A Scalable Application-Specific Processor Synthesis Methodology

Fei Sun[†], Srivaths Ravi[‡], Anand Raghunathan[‡], and Niraj K. Jha[†]
[†] Department of Electrical Engineering, Princeton University, Princeton, NJ 08544
[‡] NEC Laboratories America, Princeton, NJ 08540

Abstract—**Custom processors based on application-specific or domain-specific instruction sets are gaining popularity, and are often used to implement critical architectural blocks in complex system-on-chips. While several advances have been made in custom processor architectures, tools, and design methodologies, designers are still required to manually perform some critical tasks, such as selection of the custom instructions best suited to the given application and design constraints.**

We present a scalable methodology for the synthesis of a custom processor from an embedded software program. A key feature of the proposed methodology is its scalability, which is achieved by exploiting the structured, hierarchical nature of large software programs. We motivate the need for such a methodology, and describe the algorithms used for the critical steps, including hardware resource budgeting, local optimizations, and global exploration. Our methodology utilizes the concept of "soft" instruction templates, which can be adapted by adding operations to them or deleting operations from them at any time during the design space exploration process, allowing for global design decisions to be interleaved with fine-grained optimizations.

We have integrated our methodology in an open source compiler, and verified it using a commercial extensible processor. Experiments with several benchmarks indicate that our methodology can effectively tackle large programs. It resulted in the synthesis of high-quality custom processors that demonstrated an average speedup of 2.61X and a maximum speedup of 6.32X. The CPU times required for custom processor synthesis were quite small, indicating that the proposed techniques can be applied to embedded software programs of significant complexity.

I. INTRODUCTION

Improvements in semiconductor fabrication technologies promise to make it feasible to replace logic gates or hardware macro-blocks with microprocessors as building blocks for integrated circuit (IC) design. Such programmable solutions will provide the ability to meet short product cycles and cope with changing application functionality (*e.g.*, in areas with evolving standards). However, the rapid expansion in the market for embedded systems with tight constraints on cost, performance, size, and power consumption implies that the need to customize the architecture to the application or application domain will continue to be a primary driving requirement in system-on-chip (SoC) design.

The application-specific instruction set processor (ASIP) can provide a good trade-off between flexibility and efficiency, by tailoring the instruction set and micro-architecture to one specific application, or a set of applications for the same domain. The recent emergence of configurable and extensible processors (*e.g.*, [1], [2], [3], [4]) has met with significant success, and several complex SoCs featuring such processors (often several on the same IC) are already in production.

Acknowledgments: This work was supported by the NJCST Center for Embedded System-on-a-chip Design.

While a significant body of research, built up over a decade, exists for ASIP synthesis, most of it has focused on generating custom instruction sets and architectures from scratch, rather than specializing a given base processor. Most current commercial offerings, however, are based on the latter approach, since starting with a pre-optimized, pre-verified base architecture considerably eases the challenging tasks of design verification as well as generation of the software development tool chain. More importantly, we believe that further research is required in order to allow these techniques to scale well with application program size, to handle large, realistic embedded software applications.

Enabling the vision of the application-specific processor as the ubiquitous building block of future SoCs requires several innovations in architecture and design methodologies. These innovations should enable designers to create highly optimized instances of custom processors, which achieve high levels of processing efficiency in very short turn-around times (hardware-like efficiency from software solutions).

Despite significant advances in the supporting methodologies and automation tools for application-specific processor design (*e.g.*, re-targetable software tool chains), designers are still required to manually design custom instructions and the hardware to speed up the application, which is a slow, tedious, and error-prone process. Application-specific processor synthesis (or custom processor synthesis) attempts to address this problem by developing tools that automatically analyze the application, generate and evaluate various choices of the custom processor architecture, and select the one that best meets designers' constraints (*e.g.*, performance, area, power). Custom processor synthesis is quite challenging due to the complex inter-dependencies and tradeoffs involved in making the design decisions, which can be difficult to explore even for small applications. Moreover, the large size of real-life embedded software applications makes this task even more challenging. A moderate-sized program may have tens of thousands of lines of code and hundreds or thousands of functions. For example, consider the task of synthesizing an application-specific processor based MPEG-2 encoder/decoder (CODEC). A typical MPEG-2 decoder implementation that we analyzed contains 7,832 lines of code and 114 functions, while the encoder has 7,605 additional lines of code that comprise 95 additional functions. As shown later in the paper, the number of candidate custom instructions that can be generated to accelerate such applications is huge. Thus, any method to explore the architectural design space needs to perform highly efficient design space exploration.

A. Related Work

We first briefly discuss some previous work on ASIP design, configurable processor synthesis, and custom instruction generation and selection.

An overview of ASIP design, its benefits, challenges and barriers are described in [5], [6]. The architecture/compiler codesign of ASIPs is addressed in [7]. A method to search for a minimal-area processor core and cache, which satisfy the performance requirements of certain applications, is given in [8]. An ASIP design methodology for extending an existing processor instruction set architecture is described in [9]. A methodology for hardware-software partitioning using function call graphs is described in [10].

Instead of designing an ASIP from scratch, configurable processor synthesis tailors custom processors by adding or deleting some options, or functional units from the core processor. The notion of a flexible instruction processor (FIP) and parameterization of its modular processor templates is introduced in [11]. In [12], the PICO-NPA system that automatically synthesizes nonprogrammable accelerators (NPAs) is described. A method to partially customize the instruction set of VLIW processors for embedded applications is presented in [13]. Selection of intellectual property (IP) surrounding an ASIP core to accelerate application programs is tackled in [14]. The work in [15] deals with a method to merge datapaths to reduce interconnections. In [16], an automatic method to select among a few processor options under an area constraint is proposed.

Custom instruction generation and selection techniques try to extend the existing instruction set of the base processor in order to speed up specific applications. In [17], a semi-automated method for detection and exploitation of custom instructions for VLIW processors is provided. The work in [18] describes a method to generate complex instructions from primitive ones. In [19], a method to choose from different instruction encoding alternatives in ASIP design is proposed. Methods to generate custom instructions from operation patterns are described in [20], [21]. A technique to rapidly select custom instructions from a set of pre-synthesized custom instructions is presented in [22]. The work in [23] analyzes the impact of custom instructions on the entire system, and provides some case studies.

B. Paper Overview and Contributions

In this paper, we provide a *scalable* methodology for application-specific processor synthesis. Given one or more application programs, and the desired design constraints (*e.g.*, area), the methodology generates a custom processor architecture by extending the instruction set of a base processor, so that its performance in executing the given program is maximized. We exploit the fact that large software programs are written in a modular and hierarchical manner, and are composed of functions or procedures. We use the hierarchical structure of the program during several phases of our methodology, including hardware resource budgeting, and to structure our custom instruction selection process into local and global optimizations, much like a software compiler. The proposed methodology solves the problem of custom instruction selection recursively by traversing the function call graph - starting with smaller sub-programs at the leaves of the function call graph, finally merging them to obtain the solution for the entire program. We develop efficient and accurate macro-models to estimate the performance and area of a base processor augmented with a given set of custom instructions. Instead of considering already-selected custom instructions as atomic units, we consider adding and dropping of operations inside each custom instruction, resulting in a finer-grained (and better) approach. In order to address the size of the design space, we formulate the custom instruction pruning problem. While pruning, we not only consider each custom instruction independently, but also consider its performance and area impact with respect to all the other custom instructions.

We have implemented the proposed techniques by integrating them in an open-source compiler research framework (SUIF [24]), and have verified them using a commercial extensible processor (Tensilica's Xtensa [1]). We evaluated our tool by using it to generate custom processor architectures based on Xtensa for several embedded software programs. The resulting custom processors were subjected to logic synthesis and technology mapping to a commercial technology library, in order to estimate their area and performance. Our experiments indicate that custom processors generated using the proposed methodology can result in significant efficiency improvements in very limited synthesis times.

II. MOTIVATION

In traditional manual custom instruction design flows, designers need to read through the program, pick out the performance-critical function, analyze the function, and rewrite the function and the custom instruction description. This is a very slow and error-prone process. Moreover, it is difficult for designers to perform global tradeoffs. Hence, the custom instructions generated are often only locally optimal, but not globally optimal.

The final objective of custom instruction synthesis for extensible processors is to solve the following problem.

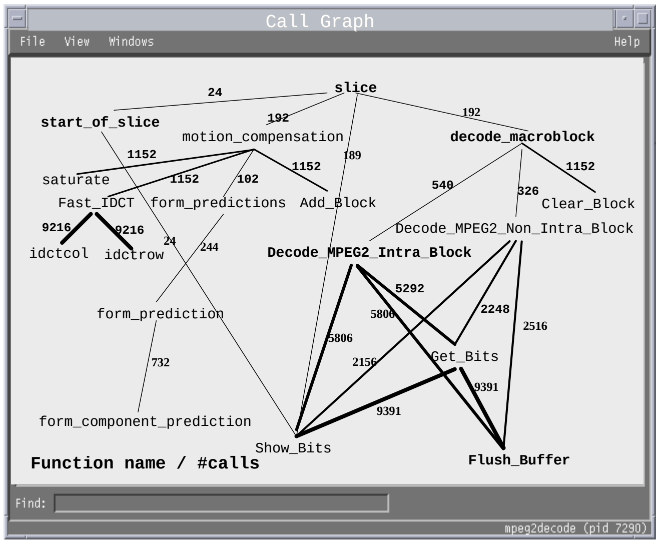
Problem 1: Given an extensible processor and an application, generate a set of custom instructions to speed up the application as much as possible while the total area of the processor is within the given area budget.

In a manual design, designers may be able to obtain custom instructions for different parts of the program, and then select a subset of these instructions to meet the area budget. However, this approach has several drawbacks. One frequently finds identical code in different parts of a large program. It is easy for a designer to overlook this fact and not be able to exploit it for larger speed-ups. It may be possible to reduce the number of operations in the derived custom instructions to try to meet the area budget. However, because of the huge search space involved, a designer is likely to just obtain a subset of the custom instructions he/she found, and not try to reduce individual custom instructions, in order to meet the area budget. Note that reducing operations in one custom instruction has an impact on how other custom instructions have to be transformed as a result.

A program may have a number of functions and a hierarchical structure. The following example illustrates the huge custom instruction design space, and why good-quality manual design is very difficult.

Example 1: Fig. 1(a) shows part of a function call graph of the `mpeg2decode` program, rooted at function `slice`. In this graph, an edge from a parent function to a child function indicates that the parent calls the child. The numbers on the edges are the times each function is called. This is not a complete call graph as some children and library function calls are not shown in the figure. Fig. 1(b) shows a table whose entries indicate the number of cycles each function consumes.

From the function call graph, we can obtain a number of custom instructions. In functions `idctcol` and `idctrow`, the largest custom instruction has 63 and 64 operations, respectively. Note that the largest custom instruction may not necessarily yield the best performance (this is because it may require too many inputs and outputs, which may reduce its performance improvement, as discussed later). If we obtain smaller custom instructions by dropping some



(a)

Function	Cycles
start_of_slice	888
form_predictions	6,401
slice	11,094
form_prediction	14,152
motion_compensation	32,868
decode_macroblock	43,295
Get_Bits	65,737
Show_Bits	76,844
Decode_MPEG2_Non_Intra_Block	88,780
Fast_IDCT	94,464
Clear_Block	152,064
Flush_Buffer	222,988
Decode_MPEG2_Intra_Block	226,937
form_component_prediction	315,987
Add_Block	353,992
saturate	374,128
idctrow	487,624
idctcol	1,116,168

(b)

Fig. 1. `mpeg2decode`: (a) function call graph, and (b) number of cycles each function consumes

operations from the largest custom instruction, we end up with more than 100,000 custom instruction candidates just for these two functions. Additionally, for functions **form_component_prediction**, **Add_Block**, and **saturate**, we can obtain 16, 11 and one largest custom instructions, respectively, that need to be investigated further.

If we only consider the largest custom instructions, we have 2^{30} possible choices for this small call graph. In the entire program, we can find nearly 300 largest custom instructions. If we consider the possible combinations of smaller custom instructions, the number of choices would run into billions. ∎

Example 1 illustrates that the design space is very large for hierarchical programs, and hence it is not possible to efficiently solve the custom processor synthesis problem directly. One approach may be to divide a large program into several smaller programs and solve the problem for the latter first. This divide-and-conquer approach can significantly reduce the design space. However, a simple divide-and-conquer algorithm may lead to a very inferior solution. For example, a straightforward divide-and-conquer algorithm would be to divide the available area among the different functions based on the number of cycles that each function consumes in the base processor. While, on the surface this may seem like a fair area allocation, the following example illustrates why this is not so.

TABLE I
Custom instructions found for the call graph in Fig.1

Function	Custom instr	Saved cycles	Area	Custom Reg.
idctcol	Ins_1	497,664	270,088	13
idctrow	Ins_2	165,888	280,606	13
saturate	Ins_3	72,238	30,576	0
Add_Block	Ins_4	1,152	81,836	4

Example 2: Table I shows some custom instructions for some of the functions in the call graph of Fig.1. The total number of cycles for all the functions is 3,674,411. The table also shows the number of cycles saved if that custom instruction were to be used, the area allocated to the function (in grids) based on the above philosophy, and the number of user-defined custom registers.

The problem with the above area allocation policy is that it allocates the area before we even know whether any custom instructions can be found for those functions and how much area they actually require. Hence, except for the few functions that yield custom instructions, the area allocated to all other functions is totally wasted. It turns out that the area allocated to function `idctcol` is much larger than the area required for its best custom instruction. If the allocated area is fixed, and we cannot move the spare area to other functions, then this also represents waste. To make matters worse, the area required to implement Ins_2 is larger than the area allocated to its function, `idctrow`. Hence, in this case we will not even be able to choose the custom instruction that can yield the best performance. We will need to shrink the custom instruction in some way to satisfy the local area requirement. Finally, in some cases, a custom instruction may yield a speed-up only if it is large enough (otherwise the overheads may swamp the benefits). This means that there is an area lower bound for a custom instruction to be useful. If it so happens that the area lower bound for Ins_2 is a little above its local area requirement, we cannot even choose Ins_2. In summary, if we use a straightforward area budgeting algorithm, because of the large discrepancy between the saved cycles, area of custom instructions, and the number of cycles required for the function that the custom instructions belong to, the result may very easily get stuck in an inferior solution. ∎

In Example 2, we assumed that each custom instruction has an area allocation, independent of the area allocated to other custom instructions. However, sometimes it is possible for two custom instructions to share area because of a similar structure, or a similar number of inputs and outputs, which further complicates the problem.

A typical RISC instruction can only have a finite number of inputs and outputs encoded in an instruction. Suppose these numbers are α and β, respectively. If we restrict the number of inputs and outputs of custom instructions to α and β, respectively, we may end up with very few custom instruction candidates and hence not obtain the complete benefit of using custom instructions. Fortunately, many commercial extensible processors, like Xtensa [1], allow more inputs and outputs for custom instructions. Besides the general-purpose register file, they allow some user-defined registers. Before custom instruction computation commences, some `write` instructions write the input data to those user-defined registers. Then the custom instruction implicitly reads the data from the user-defined registers during computation, and writes results to user-defined registers. After the computation is complete, some `read` instructions explicitly read the data from the user-defined registers.

User-defined registers are local to the processor. It is used just before and after custom instruction computation. At other times, the values in those registers are invalid. Hence, we can reuse the user-defined registers for different custom instructions, if they are guaranteed to execute sequentially, which is true for programs on single-pipeline processors. Some software transformations, such as loop-independent code motion, can increase the life length of the user-defined registers. Thus, the above constraint needs to be kept in mind while performing the transformations.

III. METHODOLOGY AND ALGORITHMS

In this section, we describe our methodology for synthesis of custom instructions. We first provide an overview of the entire design flow and then describe the important phases and steps in detail.

A. Overview

Fig. 2 illustrates the three phases involved in the automatic generation of custom instructions for hierarchical programs. The rectangular blocks denote transformation procedures, and the cylindrical blocks denote data. The intermediate representation (IR) is obtained from the SUIF compiler [24].

In Phase I, the algorithm first takes a target program (in C) as input (block **1**), and performs some target-independent transformations (block **2**) (*e.g.*, constant folding, copy propagation, common subexpression elimination, etc.) to simplify the input program. Then the profiling statistics (block **4**), program dependence graph (block **5**), and call graph (block **6**) are generated from the transformed IR (block **3**). Based on all these data, custom instruction templates are generated (block **7**). Only the largest possible template inside a basic block is generated, leaving the pruning of the template to Phase II. This significantly reduces the complexity of template generation. A lot of C programs use macros for convenience. The same macro may appear at different places in the program, resulting in the same code. Therefore, after all the largest custom instruction templates are generated, we collect information on which templates are identical and merge such templates into one (block **8**). The merged template has the attributes (*e.g.*, number of execution cycles) of all the identical templates. Finally, a modified IR with custom instruction template annotations (block **9**) is passed to Phase II.

Since the design space for realistic software programs is extremely large, we cannot use a flattened description of the application to solve Problem 1. Therefore, Phase II makes use of the program hierarchy, to divide the large program into several smaller programs, each having a much smaller design space. The custom instruction solutions for the smaller programs are recursively combined to gradually converge into a solution for the original program. For each custom instruction template, we first perform some local optimizations (block **10**). Then we mark the level in the hierarchy at which each function resides (block **11**). We start from the leaf functions (block **12**), dynamically compute the area and number of cycles saved for all the templates (block **13**), and then compute the target area for the templates for the current function and its subgraph (block **14**). Based on the target area, we scale the custom instruction templates in the function subgraph (block **15**). If all the functions at the same level have finished computation, we ascend levels in the function hierarchy and repeat the computation (block **16**). Finally, when we reach the root function, a set of custom instruction templates satisfying the area constraints has been selected (block **17**). A more detailed explanation for these steps is given in Section III-B.

Phase III performs more software transformations (block **18**) including optimizations like loop-invariant motion, and generates the custom instruction descriptions (block **21**). The original program is also modified through insertion of custom instructions (blocks **19** & **20**). Finally, the custom processor, which includes the base processor and custom instructions, is built, synthesized and verified (block **22**).

B. Details

In this section, we describe the steps in Phase II in detail. First, we mathematically formulate the problem using a flattened hierarchy in Section III-B.1 (note that this formulation is only applicable to small programs). Then we present a global exploration algorithm to reduce the design space using the program hierarchy in Section III-B.2). A detailed explanation of the cost function and other important steps in the algorithm are presented in Sections III-B.3, III-B.4, and III-B.5.

B.1 Problem formulation

Our final objective is to solve Problem 1. To have a clearer understanding of this problem, we define a custom instruction more rigorously.

Definition 1: A custom instruction (Ins) can be represented as a directed acyclic graph (DAG), $G = (V, E)$. Node $v_i \in V$ represents an operation. Edge $e_k \in E$ represents the data dependence between two nodes v_i, v_j ($e_k = (v_i \xrightarrow{s_l} v_j)$). v_i is called the data dependence predecessor of v_j (abbreviated to predecessor), while v_j is called the data dependence successor of v_i (abbreviated to successor). s_l is the symbol that connects v_i and v_j. There are two special nodes, source (v_{source}) and drain (v_{drain}). v_{source} is the predecessor of all input nodes, and v_{drain} is the successor of all output nodes. In and Out are the input and output sets of the custom instruction, respectively. $In = \{s_l | e_k = (v_{source} \xrightarrow{s_l} v_j)\}$, $Out = \{s_l | e_k = (v_i \xrightarrow{s_l} v_{drain})\}$. $|In|$ and $|Out|$ represent the number of inputs and outputs, respectively.

Definition 2: Each node $v_i \in V$ has two attributes: cycles (c_i), representing the number of cycles required to compute the operation in the base processor, and area (a_i), representing the functional unit area of the operation if it is part of a custom instruction.

Definition 3: Each custom instruction (Ins) has three attributes: execution count (EC), representing the number of times the custom instruction is executed, cycles saved (CS) and area (A). They are computed using the following equations:

$$CS = \sum_{v_i \in V} c_i - C, \tag{1}$$

$$C = SC + \max(|In| - \alpha, 0) + \max(|Out| - \beta, 0), \tag{2}$$

$$A = BA + \sum_{v_i \in V} a_i, \tag{3}$$

In Definition 3, Equation (1) computes the cycles saved in each execution. The first term gives the number of cycles needed on the base processor, and the second term gives the number of cycles needed using the custom instruction. Equation (2) shows how the second term is computed. Its first term, SC, denotes the number of scheduled cycles used by the custom instruction, which can be obtained by comparing the longest delay of the custom instruction and the processor clock period. The second and third terms give the extra cycles needed to write and read values to and from the user-defined

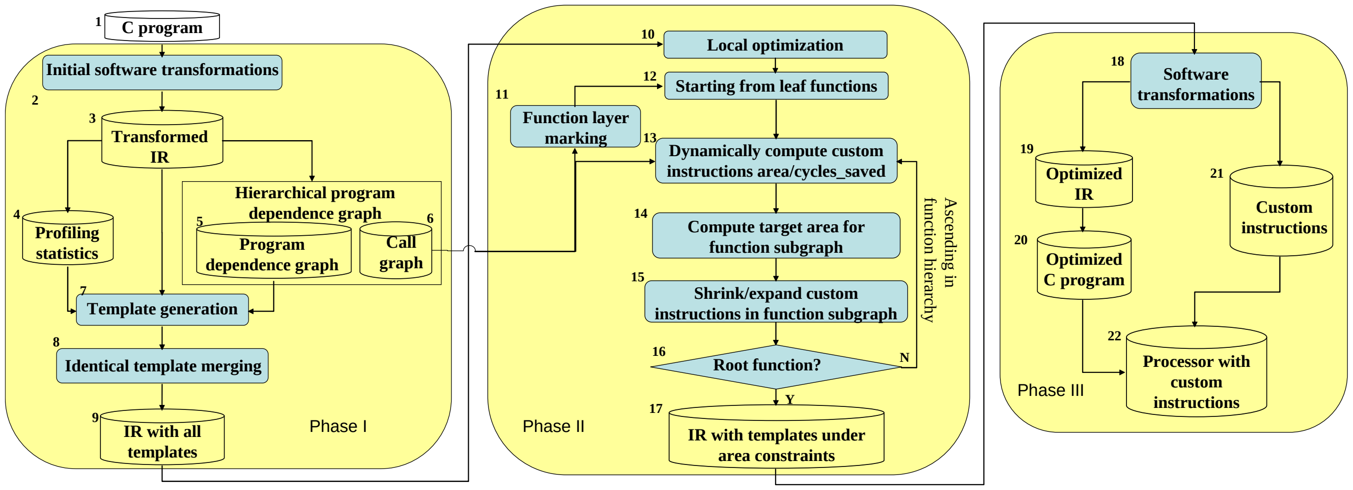

1 C program
2 Initial software transformations
3 Transformed IR
4 Profiling statistics
Hierarchical program dependence graph
5 Program dependence graph
6 Call graph
7 Template generation
8 Identical template merging
9 IR with all templates
Phase I
10 Local optimization
11 Function layer marking
12 Starting from leaf functions
13 Dynamically compute custom instructions area/cycles_saved
14 Compute target area for function subgraph
15 Shrink/expand custom instructions in function subgraph
16 Root function?
N
Y
17 IR with templates under area constraints
Ascending in function hierarchy
Phase II
18 Software transformations
19 Optimized IR
20 Optimized C program
21 Custom instructions
22 Processor with custom instructions
Phase III

Fig. 2. Custom instruction generation design flow for hierarchical programs

registers, assuming the custom instruction can encode α and β inputs and outputs in the instruction, respectively. The extra inputs have to be written to some user-defined registers before computation. These write instructions take extra cycles. Similar arguments hold for outputs. Equation (3) computes the area of the custom instruction. It is composed of two parts. BA denotes the base area of the custom instruction, required for decoding and other interface logic. The second term gives the functional unit area.

The total area of all the n custom instructions can be computed as follows:

$$AREA = \sum_{1 \le i \le n} A_i + SA, \tag{4}$$

$$SA = (\max_{1 \le i \le n}(|In_i - \alpha|) + \max_{1 \le i \le n}(|Out_i - \beta|)) \times URegArea, \tag{5}$$

Because the area to implement the user-defined register can be shared across custom instructions, we include it as a separate item for all the custom instructions. SA represents the area for user-defined registers, which is the sum of the maximum number of extra inputs and maximum number of extra outputs times unit user-defined register area, $URegArea$. The total area is the sum of the area of each custom instruction and the area of user-defined registers.

Now, Problem 1 can be restated as:

Problem 2: Given a base processor, an application, and an area budget AB, find a set of custom instructions $\{Ins_i\}$ such that $\sum_i CS_i \times EC_i$ is maximized while $AREA \le AB$.

After Phase I, a number of custom instructions $\{Ins_i\}$ becomes available, where typically $\sum_i A_i + SA > AB$. Thus, we need to select a subset of the custom instruction nodes. The problem can now be further refined as follows:

Problem 3: Given a base processor, an application, and an area budget AB, suppose n custom instructions $\{Ins_i\}$ are found and there are m_i operations in custom instruction i. We need to find an assignment of values to variables x_{ij} for node j of custom instruction i ($1 \le i \le n, 1 \le j \le m_i$), in order to maximize the following cost function:

$$CS = \sum_i CS_i \times EC_i, \tag{6}$$

where

$$CS_i = \sum_{v_{ij}} x_{ij} c_{ij} - C_i, \tag{7}$$

$$C_i = SC_i + \max(|In'_i| - \alpha, 0) + \max(|Out'_i| - \beta, 0), \tag{8}$$

$$In'_i = \{s_{im} | e_{il} = (v_{ij} \xrightarrow{s_{im}} v_{ik}), x_{ij} = 0, x_{ik} = 1\}, \tag{9}$$

$$Out'_i = \{s_{im} | e_{il} = (v_{ij} \xrightarrow{s_{im}} v_{ik}), x_{ij} = 1, x_{ik} = 0\}, \tag{10}$$

subject to

$$AREA \le AB, \tag{11}$$

where

$$AREA = \sum_i A_i + SA, \tag{12}$$

$$A_i = BA_i + \sum_{v_{ij}} x_{ij} a_{ij}, \tag{13}$$

$$SA = (\max_i(|In'_i - \alpha|) + \max_i(|Out'_i| - \beta)) \times URegArea, \tag{14}$$

$$x_{ij} = \{0, 1\}, \tag{15}$$

x_{ij} is 0 iff any node v_{rs} in any path from v_{ij} to v_{idrain} or from v_{ij} back to $v_{isource}$ has $x_{rs} = 0$.

B.2 Global exploration

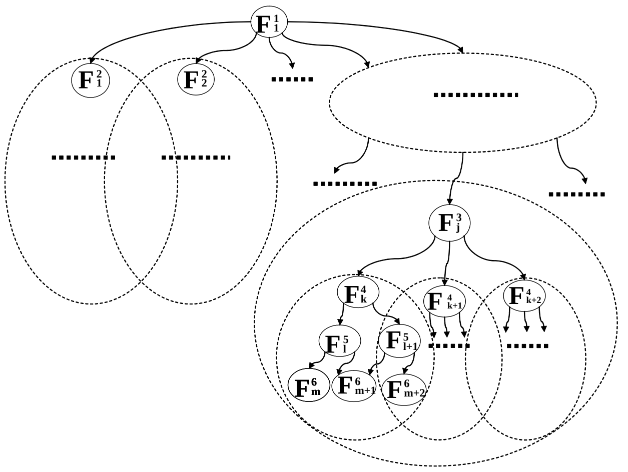
Obviously, Problem 3 is an NP-hard problem. Hence, it is not practical to use a flattened approach to solve the problem for large hierarchical programs, especially when $a_{ij} << AB << AREA$. We make use of the program hierarchy to recursively solve Problem 3 for sub-programs of the program, and finally converge to the global solution.

For each template, we first perform local optimization by dividing the template if the divided template can yield better performance and smaller area. We also perform some initial pruning of the templates. Then we break the loops in the call graph to form a DAG. We mark the levels of all the functions in the call graph, starting from the root function (main) and marking it as 1, then marking all the functions it calls as 2, and so on. Basically, a function is marked as being at level $n+1$ if the highest level of the caller functions is n.

Starting from the function at the first level, we compute the total number of cycles saved (CS) and area ($AREA$) for all the templates passed from Phase I, using Equations (6) and (12), respectively, disregarding the area constraint for now.

For each function at the current level, we again compute the total number of cycles saved (CS') and area ($AREA'$) for all the templates in the function subgraph, *i.e.*, functions directly or indirectly called by the current function. Then we compute the target area and solve Problem 3 by scaling all the templates in the function subgraph to meet the target area.

After all the functions in the current level have been processed, we decrease the level number and perform the computation again. Note that a lower level corresponds to a larger function subgraph, resulting in a larger design space search. The previous custom instruction node selection step just provides an initial solution for the current search.



(a)

Function	F_k^4	F_l^5	F_{l+1}^5	F_m^6	F_{m+1}^6	F_{m+2}^6
Cycles	6,000	41,000	9,000	56,000	14,000	24,000
Custom instr	none	Ins_1	none	Ins_2	Ins_3	Ins_4

(b)

Custom instr	Saved cycles	Area	Custom Reg.
Ins_1	13,000	18,500	2
Ins_2	53,950	36,500	4
Ins_3	10,000	14,000	1
Ins_4	50	21,000	0

(c)

Fig. 3. Illustration of global exploration: (a) call graph, (b) cycle consumption and custom instruction in each function, and (c) custom instruction properties

The following example illustrates the above algorithm.

Example 3: Fig. 3(a) depicts part of a labeled call graph of a hierarchical program. Each node is a function. The dotted ovals denote the function subgraphs. The superscript of a node represents the level number of the function, and the subscript represents the functions in the corresponding level. Fig. 3(b) shows the number of cycles and custom instruction for a function subgraph. Fig. 3(c) shows the custom instruction properties.

Suppose the target area for an entire program is 100,000 grids and initially all the custom instruction templates combined can save 200,000 cycles and consume 200,000 grids of area. During local optimization, the local target area for Ins_1 to Ins_4 are computed as 26,000, 107,900, 20,000, and 100, respectively. Hence, Ins_4 is shrunk to 0. After local optimization, suppose the area is reduced to 150,000 grids and the number of cycles saved is reduced to 160,000 cycles.

At level 6, for function F_m^6, the target area of Ins_2 is computed to be 38,914, larger than the one giving the best performance. Hence, it is not changed. For function F_{m+1}^6, the target area of Ins_3 is computed to be 8,854. Hence, the template has to shrink to occupy an area that is smaller than this value. For simplicity, let us assume that the template can scale exactly to the target area. The number of cycles saved is also reduced to 2,000 cycles due to the shrinkage. For function F_{m+2}^6, the target area is still 0. After all functions are processed, the total area is reduced to 130,000 and the total saved cycles is reduced to 140,000.

At level 5, for the function subgraph rooted at F_l^5, the total saved cycles is 68,950, and the total area for implementing Ins_1, Ins_2 and Ins_3 is 56,254. Note that this is smaller than simply adding the respective areas because of sharing of user-defined registers. The target area is now computed to be 61,070. Hence, the templates are expanded again to meet the target area. For the function subgraph rooted at F_{l+1}^5, the target area is 8,857 and hence Ins_3 is shrunk again. After all the functions are processed, the area becomes 125,000 due to the reduction in some other templates, and the total saved cycles is reduced to 133,000.

At level 4, for the function subgraph rooted at F_k^4, again the total saved cycles is 68,950, and the total area is 56,257. The computed target area is 61,562. Hence, the instructions are expanded again. This continues until the area constraint is met. ∎

B.3 Custom instruction partitioning

In Phase I of Fig. 2, we compute the largest custom instructions in a basic block, but it may not necessarily be the best candidate in terms of performance. If the custom instruction has more than α inputs and β outputs, some cycles are needed to write/read extra values to/from user-defined registers. If the number of inputs or outputs is too large, the `write` or `read` instruction will dominate the number of cycles required to execute the custom instruction. However, if the large custom instruction can be divided into several smaller custom instructions, and the execution cycles of the large custom instruction is greater than the sum of the execution cycles of the small custom instructions, the division can improve performance without significant cost. Usually, if this is beneficial, the total area for the divided custom instructions is smaller too, because of area sharing of the user-defined registers. This method can reduce the critical path as well, as shown next.

Example 4: Suppose one large custom instruction has three inputs (In_1, In_2, In_3) and two outputs (Out_1, Out_2), uses two user-defined registers, one for the input and one for the output, and takes three cycles. The code is shown below. Here, $WUserRegister$ ($RUserRegister$) refers to writing (reading) of a user-defined register.

$$WUserRegister1(In_3);$$
$$Out_2 = CustomInstr(In_1, In_2);$$
$$Out_1 = RUserRegister2();$$

If the above custom instruction is divided into two smaller custom instructions, as shown below, they will not use any user-defined registers and take only two cycles in all.

$$Out1 = CustomInstr1(In_1, In_2);$$
$$Out_2 = CustomInstr2(In_3, Out_1);$$

Thus, one cycle is saved for each execution of the custom instruction. ∎

The above example illustrates that sometimes it is beneficial to divide a large custom instruction into two or more smaller custom instructions.

B.4 Target area computation

Both in the local optimization and global exploration steps, we need to compute the target area, and scale templates to meet the target area. The modified templates form the new initial solution for the next level. Several factors need to be considered to compute the target area:

- The target area that the template occupies should be reasonable. For example, a template contributing 1% of the total cycles saved cannot be allowed to occupy 10% of the total budgeted area, if that area can be used for some other templates that can contribute to more saved cycles.
- The target area should reflect the function hierarchy structure and reduce the overall area gradually. Each level just provides an initial solution for the next level. It is better to consistently change the area by a little amount at each level, so that the new solution can converge quickly.

Based on the above factors, we first compute the local target area for each template, and shrink the template correspondingly. The local target area (LTA) is computed using the following function:

$$LTA = RF \times \frac{CS'}{CS} \times AREA \tag{16}$$

where RF is a constant relaxation factor. It is usually greater than one. Its purpose is to relax the area budget so that the custom instructions are not pruned too aggressively at the very beginning. For example, suppose the RF is set to 10, then only the custom instructions consuming more than 10 times the area they deserve are pruned. In some programs, some custom instructions only save a few cycles while consuming a significant portion of the area. They should be pruned at the very beginning. CS' is the number of cycles saved by the current template and CS is the number of cycles saved by all the templates. Since in the global exploration phase, some templates will be considered earlier than others, local optimization provides a relatively good initial solution for the next phase.

In the global exploration phase, at level i, we compute the target area based on the following function:

$$TA = \frac{CS'}{CS} \times (AREA - \frac{1}{i}(AREA - AB)) \tag{17}$$

CS' here represents the number of cycles saved by templates in the current function subgraph. The first term in Equation (17) reflects the first aspect we considered above. Ideally, the percentage of the area each template occupies should correspond to the percentage of the number of cycles saved by the template. The second term in this equation reflects the gradually reducing area. There are i steps between levels i and 1. Each time the area is reduced a little bit, *i.e.*, by $\frac{1}{i}(AREA - AB)$. The custom instructions area is gradually reduced to the area budget.

B.5 Custom instruction scaling

After the target area has been computed, all the templates in the function subgraph are shrunk or expanded to meet the target area in order to solve Problem 3. This can be done independently of global exploration. We can choose from among various approaches for this purpose, such as simulated annealing, genetic algorithm, iterative improvement or a greedy approach.

The aim of the hierarchical synthesis approach is to reduce the design space. In the beginning of this approach, the design space is small, whereas at the end of the approach, the design space is large. However, since we have a reasonably good initial solution, we do not need to use very complex heuristics to solve Problem 3. The initial solution gets refined by repeated computation with an increasingly tighter target area constraint. Thus, instead of choosing simulated annealing or genetic algorithm, we chose a simple greedy algorithm with some relaxation. As demonstrated in the experimental results section, this gave very good results.

In the algorithm, we first compare the area ($AREA'$) of the initial solution and the target area (TA). If the actual area is larger than the target area, we start deleting nodes. Otherwise, we start adding nodes. For node deletion, we first try all deletion possibilities, compute the area and cycles saved for all the templates in the subgraph, and choose the one that has the highest ratio of cycles saved over area. Note that with node deletion, the area is always smaller in the long run. If in one try, however, the area is smaller than the area budget, we compare it with the best solution computed so far. If the number of cycles saved is more than that for the best solution or they are equal but its area is smaller, we update the best solution.

For node addition, we first try all addition possibilities that lead to performance improvement, compute the area and cycles saved, and choose the one that has the highest ratio of cycles saved over area. We also keep track of the best solution computed thus far. If a better solution is found, we add the node and update the solution.

IV. EXPERIMENTAL RESULTS

We have implemented the design flow given in Fig. 2 using SUIF [24], which is a free, open source compiler research tool. Its intermediate format is platform-independent and can be transformed back to high level language (C). A number of optimizations (passes) can be applied to the intermediate format and the output written in the same format. Because of the modular property, designers can select, and order the optimizations very easily. They can also write their own passes for specific purposes. We wrote a number of passes to generate, select and optimize the custom instructions.

We calibrated the area and delay of a library of register-transfer level (RTL) components and used this information to estimate the area of custom instructions quickly in the synthesis loop (note that the final area/delay results are reported based on accurate lower-level tools). We used the simple profiling system in SUIF [25] to obtain profiling statistics (block **4** in the flow of Fig. 2), program dependence graph generator [26] to get the program dependence graph (block **5**), and some code transformation passes over SUIF to transform the code (blocks **2, 3, 6, 9, 18** and **20**). We also implemented some passes to perform some custom instruction-specific transformations (blocks **2, 7, 8, 9, 17, 18, 19** and **21**).

We verified our implementations using the Xtensa [1] platform from Tensilica. Its architecture is designed from scratch to be customizable. The designer can select a number of architecture options to enhance the base processor core. The designer can also extend the base processor by writing some custom instructions to speed up some specific applications. A complete GNU-based software tool suite and hardware synthesis and verification scripts are automatically generated to match the configuration specified in the configuration generator.

We used a GNU-based compiler to compile the program with and without custom instructions, instruction set simulator (ISS) to get the execution cycles, Tensilica Instruction Extension (TIE) compiler to convert custom instruction descriptions to RTL Verilog code, and Synopsys Design Compiler [27] to synthesize the RTL Verilog code and map it to a commercial 0.18μ technology library [28].

We evaluated our flow using six benchmarks. `AES` refers to

TABLE II

COMPARISON OF PERFORMANCE AND AREA USING OUR TOOL

Program	Execution time (ms) Original	Execution time (ms) New	Performance improvement	Processor area ($grids$) Original	Processor area ($grids$) New	# Custom instructions	Synthesis time (s)
`AES`	9.56	6.16	1.55X	433389	469184	3	67.6
`DES`	19.3	8.98	2.15X	433389	453684	3	5.0
`MD5`	9.13	5.62	1.62X	433389	622982	5	3.9
`SHA`	4.54	2.90	1.57X	433389	568852	4	6.3
`adpcm`	3.54	1.46	2.42X	433389	502435	1	3.8
`dct`	14.92	2.36	6.32X	433389	818448	2	74.3

the advanced encryption standard algorithm. `DES` refers to the data encryption standard algorithm. `MD5` denotes a message digest algorithm. `SHA` denotes a secure hash algorithm. `adpcm` refers to adaptive differential pulse code modulation. Finally, `dct` denotes a discrete cosine transform algorithm. We ran our tool on a 1.2GHz Pentium III processor with 256MB memory.

Table II summarizes the results of our experiments. It compares the execution time of the benchmarks on the original base processor, and the customized processor (with custom instructions augmenting the base processor as generated by our tool). The clock period for both processors is 6.5ns. We also include the area overhead due to the addition of the custom instructions, the number of custom instructions selected, and the time required by our tool to generate and select the custom instructions. The number of execution cycles is reported by a cycle-accurate ISS. The area is obtained from gate-level synthesis of the entire processor. Custom instruction synthesis using our tool usually takes less than a minute. Verification based on simulating and profiling the modified program on the customized processor using the Xtensa [1] software tool chain, and synthesizing the entire customized processor using Design Compiler [27] usually take several hours. The results indicate that our methodology can quickly (in several seconds to a minute) generate and prune custom instructions for realistic programs. The selected custom instructions can achieve an average speedup of 2.61X and peak speedup of 6.32X over the original processor.

V. CONCLUSIONS

Custom processor synthesis is quite challenging for real-life applications because of the complex interdependencies and tradeoffs involved in making the design decisions. In this paper, we use the modular and hierarchical structure of the program and provide a scalable methodology for application-specific processor synthesis. We have integrated our design flow in an open source compiler and verified it on a commercial extensible processor. Our experiments demonstrate that our hierarchical approach can effectively tackle large programs, achieving significant performance improvement in very limited synthesis times.

References

[1] *Xtensa microprocessor*, Tensilica Inc. (`http://www.tensilica.com`).

[2] *Jazz DSP*, Improv Systems Inc. (`http://www.improvsys.com`).

[3] *DSP core*, 3DSP Corp. (`http://www.3dsp.com`).

[4] *ARCtangent processor*, Arc International (`http://www.arc.com`).

[5] J. A. Fisher, "Customized instruction sets for embedded processors," in *Proc. Design Automation Conf.*, June 1999, pp. 253–257.

[6] K. Keutzer, S. Malik, and A. R. Newton, "From ASIC to ASIP: The next design discontinuity," in *Proc. Int. Conf. Computer Design*, Sept. 2002, pp. 84–90.

[7] D. Fischer, J. Teich, M. Thies, and R. Weper, "Efficient architecture/compiler co-exploration for ASIPs," in *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, Oct. 2002, pp. 27–34.

[8] D. Kirovski, C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Application-driven synthesis of core-based systems," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1997, pp. 104–107.

[9] K. Kucukcakar, "An ASIP design methodology for embedded systems," in *Proc. Int. Symp. HW/SW Codesign*, May 1999, pp. 17–21.

[10] M. Knieser and C. Papachristou, "COMET: A hardware-software codesign methodology," in *Proc. European Design Automation Conf.*, Sept. 1996, pp. 178–183.

[11] S. P. Seng, W. Luk, and P. Y. K. Cheung, "Flexible instruction processors," in *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, Nov. 2000, pp. 193–200.

[12] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist, and M. Sivaraman, "PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators," Tech. Rep., HP Laboratories, Palo Alto, 2001.

[13] C. Alippi, W. Fornaciari, L. Pozzi, and M. Sami, "A DAG-based design approach for reconfigurable VLIW processors," in *Proc. Design Automation & Test Europe Conf.*, Mar. 1999, pp. 778–779.

[14] H. Choi, J. H. Yi, J.-Y. Lee, I.-C. Park, and C.-M. Kyung, "Exploiting intellectual properties in ASIP designs for embedded DSP software," in *Proc. Design Automation Conf.*, June 1999, pp. 939–944.

[15] N. Moreano, G. Araujo, Z. Huang, and S. Malik, "Datapath merging and interconnection sharing for reconfigurable architectures," in *Proc. Int. Symp. System Synthesis*, Nov. 2002, pp. 38–43.

[16] T.V.K Gupta, R. Ko, and R. Barua, "Compiler-directed customization of ASIP cores," in *Proc. Int. Symp. HW/SW Codesign*, May 2002, pp. 97–102.

[17] M. Arnold and H. Corporaal, "Designing domain-specific processors," in *Proc. Int. Symp. HW/SW Codesign*, Apr. 2001, pp. 61–66.

[18] H. Choi, J.-S. Kim, C.-W. Yoon, I.-C. Park, S. H. Hwang, and C.-M. Kyung, "Synthesis of application specific instructions for embedded DSP software," *IEEE Trans. Computers*, vol. 48, no. 6, pp. 603–614, June 1999.

[19] J. E. Lee, K. Choi, and N. Dutt, "Efficient instruction encoding for automatic instruction set design of configurable ASIPs," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 2002, pp. 649–654.

[20] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "Synthesis of custom processors based on extensible platforms," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 2002, pp. 641–648.

[21] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Proc. Design Automation Conf.*, June 2003, pp. 256–261.

[22] N. Cheung, J. Henkel, and S. Parameswaran, "Rapid configuration & instruction selection for an ASIP: A case study," in *Proc. Design Automation & Test Europe Conf.*, Mar. 2003, pp. 802–807.

[23] N. Clark, W. Tang, and S. Mahlke, "Automatically generating custom instruction set extensions," in *Proc. Wkshp. Application Specific Processors*, Nov. 2002.

[24] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. A. M. Anderson, S. W. K. Tjiang, S. W. Liao, C. W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: An infrastructure for research on parallelizing and optimizing compilers," *SIGPLAN Notices*, vol. 29, no. 12, pp. 31–37, 1994.

[25] T. Callahan and J. Wawrzynek, "Simple profiling system for SUIF," in *Proc. The First SUIF Compiler Wkshp.*, Jan. 1996.

[26] J. B. Fenwick Jr. and L. L. Pollock, "Implementing an optimizing Linda compiler using SUIF," in *Proc. The First SUIF Compiler Wkshp.*, Jan. 1996.

[27] *Design Compiler*, Synopsys Inc. (`http://www.synopsys.com`).

[28] *CB-11 Cell Based IC Product Family*, NEC Electronics, Inc. (`http://www.necel.com`).