FISH: Fast Instruction SynthHesis for Custom Processors

Kubilay Atasu, Member, IEEE, Wayne Luk, Fellow, IEEE, Oskar Mencer, Member, IEEE, Can Özturan, Member, IEEE

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, (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 largest enumeration runtime 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.

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.

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], [3], [4]. A dedicated link between the custom functional units and the base processor provides an efficient communication interface. Re-using a pre-verified, 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, Stretch S6000, and IBM’s PowerPC® 405.

Modern custom instruction processors comprise parallel, deeply pipelined custom functional units including state registers, local memories, 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 Figure 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], [7], [8], [9], [10], [11], [12], [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], [15], [16], [17], [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 prior work, the techniques presented in this work do not rely on enumeration of maximal independent sets or maximal cliques. This work extends [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, XI).

II. RELATED WORK

Heuristic clustering algorithms have long been used in automated custom instruction synthesis [20], [21], [22], [23], [24]. However, the attention gradually shifts towards techniques for producing optimal solutions, such as subgraph enumeration [6], [7], [8], [9], [10], [11], [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 \( O(N^{N_{in}+N_{out}+1}) \) [27], which grows exponentially with the number of inputs and outputs. A common property of the aforementioned enumeration techniques [6], [7], [8], [9], [10], [11], [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], [16], [17], [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 \( O(2^{NC}) \), where \( NC \) 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], [18]. As a result, [26] fails to identify optimal solutions on large DFGs, although the ILP models of [13], [15], [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.

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 com-
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. AN UPPER BOUND ON THE SEARCH SPACE SIZE

In this section 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:

$$\begin{align*}
    x'_i \land a_i \land d_i &= 0, & i \in I, \\
    a_j \land d_j &= 0, & j \in J.
\end{align*}$$

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

Proof: Assume that Eq. (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 Eq. (2). We are going to show that $S$ also satisfies Eq. (1).

Suppose that a node $v_i \in V - (V_s \cup V_f)$ violates Eq. (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 Eq. (2). We note that in a convex solution, there exist three possible choices for each $v_j \in V_f$:

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 Eq. (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 Eq. (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 Eq. (2).

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

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) $a_j = 0, d_j = 1$; (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 the remaining nodes in \( V - V_f \) can be safely included in \( S \) without violating Eq. (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 \( G \) is sufficient. \( \blacksquare \)

Figure 3 depicts an example DFG where nodes \( v_4 \) and \( v_5 \) are the forbidden nodes. Because there exist only 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 \( a_5 = 1, d_5 = 0 \)); (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.

Figure 4 shows the incompatibility graph generated by Pothineni’s algorithm [16] for the DFG of Figure 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.

**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 \( V_f \):

\[
\text{Anc}(i \in I) = \{ j \in J \mid \text{There exists a path from } v_j \text{ to } v_i \} \\
\text{Desc}(i \in I) = \{ j \in J \mid \text{There exists a path from } v_i \text{ to } v_j \}
\]

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:

\[
x_i = \begin{cases} 
1 & \text{if } \text{Anc}(i) = \emptyset \land \text{Desc}(i) = \emptyset, \\
\bigwedge_{j \in \text{Anc}(i)} d_j & \text{if } \text{Anc}(i) \neq \emptyset \land \text{Desc}(i) = \emptyset, \\
\bigwedge_{j \in \text{Desc}(i)} a_j & \text{if } \text{Anc}(i) = \emptyset \land \text{Desc}(i) \neq \emptyset, \\
\bigwedge_{j \in \text{Anc}(i)} d_j \land \bigwedge_{j \in \text{Desc}(i)} a_j & \text{otherwise}.
\end{cases}
\]

Equ. (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], Chapter 5.

**VI. A 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 \( 2^{|V_f|} \).

In this section, 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; (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 in the computation of the maximal convex subgraphs, as formulated by Eq. (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 \( \text{Anc}(i) = \text{Anc}(j) \) and \( \text{Desc}(i) = \text{Desc}(j) \), the nodes \( v_i \) and \( v_j \) can be clustered without affecting the result of enumeration because Eq. (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 \( V - V_f \), Eq. (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.

The result is the clustered graph \( G' = (V', E') \), where \( V'_f \subseteq V' \) 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 Eq. 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 Eq. 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 \( G' \) is sufficient. ■

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.

Figure 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 \( d_0 \) 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.

Figure 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 indices have their \( d_i \) 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 Eq. (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 the enumerated subgraphs, too.

Figure 7 shows the \( d_i \) combinations identified during the execution trace of our enumeration algorithm on the graph of Figure 5. We observe that only six out of 16 possible combinations had to be enumerated. In fact, the worst-case
scenario of $2^{\mid V''\mid}$ can occur only if all forbidden nodes are disconnected from each other. This is illustrated in Figure 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. An example, it is also possible to invert the topological 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 $v_i$; (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 Eq. 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 non-maximal subgraphs. Moreover, [28] describes an efficient way of detecting and discarding repeated subgraphs. Note that such a step is not necessary for the purposes of 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 the 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 the 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 set-up, $M(S)$ 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 final circuit to achieve a target 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 maximization of the reduction in the schedule length by moving the source level data flow subgraph $S$ from software to hardware:

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

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 bit-width optimization, arithmetic transformations, logic minimization, technology mapping, and register re-timing. 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:

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 post-processing 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:

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

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]:

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

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 $M_2(S)$:

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

### VIII. Overall Approach

The algorithms described in this work are integrated into the Trimanar compiler [38]. Figure 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 Trimanar 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 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 very similar to the iterative subgraph generation algorithm described in [18]. The most promising subgraph first approach 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 on 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 $F(S) \times M_3(S)$. 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. RUNTIME RESULTS

The memory access, branch, and division instructions are marked as forbidden instructions in our experiments. Our algorithms are applied on ten benchmarks from multimedia and cryptography domains [39], [40], including an optimized 32-bit implementation of AES (Advanced Encryption Standard) encryption and decryption [41], and a FIPS-compliant fully unrolled DES (Data Encryption Standard) implementation [42].

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 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 set-up 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 set-up 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 search time for AES encryption, AES decryption, DES, and djpeg benchmarks. In the case of the g721encode, g721decode, rawcaudio and rawaudio
benchmarks, the set-up 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], [43]. The enumeration results of this work, and of [19] can be derived from Table III by summing up the associated set-up 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 [44]) and a public domain solver (lp_solve [45]). To provide a comparison with [16], [26], we have implemented the Bron-Kerbosch algorithm [43], 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 Eq. 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 “Enum-CMB” in Table IV. Note that the results of this work for AES encryption and DES encryption 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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Basic Preprocessing</th>
<th>Partial Clustering</th>
<th>Enhanced Clustering</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Set-up</td>
<td>Enum-TO</td>
<td>Enum-AO</td>
</tr>
<tr>
<td>AES enc.</td>
<td>0.00087</td>
<td>79.498</td>
<td>70.479</td>
</tr>
<tr>
<td>AES dec.</td>
<td>0.00161</td>
<td>537.96</td>
<td>564.71</td>
</tr>
<tr>
<td>DES</td>
<td>0.00689</td>
<td>0.24333</td>
<td>0.22559</td>
</tr>
<tr>
<td>SHA (full)</td>
<td>0.00735</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IDEA</td>
<td>0.00007</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jpeg</td>
<td>0.00009</td>
<td>0.00025</td>
<td>0.00031</td>
</tr>
<tr>
<td>g721encode</td>
<td>0.00021</td>
<td>0.00005</td>
<td>0.00006</td>
</tr>
<tr>
<td>g721decode</td>
<td>0.00021</td>
<td>0.00009</td>
<td>0.00013</td>
</tr>
<tr>
<td>rawaudio</td>
<td>0.00004</td>
<td>0.00001</td>
<td>0.00001</td>
</tr>
<tr>
<td>rawaudio</td>
<td>0.00004</td>
<td>0.00001</td>
<td>0.00001</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Related Work</th>
<th>This work</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES enc.</td>
<td>21.550</td>
<td>25.209</td>
</tr>
<tr>
<td>AES dec.</td>
<td>120.57</td>
<td>134.05</td>
</tr>
<tr>
<td>SHA</td>
<td>0.00735</td>
<td>0.00735</td>
</tr>
<tr>
<td>IDEA</td>
<td>0.00007</td>
<td>0.00007</td>
</tr>
<tr>
<td>jpeg</td>
<td>0.00021</td>
<td>0.00021</td>
</tr>
<tr>
<td>g721encode</td>
<td>0.00026</td>
<td>0.00026</td>
</tr>
<tr>
<td>g721decode</td>
<td>0.00029</td>
<td>0.00029</td>
</tr>
<tr>
<td>rawaudio</td>
<td>0.00006</td>
<td>0.00006</td>
</tr>
<tr>
<td>rawaudio</td>
<td>0.00006</td>
<td>0.00006</td>
</tr>
</tbody>
</table>
(namely $G''$) that is composed of clusters of forbidden nodes only, i.e., a graph with $|V'|$ nodes. Our work derives the upper bound on the number of maximal convex subgraphs as $2^{|V'|}$. 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, Figure 4). It is well known that the number of maximal cliques of a graph with $|V' - V|_F$ nodes is bounded by $3^{|V' - V|_F / 3}$ [43]. 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 [44] 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.

Figure 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 $2 \cdot (31)^5 \approx 2M$ combinations, which takes around 2.8 sec to execute. On the other hand, CPLEX finds the optimal solution in only 0.056 sec, providing us with a second and more efficient solution alternative.

Table V provides additional comparisons between our enumeration algorithms and those described in [19], [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 rijndaell are around 10 sec, an order of magnitude larger than those we present under the [28] column. Such a slow-down is mainly due to the overhead of storing all 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 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 sec for the Enum column, taking into account all 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 [46]. 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 130nm 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.
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.

Figures 10 and 11 demonstrate the area and performance trade-offs for the DES, and g721encode 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 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).

Figure 12 shows the per-cent 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 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 rawaudio 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 rawaudio benchmarks around two percent difference is 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

Figure 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 Figure 13 implements the combinational logic between the memory access 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 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. 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 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 Figure 12.

Note that, although the custom instruction generated for DES example is highly re-usable and has a low area overhead, the custom instructions generated for SHA have a high area overhead and almost no re-use 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 re-usable subgraphs. Ideally, a combination of these two approaches
should be evaluated to identify the most promising area and performance trade-offs for custom instruction processors.

XII. Conclusions

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 data-flow graphs 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 runtime 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 post-processing step results in a 30 fold speed-up with respect to a single-issue base processor.

Current and future work includes (1) exploration of better estimators for critical path computation that take into account arithmetic optimizations, bit-width 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; (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 [47], integrating our design flow into synthesis tools for heterogeneous multiprocessor SoC devices [48, 49], and combining our algorithms with fast synthesis techniques [50, 51] to enable dynamic hardware/software partitioning and run-time reconfiguration.

XIII. Acknowledgements

The support of UK EPSRC and IBM Research is gratefully acknowledged. We thank Charlotte Bolliger, Anne-Marie Cromack, and Christoph Hagleiter from IBM Research - Zurich, and the anonymous reviewers for their valuable comments.

References

Kubilay Atasu (’S’03-M’08) received the B.Sc. degree in computer engineering from Boğaziçi University, Istanbul, Turkey, in 1999, 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 Bogazici 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.