basic Preprocessing			Partial Clustering			Enhanced Clustering		
	Set-up	Enum-TO	Enum-AO	Set-up	Enum-TO	Enum-AO	Set-up	Enum-TO	Enum-AO
AES enc.	0.00087	79.498	70.479	0.00120	21.549	30.519	0.00164	2.8014	3.4747
AES dec.	0.00161	537.96	564.71	0.00214	120.57	150.64	0.00265	1.5852	1.9655
DES	0.00689	0.24333	0.22559	0.00890	0.13334	0.13348	0.01202	0.02253	0.02288
SHA (full)	0.00735	0	0	0.00735	0	0	0.00735	0	0
IDEA	0.00007	0	0	0.00007	0	0	0.00007	0	0
djpeg	0.00009	0.00025	0.00031	0.00012	0.00008	0.00013	0.00019	0.00001	0.00001
g721encode	0.00021	0.00005	0.00006	0.00024	0.00002	0.00003	0.00036	0.00001	0.00001
g721decode	0.00021	0.00009	0.00013	0.00027	0.00003	0.00004	0.00036	0.00001	0.00001
rawcaudio	0.00004	0.00001	0.00001	0.00005	0.00001	0.00001	0.00012	0.00001	0.00001
rawdaudio	0.00004	0.00001	0.00002	0.00005	0.00001	0.00001	0.00013	0.00001	0.00001

TABLE IV
RUN-TIME TO COMPUTE THE SUBGRAPH THAT MAXIMIZES $M_1(S)$ IN THE LARGEST BASIC BLOCK (IN SECONDS). RUN-TIME RESULTS FOR A COMBINATION OF OUR ENUMERATION ALGORITHM AND SELECTED OPTIMIZATIONS FROM [28] ARE GIVEN IN THE ENUM-CMB COLUMN

	Related Work			This work			
Benchmark	Atasu et al. [19]	Li et al. [28]	Bron-Kerbosch [42]	ILP (CPLEX)	ILP (lpsolve)	Enum	Enum-CMB
AES enc	21.550	25.209	3.6332	0.05642	0.4600	2.8031	1.1987
AES dec	120.57	134.05	3.0706	0.23837	1.6080	1.5879	0.67775
DES	0.14224	0.03354	0.03785	5.4520	171.63	0.03456	0.01533
SHA	0.00735	0.00735	0.00735	0.13555	2.8680	0.00735	0.00735
IDEA	0.00007	0.00007	0.00008	0.00906	0.02800	0.00007	0.00007
djpeg	0.00021	0.00029	0.00035	0.00854	0.03200	0.00020	0.00020
g721encode	0.00026	0.00027	0.00035	0.02472	0.10800	0.00026	0.00026
g721decode	0.00029	0.00030	0.00044	0.02475	0.11600	0.00030	0.00030
rawcaudio	0.00006	0.00006	0.00010	0.00911	0.02000	0.00005	0.00005
rawdaudio	0.00006	0.00006	0.00016	0.00709	0.01600	0.00005	0.00005

search time for AES encryption, AES decryption, DES, and djpeg benchmarks. In the case of the g721encode, g721decode, rawcaudio and rawdaudio benchmarks, the setup time dominates the search time, and the Basic Preprocessing option is usually the fastest. "Enum-AO" improves the run-time slightly only in the case of the Basic Preprocessing option and only for AES encryption and DES. In the remaining cases, it increases the run-time with respect to "Enum-TO", owing to the additional complexity introduced in picking the next node to branch. Table III shows that the fastest options are "Enum-TO" combined with the Basic Preprocessing or the Enhanced Clustering options.

Table IV compares the run-time of our ILP approach (Section V) and our enumeration algorithm (Section VI) with the run-time of the enumeration algorithms described in [19], [28], [42]. The enumeration results of this work, and of [19] can be derived from Table III by summing up the associated setup and search times. Note that [19] combines "Enum-TO" and Partial Clustering options. Our work also uses "Enum-TO" instead of the more costly "Enum-AO" option. However, we choose between Basic Preprocessing and Enhanced Clustering options depending on the graph characteristics. If the number of forbidden nodes is smaller than eight after Basic Preprocessing, our algorithm skips Enhanced Clustering, and effectively uses the Basic Preprocessing option. Otherwise, Enhanced Clustering is used. Such an approach gives us the best run-time results. We have also included the run-time of the ILP using both a commercial solver (CPLEX [43]) and a public domain solver.[2] To provide a comparison with [16], [17], we have implemented the Bron–Kerbosch algorithm [42], which is known as one of the fastest and most scalable maximal clique enumeration algorithms. To provide a fair comparison between different techniques, we keep track of only one subgraph that maximizes $M_1(S)$ in each case.

Table IV shows that [28] significantly improves the run-time of "Enum-AO" combined with Basic Preprocessing option although it generates the same search tree. The improvement is due to the optimizations in the data structures that simplify the computation of (3). We have also integrated the same optimization into our algorithms and observed a considerable improvement in our run-time results. The results for the combined method are given in the column labeled as "Enum-CMB" in Table IV. Note that the results of this work for AES encryption and AES decryption benchmarks are significantly better than both [19] and [28] thanks to the Enhanced Clustering technique introduced in this work, which reduces the exponential complexity of enumeration. However, a combination of our algorithm with selected optimizations from [28] provides the best enumeration run-time results.

Table IV results indicate that enumerating maximal cliques using the Bron–Kerbosch algorithm produces run-time results comparable to those produced by our algorithms. Although both techniques benefit from node clustering (Section VI.A), the Enhanced Clustering technique presented in this work is only applicable if the search tree is constructed in the way described in Section VI-B. As a result, the Simple Clustering option is used in combination with Bron–Kerbosch algorithm in our experiments as described in [16], [17]. Note that our search tree construction technique makes use of a graph (namely G'') that is composed of clusters of forbidden nodes only, i.e., a graph with $|V_f'|$ nodes. Our work derives the upper bound on the number of

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

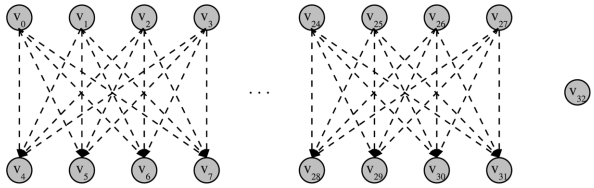

Fig. 9. Connectivity of forbidden nodes for the largest basic block of AES.

maximal convex subgraphs as $2^{|V'_f|}$. However, the independent set and the clique enumeration algorithms described in [16], [17] rely on a graph representation that is composed of clusters of nodes that are not forbidden, i.e., a graph with $|V' - V'_f|$ nodes (see, for example, Fig. 4). It is well known that the number of maximal cliques of a graph with $|V' - V'_f|$ nodes is bounded by $3^{(|V'-V'_f|/3)}$ [42]. Based on the numbers given in Table II, the upper bound derived by our work is a tighter one. On the other hand, the two techniques appear to be equally effective in reducing the search space.

Table IV also demonstrates that, except for AES encryption and AES decryption, enumeration is several orders of magnitude faster than the ILP-based approach. The advantage of using enumeration is more evident when lpsolve is used instead of CPLEX. In fact, CPLEX [43] can automatically recognize constraint matrices that correspond to clique and independent set problems and includes efficient solvers for these problems. We observe that in the case of AES encryption and AES decryption, ILP is faster than enumeration even if lpsolve is used. In these two cases, enumerating all possible maximal convex subgraphs does not appear to be the most efficient way of finding the most promising subgraph. ILP solvers usually apply an objective guided search strategy. Branch-and-bound and branch-and-cut type algorithms used by the ILP solvers reduce the search space not only based on the constraints, but also based on the definition of the metric function, without enumerating every feasible solution.

Fig. 9 illustrates why enumeration is relatively inefficient for AES encryption. On the largest basic block of AES encryption, G'' is composed of 33 nodes after node clustering. This graph is composed of five disconnected groups of nodes, one of which is composed of only a single node. The remaining four groups are identical, and each one independently requires evaluation of 31 possible d_i combinations. In total, our enumeration algorithm evaluates $2 * (31)^4 \approx 2M$ combinations, which takes around 2.8 sec to execute. On the other hand, CPLEX finds the optimal solution in only 0.056 s, providing us with a second and more efficient solution alternative.

Table V provides additional comparisons between our enumeration algorithms and those described in [19] and [28]. Note that the four techniques evaluated in Table V build similar search trees, and each one is adapted to compute a single subgraph that maximizes $M_1(S)$. Note also that the run-time results presented in [28] for rijndael are around 10 s, an order of magnitude larger than those we present under the [28] column. Such a slowdown is mainly due to the overhead of storing all of the enumerated subgraphs and removing the redundant ones, which our algorithms avoid. Our algorithms store only the best subgraph during enumeration, and apply enumeration multiple

TABLE V
SINGLE ENUMERATION RUN-TIME FOR RIJNDAEL [38] (IN SECONDS)

function	BB id	$\|V_f\|$	$\|V\|$	[19]	[28]	Enum	Enum-CMB
_encrypt	4808	48	172	0.96	1.22	0.027	0.013
_encrypt	15035	58	196	1.29	1.22	0.037	0.015
_encrypt	20435	54	189	0.92	0.70	0.028	0.010

TABLE VI
TOTAL RUN TIME (IN SECONDS). CPLEX IS USED AS THE ILP SOLVER

Benchmark	# Inst.	# BBs	ILP (M_2)	ILP (M_1)	Enum (M_1)
AES enc	735	27	0.58871	0.25240	5.6109
AES dec	1011	28	1.0251	0.50534	3.2561
DES	1235	45	210.03	30.892	0.38020
SHA	1339	30	0.45155	0.32819	0.11164
IDEA	595	65	0.07450	0.04175	0.00479
djpeg	5503	957	1.0170	0.77304	0.05980
g721encode	892	85	1.2475	0.62377	0.03509
g721decode	864	79	1.3006	0.74369	0.03379
rawcaudio	119	13	0.06766	0.03284	0.00284
rawdaudio	102	11	0.06569	0.02990	0.00267

times until all DFG nodes are covered. Such an approach usually results in a very small run-time overhead. Often, a few iterations are sufficient for each basic block, and the enumeration run-time drops significantly after the first iteration. For each of the three basic blocks evaluated, the total run-time observed is smaller than 0.04 s for the Enum column, taking into account all of the iterations needed.

Finally, Table VI shows the total run-time of our subgraph-generation algorithm for our initial ten benchmarks. For each benchmark, the total number of instructions and the total number of basic blocks are also provided. The table shows the run-time of ILP, using both $M_1(S)$ and $M_2(S)$ as metric functions, and CPLEX as the solver. We also provide the run-time of our enumeration algorithm using $M_1(S)$ as the metric function. We observe that the use of $M_1(S)$ reduces the run-time of the ILP usually by half. However, in the case of DES, the run-time improvement is around seven fold. Enumeration is slower than ILP in the case of AES encryption and AES decryption, but it is three fold faster for SHA, and about an order of magnitude faster for the remaining seven benchmarks.

X. DESIGN SPACE EXPLORATION RESULTS

This work assumes a single-issue baseline machine with predication support containing 32 32-bit general-purpose registers and 32 single-bit predicate registers, based on the HPL-PD architecture [44]. Software latency of integer multiplication instructions is defined to be two cycles, and software latency of the rest of the integer operations is defined to be a single cycle. Custom instructions are synthesized to UMC's 130 nm standard cell library using Synopsys Design Compiler. Note that custom instructions can be pipelined in order not to increase the cycle time of the base processor, which is estimated to be around the critical path delay of a 32-bit carry propagate adder.

Figs. 10 and 11 demonstrate the area and performance tradeoffs for the DES, and IDEA benchmarks using $M_1(S)$ as the metric function in subgraph generation and $M_3(S)$ as the metric function in subgraph selection. The execution cycle count without using custom instructions is normalized to a hundred and the percent reduction in the execution cycles using

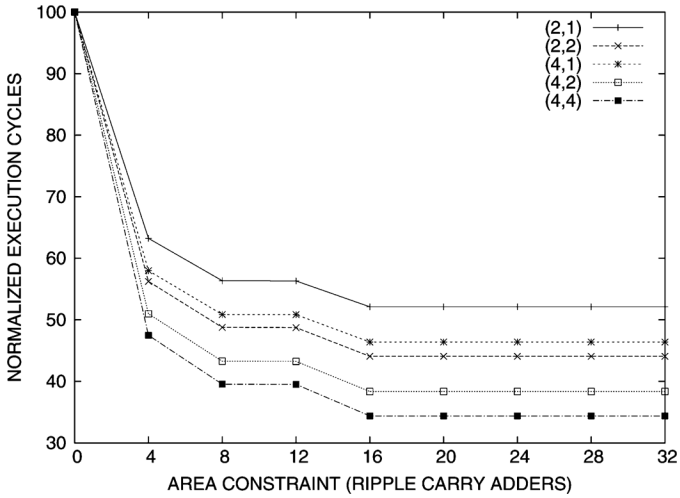

Fig. 10. DES: effect of increasing the number of register file ports on the performance for a range of area constraints.

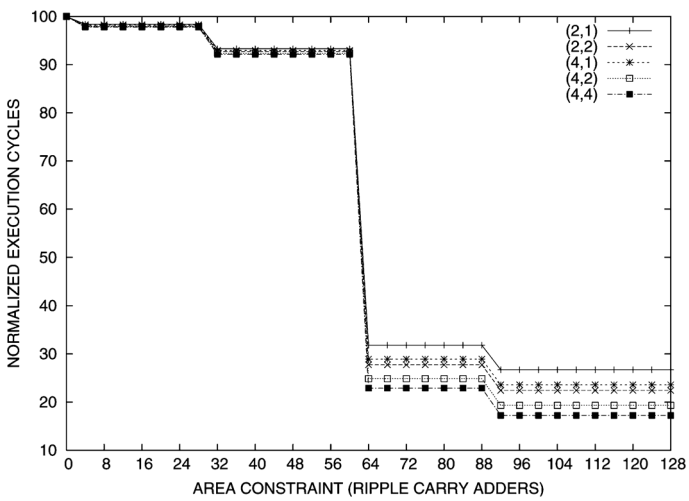

Fig. 11. IDEA: effect of increasing the number of register file ports on the performance for a range of area constraints.

custom instructions is shown for a range of area constraints. Five register file read and write port combinations are evaluated: (2,1), (2,2), (4,1), (4,2), and (4,4). The combination (2,1) stands for a register file with two read ports and a single write port. Increasing the number of read and write ports supported by the register file decreases the communication cost of the custom instructions (i.e., $C(S)$ in the computation of $M_3(S)$) and increases the speed-up. In the case of DES, we observe that an area equivalent to 16 ripple carry adders (RCAs) is sufficient to achieve the highest speed-up. In the case of IDEA, a significant reduction in the execution cycles can be achieved at a cost of 64 RCAs, and an additional area budget of 28 RCAs results in further reduction. Such stepwise behavior is mainly due to a few large and frequently executed custom instructions (i.e., subgraphs).

Fig. 12 shows the percent reduction in the execution cycles of our ten benchmarks using custom instructions. For each benchmark, two columns are provided. The column on the right shows the results found using $M_1(S)$ in subgraph generation and the column on the left shows the results found using $M_2(S)$ in subgraph generation. Note that the ILP model of Section V and

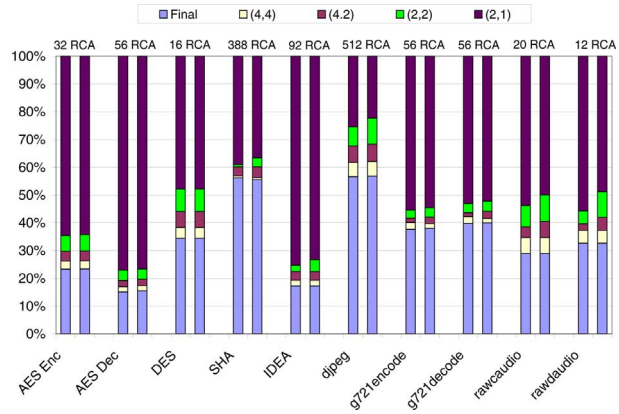

Fig. 12. Reduction in the number of execution cycles. Additional register file ports improve the results. Two columns are provided for each benchmark: the column on the right shows the results of $M_1(S)$ and the column on the left shows the results of $M_2(S)$. Area costs are shown in ripple carry adders.

the enumeration algorithm of Section VI compute equivalent results for the same merit function definition $M(S)$. The stacked columns show the results computed for four register file read and write port combinations: (2,1), (2,2), (4,2), and (4,4). At the top of each column, the cell area for the associated custom datapath is given in equivalent RCA area costs. For AES Encryption, AES Decryption, DES, g721encode, and g721decode benchmarks, the results of $M_1(S)$ and of $M_2(S)$ differ only marginally. The maximum difference observed for these five benchmarks is less than one percent. For the remaining five benchmarks, the results of $M_2(S)$ are better when the register file ports are limited to two read ports and one write port. However, as the register file port constraints are relaxed, the results of $M_1(S)$ and $M_2(S)$ again become indistinguishable. For a register file with two read ports and one write port, we observe between two to seven percent difference for the SHA, IDEA, djpeg, rawcaudio, and rawdaudio benchmarks. On the other hand, for a register file with two read ports and two write ports, the difference is again smaller than one percent for eight of the benchmarks. Only in the case of rawcaudio and rawcaudio benchmarks is around two percent difference observed. Given a register file port constraint of (4,2) or (4,4), the difference in all cases is even less than one percent and, occasionally, zero. For some benchmarks, such as SHA and g721encode, in some cases, $M_1(S)$ finds better results than $M_2(S)$. We observe that because $M_2(S)$ takes into account the communication cost $C(S)$, it is advantageous over $M_1(S)$ when the register file port constraints are tight. However, in general, the two metric functions produce similar results. In particular, if a basic block is sufficiently large, $M_2(S)$ almost always computes the same subgraph computed by $M_1(S)$, i.e., the maximum convex subgraph of the basic block.

XI. Examples of Custom Instructions

Fig. 13 shows the most promising custom instruction our algorithms automatically identify from the DES C code. Note that, in this case, $M_1(S)$ and $M_2(S)$ compute equivalent results. The software implementation fully unrolls DES round transformations within a single basic block, which results in 822 base processor instructions. The custom instruction of Fig. 13 implements the combinational logic between the memory access

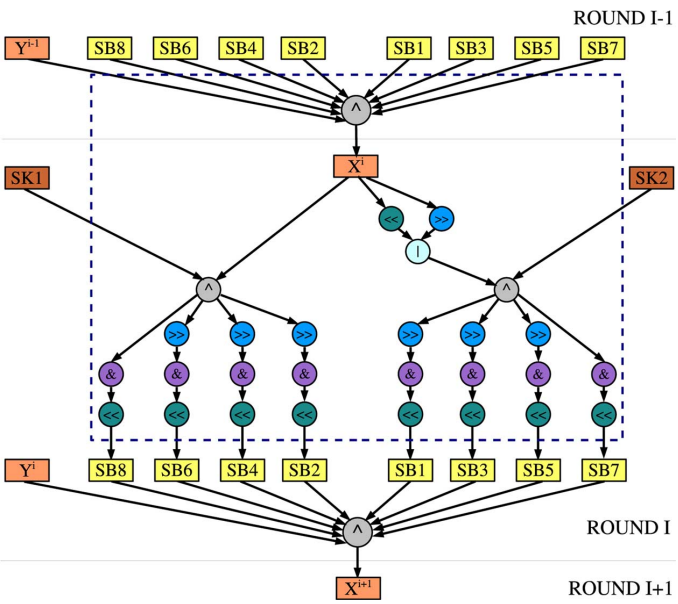

Fig. 13. Most promising custom instruction for DES. Although the custom instruction has eleven input and nine output operands, the use of local memories reduces the communication cost $C(S)$ to zero.

layers of two consecutive DES rounds. Our algorithms automatically identify 15 equivalent instances of this instruction in the same basic block. We note that X and Y represent the DES encryption state. Eight of the inputs of the custom instruction are static lookup table (LUT) entries (SBs), and two of the inputs (SK1, SK2) contain the DES round key. Accordingly, eight of the outputs contain the addresses of the LUT entries that should be fetched for the next round. We observe that the size of the LUTs is rather small (256 bytes only). We could avoid all of the related main memory accesses and address calculations by embedding the eight LUTs 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. Although the custom instruction has eleven input and eight output operands, the use of local memories reduces its communication cost $C(S)$ to zero. These optimizations reduce the size of the core basic block of DES from 822 to 22 instructions, which incorporate only five base processor instructions and three different types of custom instructions. Our synthesis results show that each of these custom instructions has only a single cycle latency $HW(S)$. The result is a 30-fold speed-up.

Our second example is SHA. The most time-consuming part of SHA is the compression function, which is applied 80 times within a loop body. This loop body is often unrolled to improve the software performance of SHA. In this work, we evaluate five different implementations: the first implementation does not unroll the main loop of SHA; the next three implementations have their loops unrolled two, five, and ten times. The fifth implementation has the SHA main loop fully unrolled. In all five cases, our algorithms identify only a single maximal convex subgraph within the loop body. We observe that unrolling the loop body results in the identification of larger DFG subgraphs. Such subgraphs often translate into an increase in the latency and the area cost of custom instructions.

TABLE VII
RELATIVE LATENCY AND AREA COEFFICIENTS FOR VARIOUS OPERATORS BASED ON SYNTHESIS RESULTS USING UMC'S 130-NM PROCESS

Operator	Latency	Area
32-bit + 32-bit adder	1.000	1.000
32-bit * 32-bit multiplier	1.524	18.463
32-bit and	0.010	0.236
32-bit xor	0.029	0.415
32-bit shifter	0.295	1.977
32-bit shifter (constant)	0.000	0.000
32-bit comparator (eq)	0.095	0.512
32-bit comparator (geq)	0.552	0.632

TABLE VIII
ESTIMATED *VERSUS* SYNTHESIZED LATENCY AND AREA RESULTS FOR THE MOST PROMISING CUSTOM INSTRUCTIONS WHILE UNROLLING SHA

	Est. Latency	Synt. Latency	Est. Area	Synt. Area
SHA	3.035	1.095	7.658	5.554
SHA (2)	6.047	2.099	13.316	8.581
SHA (5)	15.085	5.119	30.290	21.266
SHA (10)	30.149	10.393	58.580	42.405
SHA (full)	242.037	82.495	433.981	345.440

Table VII presents the synthesis results obtained for various source-level operations using Synopsys Design Compiler. The latency and area results given are normalized to the latency and area results of a 32-bit adder. Note that $M_2(S)$ estimates the latency $HW(S)$ of custom instructions as the latency of the longest path length between the source and destination operands of the source-level DFG subgraph S using the latency coefficients (h_i) of Table VII. Recently, algorithms for pipelining source-level DFGs have also been described [14], [17], which use similar estimation techniques. Table VIII shows that such approaches can be highly inaccurate because the target hardware libraries and the optimizations applied by the synthesis tools are not taken into account. Table VIII compares the synthesized latency $(HW_S(S))$ and area results of the custom instructions for five SHA implementations with the estimated latency $(HW(S))$ and area results. Note that area estimations are computed by assuming a cumulative cost model and using the coefficients of Table VII. A large gap between the estimations and the synthesis results can be observed. This gap becomes even larger as the main loop of SHA gets unrolled, and larger DFG subgraphs are identified. Assuming that the target cycle time is around the delay of a 32-bit adder, the difference between the estimated latency and the synthesized latency goes up to 160 cycles for fully unrolled SHA. We conclude that estimating $HW(S)$ at the source level can be highly inaccurate, which also explains the similarity between $M_1(S)$ and $M_2(S)$ results in Fig. 12.

Note that, although the custom instruction generated for DES example is highly reusable and has a low area overhead, the custom instructions generated for SHA have a high area overhead and almost no reuse potential. As illustrated in [18], it is in fact possible to find more area-efficient custom processor configurations for SHA, which also offer a reasonably good performance, by searching for smaller and more reusable subgraphs. Ideally, a combination of these two approaches should be evaluated to identify the most promising area and performance tradeoffs for custom instruction processors.

XII. Conclusion

This paper presented FISH, a novel approach for improving the efficiency of automatically generated custom instruction processors and the associated theoretical and practical results. FISH introduces fast custom instruction synthesis methods based on a novel subgraph enumeration algorithm and an integer linear programming formulation. FISH evaluates maximal convex subgraphs of source-level DFGs as custom instruction candidates. The use of multiple different merit functions for ranking custom instruction candidates is enabled within a design space exploration framework. The run-time results show that the search space reduction techniques described in this work result in the fastest enumeration algorithms among the state-of-the-art maximal convex subgraph enumeration algorithms, including those based on maximal clique enumeration. In addition, FISH derives an upper bound on the number of maximal convex subgraphs in a DFG and on the complexity of enumeration that is tighter than the respective bound known for maximal clique enumeration.

In most of the cases, enumeration is faster than ILP. However, in those cases where enumeration is relatively inefficient, ILP provides a fast alternative. Our experiments show that in most of the cases a simple merit function, which estimates only the software costs of the subgraphs, can be as good as a more complex merit function that also estimates the hardware execution latencies and the communication costs. In particular, in large basic blocks, the two merit functions compute the same subgraphs most of the time. We show that, for DES, combining our subgraph generation and selection approach with a simple postprocessing step results in a 30-fold speed-up with respect to a single-issue base processor.

Current and future work includes: 1) developing better estimators for critical path computation that take into account arithmetic optimizations, word-length optimizations, target hardware libraries, and wire delays; 2) exploring data-layout and loop transformations and applying our techniques on larger code segments beyond predicated basic blocks; and 3) enhancing our tool chain to implement resource-constrained scheduling algorithms for pipelining register file accesses [14], [17]. Possible further extensions include supporting design development with power and energy constraints [45], integrating our design flow into synthesis tools for heterogeneous multiprocessor SoC devices [46], [47] and combining our algorithms with fast synthesis techniques [48], [49] to enable dynamic hardware/software partitioning and run-time reconfiguration.

Acknowledgment

The authors would like to thank C. Bolliger, A.-M. Cromack, and C. Hagleitner, IBM Research—Zurich, and the anonymous reviewers for their valuable comments.

References

[1] J. Henkel, "Closing the SoC design gap," *Computer*, vol. 36, no. 9, pp. 119–121, Sep. 2003.

[2] M. Gschwind, "Instruction set selection for ASIP design," in *Proc. CODES*, 1999, pp. 7–11.

[3] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood, "Lx: A technology platform for customizable vliw embedded processing," in *Proc. ISCA*, 2000, pp. 203–213.

[4] R. E. Gonzalez, "Xtensa: A configurable and extensible processor," *IEEE Micro*, vol. 20, no. 2, pp. 60–70, 2000.

[5] G. Martin, "Recent developments in configurable and extensible processors," in *Proc. ASAP*, Sep. 2006, pp. 39–44.

[6] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Proc. 40th DAC*, Anaheim, CA, Jun. 2003, pp. 256–261.

[7] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *Proc. FPGA*, Monterey, CA, Feb. 2004, pp. 183–189.

[8] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors," in *Proc. CASES*, Sep. 2004, pp. 69–78.

[9] L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Trans. Comput-Aided Des. (CAD) Integr. Circuits Syst.*, vol. 25, no. 7, pp. 1209–1229, Jul. 2006.

[10] P. Yu and T. Mitra, "Disjoint pattern enumeration for custom instructions identification," in *Proc. FPL*, Aug. 2007, pp. 273–278.

[11] X. Chen, D. L. Maskell, and Y. Sun, "Fast identification of custom instructions for extensible processors," *IEEE Trans. Comput-Aided Des. (CAD) Integr. Circuits Syst.*, vol. 26, no. 2, pp. 359–368, Feb. 2007.

[12] G. Gutin, A. Johnstone, J. Reddington, E. Scott, and A. Yeo, "An algorithm for finding input-output constrained convex sets in an acyclic digraph," in *Proc. WG*, 2008, pp. 206–217.

[13] K. Atasu, G. Dündar, and C. Özturan, "An integer linear programming approach for identifying instruction-set extensions," in *Proc. CODES + ISSS*, Jersey City, NJ, Sep. 2005, pp. 172–177.

[14] L. Pozzi and P. Ienne, "Exploiting pipelining to relax register-file port constraints of instruction-set extensions," in *Proc. CASES*, 2005, pp. 2–10.

[15] K. Atasu, R. G. Dimond, O. Mencer, W. Luk, C. Özturan, and G. Dündar, "Optimizing instruction-set extensible processors under data bandwidth constraints," in *Proc. DATE*, Nice, France, Apr. 2007, pp. 588–593.

[16] N. Pothineni, A. Kumar, and K. Paul, "Application specific datapath with distributed I/O functional units," in *Proc. VLSI Design*, Hyderabad, India, Jan. 2007, pp. 551–558.

[17] A. K. Verma, P. Brisk, and P. Ienne, "Rethinking custom ISE identification: A new processor-agnostic method," in *Proc. CASES*, Salzburg, Austria, Sep. 2007, pp. 125–134.

[18] K. Atasu, C. Özturan, G. Dündar, O. Mencer, and W. Luk, "CHIPS: Custom hardware instruction processor synthesis," *IEEE Trans. Comput.-Aided Des. (CAD) Integr. Circuits Syst.*, vol. 27, no. 3, pp. 528–541, Mar. 2008.

[19] K. Atasu, O. Mencer, W. Luk, C. Özturan, and G. Dündar, "Fast custom instruction identification by convex subgraph enumeration," in *Proc. ASAP*, Leuven, Belgium, Jul. 2008, pp. 1–6.

[20] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man, "Instruction set definition and instruction selection for ASIPs," in *Proc. ISSS*, 1994, pp. 11–16.

[21] R. Kastner, S. Ogrenci-Memik, E. Bozorgzadeh, and M. Sarrafzadeh, "Instruction generation for hybrid reconfigurable systems," in *Proc. ICCAD*, 2001, pp. 127–130.

[22] N. Clark, H. Zhong, and S. Mahlke, "Processor acceleration through automated instruction set customization," in *Proc. MICRO*, 2003, pp. 184–88.

[23] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "Custom-instruction synthesis for extensible-processor platforms," *IEEE Trans. Comput.-Aided (CAD) Integr. Circuits Syst.*, vol. 23, no. 2, pp. 216–228, Feb. 2004.

[24] P. Biswas, S. Banerjee, N. Dutt, L. Pozzi, and P. Ienne, "ISEGEN: Generation of high-quality instruction set extensions by iterative improvement," in *Proc. DATE*, Mar. 2005, pp. 1246–1251.

[25] R. Leupers, K. Karuri, S. Kraemer, and M. Pandey, "A design flow for configurable embedded processors based on optimized instruction set extension synthesis," in *Proc. DATE*, Munich, Germany, Mar. 2006, pp. 581–586.

[26] A. K. Verma, P. Brisk, and P. Ienne, "Fast, quasi-optimal, and pipelined instruction-set extensions," in *Proc. ASPDAC*, Mar. 2008, pp. 334–339.

[27] J. Reddington, G. Gutin, A. Johnstone, E. Scott, and A. Yeo, "Better than optimal: Fast identification of custom instruction candidates," in *Proc. CSE (2)*, 2009, pp. 17–24.

[28] T. Li, Z. Sun, W. Jigang, and X. Lu, "Fast enumeration of maximal valid subgraphs for custom-instruction identification," in *Proc. CASES*, 2009, pp. 29–36.

[29] P. Biswas, V. Choudhary, K. Atasu, L. Pozzi, P. Ienne, and N. Dutt, "Introduction of local memory elements in instruction set extensions," in *Proc. DAC*, 2004, pp. 729–734.
[30] N. Moreano, E. Borin, C. de Souza, and G. Araujo, "Efficient datapath merging for partially reconfigurable architectures," *IEEE Trans. Comput.-Aided Des. (CAD) Integr. Circuits Syst.*, vol. 24, no. 7, pp. 969–980, Jul. 2005.
[31] N. Cheung, S. Parameswaran, J. Henkel, and J. Chan, "MINCE: Matching instructions using combinational equivalence for extensible processor," in *Proc. DATE*, Feb. 2004, pp. 1020–1027.
[32] A. Chattopadhyay, W. Ahmed, K. Karuri, D. Kammler, R. Leupers, G. Ascheid, and H. Meyr, "Design space exploration of partially re-configurable embedded processors," in *Proc. DATE*, Apr. 2007, pp. 319–324.
[33] S.-K. Lam and T. Srikanthan, "Rapid design of area-efficient custom instructions for reconfigurable embedded processing," *J. Syst. Architecture*, vol. 55, no. 1, pp. 1–14, 2009.
[34] S. P. Seng, W. Luk, and P. Y. K. Cheung, "Run-time adaptive flexible instruction processors," in *Proc. FPL*, Sep. 2002, pp. 545–555.
[35] L. Bauer, M. Shafique, S. Kramer, and J. Henkel, "RISPP: Rotating instruction set processing platform," in *Proc. DAC*, Jun. 2007, pp. 791–796.
[36] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner, "An architecture framework for transparent instruction set customization in embedded processors," in *Proc. ISCA*, Washington, DC, 2005, pp. 272–283.
[37] K. Atasu, "Hardware/software partitioning for custom instruction processors" Ph.D. dissertation, Dept. Comput. Eng., Bogazici Univ., Istanbul, Turkey, 2007 [Online]. Available: http://www.doc.ic.ac.uk/~atasu/Kubilay_Atasu_PhD_Thesis.pdf
[38] M. Guthaus *et al.*, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," Univ. of Michigan, Ann Arbor [Online]. Available: http://www.eecs.umich.edu/mibench/
[39] C. Lee *et al.*, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. MICRO*, Dec. 1997, pp. 330–335.
[40] K. Atasu, M. Macchetti, and L. Breveglieri, "Efficient AES implementations for ARM based platforms," in *Proc. SAC*, 2004, pp. 841–845.
[41] "XySSL—DES and Triple-DES Source Code," [Online]. Available: http://xyssl.org/
[42] C. Bron and J. Kerbosch, "Algorithm 457: Finding all cliques of an undirected graph," *Commun. ACM*, vol. 16, no. 9, pp. 575–577, 1973.
[43] ILOG CPLEX Optimization Software. ILOG. [Online]. Available: http://www.ilog.com/products/cplex/
[44] V. Kathail *et al.*, HPL-PD Architecture Specification, Version 1.0 HP Labs Tech. Rep. HPL-93-80R1, 1993.
[45] Y. Fei, S. Ravi, A. Raghunathan, and N. K. Jha, "A hybrid energy-estimation technique for extensible processors," *IEEE Trans. Comput.-Aided Des. (CAD) Integr. Circuits Syst.*, vol. 23, no. 5, pp. 652–664, May 2004.
[46] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "Application-specific heterogeneous multiprocessor synthesis using extensible processors," *IEEE Trans. Comput.-Aided Des. (CAD) Integr. Circuits Syst.*, vol. 25, no. 9, pp. 1589–1602, Sep. 2006.
[47] S. L. Shee and S. Parameswaran, "Design methodology for pipelined heterogeneous multiprocessor system," in *Proc. DAC*, Jun. 2007, pp. 811–816.
[48] R. Lysecky, F. Vahid, and S. X.-D. Tan, "Dynamic FPGA routing for just-in-time FPGA compilation," in *Proc. DAC*, 2004, pp. 954–959.
[49] R. Lysecky and F. Vahid, "Design and implementation of a MicroBlaze-based warp processor," *ACM Trans. Embed. Comput. Syst.*, vol. 8, no. 3, pp. 1–22, 2009.

Kubilay Atasu (S'03–M'08) received the B.Sc. degree in computer engineering from Boğaziçi University, Istanbul, Turkey, in 2000, the M.Eng. degree in embedded systems design from University of Lugano, Lugano, Switzerland, in 2002, and the Ph.D. degree in computer engineering from Boğaziçi University, Istanbul, Turkey, in 2007.

From 2005 to 2008, he was a Research Associate with the Department of Computing, Imperial College London, London, U.K. Since 2008, he has been with Systems Department, IBM Research—Zurich, Ruschlikon, Switzerland. His research interests include custom processors, electronic design automation, and programmable accelerator engines for packet processing algorithms.

Dr. Atasu was the recipient of a Best Paper Award at the Design Automation Conference in 2003 and the IEEE International Conference on Application-Specific Systems, Architectures, and Processors in 2008.

Wayne Luk (S'85–M'89–SM'06–F'09) received the M.A., M.Sc., and D.Phil. degrees in engineering and computing science from the University of Oxford, Oxford, U.K., in 1984, 1985, and 1989, respectively.

He is a Professor of Computer Engineering with the Department of Computing, Imperial College London, London, U.K., and leads the Custom Computing Group there. He is also a Visiting Professor with Stanford University, Stanford, CA. His research interests include theory and practice of customizing hardware and software for specific application domains, such as graphics and image processing, multimedia, and communications. Much of his current work involves high-level compilation techniques and tools for parallel computers and embedded systems, particularly those containing reconfigurable devices such as field-programmable gate arrays.

Oskar Mencer (M'09) received the B.S. degree in computer engineering from The Technion, Haifa, Israel, in 1994, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1997 and 2000, respectively.

He founded MAXELER Technologies, in 2003, after three years as a member of Technical Staff in the Computing Sciences Research Center, Bell Labs. Since 2004, he has been a full-time member of the academic staff in the Department of Computing, Imperial College London, London, U.K., and leads the computer architecture research group there. His research interests include computer architecture, computer arithmetic, very large scale integration (VLSI) micro-architecture, VLSI computer-aided design (CAD), and reconfigurable (custom) computing. More specifically, he is interested in exploring application-specific representation of computation at the algorithm level, the architecture level, and the arithmetic level.

Can Özturan received the Ph.D. degree in computer science from Rensselaer Polytechnic Institue, Troy, NY, in 1995.

After working as a Post-Doctoral Staff Scientist with the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, he joined the Department of Computer Engineering, Boğaziçi University, Istanbul, Turkey, as a Faculty Member in 1996. His research interests are parallel processing, scientific computing, graph algorithms, and grid computing.

Günhan Dündar was born in Istanbul, Turkey, in 1969. He received the B.S. and M.S. degrees from Boğaziçi University, Istanbul, Turkey, in 1989 and 1991, respectively, and the Ph.D. degree from Rensselaer Polytechnic Institute, Troy, NY, in 1993, all in electrical engineering.

Since 1994, he has been with Department of Electrical and Electronics Eng. Boğaziçi University, Istanbul, Turkey, where he is currently a Professor. During 1994, he was with the Turkish Navy at the Naval Academy and, during 2003, he was with EPFL, Switzerland, both on leave from Boğaziçi University. He has authored or coauthored more than 100 publications in international journals and conferences. His research interests include analog IC design, electronic design automation, and neural networks.