CHIPS: Custom Hardware Instruction Processor Synthesis

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

***Abstract*—This paper describes an integer-linear-programming (ILP)-based system called Custom Hardware Instruction Processor Synthesis (CHIPS) that identifies custom instructions for critical code segments, given the available data bandwidth and transfer latencies between custom logic and a baseline processor with architecturally visible state registers. Our approach enables designers to optionally constrain the number of input and output operands for custom instructions. We describe a design flow to identify promising area, performance, and code-size tradeoffs. We study the effect of input/output constraints, register-file ports, and compiler transformations such as if-conversion. Our experiments show that, in most cases, the solutions with the highest performance are identified when the input/output constraints are removed. However, input/output constraints help our algorithms identify frequently used code segments, reducing the overall area overhead. Results for 11 benchmarks covering cryptography and multimedia are shown, with speed-ups between 1.7 and 6.6 times, code-size reductions between 6% and 72%, and area costs ranging between 12 and 256 adders for maximum speed-up. Our ILP-based approach scales well: benchmarks with basic blocks consisting of more than 1000 instructions can be optimally solved, most of the time within a few seconds.**

***Index Terms*—Application-specific instruction-set processors (ASIPs), custom instructions, customizable processors, extensible processors, integer linear programming (ILP), optimization algorithms.**

I. Introduction

EMBEDDED systems are dedicated to an application domain, produced in high volume, and built under strict cost constraints. The design space is often large, and many interacting components can be involved. Maximizing the performance while minimizing the transistor area and the power consumption are usually the main goals of the design. Designers carefully analyze the characteristics of the target application and fine-tune the implementation to achieve the best tradeoffs. General-purpose processors are often not flexible enough for adapting to the strict area, performance, and power-consumption requirements of embedded applications. Hence, a common approach in the design of embedded systems is to implement the control-dominated tasks on a general-purpose processor and the computation-intensive tasks on custom hardware, in application-specific integrated-circuit or field-programmable gate-array technology.

Application-specific instruction-set processors (ASIPs) provide a good compromise between general-purpose processors and custom-hardware designs. The traditional approach in developing ASIPs involves the design of a complete instruction-set architecture (ISA) for a given application. The processor and the compiler are synthesized based on a high-level ISA description. Target's CHESS compiler based on the nML [1] architecture description language (ADL) and Coware's LISATek based on LISA ADL [2] are among the commercial examples. A more recent approach assumes a preverified preoptimized base processor with a basic instruction set. The base processor is augmented with custom-hardware units that implement application-specific instructions. A dedicated link between custom units and the base processor provides an efficient communication interface. Reusing a preverified preoptimized base processor reduces the design complexity and the time to market. Several commercial examples exist, such as Tensilica Xtensa, Altera Nios II, Xilinx MicroBlaze, ARC 700, MIPS Pro Series, and ARM OptimoDE.

We believe that the availability of automated techniques for the synthesis of custom instructions has tremendous value in coping with the time-to-market pressure and in reducing the design cost of ASIPs. The automation effort is motivated by manual-design examples, such as in [3] and [4], which demonstrate the importance of identifying coarse grain and frequently used code segments. Tensilica's XPRES Compiler and CoWare's CORXpert are among the successful commercial tools offering automated solutions.

In this paper, we target customizable architectures similar to Tensilica Xtensa processor, where the data bandwidth between the base processor and the custom logic can be constrained by the available register-file ports. Our approach is also applicable to architectures where the data bandwidth is limited by dedicated data-transfer channels, such as the Fast Simplex Link channels of Xilinx MicroBlaze processor. We improve upon the

Manuscript received March 26, 2007; revised July 1, 2007 and August 22, 2007. This work was supported in part by U.K. EPSRC under Research Projects EP/C509625/1 and EP/C549481/1 and in part by Boğaziçi University under Research Project 05HA102D. The work of K. Atasu was supported in part by TUBITAK under the National Ph.D. Scholarship Program. This paper was recommended by Associate Editor G. E. Martin.

K. Atasu was with the Department of Computer Engineering, Boğaziçi University, 34342 Istanbul, Turkey. He is now with the Department of Computing, Imperial College London, SW7 2BZ London, U.K. (e-mail: atasu@boun.edu.tr; atasu@doc.ic.ac.uk).

C. Özturan is with the Department of Computer Engineering, Boğaziçi University, 34342 Istanbul, Turkey (e-mail: ozturaca@boun.edu.tr).

G. Dündar is with the Department of Electrical and Electronics Engineering, Boğaziçi University, 34342 Istanbul, Turkey (e-mail: dundar@boun.edu.tr).

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).

Digital Object Identifier 10.1109/TCAD.2008.915536

state-of-the-art algorithms by formulating and solving the key step of custom-instruction generation as a formal optimization problem. Our main contributions are as follows:

1) a novel custom-instruction generation technique, which optionally constrains the number of input and output operands for custom instructions and explicitly evaluates the data-transfer costs;
2) an integer-linear-programming (ILP)-based approach, which, unlike previous work [11]–[17] and [22]–[25], does not rely on heuristic clustering algorithms or input/output constraints for the reduction of the search space;
3) a demonstration that our solutions scale well: basic blocks with more than a thousand instructions can be optimally solved with or without input/output constraints;
4) an evaluation of the impact of area and register-file-port constraints on the execution cycle count and code size for various multimedia and cryptography benchmarks.

II. Background and Related Work

Automatic hardware/software partitioning is a key problem in the hardware/software codesign of embedded systems. The traditional approach assumes a processor and a coprocessor integrated through a bus interface [7]–[9]. The system is represented as a graph, where the graph nodes represent tasks or basic blocks, and the edges are weighted based on the amount of communication between the nodes. The hardware/software-partitioning problem under area and schedule-length constraints is formulated as an ILP problem in [9]. In [10], Arato shows that hardware/software partitioning under area and schedule-length constraints is NP-hard.

Custom-instruction processors are emerging as an effective solution in the hardware/software codesign of embedded systems. In the context of custom-instruction processors, hardware/software partitioning is done at the instruction-level granularity. Application basic blocks are transformed into data-flow graphs (DFGs), where the graph nodes represent instructions similar to those in assembly languages, and the edges represent data dependencies between the nodes. Profiling analysis identifies the most time-consuming basic blocks. Code transformations, such as loop unrolling and if-conversion, selectively eliminate control-flow dependencies and merge application basic blocks. Custom instructions provide efficient hardware implementations for frequently executed DFG subgraphs.

The partitioning of an application into base-processor instructions and custom instructions is done under certain constraints. First, there is a limited area available in the custom logic. Second, the data bandwidth between the base processor and the custom logic is still limited (see Fig. 1), and the data-transfer costs have to be explicitly evaluated. Next, only a limited number of input and output operands can be encoded in a fixed-length instruction word. Further restrictions on the structure of the custom instructions are needed to guarantee a feasible schedule for the instruction stream.

Automatic identification of custom instructions has remained an active area of research for more than a decade. The main-

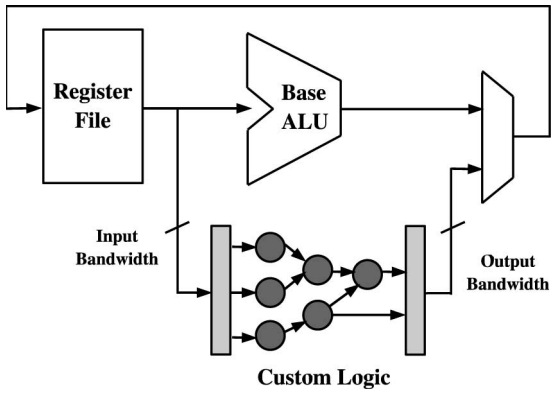

Fig. 1. Datapath of the custom-instruction processor: The data bandwidth may be limited by the available register-file ports or by the dedicated data-transfer channels.

stream approach divides the custom-instruction identification problem into the following two phases: 1) generation of a set of custom-instruction templates and 2) selection of the most profitable templates under area or schedule-length constraints. Most of the early research and some of the recent work [11]–[16] relied on incremental clustering of related DFG nodes in order to generate a set of custom-instruction templates. Alippi *et al.* [17] introduced the MaxMISO algorithm, which partitions a DFG into maximal-input single-output subgraphs in linear run-time. Binh *et al.* [18] proposed a branch-and-bound-based algorithm for the selection problem in order to minimize the area cost under schedule-length and power-consumption constraints. Sun *et al.* [19] imposed no explicit constraints on the number of input and output operands for custom instructions and formulated custom instruction selection as a mathematical-optimization problem.

Cheung *et al.* [20] generated custom-instruction templates based on exhaustive search. The exhaustive approach is not scalable, since the number of possible templates (DFG subgraphs) grows exponentially with the size of the DFGs. Atasu *et al.* [21] introduced constraints on the number of input and output operands for subgraphs and showed that application of a constraint-propagation technique could significantly reduce the exponential search space. Additionally, Atasu *et al.* proposed a greedy algorithm, which iteratively selects nonoverlapping DFG subgraphs having maximal speed-up potential based on a high level metric. The proposed technique is often limited to DFGs with a few hundred nodes, and the input/output constraints must be tight enough to reduce the exponential worst case time complexity. The work of Atasu *et al.* showed that clustering based approaches (e.g., [14]) or single output operand restriction, (e.g., [17]) could severely reduce the achievable speed-up using custom instructions.

Cong *et al.* [23] proposed a dynamic-programming-based algorithm, which enumerates single-output subgraphs with a given number of inputs. Yu and Mitra [24] show that subgraph enumeration under input and output constraints can be done much faster if the additional connectivity constraint is imposed on the subgraphs. Pozzi *et al.* [22] further optimized the algorithm of Atasu [21] and show that enumerating connected subgraphs only can substantially reduce the speed-up potential.

In [25], Biswas *et al.* proposed an extension to the Kernighan–Lin heuristic, which is, again, based on input and output constraints. This approach does not evaluate all feasible subgraphs. Therefore, an optimal solution is not guaranteed. In [26], Bonzini and Pozzi derived a polynomial bound on the number of feasible subgraphs if the number of inputs and outputs for the subgraphs are fixed. However, the complexity grows exponentially as the input/output constraints are relaxed. Performance of the proposed algorithm is reported to be similar to the performance of the enumeration algorithm described in [21] and [22].

Leupers and Marwedel [27] described a code-selection technique for irregular datapaths with complex instructions. Clark *et al.* [16] formulated the problem of matching a library of custom-instruction templates with application DFGs as a subgraph isomorphism problem. Peymandoust *et al.* [28] proposed a polynomial-manipulation-based technique for the matching problem. Cong *et al.* [23] made use of isomorphism testing to determine whether enumerated DFG subgraphs are structurally equivalent. Cheung *et al.* [29] employed model equivalence checking to verify whether generated subgraphs are functionally equivalent to a predesigned set of library components.

Clark *et al.* [30] proposed a reconfigurable array of functional units tightly coupled with a general-purpose processor that can accelerate data-flow subgraphs. A microarchitectural interface and a compilation framework allow a transparent instruction-set customization. Dimond *et al.* [31] introduced a customizable soft processor with multithreading support. Techniques that can exploit structural similarities across custom instructions for area-efficient synthesis are described in [32]–[34].

The speed-up obtainable by custom instructions is limited by the available data bandwidth between the base processor and custom logic. Extending the core register file to support additional read and write ports improves the data bandwidth. However, additional ports result in increased register-file size, power consumption, and cycle time. The Tensilica Xtensa [5] uses custom state registers to explicitly move additional input and output operands between the base processor and custom units. Binding of base-processor registers to custom state registers at compile time reduces the amount of data transfers. Use of shadow registers [35] and exploitation of forwarding paths of the base processor [36] can improve the data bandwidth.

Another potential complexity in the design of custom-instruction processors is the difficulty of encoding multiple input and output operands within a fixed-length instruction word. Issuing explicit data-transfer instructions to and from custom state registers is a way of encoding the additional input and output operands. An orthogonal approach proposed by Lee *et al.* [37] restricts the input and output operands for custom instructions to a subset of the base-processor registers. Tensilica Xtensa LX processors [6] introduce flexible instruction-encoding support for multioperand instructions, known as FLIX, in order to address the encoding problem.

In this paper, we extend the material described in [53] and [54]. We assume a baseline machine with architecturally visible state registers and support for explicit data-transfer instructions. We do not constrain the number of input and output operands for custom instructions. However, we explicitly account for the data-transfer cycles between the base processor and the custom logic if the number of inputs or outputs exceed the available register-file ports. We explore compiler transformations, such as if-conversion [45] and loop unrolling, that can eliminate control dependencies, and we apply our algorithms on predicated basic blocks.

TABLE I
COMPARISON WITH SOME STATE-OF-THE-ART TECHNIQUES

	[21], [22]	[23]	[24]	Our work
Controllability of inputs	√	√	√	√
Controllability of outputs	√		√	√
Support for disconnectedness	√			√
Removal of I/O constraints				√

Today's increasingly advanced ILP solvers, such as CPLEX [42], are often able to efficiently solve problems with thousands of integer variables and tens of thousands of linear constraints. To take advantage of this widely used technology, we formulate the custom-instruction identification problem as an ILP in Section IV. We show that the number of integer variables and the number of linear constraints used in our ILP formulation grow only linearly with the size of the problem. In Section V, we integrate our ILP-based solution into an iterative algorithm, used also in [21] and [22], which reduces the search space based on a most profitable subgraph first approach.

In Section VII-C, we provide two real-life examples on which the algorithms of [21] and [22] miss optimal solutions since the subgraph-enumeration algorithm fails to complete within 24 h when the input/output constraints are loose or removed. Optimizing the same high-level metric, our ILP approach locates optimal solutions in only a few seconds in both cases. We provide a detailed run-time comparison in Section VII-H. Table I compares our approach with some state-of-the-art techniques in terms of the supported features.

III. OVERALL APPROACH

Fig. 2 shows our tool chain called Custom Hardware Instruction Processor Synthesis (CHIPS). We use the Trimaran [41] framework to generate the control- and data-flow information and to achieve basic-block level profiling of a given application. Specifically, we work with Elcor, the back-end of Trimaran. We operate on the Elcor intermediate representation after the application of classical compiler optimizations. We implement an if-conversion pass that selectively eliminates control-flow dependencies due to conditional branches. Immediately prior to register allocation, we apply our algorithms to identify the custom instructions.

Section IV describes a scalable ILP formulation that identifies the most promising data-flow subgraphs as custom-instructions templates. Our ILP formulation guarantees a feasible schedule for the generated templates. Furthermore, our ILP formulation explicitly calculates the data-transfer costs and critical-path delays and identifies the templates that reduce the schedule length most. We use the industry-standard CPLEX Mixed Integer Optimizer [42] within our algorithms to solve the ILP problems we generate.

We iteratively generate a set of custom-instruction templates based on ILP as described in Section V. We group structurally

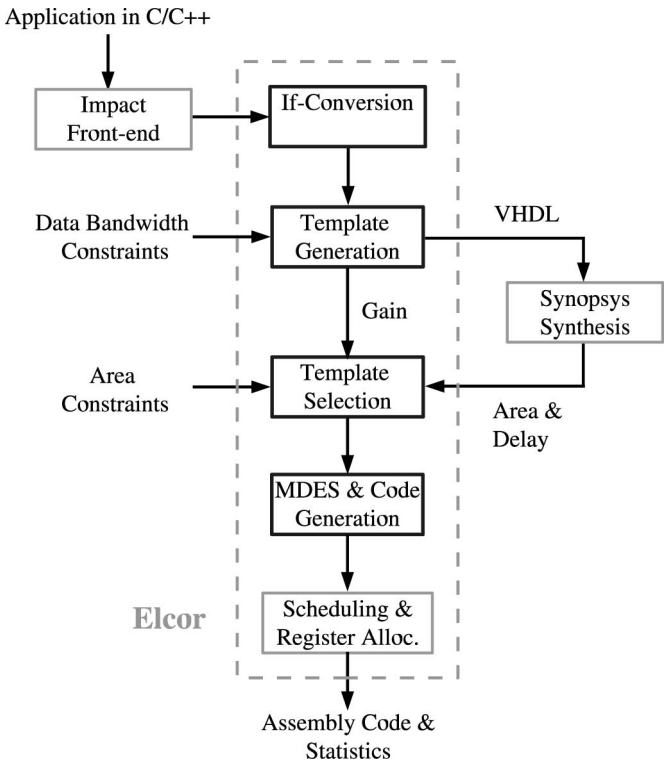

Fig. 2. CHIPS: We integrate our algorithms into Trimaran [41]. Starting with C/C++ code, we automatically generate behavioral descriptions of custom instructions in VHDL, a high-level MDES, and assembly code.

equivalent templates within isomorphism classes as custom-instruction candidates. We generate the behavioral descriptions of custom-instruction candidates in VHDL, and we produce area and delay estimates using Synopsys Design Compiler. We select the most profitable candidates under area constraints based on a Knapsack model described in Section VI.

Once the most profitable custom-instruction candidates are selected under area constraints, we automatically generate high-level machine descriptions (MDES) [43] supporting the selected candidates. After the generation of MDES, we insert the custom instructions in the code and replace the matching subgraphs. Finally, we apply standard Trimaran scheduling and register-allocation passes, and we produce the assembly code and scheduling statistics.

IV. Definitions and Problem Formulation

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

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

We associate with every graph node v_i a software latency s_i and a hardware latency h_i, where s_i is an integer and h_i is real. Software latencies give the time in clock cycles that it takes to execute the operations on the pipeline of the base processor. Hardware latencies are given in a unit of clock cycles. These values are calculated by synthesizing the operators and normalizing their delays to a target cycle time.

Given RF_{in} read ports and RF_{out} write ports supported by the core register file, we assume that first RF_{in} input operands can be read and that first RF_{out} output operands can be written back free of cost by the custom instructions. We assume that the cost of transferring additional RF_{in} inputs is c_1 cycles, and the cost of transferring additional RF_{out} outputs is c_2 cycles.

We use the following indexes in our formulations:

I_1 Indexes for nodes $v_i^{\text{in}} \in V^{\text{in}}$;
I_2 Indexes for nodes $v_i \in V$;
I_3 Indexes for nodes $v_i \in V^{\text{out}}$;
I_4 Indexes for nodes $v_i \in V/V^{\text{out}}$.

We define the set of immediate successors of nodes in V^{in} as follows:

$$Succ(i \in I_1) = \left\{ j \in I_2 | \exists\, e \in E^{\text{in}} : e = (v_i^{\text{in}}, v_j) \right\}.$$

We define the set of immediate successors and the set of immediate predecessors of nodes in V as follows:

$$Succ(i \in I_2) = \{ j \in I_2 | \exists\, e \in E : e = (v_i, v_j) \}$$
$$Pred(i \in I_2) = \{ j \in I_2 | \exists\, e \in E : e = (v_j, v_i) \}.$$

A. Calculation of Input Data Transfers

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

$$N_{\text{in}} = \sum_{i \in I_1} \left(\bigvee_{j \in Succ(i)} x_j \right) + \sum_{i \in I_2} \left(x_i' \wedge \left(\bigvee_{j \in Succ(i)} x_j \right) \right). \quad (1)$$

We calculate the number of additional data transfers from the core register file to the custom logic as DT_{in}

$$DT_{\text{in}} \geq N_{\text{in}}/RF_{\text{in}} - 1, \qquad DT_{\text{in}} \in Z^+ \cup \{0\}. \quad (2)$$

A constraint on the maximum number of input operands can be imposed as follows:

$$N_{\text{in}} \leq \text{MAX}_{\text{in}}. \quad (3)$$

B. Calculation of Output Data Transfers

We introduce an integer decision variable N_{out} to compute the number of output operands of a template T. A node

$v_i \in V^{\text{out}}$, defining an output operand of the basic block, defines an output operand of T if it is in T. A node $v_i \in V/V^{\text{out}}$ defines an output operand of T if it is in T, and it has at least one immediate successor not in T

$$N_{\text{out}} = \sum_{i \in I_3} x_i + \sum_{i \in I_4} \left(x_i \wedge \left(\bigvee_{j \in Succ(i)} x'_j \right) \right). \quad (4)$$

We calculate the number of additional data transfers from the custom logic to the core register file as DT_{out}

$$DT_{\text{out}} \geq N_{\text{out}}/RF_{\text{out}} - 1, \qquad DT_{\text{out}} \in Z^+ \cup \{0\}. \quad (5)$$

A constraint on the maximum number of output operands can be imposed as follows:

$$N_{\text{out}} \leq \text{MAX}_{\text{out}}. \quad (6)$$

C. Convexity Constraint

For each node $v_i \in V$, we introduce two new decision variables A_i and D_i. A_i represents whether v_i has an ancestor in $T(A_i = 1)$ or not $(A_i = 0)$. Similarly, D_i represents whether v_i has a descendant in $T(D_i = 1)$ or not $(D_i = 0)$.

A node has an ancestor in T if it has at least one immediate predecessor that is already in T or having an ancestor in T

$$A_i = \begin{cases} 0 & \text{if } Pred(i) = \emptyset \\ \left(\bigvee_{j \in Pred(i)} (x_j \vee A_j) \right) & \text{otherwise.} \end{cases} \quad (7)$$

A node has a descendant in T if it has at least one immediate successor that is already in T or having a descendant in T

$$D_i = \begin{cases} 0 & \text{if } Succ(i) = \emptyset \\ \left(\bigvee_{j \in Succ(i)} (x_j \vee D_j) \right) & \text{otherwise.} \end{cases} \quad (8)$$

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

$$x'_i \wedge A_i \wedge D_i = 0, \qquad i \in I_2. \quad (9)$$

D. Critical-Path Calculation

We estimate the execution latency of a template T on the custom logic by quantizing its critical-path length. We calculate the critical-path length by applying an as-soon-as-possible scheduling without resource constraints.

For each node $v_i \in V$, we introduce a real decision variable $l_i \in R$, which represents the time in which the result of v_i becomes available when T is executed on custom logic, assuming all of its input operands are available at time zero

$$\begin{aligned} l_i &\geq h_i x_i && \text{if } Pred(i) = \emptyset \\ l_i &\geq l_j + h_i x_i, && j \in Pred(i). \end{aligned} \quad (10)$$

The largest l_i value gives us the critical-path length of T. We introduce an integer decision variable $L(T) \in Z^+$ to quantize the critical-path length

$$L(T) \geq l_i, \qquad i \in I_2. \quad (11)$$

E. Objective

Our objective is to maximize the decrease in the schedule length by moving template T from software to the custom logic. We estimate the software cost of T as the sum of the software latencies of the instructions contained in T. When the template is executed on the custom logic, the number of cycles required to transfer its input and output operands from and to the core register file are $c_1 DT_{\text{in}}$ and $c_2 DT_{\text{out}}$, respectively. $L(T)$ provides the estimated execution latency of the template on the custom logic once all of its inputs are ready. The objective function of the ILP is defined as follows:

$$Z(T) = \max \sum_{i \in I_2} (s_i x_i) - c_1 DT_{\text{in}} - c_2 DT_{\text{out}} - L(T). \quad (12)$$

F. Scalability of the Model

Our ILP model scales linearly with the size of the problem. The overall problem is represented using $O(|V \cup V^{\text{in}}|)$ integer decision variables and $O(|E \cup E^{in}|)$ linear constraints.

V. Template Generation

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

Providing a good upper bound on the value of the objective function can greatly enhance the performance of the ILP solver without affecting the optimality of the solution. ILP solvers rely on well-known optimization techniques such as branch-and-bound and branch-and-cut for efficiently exploring the search space. These techniques build a search tree, where the nodes represent subproblems of the original problem. Given a good upper bound on the objective value, the number of branches and the size of the search tree can be significantly reduced. In our case, we observed that the relaxation of the convexity constraint simplifies the problem and allows us to obtain good upper bounds on the objective value of the unrelaxed problem within a short time. Moreover, given the iterative nature of the template-generation algorithm, the objective value of the previous iteration provides a second and sometimes tighter upper bound.

A formal description of our approach is given in Fig. 3. We first solve the relaxed problem, where the convexity constraint is not imposed on the templates. If the identified template is convex, we add it to our template pool. Otherwise, the solution identified provides an upper bound on the objective value of the unrelaxed problem. We solve the problem once more with the convexity constraint imposed using the improved upper bound. Initially, the upper bound is set to the value of the maximum

```
PROCEDURE TEMPLATE_GENERATION
Given 𝒢 : The set of application basic blocks
Generate 𝒯 : The set of custom instruction templates
G : Current basic block
T : Current custom instruction template
Objective : Current value of the objective function
Upper_Bound : Current upper bound on the objective value
BEGIN
    𝒯 = ∅;
    FOR G IN 𝒢 DO
        Objective = MAX_INT;
        Upper_Bound = MAX_INT;
        WHILE (Objective > 0) DO
            Generate the relaxed ILP problem for G;
            Impose (Objective ≤ Upper_Bound);
            Solve ILP, calculate Objective and extract T;
            Upper_Bound = MIN(Objective, Upper_Bound) ;
            IF (T is NOT convex)
                Add the convexity constraint to the ILP;
                Impose (Objective ≤ Upper_Bound);
                Solve ILP, calculate Objective and extract T;
                Upper_Bound = MIN(Objective, Upper_Bound) ;
            END IF
            IF (Objective > 0)
                𝒯 = 𝒯 ∪ {T};
                Collapse T into a single node in G;
                Mark the new node as invalid;
            END IF
        END WHILE
    END FOR
END
```

Fig. 3. Template-generation algorithm iteratively solves a set of ILP problems. Providing a good upper bound on the objective value can dramatically reduce the solution time.

integer (MAX_INT). As the iterations proceed, the DFG gets smaller, the upper bound gets tighter, and these factors usually decrease the solution time.

The objective of the iterative template-generation algorithm is to generate custom-instruction templates covering application DFGs as much as possible while avoiding the exponential computational complexity of the subgraph-enumeration techniques. Not allowing overlapping between templates guarantees that the number of iterations will be $O(N_{\text{tot}})$, where N_{tot} represents the total number of instructions in an application. In practice, the number of iterations is much smaller than N_{tot} as the templates we generate are often coarse grain.

At each iteration, we choose the template that provides the highest objective value (i.e., the most profitable subgraph). Although heuristic in nature, the iterative approach results in reasonably good code coverage. In [38], Clark *et al.* combined the subgraph-enumeration algorithm of [21] with a unate-covering-based code-selection approach. The improvement in speed-up over the iterative approach is reported as 1% only.

VI. Template Selection

Once the template generation is done, we calculate the isomorphism classes using the nauty package [40]. We assume that the set of generated templates $\mathcal{T}$ is partitioned into N_G distinct isomorphism classes

$$\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2 \cup \cdots \cup \mathcal{T}_{N_G}. \tag{13}$$

An isomorphism class defines a custom-instruction candidate that can implement all the templates included in that class. Once isomorphism classes are formed, we generate behavioral descriptions of the custom-instruction candidates in VHDL. We apply high-level synthesis, and we associate an area estimate $A(\mathcal{T}_i)$ and a normalized critical-path estimate $D(\mathcal{T}_i)$ with each custom-instruction candidate $\mathcal{T}_i$.

The value of the objective function $Z(T)$ described in Section IV-E provides an initial estimation of the reduction in the schedule length by a single execution of the template $T \in \mathcal{T}_i$ on the custom logic. We replace the estimated critical-path length $L(T)$ with the more accurate result $D(\mathcal{T}_i)$ we obtain from high-level synthesis in order to generate a refined estimation of the reduction in the schedule length for the custom-instruction candidate $\mathcal{T}_i$

$$Z(\mathcal{T}_i) = Z(T) + L(T) - D(\mathcal{T}_i). \tag{14}$$

Given that a template $T \in \mathcal{T}_i$ is executed $F(T)$ times by a typical execution of the application, the total number of executions of a custom-instruction candidate $\mathcal{T}_i$ is calculated as follows:

$$F(\mathcal{T}_i) = \sum_{T \in \mathcal{T}_i} F(T), \qquad i \in \{1, \ldots, N_G\}. \tag{15}$$

The overall reduction in the schedule length of the application by implementing $\mathcal{T}_i$ as a custom instruction is estimated as follows:

$$G(\mathcal{T}_i) = Z(\mathcal{T}_i) * F(\mathcal{T}_i). \tag{16}$$

We formulate the problem of selecting the most profitable custom-instruction candidates under an area constraint A_{MAX} as a Knapsack problem and solve it using ILP solvers

$$\begin{aligned} &\max \sum_{i \in \{1,\ldots,N_G\}} G(\mathcal{T}_i) y_i \\ &\text{s.t.} \sum_{i \in \{1,\ldots,N_G\}} A(\mathcal{T}_i) y_i \le A_{\text{MAX}} \\ &\qquad y_i \in \{0,1\}, \qquad i \in \{1, \ldots, N_G\} \end{aligned} \tag{17}$$

where the binary decision variable y_i represents whether the candidate $\mathcal{T}_i$ is selected ($y_i = 1$) or not ($y_i = 0$).

Naturally, the number of custom-instruction candidates is smaller than the number of custom-instruction templates we generate. Hence, the number of integer decision variables used in the Knapsack formulation is $O(N_{\text{tot}})$ and, in practice, much smaller than N_{tot}. This allows us to solve the Knapsack problem optimally in all practical cases.

Our approach does not guarantee a globally optimal solution. A truly optimal solution could be possible by enumerating all possible subgraphs within the application DFGs. However, this approach is not computationally feasible, since the number of possible subgraphs grows exponentially with the size of the DFGs. In [39], Yu and Mitra enumerate only connected subgraphs having up to four input and two output operands and do not allow overlapping between selected subgraphs. Although the search space is significantly reduced by these restrictions, it

is reported that optimal selection using ILP solvers occasionally does not complete within 24 h.

VII. Experiments and Results

A. Experiment Setup

We evaluate our technique by using Trimaran scheduling statistics to estimate the execution cycles and by using Synopsys synthesis to estimate the area and delay for custom instructions.

1) MDES and Code Generation: We define a single-issue baseline machine with predication support including 32 32-b general-purpose registers and 32 1-b predicate registers. Our baseline machine implements the HPL-PD architecture [44] based on a high-level MDES model [43]. MDES requires specifications of the operation formats, resource usages, scheduling alternatives, execution latencies, operand read and write latencies, and reservation table entries for the instructions supported by the architecture. We automatically generate the MDES entries for the custom instructions identified by our algorithms. We implement custom-instruction replacement using a technique similar to the one described by Clark *et al.* in [16]. We apply standard Trimaran scheduling and register-allocation passes, and we produce the assembly code and scheduling statistics.

We assume two-cycle software latencies for integer multiplication instructions and single-cycle software latencies for the rest of the integer operations. We do not allow division operations to be part of custom instructions due to their high latency and area overhead. We exclude support for memory-access instructions as part of custom instructions as well, in order to avoid nondeterministic latencies due to the memory system and the necessary control circuitry. We assume single-cycle data-transfer latencies between general-purpose registers and custom units ($c_1 = c_2 = 1$). We assume single-cycle copy and update operations for transferring the predicate register-file contents to and from the custom logic. We assume that given RF_{in} read ports and RF_{out} write ports supported by the register file, RF_{in} input operands and RF_{out} output operands can be encoded within a single-instruction word.

2) Synopsys Synthesis: We calculate the hardware latencies of various arithmetic and logic operations (i.e., h_i values described in Section IV) by synthesizing on United Microelectronics Corporation's (UMC's) 130-nm standard cell library using Synopsys Design Compiler and normalizing to the delay of a 32-b ripple carry adder (RCA). Table II shows the relative latency and area coefficients for some selected operators. Once our algorithms identify the custom-instruction candidates, we automatically generate their VHDL descriptions and synthesize on the same library. If the critical-path delay of a candidate is larger than the delay of a 32-b RCA, it is pipelined to ensure a fixed clock frequency.

3) Benchmarks: We have applied our algorithms on a number of cryptography and media benchmarks. We have used highly optimized 32-b implementations of Advanced Encryption Standard (AES) encryption and decryption described in [46], a FIPS-46-3 compliant fully unrolled Data Encryption Standard (DES) implementation [47], a loop-based and a fully unrolled secure-hash-algorithm (SHA) implementation from MiBench [48] and several other benchmarks from MediaBench [49].

TABLE II
Relative Latency and Area Coefficients for Various Operators Based on Synthesis Results on 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

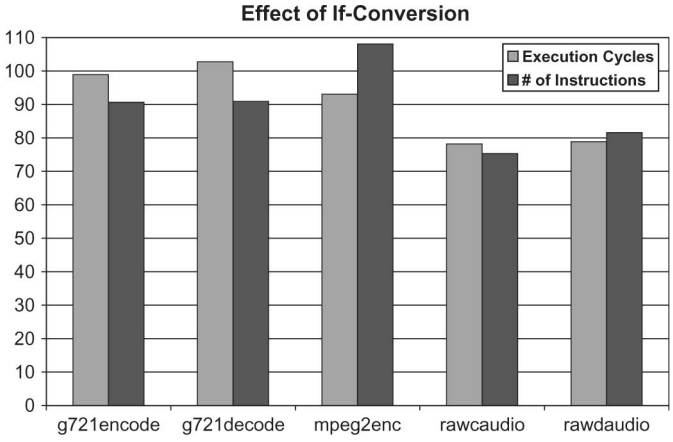

Fig. 4. We apply an if-conversion pass before identifying custom instructions. This reduces the number of execution cycles and the code size in most of the cases.

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

B. If-Conversion Results

We implement an if-conversion pass to selectively eliminate the control-flow dependencies. This improves the scope of our algorithms and enables us to identify coarser grain custom instructions. We apply if-conversion only on the most time consuming functions of the application. We partition control-flow graphs into maximal single-entry single-exit regions and convert each such region into a predicated basic block.

The results of our if-conversion pass on five MediaBench benchmarks are shown in Fig. 4. We observe that the number of execution cycles and the number of instructions in the code are reduced in most of the cases, although this is not our main objective. The remaining benchmarks are only marginally affected, particularly because they already consist of large basic blocks and contain few control-flow changes.

Table III shows the total number of basic blocks, the total number of instructions, and the number of instructions within the largest basic block for each benchmark. This information is collected after the application of if-conversion and considers basic blocks with positive execution frequencies only.

TABLE III
INFORMATION ON BENCHMARKS: BB REPRESENTS BASIC BLOCK

Benchmark	# of BBs	# of Instrs	Largest BB
AES encryption	27	735	317
AES decryption	28	1011	501
DES	45	1235	822
SHA (loop)	38	302	24
SHA (fully unrolled)	30	1339	1155
IDEA	65	595	96
djpeg	957	5503	92
g721encode	85	892	131
g721decode	79	864	131
mpeg2enc	147	2861	568
rawcaudio	13	119	54
rawdaudio	11	102	45

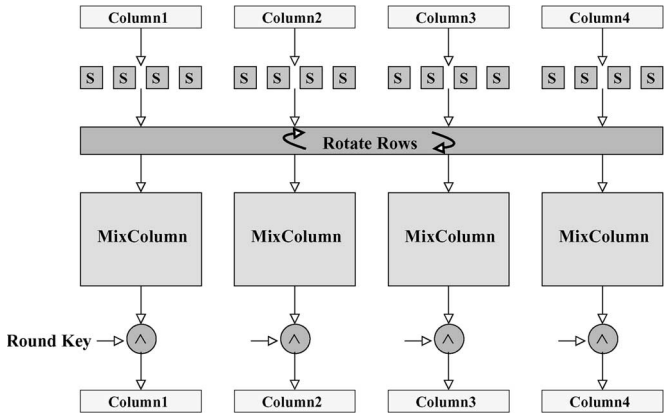

Fig. 5. AES round transformation. Given an input constraint of four and an output constraint of four, our algorithms successfully identify the four parallel MixColumn Transformations within the round transformation as the most promising custom-instruction candidate. None of the algorithms described in [22] are able to identify this solution.

C. Examples of Custom Instructions

Our first example is the AES. The core of the AES encryption is the AES round transformation (see Fig. 5), which operates on a 128-b state. The state is often stored in four 32-b registers called as the columns. The most compute-intensive part of AES encryption is the MixColumn transformation that is applied separately on each column. The AES implementation we use unrolls two round transformations within a loop, resulting in the largest basic block of the application. A second basic block incorporates an additional round transformation, resulting in a total number of 12 MixColumn transformations in the code.

Given an input constraint of one and an output constraint of one, our algorithms successfully identify all 12 instances of the MixColumn transformation in the code as the most promising custom instruction for the application. Given an input constraint of two and an output constraint of two, our algorithms successfully identify two parallel MixColumn transformations within a round transformation as the most promising custom instruction, finding all six instances in the code. Given an input constraint of four and an output constraint of four, our algorithms successfully identify the four parallel MixColumn transformations within a round transformation, and all three instances of the four-input four-output custom instruction are matched. In all cases, our algorithms identify optimal solutions

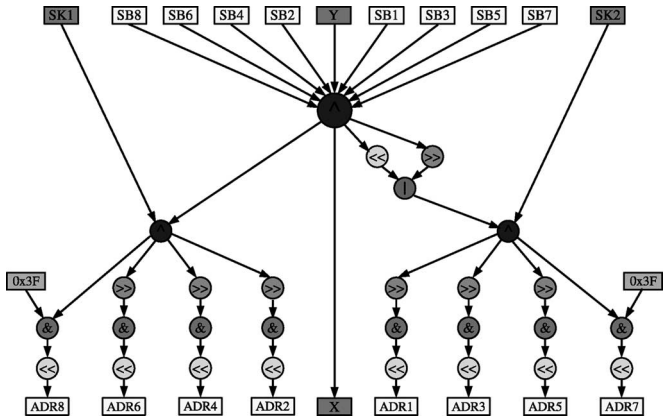

Fig. 6. Optimal custom instruction implementing the DES rounds. Eight of the inputs (SBs) are substitution table entries and eight of the outputs are the addresses of the substitution table entries that should be fetched for the next round. SK1 and SK2 contain the round key. Y represents the second half of the current state, and the first half of the state for the next round is generated in X. Fifteen instances of the same instruction are automatically identified from the C code.

within a few seconds. On the other hand, the subgraph-enumeration algorithm of [22] fails to complete for a four-output constraint, and none of the approximate algorithms described in [22] are able to identify four MixColumn transformations in parallel.

The synthesis results show that the critical-path delay of the MixColumn transformation is around one-fourth of a 32-b RCA, and its area cost is less than the area cost of two 32-b RCAs. The transformation can be implemented as a single-cycle instruction. In fact, it is the most likely choice for a manual designer as a custom instruction, as shown by Seng in [4]. Given a register file with two read ports and two write ports, we could as well implement two parallel MixColumn transformations as a single-cycle instruction. Obviously, this solution would incur two times more area overhead. Given a register file with four read ports and four write ports, we could even implement four parallel MixColumn transformations as a single-cycle instruction depending on our area budget.

In Fig. 6, we show the most promising custom instruction our algorithms automatically identify from the DES C code when no constraints are imposed on the number of input and output operands. An analysis reveals that the custom instruction implements the complete data processing within the round transformations of DES. It has 11 inputs and 9 outputs, 15 instances are automatically matched in the C code. To our knowledge, no other automated technique has been able to achieve a similar result. Subgraph-enumeration algorithms, such as in [21], [22], and [24], are impracticable when the input/output constraints are removed or loose and fail to identify custom instructions such as the one shown in Fig. 6.

An analysis shows that eight of the inputs of the custom instruction of Fig. 6 are static lookup-table entries (SBs), and eight of the outputs (ADRs) contain addresses of the lookup-table entries that should be fetched for the next round. Two of the inputs (SK1, SK2) contain the DES round key, the input Y and the output X represents the DES encryption state. The custom instruction implements 35 base-processor instructions, which are mostly bitwise operations. The synthesis results show

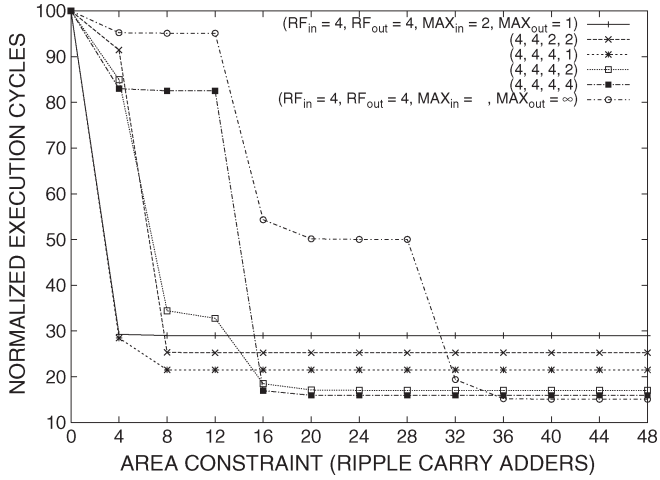

Fig. 7. AES decryption: Percent reduction in the execution cycles. Register file supports four read ports and four write ports (i.e., $RF_{\rm in} = 4$, $RF_{\rm out} = 4$). An input constraint of $\rm MAX_{in}$ and an output constraint of $\rm MAX_{out}$ can be imposed on the custom instructions, or these constraints can be removed (i.e., $\rm MAX_{in} = \infty$, $\rm MAX_{out} = \infty$).

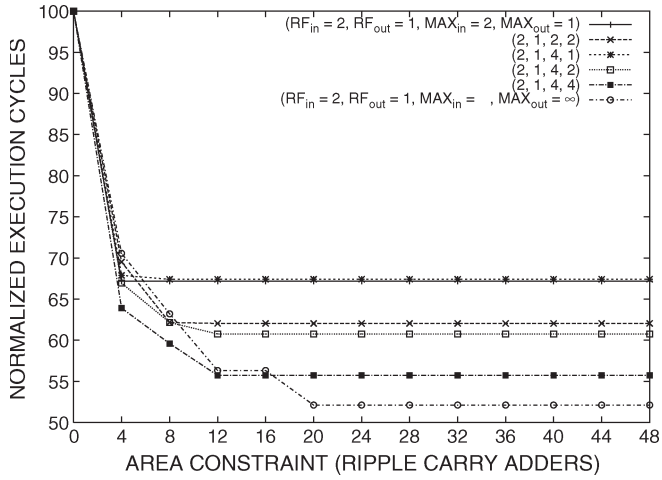

Fig. 8. DES: Percent reduction in the execution cycles. Register file supports two read ports and one write port (i.e., $RF_{\rm in} = 2$, $RF_{\rm out} = 1$). An input constraint of $\rm MAX_{in}$ and an output constraint of $\rm MAX_{out}$ can be imposed on the custom instructions, or these constraints can be removed (i.e., $\rm MAX_{in} = \infty$, $\rm MAX_{out} = \infty$).

that the critical path of the custom instruction is around one-eight of the critical path of a 32-b RCA. Hence, the custom instruction can be executed within a single cycle. However, as the register file has a limited number of read and write ports, we need additional data-transfer cycles to transfer the input and output operands between the core register file and the custom units. In practice, the granularity of the custom instruction is coarse enough to make it profitable despite the data-transfer overhead, and this overhead is explicitly calculated by our algorithms.

D. Effect of Input/Output Constraints

In this paper, we use input/output constraints to control the granularity of the custom instructions and to capture structural similarities within an application. Our motivation is that applications often contain repeated code segments that can be characterized by the number of input and output operands. When the input/output constraints are tight, we are more likely to identify fine-grain custom instructions. As we demonstrate in Section VII-G, fine-grain custom instructions often have more reuse potential. Relaxation of the constraints results in coarser grain custom instructions (i.e., larger data-flow subgraphs). Coarse-grain instructions are likely to provide higher speed-up, although at the expense of increased area.

In Figs. 7 and 8, we analyze the effect of different input and output constraints (i.e., $\rm MAX_{in}$, $\rm MAX_{out}$) on the speed-up potentials of custom instructions. For each benchmark, we scale the initial cycle count down to 100, and we plot the percent decrease in the cycle count by introducing custom instructions for a range of area constraints (up to 48 RCAs). At the end of this analysis, we locate the Pareto optimal points (i.e., input/output combinations) that maximize the cycle-count reduction at each area constraint.

In Fig. 7, we assume a register file with four read ports and four write ports, and we explore the achievable speed-up for AES decryption. The main difference between AES decryption and AES encryption is the InvMixColumn transformations that replace MixColumn transformations in the round transformation. The area cost of the InvMixColumn transformation is around the area cost of four RCAs. Fig. 7 shows that, at an area constraint of four adders, the Pareto optimal solution is obtained using four-input one-output custom instructions. On the other hand, at an area constraint of 16 adders, four-input four-output custom instructions provide the Pareto optimal solution. This solution implements four InvMixColumn transformations in parallel as a single-cycle instruction. We observe that removing the input/output constraints completely improves the performance slightly until an area constraint of 40 adders.

In Fig. 8, we assume a register file with two read ports and single write port, and we explore the achievable speed-up for DES. We observe that, when the area budget is below 16 adders, Pareto optimal solutions are generated by four-input four-output custom instructions. However, we obtain the highest reduction in the execution cycles when the input/output constraints are removed, at an area cost of 20 adders.

E. Effect of Register-File Ports

In Figs. 9 and 10, we demonstrate the improvement in performance using additional register-file ports. We scale the initial cycle count down to 100, and we plot the percent reduction in the execution cycles for a range of area constraints. For each $(RF_{\rm in}, RF_{\rm out})$ combination, we explore the following six different $(\rm MAX_{in}, MAX_{out})$ combinations: (2, 1), (2, 2), (4, 1), (4, 2), (4, 4), and (∞, ∞). At each area constraint, we choose the Pareto optimal solution given by one of the $(\rm MAX_{in}, MAX_{out})$ combinations. A monotonic decrease in the execution cycles with the increasing number of register-file ports is clearly shown in Figs. 9 and 10. We observe that a register file with two read ports and two write ports is often more beneficial than a register file with four read ports and a single write port. Additionally, a register file with four read ports and two write ports generates favorable design points.

dator

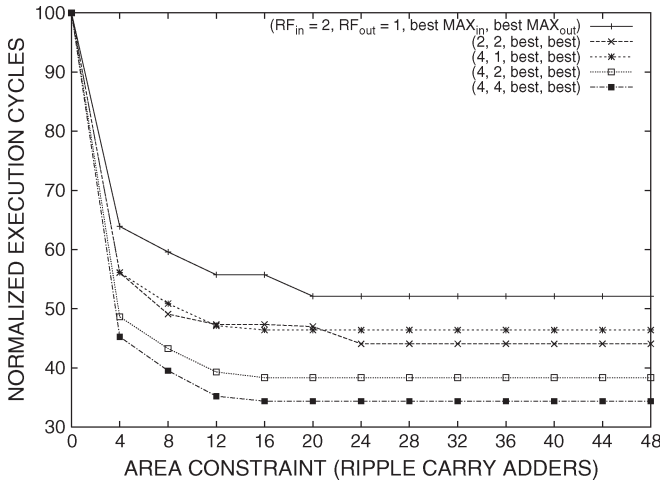

Fig. 9. DES: Effect of increasing the number of register-file ports (i.e., RF_{in} and RF_{out}) on the performance. At each area constraint, we choose the best MAX_{in}, MAX_{out} combination that minimizes the execution time.

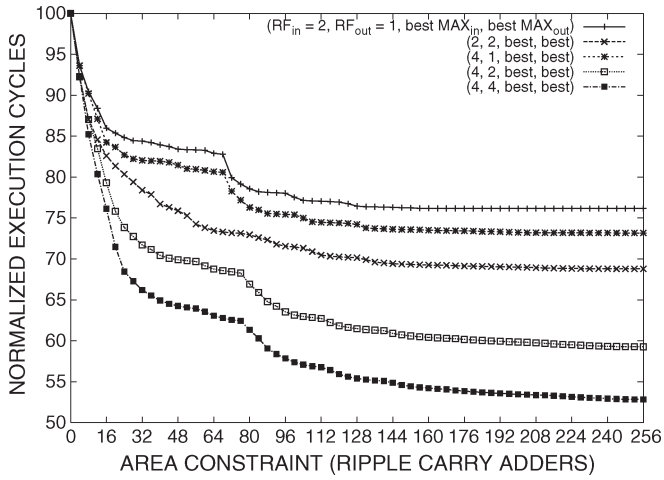

Fig. 10. Djpeg: Effect of increasing the number of register-file ports (i.e., RF_{in} and RF_{out}) on the performance. At each area constraint, we choose the best MAX_{in}, MAX_{out} combination that minimizes the execution time.

In Fig. 11, we analyze the four functions that constitute 92% of the run-time of the djpeg benchmark. Given sufficient register-file ports and area, custom instructions provide more than 2.8 times speed-up for *jpeg_idct_islow* and *h2_v2_fancy_upsample* functions, whereas the acceleration for other functions is limited. The last column shows that given four read and four write ports, we achieve a 47% reduction in the execution cycles of the overall application at an area cost of 256 adders. This translates to a 1.89 times overall speed-up. We observe that most of the area is consumed on accelerating the *jpeg_idct_islow* function (172 adders for maximal speed-up). Fig. 10 shows the area-delay tradeoffs in the design space in more detail. As an example, we could choose to use a register file with four read and two write ports and achieve an overall speed-up of 1.63 times at an area cost of 128 adders.

F. Effect of Loop Unrolling

In Fig. 12, we demonstrate the effect of loop unrolling on the performance of SHA and on the quality of the custom

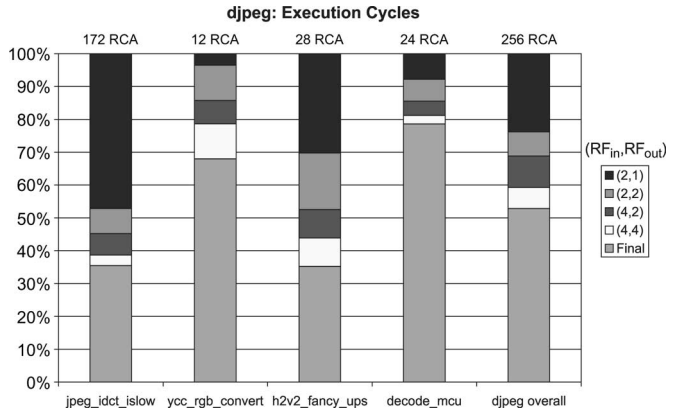

Fig. 11. Djpeg: Increasing the number of register-file ports (i.e., RF_{in}, RF_{out}) improves the performance. First four columns depict the maximal achievable speed-up for the four most time consuming functions. The last column depicts the maximal achievable speed-up for the overall application. Respective area costs are given in terms of RCAs.

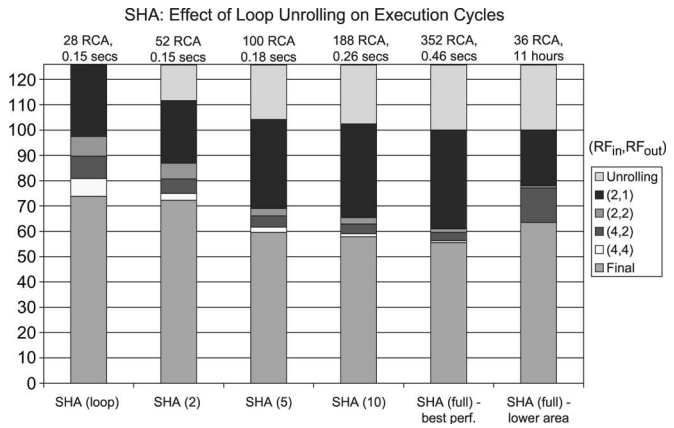

Fig. 12. Loop unrolling improves the performance and enables coarser grain custom instructions. SHA (2) represents the SHA implementation where the main loop is unrolled by two. Area costs in terms of RCAs and the solution times required to identify the custom instructions are shown.

instructions our algorithms generate. We consider the following five different SHA implementations: the first implementation does not unroll the loops; the next three implementations have their loops unrolled two, five, and ten times; and the fifth implementation has all SHA loops fully unrolled. We impose no constraints on the number of inputs and outputs for custom instructions (i.e., $\text{MAX}_{\text{in}} = \infty$, $\text{MAX}_{\text{out}} = \infty$) on the first five columns. The last column again targets the fully unrolled implementation and imposes $\text{MAX}_{\text{in}} = 4$ and $\text{MAX}_{\text{out}} = 2$.

We observe that the solution time required to identify the custom instructions is less than 0.5 s for the first five columns. Loop unrolling increases the size of the basic blocks and results in coarser grain custom instructions. The number of execution cycles monotonically decreases with the amount of unrolling. However, the area overhead also monotonically increases. We observe that, in the last column, by imposing constraints on the number of input and outputs, we identify smaller custom instructions, but we find several equivalent instances of these instructions in the code. The result is a fair speed-up at a reduced area cost. The last column provides a speed-up of 1.56 times over the highest performing software

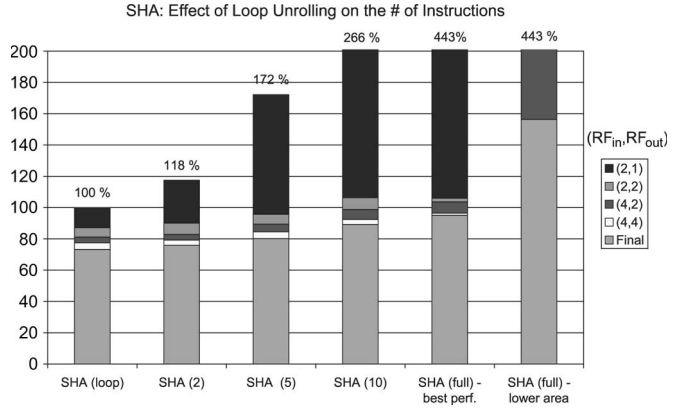

Fig. 13. Loop unrolling increases the number of instructions in the code (up to 443%). Compression due to custom instructions often compensates for this effect.

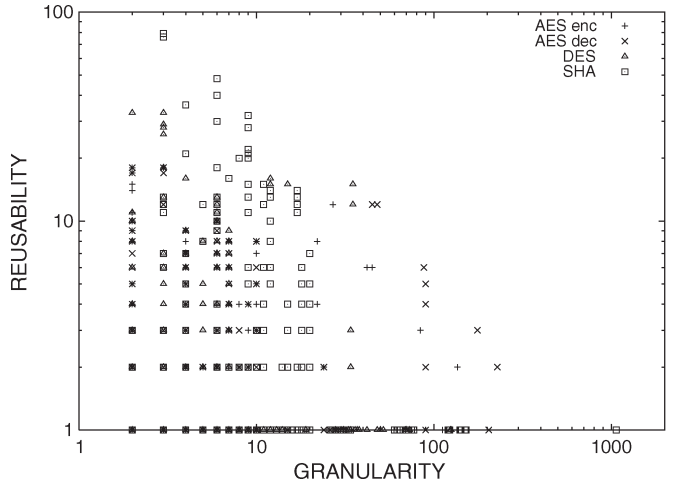

Fig. 14. Granularity versus reusability. Each point represents a custom-instruction candidate.

implementation at the cost of only 36 adders. In this case, the time required to identify the custom instructions by our algorithms is around 11 h.

In Fig. 13, we consider the same design points shown in Fig. 12, and we analyze the effect of loop unrolling on the code size. We observe that, although loop unrolling increases the number of instructions, the compression due to the use of custom instructions often compensates for this effect. We note that the third column, where the main loop of SHA is unrolled five times, provides a speed-up of 1.68 times over the highest performing software implementation at an area cost of 100 adders and results in a code-size reduction of 20% over the most compact software implementation. The associated solution time is less than 0.2 s.

G. Granularity Versus Reusability

We define the granularity of a custom-instruction candidate as the number of base-processor instructions contained in it. We define the reusability of a custom-instruction candidate as the number of structurally equivalent instances of the candidate identified in the application DFGs. Fig. 14 shows the granularity of the custom-instruction candidates we have generated from four cryptography benchmarks versus their reusability. We observe that candidates with high reusability are often fine grained, and coarse-grain candidates usually have limited reusability. In one case, we identify a candidate consisting of 1065 base-processor instructions, which has only a single instance in the code. In another case, we identify a candidate consisting of only three base-processor instructions, which is reused 80 times. Another candidate identified by our algorithms consists of 45 base-processor instructions, and it has 12 instances in the code. Exploring different granularities in this manner allows us to identify the most promising area and performance tradeoffs within the design space.

H. Run-Time Results

The runtime of our tool chain is dominated by the template-generation algorithm described in Section V that solves a series of ILP problems. At each iteration, we first solve a relaxed problem, where the convexity constraint is not imposed on the custom-instruction templates. If the solution to the relaxed problem is not a convex template, we impose the convexity constraint and solve once more. Often, the first iteration of the template-generation algorithm is the most time consuming one. Table IV describes ILP statistics associated with the first iteration of the template-generation algorithm on the largest basic blocks of four cryptography benchmarks for four $(\mathrm{MAX_{in}}, \mathrm{MAX_{out}})$ combinations [i.e., (4, 1), (4, 2), (4, 4), and (∞, ∞)]. While generating the results, we set $(RF_{\mathrm{in}} = 2, RF_{\mathrm{out}} = 1)$ if $(\mathrm{MAX_{in}} = \infty, \mathrm{MAX_{out}} = \infty)$. Otherwise, we set $(RF_{\mathrm{in}} = \mathrm{MAX_{in}}, RF_{\mathrm{out}} = \mathrm{MAX_{out}})$.

We observe that the solution time is usually a few seconds. Quite often, the relaxed problem generates a convex template, and we skip solving the original problem with the convexity constraint. The solution time may exceed 1 h in some cases as it happens for fully unrolled SHA when $\mathrm{MAX_{out}} = 2$ or $\mathrm{MAX_{out}} = 4$. In these two cases, we observe that the upper bound provided by the relaxed problem considerably pulls down the solution time for the original problem. We observe that our algorithms are extremely efficient when the input and output constraints are removed (i.e., $\mathrm{MAX_{in}} = \infty$, $\mathrm{MAX_{out}} = \infty$). The first iteration of the template-generation algorithm on fully unrolled SHA takes only 0.22 s.

In Table V, we show the solution times for the first iteration of the exact algorithm of [22] on the same benchmarks and for the same $(\mathrm{MAX_{in}}, \mathrm{MAX_{out}})$ combinations given in Table IV. We also show the solution times required by our algorithm, which are sums of the solution times of the relaxed and original problems from Table IV. We note that given $(RF_{\mathrm{in}} = \mathrm{MAX_{in}}, RF_{\mathrm{out}} = \mathrm{MAX_{out}})$, the objective function of our ILP formulation is equivalent to the merit function of [22]. We observe that the algorithm of [22] is, in general, efficient when the input/output constraints are tight (i.e., $\mathrm{MAX_{out}} = 1$ or $\mathrm{MAX_{out}} = 2$). However, it fails to complete for DES within 24 h even if $\mathrm{MAX_{out}} = 1$. The algorithm of [22] becomes extremely inefficient when the constraints are loose (i.e., $\mathrm{MAX_{out}} = 4$) or removed (i.e., $\mathrm{MAX_{out}} = \infty$) and fails to complete for all four benchmarks within 24 h. Our algorithm is faster in most of the cases and successfully completes in all

TABLE IV
SIZE OF THE LARGEST BASIC BLOCK (BB), THE NUMBER OF INTEGER DECISION VARIABLES (VARS), AND THE NUMBER OF LINEAR CONSTRAINTS (CONSTRS) FOR THE RELAXED PROBLEM AND FOR THE ORIGINAL PROBLEM WITH THE CONVEXITY CONSTRAINT. WE SHOW THE SOLUTION TIMES ASSOCIATED WITH THE FIRST ITERATION OF THE TEMPLATE-GENERATION ALGORITHM FOR FOUR ($\mathrm{MAX_{in}}, \mathrm{MAX_{out}}$) COMBINATIONS

Benchmark	BB size	Relaxed problem						Original problem with convexity constraints					
		Vars	Constrs	Solution time (seconds)				Vars	Constrs	Solution time (seconds)			
				(4,1)	(4,2)	(4,4)	(∞,∞)			(4,1)	(4,2)	(4,4)	(∞,∞)
AES enc.	317	879	1872	0.06	0.12	0.05	0.03	1403	4124	*skipped*	*skipped*	0.78	0.14
AES dec.	501	1591	3508	0.25	0.23	0.13	0.06	2483	7404	*skipped*	0.55	2.98	0.37
DES	822	1962	4043	2.75	0.42	0.09	0.08	3417	9760	*skipped*	*skipped*	9.78	11.23
SHA (full)	1155	3777	8885	193	5633	5116	0.22	5899	18524	*skipped*	4122	1839	*skipped*

TABLE V
RUN-TIME COMPARISON WITH THE EXACT ALGORITHM OF [22]. WE SHOW THE SOLUTION TIMES IN SECONDS FOR FOUR ($\mathrm{MAX_{in}}, \mathrm{MAX_{out}}$) COMBINATIONS. REFERENCE [22] FAILS TO COMPLETE WITHIN 24 h FOR ALL FOUR BENCHMARKS GIVEN $\mathrm{MAX_{out}} = 4$ AND $\mathrm{MAX_{out}} = \infty$

Benchmark	[22]		Our work			
	(4,1)	(4,2)	(4,1)	(4,2)	(4,4)	(∞,∞)
AES enc.	0.43	397	0.06	0.12	0.83	0.17
AES dec.	1.05	1417	0.25	0.78	3.11	0.43
DES	-	-	2.75	0.42	9.87	11.31
SHA (full)	3.94	317	193	9755	6955	0.22

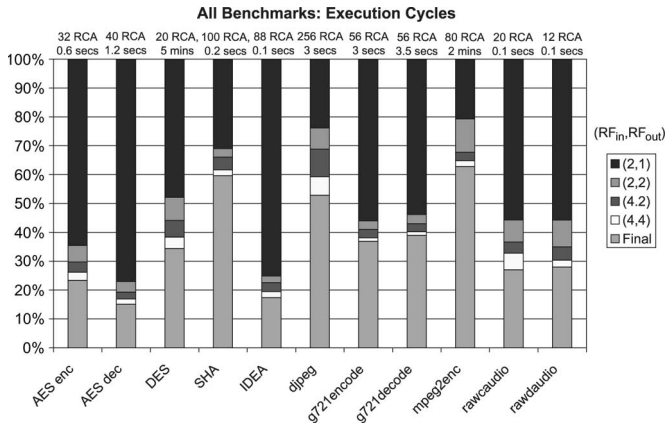

Fig. 15. All benchmarks: Increasing the number of register-file ports (i.e., RF_{in}, RF_{out}) improves the performance. Area costs in terms of RCAs and solution times are shown. For each benchmark, the highest performing code with and without compiler transformations is taken as the base. The largest solution time observed is only 5 min.

of the cases. We have obtained optimal ILP results in all of our experiments.

I. Overall Results

In Fig. 15, we describe the reduction in the execution-cycle count for all the benchmarks from Table III while increasing the number of register-file ports. The area costs and the solution times are given on top of the columns for each benchmark. Using only a limited amount of hardware resources, we obtain a speed-up of up to 4.3 times for AES encryption, 6.6 times for AES decryption, 2.9 times for DES, 5.8 times for IDEA, 2.7 times for g721decode, 1.7 times for mpeg2encode, and 4.7 times for rawcaudio. Except for a few cases, we obtain the highest performing solutions when we remove the input/output constraints. In most cases, the highest performing solution is found in only a few seconds, but it may take up to a few minutes

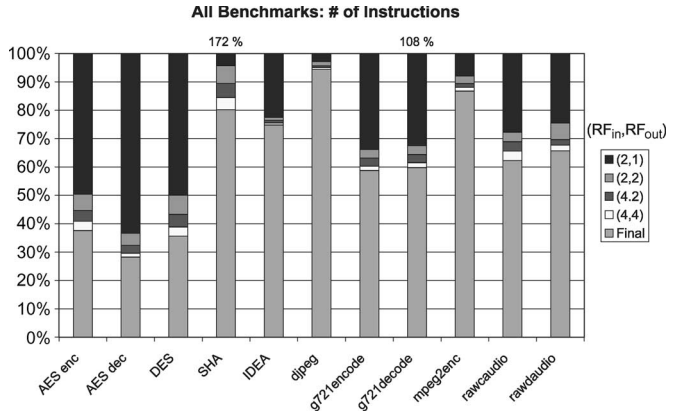

Fig. 16. All benchmarks: Increasing the number of register-file ports (i.e., RF_{in}, RF_{out}) reduces the number of instructions in the code. For each benchmark, the smallest code with and without compiler transformations is taken as the base.

as observed for DES and mpeg2enc. For the same design points, Fig. 16 shows the reduction in the total number of instructions in the code. A reduction of up to 72 large benchmarks with small kernels, the code-size reduction can be as small as 6%, as it is observed for djpeg.

VIII. CONCLUSION

In this paper, we describe an ILP-based system called CHIPS for identifying custom instructions given the available data bandwidth and transfer latencies between the base processor and the custom logic. We iteratively solve a set of ILP problems in order to generate a set of custom-instruction templates. At each iteration, ILP orders feasible templates based on a high-level metric and picks the one that offers the highest reduction in the schedule length. The iterative algorithm aims to maximize the code covered by custom instructions and guarantees that the number of generated custom-instruction templates is at most linear in the total number of instructions within the application. After template generation, we identify structurally equivalent templates based on isomorphism testing, and we select the most profitable templates under area constraints based on a Knapsack model.

Our approach involves a baseline machine with architecturally visible state registers. We enable designers to optionally constrain the number of input and output operands for custom instructions. We demonstrate that our algorithms are able to handle benchmarks with large basic blocks consisting of more than 1000 instructions, with or without the input/output

constraints. Our experiments show that the removal of input/output constraints results in the highest performing solutions. We demonstrate that these solutions cannot be covered by the subgraph-enumeration algorithms of [21] and [22], which rely on input/output constraints for reducing the search space. On the other hand, we observe that input/output constraints help us identify frequently used code segments and efficiently explore the area/performance tradeoffs in the design space.

Our approach does not guarantee a globally optimal solution. Whether a truly optimal algorithmic flow exists is still an open-research question. We believe that our solution can still be improved by the following extensions: 1) combining our approach with pattern-matching techniques [16], [27]–[29] in order to improve the utilization of the custom instructions we generate and 2) integrating our approach with datapath-merging techniques [32]–[34] in order to exploit partial resource sharing across custom instructions for area-efficient synthesis.

We are currently exploring techniques that enable custom instructions to access memory hierarchy [50], including automated partitioning of the program data between on- and off-chip memories. We hope to extend our approach to cover run-time reconfigurable processors [51] and heterogeneous multiprocessor systems [52].

ACKNOWLEDGMENT

The authors would like to thank N. Clark for his help in using Trimaran.

REFERENCES

[1] A. Fauth *et al.*, "Describing instruction set processors using nML," in *Proc. Eur. Des. Test Conf.*, Mar. 1995, pp. 503–507.
[2] A. Hoffmann *et al.*, *Architecture Exploration for Embedded Processors With LISA*. Norwell, MA: Kluwer, 2002.
[3] P. Faraboschi *et al.*, "Lx: A technology platform for customizable VLIW embedded processing," in *Proc. ISCA*, Vancouver, BC, Canada, Jun. 2000, pp. 203–213.
[4] S. P. Seng *et al.*, "Run-time adaptive flexible instruction processors," in *Proc. FPL*. Montpellier, France, Sep. 2002, pp. 545–555.
[5] D. Goodwin and D. Petkov, "Automatic generation of application specific processors," in *Proc. CASES*, San Jose, CA, Nov. 2003, pp. 137–147.
[6] G. Martin, "Recent developments in configurable and extensible processors," in *Proc. ASAP*, Steamboat Springs, CO, Sep. 2006, pp. 39–44.
[7] R. K. Gupta and G. D. Micheli, "System-level synthesis using reprogrammable components," in *Proc. EURO-DAC*, 1992, pp. 2–7.
[8] R. Ernst *et al.*, "Hardware-software cosynthesis for microcontrollers," *IEEE Des. Test Comput.*, vol. 10, no. 4, pp. 64–75, Dec. 1993.
[9] R. Niemann and P. Marwedel, "An algorithm for hardware/software partitioning using mixed integer linear programming," *Des. Autom. Embed. Syst.*, vol. 2, no. 2, pp. 165–193, Mar. 1997.
[10] P. Arato *et al.*, "Hardware–software partitioning in embedded system design," in *Proc. Int. Symp. Intell. Signal Process.*, Sep. 2003, pp. 197–202.
[11] J. Van Praet *et al.*, "Instruction set definition and instruction selection for ASIPs," in *Proc. 7th Int. Symp. High-Level Synthesis*, 1994, pp. 11–16.
[12] I.-J. Huang and A. M. Despain, "Synthesis of application specific instruction sets," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 14, no. 6, pp. 663–675, Jun. 1995.
[13] M. Arnold and H. Corporaal, "Designing domain specific processors," in *Proc. 9th Int. Workshop HW/SW Codesign*, Apr. 2001, pp. 61–66.
[14] M. Baleani *et al.*, "HW/SW partitioning and code generation of embedded control applications on a reconfigurable architecture platform," in *Proc. 10th Int. Workshop HW/SW Codesign*, May 2002, pp. 151–156.
[15] P. Brisk *et al.*, "Instruction generation and regularity extraction for reconfigurable processors," in *Proc. CASES*, Oct. 2002, pp. 262–269.
[16] N. Clark *et al.*, "Processor acceleration through automated instruction set customization," in *Proc. MICRO*, San Diego, CA, Dec. 2003, pp. 184–188.
[17] C. Alippi *et al.*, "A DAG based design approach for reconfigurable VLIW processors," in *Proc. DATE*, Munich, Germany, Mar. 1999, pp. 778–779.
[18] N. N. Binh *et al.*, "A hardware/software partitioning algorithm for designing pipelined ASIPs with least gate counts," in *Proc. DAC*, 1996, pp. 527–532.
[19] F. Sun *et al.*, "A scalable application-specific processor synthesis methodology," in *Proc. ICCAD*, San Jose, CA, Nov. 2003, pp. 283–290.
[20] N. Cheung *et al.*, "INSIDE: INstruction Selection/Identification and Design Exploration for extensible processors," in *Proc. ICCAD*, 2003, pp. 291–297.
[21] K. Atasu *et al.*, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Proc. DAC*, Jun. 2003, pp. 256–261.
[22] L. Pozzi *et al.*, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 7, pp. 1209–1229, Jul. 2006.
[23] J. Cong *et al.*, "Application-specific instruction generation for configurable processor architectures," in *Proc. FPGA*, Feb. 2004, pp. 183–189.
[24] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors," in *Proc. CASES*, Sep. 2004, pp. 69–78.
[25] P. Biswas *et al.*, "ISEGEN: Generation of high-quality instruction set extensions by iterative improvement," in *Proc. DATE*, 2005, pp. 1246–1251.
[26] P. Bonzini and L. Pozzi, "Polynomial-time subgraph enumeration for automated instruction set extension," in *Proc. DATE*, Apr. 2007, pp. 1331–1336.
[27] R. Leupers and P. Marwedel, "Instruction selection for embedded DSPs with complex instructions," in *Proc. EURO-DAC*, 1996, pp. 200–205.
[28] A. Peymandoust *et al.*, "Automatic instruction set extension and utilization for embedded processors," in *Proc. ASAP*, Jun. 2003, pp. 108–118.
[29] N. Cheung *et al.*, "MINCE: Matching INstructions using combinational equivalence for extensible processor," in *Proc. DATE*, 2004, pp. 1020–1027.
[30] N. Clark *et al.*, "An architecture framework for transparent instruc tion set customization in embedded processors," in *Proc. ISCA*, 2005, pp. 272–283.
[31] R. Dimond *et al.*, "Application-specific customisation of multi-threaded soft processors," *Proc. Inst. Electr. Eng.—Computers Digital Techniques*, vol. 153, no. 3, pp. 173–180, May 2006.
[32] W. Geurts *et al.*, *Accelerator Data-Path Synthesis for High-Throughput Signal Processing Applications*. Norwell, MA: Kluwer, 1997.
[33] P. Brisk *et al.*, "Area-efficient instruction set synthesis for reconfigurable system-on-chip designs," in *Proc. DAC*, Jun. 2004, pp. 395–400.
[34] N. Moreano *et al.*, "Efficient datapath merging for partially reconfigurable architectures," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 7, pp. 969–980, Jul. 2005.
[35] J. Cong *et al.*, "Instruction set extension with shadow registers for configurable processors," in *Proc. FPGA*, Feb. 2005, pp. 99–106.
[36] R. Jayaseelan *et al.*, "Exploiting forwarding to improve data bandwidth of instruction-set extensions," in *Proc. DAC*, Jul. 2006, pp. 43–48.
[37] J. Lee *et al.*, "Efficient Instruction encoding for automatic instruction set design of configurable ASIPS," in *Proc. ICCAD*, Nov. 2002, pp. 649–654.
[38] N. Clark *et al.*, "Scalable subgraph mapping for acyclic computation accelerators," in *Proc. CASES*, Oct. 2006, pp. 147–157.
[39] P. Yu and T. Mitra, "Satisfying real-time constraints with custom instructions," in *Proc. CODES+ISSS*, Jersey City, NJ, Sep. 2005, pp. 166–171.
[40] *Nauty Package*. [Online]. Available: http://cs.anu.edu.au/people/bdm/nauty
[41] Trimaran. [Online]. Available: http://www.trimaran.org
[42] ILOG CPLEX. [Online]. Available: http://www.ilog.com/products/cplex/
[43] S. Aditya *et al.*, "Elcor's machine description system," Hewlett-Packard Lab., Palo Alto, CA, HP Labs Tech. Rep., HPL-98-128, 1998.
[44] V. Kathail *et al.*, "HPL-PD architecture specification," Hewlett-Packard Lab., Palo Alto, CA, HP Labs Tech. Rep., HPL-93-80R1, 1993. Version 1.0.
[45] J. R. Allen *et al.*, "Conversion of control dependence to data dependence," in *Proc. 10th ACM Symp. Principles Program. Languages*, Jan. 1983, pp. 177–189.
[46] K. Atasu *et al.*, "Efficient AES implementations for ARM based platforms," in *Proc. ACM SAC*, Mar. 2004, pp. 841–845.
[47] *XySSL—DES and Triple-DES Source Code*. [Online]. Available: http://xyssl.org/

[48] M. Guthaus *et al.*, *MiBench: A Free, Commercially Representative Embedded Benchmark Suite*. [Online]. Available: http://www.eecs.umich.edu/mibench/
[49] C. Lee *et al.*, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. MICRO*, Dec. 1997, pp. 330–335.
[50] P. Biswas *et al.*, "Automatic identification of application-specific functional units with architecturally visible storage," in *Proc. DATE*, Mar. 2006, pp. 1–6.
[51] L. Bauer *et al.*, "RISPP: Rotating instruction set processing platform," in *Proc. DAC*, San Diego, CA, Jun. 2007, pp. 791–796.
[52] S. L. Shee and S. Parameswaran, "Design methodology for pipelined heterogeneous multiprocessor system," in *Proc. DAC*, San Diego, CA, Jun. 2007, pp. 811–816.
[53] K. Atasu *et al.*, "An integer linear programming approach for identifying instruction-set extensions," in *Proc. CODES+ISSS*, Sep. 2005, pp. 172–177.
[54] K. Atasu *et al.*, "Optimizing instruction-set extensible processors under data bandwidth constraints," in *Proc. DATE*, Apr. 2007, pp. 588–593.

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 the University of Lugano, Lugano, Switzerland, in 2002, and the Ph.D. degree in computer engineering from Boğaziçi University in 2007.

From 2002 to 2003, he was a Research Assistant with the Swiss Federal Institute of Technology Lausanne, Lausanne, Switzerland. In summer 2006, he was a Visiting Researcher with Stanford University, Stanford, CA. Since 2005, he has been a Research Associate with the Department of Computing, Imperial College London, London, U.K. His research interests include customizable processors, electronic design automation, computer arithmetic, and reconfigurable computing.

Dr. Atasu was the recipient of a Best Paper Award in embedded systems category at the Design Automation Conference in 2003.

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

He was a Postdoctoral Staff Scientist at the Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA. In 1996, he became a faculty member in the Department of Computer Engineering, Boğaziçi University, Istanbul, Turkey, where he is currently an Associate Professor. His research interests include parallel processing, scientific computing, graph algorithms, and grid computing.

Günhan Dündar (S'90–M'95) was born in Istanbul, Turkey, in 1969. He received the B.S. and M.S. degrees in electrical engineering from Boğaziçi University, Istanbul, in 1989 and 1991, respectively, and the Ph.D. degree in electrical engineering from Rensselaer Polytechnic Institute, Troy, NY, in 1993.

Since 1994, he has been with the Department of Electrical and Electronics Engineering, Boğaziçi University, where he is currently a Professor and Department Chair. In 1994, he was with the Turkish Navy at the Naval Academy, and in 2003, he was with École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, both on leave from Boğaziçi University. He has more than 100 publications in international journals and conferences. His research interests include analog IC design, electronic design automation, and neural networks.

Oskar Mencer (S'96–M'00) received the B.S. degree in computer engineering from Technion, Israel Institute of Technology, 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.

After three years as a member of the technical staff with the Computing Sciences Research Center, Bell Laboratories, Murray Hill, NJ, he founded MAXELER Technologies, New York, NY, in 2003. Since 2004, he is a full-time member of the academic staff with the Department of Computing, Imperial College London, London, U.K., where he also leads the Computer Architecture Research Group. His research interests include computer architecture, computer arithmetic, very large scale integration (VLSI) microarchitecture, VLSI computer-aided design, 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.

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

He is currently a Professor of computer engineering with the Department of Computing, Imperial College London, London, U.K., where he also leads the Custom Computing Group. 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.