Synthesis of Custom Processors based on Extensible Platforms

Fei Sun†, Srivaths Ravi‡, Anand Raghunathan‡, and Niraj K. Jha†
† Department of Electrical Engineering, Princeton University, Princeton, NJ 08544
‡ C&C Research Labs, NEC USA, Princeton, NJ 08540

Abstract— **Efficiency and flexibility are critical, but often conflicting, design goals in embedded system design. The recent emergence of extensible processors promises a favorable tradeoff between efficiency and flexibility, while keeping design turnaround times short. Current extensible processor design flows automate several tedious tasks, but typically require designers to manually select the parts of the program that are to be implemented as custom instructions.**

In this work, we describe an automatic methodology to select custom instructions to augment an extensible processor, in order to maximize its efficiency for a given application program. We demonstrate that the number of custom instruction candidates grows rapidly with program size, leading to a large design space, and that the quality (speedup) of custom instructions varies significantly across this space, motivating the need for the proposed flow. Our methodology features cost functions to guide the custom instruction selection process, as well as static and dynamic pruning techniques to eliminate inferior parts of the design space from consideration. Further, we employ a two-stage process, wherein a limited number of promising instruction candidates are first selected, and then evaluated in more detail through cycle-accurate instruction set simulation and synthesis of the corresponding hardware, to identify the custom instruction combinations that result in the highest program speedup or maximize speedup under a given area constraint.

We have evaluated the proposed techniques using a state-of-the-art extensible processor platform, in the context of a commercial design flow. Experiments with several benchmark programs indicate that custom processors synthesized using automatic custom instruction selection can result in large improvements in performance (upto 5.4X, average of 3.4X), energy (upto 4.5X, average of 3.2X), and energy-delay product (upto 24.2X, average of 12.6X), while speeding up the design process significantly.

I. INTRODUCTION

Efficiency and flexibility are two of the most important driving factors in embedded system design. Efficient implementations are required to meet the tight cost, timing and power constraints present in embedded systems. Flexibility, albeit tough to quantify, is equally important — it allows system designs to be easily modified or enhanced in response to bugs, evolution of standards, market shifts, or user requirements, during the design cycle and even after production. Various implementation alternatives for a given function, ranging from custom-designed hardware to software running on embedded processors, provide a system designer with differing degrees of efficiency and flexibility. Unfortunately, it is often the case that these are conflicting design goals. While efficiency is obtained through custom hardwired implementations, flexibility is best provided through programmable implementations. Hardware/software partitioning — separating a system's functionality into embedded software (running on programmable processors), and custom hardware (implemented as co-processors or peripheral units) — is one approach to achieve a good balance between flexibility and efficiency [1]. However, the increasing scale and complexity of embedded Systems-on-Chips (SOCs), together with decreasing market cycles, provide a constant push towards increasing embedded software content.

While it is known that application-specific instruction set processors (ASIPs) provide a good tradeoff between efficiency and flexibility, their relatively large design turnaround times (compared to software implemented on pre-designed and pre-verified embedded processors) have, in part, prevented their wide use in SOCs. The recent availability of configurable and extensible processors [1] promises a favorable tradeoff between efficiency and flexibility, while keeping design turnaround times short. The emergence and success of companies that offer configurable and extensible embedded microprocessor and DSP cores as a primary product (*e.g.*, Xtensa from Tensilica [3], ARCtangent from ARC [4], Jazz from Improv Systems [5], SP-5flex from 3DSP [6], Carmel DSP from Siemens [7], *etc.*) testifies to the benefits of this approach.

Realizing the potential of extensible processors requires the development of supporting tools and methodologies that enable system designers to achieve design turnaround times that are comparable to all-software solutions. State-of-the-art extensible processor design flows automate several tedious tasks and aid the design process by providing a high-level interface to perform processor customization, automatic generation of register-transfer level (RTL) hardware descriptions, a re-targetable software tool chain for the customized processor (including compilers, assemblers, debuggers, and binary utilities), and design automation scripts to support verification, logic synthesis, and physical design. However, they *require designers to manually select parts of the program and design the custom instructions that implement them. As shown in this paper, these are daunting tasks for large programs, and are further complicated when performance improvements need to be attained subject to other considerations such as constraints on hardware area overheads and clock period.* Our work addresses the above need by providing a systematic methodology and automation algorithms for the application-specific customization of extensible processors.

A. Related Work

We briefly trace related research work along the lines of ASIP architectures and overall design methodologies, application-specific instruction set selection, compilation techniques for ASIPs, and low power ASIP design.

A good overview of the benefits and challenges involved in ASIP design is contained in [8]. Early works on architectural synthesis for ASIPs are contained in [9], [10]. In [11], a design system for synthesizing simple pipelined ASIPs is described. The work in [12] deals with the selection of intellectual property (IP) surrounding an ASIP core to accelerate application programs. A method to design parallel and scalable ASIP architectures, suitable for reactive systems, is presented in [13]. In [14], an early design space exploration methodology is given for clustered very long instruction word (VLIW) data paths in the context of specific target appli-

Acknowledgments: This work was supported in part by DARPA under contract no. DAAB07-00-C-L516 and in part by the NJCST Center on Embedded System-on-a-Chip Design.

[1]Following the terminology used in [2], a configurable processor offers the designer the possibility to customize the basic processor architecture, while an extensible processor allows the designer to extend the basic instruction set through the addition of custom application-specific instructions.

cations. Automatic architectural synthesis of VLIW processors is targeted in [15]. Design of area-efficient hardware blocks of an ASIP is tackled in [16], [17]. In [18], an ASIP design methodology is given for customizing an existing processor instruction set and architecture.

Instruction set selection refers to the problem of defining a custom instruction set that will result in the most efficient processing for a given application or domain. The work in [19] presents a technique to generate multi-cycle application specific instructions for DSP applications. Instruction set design and selection are treated as a scheduling problem in [20], and as a module selection or operation coupling problem in [11], [21]. ASIP instruction set optimization under functional unit sharing constraints is addressed in [22]. A hardware-software co-design approach for instruction set selection is proposed in [23]. A method to automatically detect recurring operation patterns to obtain custom instructions is presented in [24].

Efficient retargetable compilers and software tool chains are necessary to enable the use of ASIPs. These are discussed in [25].

Recent work has also focused on reducing power consumption in ASIPs. A concept of instruction sub-setting is introduced in [26] to create an ASIP from a more general processor, such as a DSP. In [27], a low power ASIP is synthesized from a customized ASIC using power estimation techniques derived from high-level synthesis. Synthesis of power-efficient hypermedia processors is addressed in [28]. Case studies of power optimization of ASIP cores are presented in [29], [30].

B. Paper Overview and Contributions

The contributions of this work include:

- Instead of designing an entire instruction set, we propose an automatic methodology for the selection of custom instructions to augment an extensible processor in order to maximize its efficiency for a given application program [2].
- We demonstrate the need for such a methodology by illustrating the size and complexity of the custom instruction design space, and present an analysis of the issues and tradeoffs involved in instruction set extension.
- We develop cost functions and pruning techniques to guide the custom instruction selection process.
- Unlike most previous work, our methodology involves actual logic synthesis and cycle-accurate instruction set simulation of promising candidate custom instructions to accurately estimate performance benefits and hardware overheads.
- We not only consider every custom instruction independently, but also consider combinations of custom instructions, and choose a combination that can maximize performance under area constraints.
- We have performed experiments, in the context of a commercial design flow (using Tensilica's Xtensa), indicating that custom processors generated through the proposed methodology result in significant improvements in performance, energy, and energy-delay product. Our results are derived from post-synthesis technology-mapped netlists of the entire original and optimized processor cores.

II. MOTIVATION

Current extensible processor flows typically require designers to manually design the custom instructions that are used to augment the base processor platform. Given an application program, this involves profiling of the program source code on the base processor, identification of the most performance-critical sections of the program (the "hot spots"), formulating custom instructions by specifying new hardware resources and the operations they perform, and re-writing the source code of the program [3] to directly invoke the custom instruction. While these steps are somewhat simplified by profiling tools, and through the specification of the custom instruction hardware at a high level of abstraction, they are still daunting tasks for large programs, and are further complicated when performance needs to be optimized subject to constraints on hardware overhead.

```
/* This function swaps the order of bytes in s if
 * argument do_swap is non-zero
 */
static unsigned
BYTESWAP(unsigned s, unsigned char do_swap)
{
    unsigned ss, s1, s2, s3, s4,
             s5, s6, s7, s8, result;

    s1 = s<<24;
    s2 = s<<8;
    s4 = s>>8;
    s6 = s>>24;
    s3 = s2 & 0xff0000;
    s5 = s4 & 0xff00;
    s7 = s1 | s3;
    s8 = s5 | s6;
    ss = s7 | s8;
    /*Global count of #words processed*/
    if(do_swap) SWAPPED_COUNT++;
    result = do_swap ? ss : s;

    return result;
}
```

Fig. 1. An example function used to demonstrate the size and complexity of the custom instruction design space

The number of candidate custom instructions for a given program grows exponentially with the program size. It is not uncommon for functions with a few tens of operations to contain several hundred instruction candidates. We used the tools developed in our paper to identify all possible (unique) instruction candidates for a C function `BYTESWAP()` derived from the Tensilica training kit, which is shown in Fig. 1. As the name indicates, this function swaps the order of bytes in a word (*e.g.*, for endian conversion). The function, although quite small, contains **482** potential custom instructions! Most realistic programs contain a function hierarchy with a large number of functions. In well-optimized software, it is also often the case that there are several "critical" functions, *i.e.*, no single or small set of functions is responsible for a large fraction of the total program execution time [31]. In such scenarios, a *combination of several custom instructions* may be necessary to achieve the desired performance. Two custom instructions that appear to lead to the same speedups may result in hardware that impacts the critical path (and hence the clock period) to different extents. If the addition of a custom instruction causes a violation of the original processor's clock period, the designer can choose one of the following options: (i) reduce the amount of computation performed in the instruction, (ii) split it into multiple instructions, (iii) multi-cycle the execution of the instruction, or (iv) simply accept the clock period penalty. Different choices may be optimal, depending on various factors.

When constraints on hardware overheads are present (which is frequently the case), the designer needs to judiciously select the hardware resources employed. In fact, given a set of operations to be performed by a custom instruction, there exists an area-delay tradeoff based on the number of resources used. Similar arguments apply to storage resources (registers, lookup tables, *etc.*) used in custom instructions. *When multiple instructions need to be selected from a large set of candidates, the tradeoffs involved are complex, and can be difficult to identify manually, leading to a need for methodologies such as the one proposed in this paper.*

Example 1: In order to illustrate the variation of speedup over the set of all candidate instructions (the "design space"), we generated and evaluated all possible custom instructions for the `BYTESWAP()` function. The function iterates 10,000 times and consumes 130K cycles on the base processor core. Fig. 2 plots the execution cycle savings resulting from each of the 482 candidate instructions. The instructions are ordered according to their size, location in the source code, and relative importance in the profile

[2]Although we directly target performance improvements, we demonstrate that energy and energy-delay product are also significantly reduced.

[3]While it is conceivable that a compiler could automatically identify portions of the program to map to the new instructions, in practice, this is difficult for highly complex instructions, requiring manual re-writing or annotation of source code.

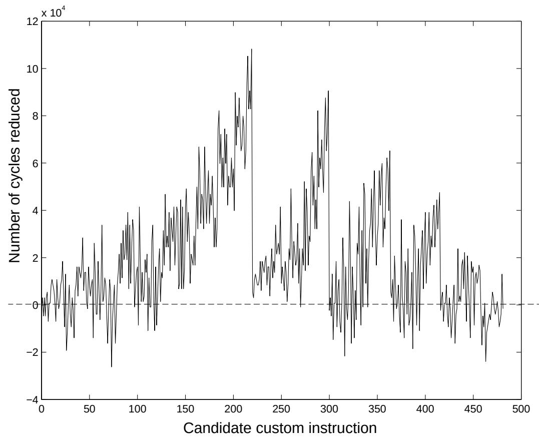


Fig. 2. Performance variation across the custom instruction design space for the BYTESWAP() example

- metrics that a designer would typically use to examine candidate custom instructions. Fig. 2 indicates that there is a large variation in quality across the custom instruction design space. Some custom instructions even increase the number of cycles. The figure shows that the nature of the design space is quite complex, underlining the need for automatic exploration techniques. ∎

Although manual creation of custom instructions allows for human ingenuity to be applied to create high-quality hardware, considering the large and complex design space of custom instructions, designers can benefit significantly from tools and methodologies to explore this design space automatically. The proposed methodology could be used to explore the large design space and short-list the most promising instruction candidates, which human designers can further refine.

III. BACKGROUND

We chose Tensilica's Xtensa [3] as our target platform, since its architecture was designed from scratch to be customizable, allowing it to be efficiently tailored to target applications, to obtain SOCs with optimized performance, power, code size and die size. Its key features are as follows:

- The Xtensa processor provides the means to select a wide range of architectural parameters in the base processor core, *e.g.*, whether to include generic instructions (*e.g.*, multiply-accumulate), floating point co-processors, configure the register file and memory/cache architecture, configure exception/interrupt mechanisms, and include test and debugging support.
- The designer can extend the base processor by designing custom instructions for application-specific computations.
- A GNU-based software tool suite is automatically generated to match the exact configuration specified in the processor generator, including a GNU C/C++ compiler, assembler, linker, debugger, diagnostics, reference test benches, cycle-accurate instruction set simulator (ISS), and standard libraries. This enables rapid design, verification, and integration of application-specific hardware and software, and removes a major bottleneck that has prevented consideration of ASIPs as the processing element of choice in SOC design.

Designers use the Tensilica Instruction Extension (TIE) language [2] to define custom instructions. The instructions can be either single-cycle or multi-cycled. Instead of invoking custom instructions at the assembly level, calls to TIE instructions can be directly inserted into high-level language (*e.g.*, C, C++) descriptions of the application program. This eases designers' burden so that they can put more emphasis on the functionality of the program and select the best instructions. The Xtensa instruction set defines a limited opcode range and encoding formats for custom instructions. TIE instructions can have at most two input and one output operand fields in the instruction. If a custom instruction needs additional inputs and/or outputs, it can implicitly read them from or write them to some internal state registers, which are defined in the TIE specification.

IV. METHODOLOGY AND ALGORITHMS

In this section, we describe our design methodology for automatically generating custom instructions to get performance improvements and/or energy reductions. We first provide an overview and then describe the important steps in detail.

A. Overview

Fig. 3 outlines the design flow of automatic custom instruction generation. It takes as input the application program to be optimized (in C), and outputs the selected custom instructions and modified C program. Given a C program as input, step **1** gen-

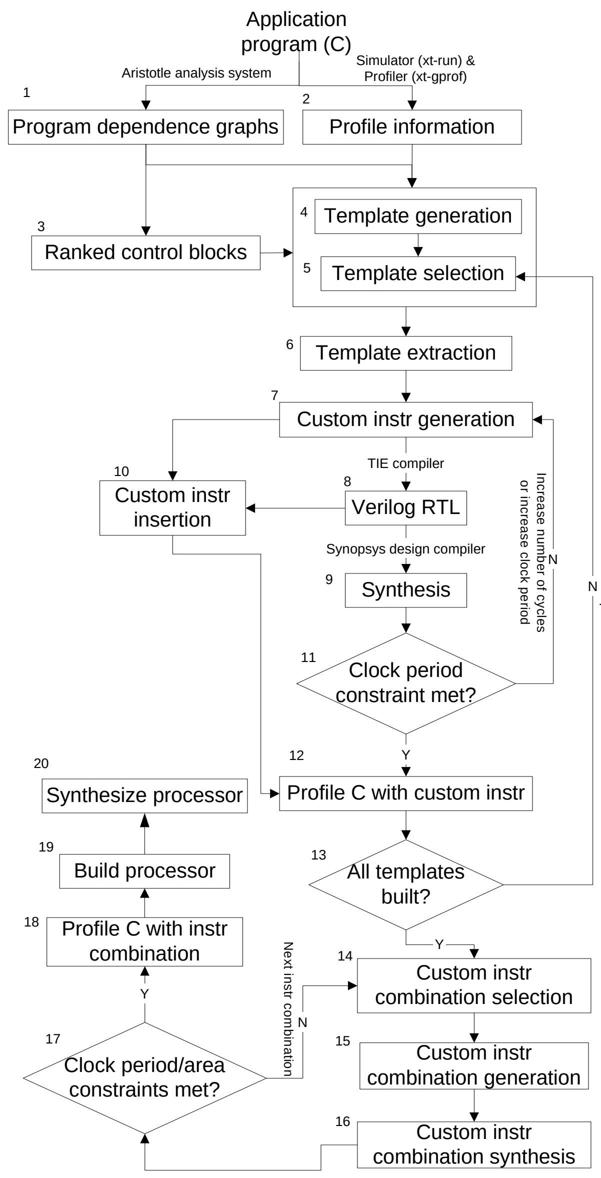

Application program (C)
Aristotle analysis system
Simulator (xt-run) & Profiler (xt-gprof)
1
Program dependence graphs
2
Profile information
3
Ranked control blocks
4
Template generation
5
Template selection
6
Template extraction
7
Custom instr generation
TIE compiler
8
Verilog RTL
10
Custom instr insertion
Synopsys design compiler
9
Synthesis
11
Clock period constraint met?
N
Increase number of cycles or increase clock period
Next template
N
Y
12
Profile C with custom instr
13
All templates built?
Y
14
Custom instr combination selection
15
Custom instr combination generation
16
Custom instr combination synthesis
17
Clock period/area constraints met?
N
Next instr combination
Y
18
Profile C with instr combination
19
Build processor
20
Synthesize processor

Fig. 3. Automatic custom instruction generation flow

erates the *program dependence graphs* [32], which include: (i) the *control dependence graph* to identify the control blocks of the program, (ii) the *data dependence graph* to get data predecessor and successor information, and (iii) the *control flow graph* to indicate how control flows through the application. At the same time, the program is simulated and then profiled both at the function level and line level, to determine where the hot spots are (step **2**). Step **3** ranks the control blocks of the program in descending order of potential for improvement. The ranking criteria may include performance (from a profiler), energy (from an energy estimator) or energy-delay product (from both), depending on the optimization objective. In this work, we focus on performance as the metric to drive custom instruction selection. However, our results (Section V) show that energy and energy-delay product are also significantly reduced in the process. A *control block* is a sequence of program statements or control blocks that can be executed sequentially. It is different from a *basic block* in that a control block can recursively contain other control blocks, while a basic block cannot contain other basic blocks. In our work, custom instructions are selected inside control blocks. However, it is possible to cross control block boundaries in the case of conditional statements that can be transformed to equivalent arithmetic/logic expressions (*e.g.*, `if` and `case` statements that can be translated into "?" and ":" operators).

Steps **4** and **5** generate and select custom instruction templates, respectively. Although these two steps are explained separately for the sake of clarity, their implementation may be combined for the sake of efficiency. A *template* is a set of program statements that is a candidate for implementation as a custom instruction. Since the number of templates grows exponentially with program size, it is necessary to prune the search space. Our pruning techniques are explained in detail in Sections IV-B.1 and IV-B.2.

Each promising candidate selected in step **5** is extracted from the C program (step **6**), and transformed to a format (TIE) that describes the opcode, operand, states, user-registers, computations, *etc.* (step **7**). It is then compiled (using Tensilica's TIE compiler [3]) to get the Verilog RTL description of the additional hardware that will augment the base processor (step **8**). The RTL description is synthesized (using Synopsys Design Compiler [33]) to get the timing and area information (step **9**). If the new instruction cannot be fit in the base processor core's clock period, either the number of cycles used by the new instruction is increased, or the clock period is increased, and the custom instruction generation phase is repeated (step **11**). Hence, for a single custom instruction template, there may be several versions with varying clock period and number of cycles. At the same time, the original C code is transformed by replacing the appropriate statements with a call to the custom instruction (step **10**). Then, for each version of the custom instruction, the new C program is compiled and profiled using a cycle-accurate ISS to get the performance improvement (step **12**). Steps **5** through **12** are iterated for every selected template.

After each individual custom instruction has been verified, a subset (combination) of instructions is chosen to get the maximum performance improvement under the given area constraint, depending on the selection criteria (step **14**). This step is described in further detail in Section IV-B.4. The hardware corresponding to the selected custom instruction combination is built and synthesized (steps **15** and **16**). If the timing and/or area constraint is not satisfied, the next best custom instruction combination is selected (step **17**). Otherwise, the modified C program is compiled and profiled again to get the final performance improvement and/or energy reduction (step **18**). After having selected the custom instruction combination, the whole processor is built and synthesized (steps **19** and **20**).

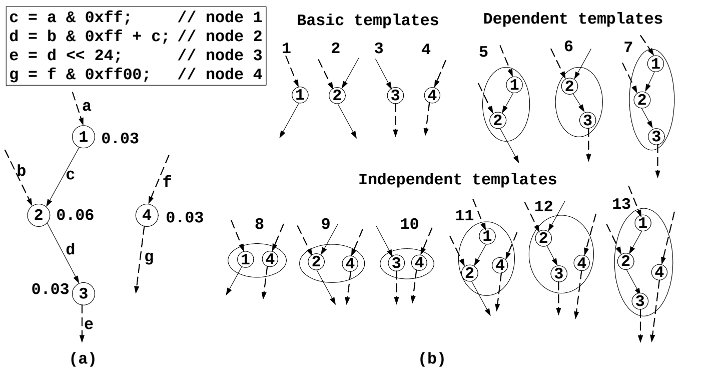
B. Details

In this section, we describe in detail the important steps of our algorithm. Section IV-B.1 describes the template generation method, Section IV-B.2 describes the template selection algorithm, Section IV-B.3 describes the instrumentation of the program to invoke the custom instructions, and Section IV-B.4 details the custom instruction combination selection method.

B.1 Template generation

Although template generation and selection are represented as distinct steps in Fig. 3, in our implementation, they are interleaved in order to improve efficiency. In the generation phase, some templates, which have low potential for performance improvement, are not generated. We refer to this as a *static pruning technique*.

We propose to generate templates in three phases. In the first phase, we can generate *basic templates*. A basic template consists of a single node in the program dependence graphs that satisfies the given selection (pruning) criteria. In the second phase, we generate *dependent templates*. A dependent template is a fully connected sub-graph of the data dependence graph. Hence, each node of a dependent template is connected to some other node in the template through a variable. Dependent templates are generated by using a basic template as a seed, checking data dependencies of the basic template, and including combinations of data dependence predecessors and successors if they satisfy the selection criteria. In the third phase, we generate *independent templates*. In this step, we use both basic and dependent templates as seeds, and add nodes that are independent of the seed template. The following example illustrates the template generation process.



```
c = a & 0xff;     // node 1
d = b & 0xff + c; // node 2
e = d << 24;      // node 3
g = f & 0xff00;   // node 4
```

Fig. 4. Illustration of the template generation process: (a) code fragment and its data dependence graph, and (b) generated templates

Example 2: Fig. 4(a) shows a small fragment of C code corresponding to a single control block, and its data dependence graph. Each node represents a single statement in the C program. Each node also has a weight that represents the fraction of the total program execution time spent in that node. The dotted lines in Fig. 4(a) indicate data dependencies with operations that belong to other control blocks. Fig. 4(b) shows all possible templates that can be generated from this graph. Nodes 1, 2, 3, and 4 form basic templates. Templates 5, 6, and 7 are generated in the dependent template generation phase. The independent template generation phase generates templates 8 through 13. It combines every basic or dependent template with templates they are independent of, *i.e.*, template 4.

Note that nodes 1 and 3 cannot be combined to form a template. To explain that, we need to consider the fact that all the statements corresponding to a template will be replaced by a single statement (call to the custom instruction) in the optimized C program, *i.e.*, they are merged into a single node in the dependence graph. In that case, node 2 will be the data predecessor of the new node (since it generates data that is used by node 3), as well as its data successor (since it uses data generated by node 1). That will introduce a data dependence cycle inside a control block, which is illegal, since it changes the program's functionality. ∎

In Example 2, we enumerated all possible templates for the sake of clarity. In practice, the number of candidate templates may potentially be very large, even for programs of moderate size. Hence, it is necessary to use pruning criteria to select good templates while discarding less promising ones. Any metric to evaluate the templates should consider the following factors:

- Amdahl's law [34] suggests that the fraction of the original program's execution time that a template accounts for presents a bound on the performance improvement achievable when it is converted into a custom instruction. Hence, templates that have a larger cumulative weight are more desirable.
- Blindly applying the first criterion would result in the degenerate solution where the largest possible template is always chosen. However, it is often the case that the largest template does not

result in the best speedup. That is in part because each template has an inherently different scope for optimization, *i.e.*, template efficiency when implemented as a custom instruction. Given two templates that account for the same fraction of total execution time on the original processor, the number of cycles required to execute them when implemented as a custom instruction is an indicator of their optimization potential.

- Many extensible processors, including the Xtensa processor, impose a limit on the number of operand fields that can be specified in the instruction format. Also, the general-purpose register file in the processor has a specific number of read and write ports, imposing a limit on the number of general-purpose registers that can be used in a custom instruction. This bottleneck can be overcome by defining custom registers (called *state or user-defined registers*) whose use is hardwired into the instruction. However, the use of state registers imposes an additional overhead. When other computations generate (or use) data that are used (or generated) by the custom instruction, the contents of the state registers need to be written to or read from either memory or the processor's general-purpose registers. The overhead for data transfer is determined by the number of "excess" input and output variables of a given instruction template.

Considering all the above factors, we use the following equation to rank candidate templates:

$$Priority = \frac{OriginalTime}{\max(In-\alpha,0)+\max(Out-\beta,0)+\gamma} \tag{1}$$

In the above equation, $OriginalTime$ is the fraction of the total execution time of the original program spent in the template, In and Out are the number of inputs and outputs of the template, respectively, α is the number of inputs that can be encoded in the instruction, β is the number of outputs that can be encoded, and γ is the number of cycles required by the template when implemented as a custom instruction. The numerator in Equation (1) is automatically computed from the line-by-line profile information. The denominator is an estimate of the number of cycles required by the custom instruction in each invocation. Since one instruction can have at most α inputs and β outputs specified in the instruction (the exact values of α and β are dependent on the processor architecture), if the number of inputs is greater than α, a cycle is needed for each additional input to load it into a user-defined state register. If the number of inputs is less than α, this term is zero. A similar explanation holds for the number of additional outputs.

It bears mentioning that the $Priority$ metric presented in Equation (1) is a coarse-grained metric, because it does not consider detailed architectural effects such as pipeline stalls. Depending on the program structure, compiler, and base processor architecture, some templates may cause pipeline delays, while others may not. Also, because of pipeline stalls, the time spent in storing values to state registers or reading values from state registers can be masked in some templates but not in others, resulting in variations in the speedup obtained by seemingly similar templates. Estimating pipeline stalls at the C program level is quite difficult, since it requires a lot of information regarding the processor's micro-architecture and compiler optimizations. However, for our purpose, an approximate metric suffices, since we employ it only as a pruning criterion, and to identify groups of promising templates. As indicated in Fig. 3, we actually evaluate the most promising templates using a cycle-accurate ISS and synthesis of the additional hardware, and select the best from among them.

Since templates having higher values of the $Priority$ metric are likely to get more performance speedup or energy reduction, we first consider those templates as seeds when generating new templates. In order to achieve this, we preserve a ranked index of the templates, and traverse the list in decreasing order of $Priority$ when choosing a seed. Further, we set a $Priority$ threshold to determine the templates considered for further analysis, those below the threshold being discarded. For the sake of computational efficiency, the threshold mechanism is dynamically enforced during template creation itself (rather than as a post-processing step). While generating templates, we preserve the highest priority ($Priority_{highest}$) seen thus far. After we generate a new template, we compute its priority and compute the ratio $\frac{Priority_{current}}{Priority_{highest}}$. If the ratio is below the threshold, we do not add it to the template list. Fig. 5 shows the relationship between the threshold ratio and the number of templates generated for an example program, `RGBtoCMYK`, which performs pixel color conversion. The number of templates displays a sharp decrease if the threshold ratio is above 0.2. If the threshold ratio is set too high, the searched design space is limited and the solution may be trapped into local optima. On the other hand, if the threshold ratio is set too low, not many templates are pruned out. In our experiments, we found that setting the threshold ratio between 0.1 and 0.15 achieved reasonable reductions in the number of templates generated, with negligible or no impact on quality.

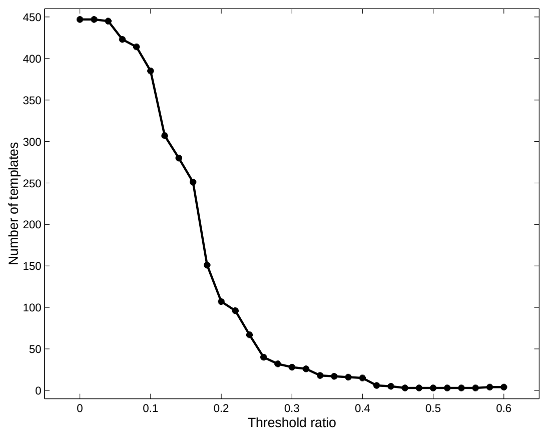


Fig. 5. Number of templates *vs.* threshold ratio

B.2 Template selection

After all the templates in a control block have been generated, the template selection step separates those templates into several groups based on ranges of the $Priority$ values (*e.g.*, all templates having $Priority$ within 5% of $Priority_{highest}$ form the first group, *etc.*). This "binning" is performed because, as mentioned earlier, the $Priority$ metric is only a coarse-grained indicator of the actual performance improvement, and hence more detailed evaluation is necessary to discern the best template from a group of templates with similar $Priority$ values. We examine templates group-by-group, and all the templates within a group are evaluated in detail (using the ISS and hardware synthesis), without regard to their exact $Priority$ value. We first attempt to generate custom instructions from all templates in the highest priority group (steps **6** to **12** of Fig. 3). If all templates from a group fail to generate a custom instruction (*e.g.*, because of timing and area constraint violations), we move to the next best group. This procedure continues until at least one template from a group succeeds in generating a custom instruction, or all groups have been tried.

Ideally, the custom instruction should fit in one cycle at the original processor core's clock period. However, sometimes the additional hardware required to implement the custom instruction results in an increase in the critical path and violates the original clock period constraint. In such situations, our tool tries two options: (i) increasing the clock period (this is equivalent to slowing down all the instructions), and (ii) increasing the number of cycles for the custom instruction until the original clock period constraint is satisfied. Specifically, we first generate the custom instruction as a single-cycle instruction, and find the smallest clock period it can be synthesized to fit in. If the clock period thus obtained is greater than the base processor core's clock period, we iteratively increase the number of cycles of the custom instruction until increasing cycle count further does not help reduce the critical path. For each number of cycles, we find the shortest clock period that can accommodate the custom instruction. Hence, each selected template may result in several *different versions of a cus-*

tom instruction, each having a different clock period and number of cycles.

For each control block in the application program, all successfully generated custom instructions are compared in terms of their actual speedup and area, and the ones that best fit the selection criteria are chosen and passed on to the subsequent phase in which instruction combinations are selected. If we only consider the number of execution cycles as the objective (without regard to area or clock period), it is only necessary to preserve disjoint templates that result in the best speedup. However, if we also consider area and clock period as constraints or objectives, we can apply the notion of Pareto optimality [35] to remove inferior or dominated templates from further consideration. For example, suppose that execution cycles and clock period are the only parameters of interest. Consider two templates, A and B, where the set of nodes that constitute A is a superset of those that constitute B. If both A and B can be implemented without increasing the base processor core's clock period, we retain A and discard B. Naturally, as more dimensions (area, energy, *etc.*) are added, Pareto optimality translates into stronger conditions that need to be satisfied to discard a candidate, resulting in a larger number of candidate instructions being passed on to the subsequent phase.

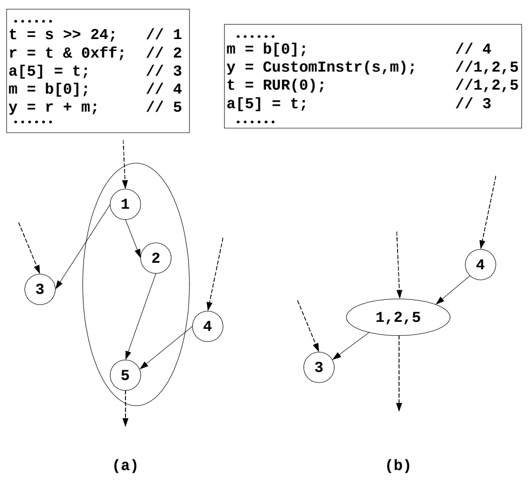
B.3 Custom instruction insertion

When a template is selected and needs to be evaluated in detail using an ISS, a call to the custom instruction needs to be inserted back into the original C program (step **10** of Fig. 3). In this process, care must be taken to preserve the functionality of the original program. A conservative approach is to place the call to the new instruction such that all control and data dependencies of the original program are satisfied. In other words, the new instruction is placed after the position of the latest data dependence predecessor, and before the position of the earliest data dependence successor. However, for some templates, no proper location for insertion can be found using only the above method, as illustrated in the following example.

Example 3: Fig. 6(a) shows an example code fragment, and its data dependence graph. As can be seen from the code, the control flow dependencies are $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$. Suppose that the template under consideration is $\{1, 2, 5\}$. Note that the template becomes a single node in the modified data dependence graph. Node 3 is the data dependence successor of the new node, and node 4 is its data dependence predecessor. In other words, the data flow after insertion of the custom instruction would be $4 \rightarrow \{1, 2, 5\} \rightarrow 3$, which is in contradiction with the control flow of the original program. However, if nodes 3 and 4 are independent, it does not matter which one executes first. If we can exchange the order of nodes 3 and 4, we find that the template can be inserted back into the C program without affecting functional correctness, as shown in Fig. 6(b). In the modified C code, one output is implicitly written to a state register. The statement `t=RUR(0)` in Fig. 6(b) assigns the value from state register 0 to variable `t`. ∎

The above example illustrated that the insertion of calls to a custom instruction may require the re-ordering of other program statements that are not part of the template itself. This requires that the data dependence information inside a control block be complete and exact. In practice, it is known that exact data dependence graph extraction is difficult when array or pointer accesses are present [36]. One solution is to have a pre-processing step, which adds code (i) at the beginning of a set of computations involving arrays and pointers, to read all required values into temporary scalar variables, and (ii) at the end, to write back temporary scalar variables into arrays or pointer contents. If the compiler is smart enough, the penalty of introducing the additional variables can be reduced to a minimum.

If the template requires state or user-defined registers, reads from and writes to state registers need to be performed explicitly in the application program. This code is also automatically generated by our tool as part of the custom instruction insertion step. A typical solution is to write program variable values into state registers before computation, and read state register contents into variables after computation. This scheme minimally interferes



```
......
t = s >> 24;   // 1
r = t & 0xff;  // 2
a[5] = t;      // 3
m = b[0];      // 4
y = r + m;     // 5
......
```

```
......
m = b[0];               // 4
y = CustomInstr(s,m);   //1,2,5
t = RUR(0);             //1,2,5
a[5] = t;               // 3
......
```

Fig. 6. Illustration of template insertion: (a) original code fragment and its data dependence graph, and (b) modified program to enable insertion

with the register assignment for general-purpose registers, which is performed by the compiler. However, the overhead associated with transfer of data to/from state registers can be minimized if we carefully select the variables that are put into state registers. We illustrate this through the following example.

Example 4: Fig. 7(a) shows a fragment of C code. Note that variable `offset` is assigned a value before the loop and is not changed during the loop. Fig. 7(b) shows the modified program after a custom instruction is inserted to perform the computation `offset + i*j`. This expression requires three input operands (`offset, i, j`), while only two operands can be read at a time from general-purpose registers. Hence, an additional (user-defined) state register is created to store one of the operands. We have three natural choices that follow. Fig. 7(b) illustrates the case where we choose to store variable `offset` in the state register. The statement `WUR(offset,0)` in Fig. 7(b) stores the value of variable `offset` in state register 0. It is better to put `offset` in the state register (and not `i` or `j`), because the value of both `i` and `j` are changed inside the loop, while `offset` is not. Most compilers can detect that the `WUR` operation is loop-invariant, and move it outside the loop. As a result, the cycle count associated with the custom instruction is reduced from two to one. A similar argument holds for the assignment of output variables to state registers. ∎

```
...
offset = t + 1;
for(i=0;i<100;i++)
         :
{        :
    j = ...
          :
          :
          :
          :
    result = offset + i * j;
}
...
```

(a)

```
...
offset = t + 1;
for(i=0;i<100;i++)
         :
{        :
    j = ...
           :
           :
    WUR(offset,0);
    result = CustomInstr(i,j);
}
...
```

(b)

Fig. 7. Illustration of the issues involved in data transfers to/from user-defined registers: (a) a fragment of C code, and (b) modified C code with custom instruction

B.4 Custom instruction combination selection

After template selection, there may still be several custom instruction candidates in the program, with each candidate having several versions (due to variations in clock period and number of clock cycles). The next step is to select a subset (or combination) of custom instruction candidates that best satisfies the performance, area and energy requirements. The inclusion of one custom instruction could either reduce or enhance the performance/energy benefits of another custom instruction. Thus, the custom instruction candidates that have survived scrutiny so far also need to be evaluated considering their inter-dependencies. Clearly, this search space is large. Hence, methods to efficiently explore the search space need to be employed. If all the selected custom instructions are non-overlapping, and if the optimization criterion is maximizing performance under an area constraint, the selection problem can be stated as follows:

Problem 1: Given a set of non-overlapping custom instructions, with each instruction having several versions (not selecting the instruction can also be considered as a degenerate version with zero area overhead and no performance benefit), find a version for each instruction such that the performance is maximized while area is under a certain threshold.

A more mathematical representation is as follows:

Problem 2: Given n non-overlapping custom instructions, custom instruction i ($1 \le i \le n$) has $m_i + 1$ versions, version j of instruction i ($0 \le j \le m_i$) has the following attributes: number of cycles SC_{ij}, clock period CP_{ij}, area overhead CA_{ij}, and cycles reduction CR_{ij}. Not selecting the instruction is assigned to version 0, with $SC_{i0} = 0$, $CP_{i0} = CP_{orig}$, $CA_{i0} = 0$, and $CR_{i0} = 0$. Find an assignment of values to variables x_{ij} for all i, j, to minimize the following cost function:

$$f = \frac{(Cycles_{orig} - \sum_{i=1}^{n}\sum_{j=0}^{m_i} x_{ij}CR_{ij}) \max_{1\le i\le n, 0\le j\le m_i} x_{ij}CP_{ij}}{Cycles_{orig}CP_{orig}} \quad (2)$$

subject to:

$$\sum_{j=0}^{m_i} x_{ij} = 1, \qquad 1 \le i \le n, \quad (3)$$

$$\sum_{i=1}^{n}\sum_{j=0}^{m_i} x_{ij}CA_{ij} \le AREA, \quad (4)$$

$$x_{ij} \textit{ is } 0 \textit{ or } 1, \qquad 1 \le i \le n, 0 \le j \le m_i \quad (5)$$

In the above equations, $Cycles_{orig}$ is the number of cycles of the original program running on the base (unaugmented) processor core. $AREA$ is the maximum total area allowed for the custom instruction hardware.

Equation (3) ensures that exactly one version of each custom instruction is chosen. Equation (4) makes sure that the area of the selected custom instructions does not exceed the maximum area constraint. The cost function (Equation (2)) is the ratio of the execution time of the program with custom instructions to the execution time of the original program. The first factor in the numerator is the number of cycles of the program with custom instructions, and the second factor is the clock period of the new processor.

Note that the above formulation is only an approximation, and makes certain assumptions about how the area overheads and clock period will behave when combinations of custom instructions are included. Specifically, we assume that the area overhead will behave in an additive manner (we can also consider additive behavior with a constant shared overhead), and that the clock period is governed by a max-function. Since this approximation may introduce some error, we find not just the best solution, but the best k ones, using a branch-and-bound algorithm, and evaluate all of them through logic synthesis.

The branch-and-bound algorithm for custom instruction combination selection works as follows. First, all custom instructions are sorted in descending order of the metric $\max_j \left(\frac{CR_{ij}}{CA_{ij}}\right)$. The order computed above is used for branching, *i.e.*, we make branching decisions on instructions strictly in the above order. Each branching decision consists of choosing a specific version of the instruction under consideration. At each point visited in the branch-and-bound decision tree, we compute three values: (i) the current cost function f, (ii) a lower bound of the cost function f_{lo}, which is the cost function obtained by including all custom instructions that have not been visited yet, while the maximum clock period remains as the current clock period, and (iii) total area of already selected custom instructions CA. We pop the next custom instruction from the sorted list, go through each version and compute the same three values again. If CA is greater than the maximum area constraint, we bound. If the lower bound f_{lo} is worse than the best k solutions, we bound, otherwise, we consider the next custom instruction in the sorted list for branching. If the current f is within the k best solutions seen thus far, the current solution is stored in the result array.

The above procedure can be easily extended as follows to the case when candidate custom instructions are generated from overlapping templates. Suppose that, at a given point in the decision tree, a custom instruction (say, instruction i) is chosen. We find all the other custom instructions that have overlap with instruction i, and force the procedure to avoid choosing them. If we backtrack to a different part of the decision tree and reverse the decision to include instruction i, these constraints are removed.

V. EXPERIMENTAL RESULTS

We have implemented the flow described in Section IV by integrating several commercial and public-domain tools with our custom tools. Our tool takes a C program as input and outputs custom instructions and the modified C program. The data between commercial tools and our program are exchanged through files and scripts. The GNU-based compiler, simulator, and profiler tools provided by Tensilica are used to simulate the program and gather information about execution cycles (steps **2**, **12**, and **18** in Fig. 3). We use the Aristotle analysis system [32] to generate the program dependence graphs (step **1** in Fig. 3). The program dependence graphs are generated at the source code level. Hence, it is easy to back-annotate to the original C program.

After a promising new instruction or instruction combination is identified, our tool automatically outputs them in TIE format, and invokes Tensilica's TIE compiler to transform the instruction specification to RTL Verilog code (steps **7** and **15** in Fig. 3). We then use Synopsys Design Compiler [33] to synthesize the RTL circuit and map it to NEC's commercial 0.18μ technology library [37] (steps **9** and **16** in Fig. 3). The area and clock period information extracted from the synthesized, mapped netlists are used to drive the selection of the final instruction combination that is used to augment the processor.

We evaluated the proposed techniques using six example benchmarks. `BYTESWAP` is a function to swap the order of bytes in a word. It is mostly used for little-endian to big-endian conversion and vice versa. `Add4` adds the value of four bytes in one word and returns the sum. `RGBtoCMYK` is a color conversion program. `Alphablend` blends two 24-bit pixels. `PopCount` implements the population count function, which counts the number of 1's in a word. `Rand` is a function for ISAAC (indirection, shift, accumulate, add, and count), which is used as a fast cryptographic random number generator. Our experiments are run on a 440 MHz SUN Ultra10 workstation with 1 GB main memory. Area constraint is set to 10% of the original processor's total area in all experiments.

The time to completely generate and select custom instructions from original C programs (steps **1** to **18** in Fig. 3) varies from less than an hour to over six hours, depending on the number of iterations. Most of the time in the design flow is spent in synthesis (Design Compiler [33]: steps **9** and **16** in Fig. 3), simulation (xt-run [3]) and profiling (xt-gprof [3]: steps **2**, **12**, and **18** in Fig. 3)[4].

Table I summarizes the results of our experiments. It compares the execution time, energy, and energy-delay product of the bench-

[4]The CPU times are quite reasonable considering that we are not only performing instruction selection, but also a complete synthesis of the optimized processor.

TABLE I

Area, performance and energy results for processors generated by the proposed tool

Program	Original				New				Speedup	Energy·Delay reduction
	Time (ms)	Energy (μJ)	Energy·Delay $(ms \cdot \mu J)$	Area $(grids)$	Time (ms)	Energy (μJ)	Energy·Delay $(ms \cdot \mu J)$	Area $(grids)$		
`BYTESWAP`	0.958	101.5	97.2	435347	0.397	42.1	16.7	432496	2.4X	5.8X
`Add4`	0.532	63.8	33.9	-do-	0.327	35.0	11.4	445216	1.6X	3.0X
`RGBtoCMYK`	2.073	193.0	400.1	-do-	0.387	42.6	16.5	446314	5.4X	24.2X
`Alphablend`	2.728	298.4	814.0	-do-	0.531	73.4	39.0	458953	5.1X	20.9X
`PopCount`	0.901	90.1	81.2	-do-	0.217	20.0	4.3	438346	4.2X	18.9X
`Rand`	2.063	253.7	523.4	-do-	1.277	159.6	203.8	436995	1.6X	2.6X

mark programs, running on a base processor (without any TIE extensions), and on the customized processors generated by our tool. We also report the area overheads incurred due to the addition of extra hardware to the processor. Note that the only difference between the two processor versions used for each program is the presence of custom instructions - all other processor parameters are kept unchanged. The results in Table I are based on: (i) execution cycles reported by the cycle-accurate ISS, (ii) clock period and area information derived from the synthesized, mapped netlists of the complete processor cores, and (iii) power estimates provided by running the commercial tool WattWatcher from Sente Inc. [38]. The results indicate that processors customized using instructions automatically generated by our tool can achieve a performance improvement of upto 5.4X (average of 3.4X) over the base processor cores. Energy consumption is reduced by upto 4.5X (average of 3.2X), energy-delay product is reduced by upto 24.2X (average of 12.6X), while average area increase is only 1.8%.

VI. CONCLUSIONS

Current design flows based on extensible processors require designers to manually identify and design custom instructions to accelerate parts of the application program. In this work, we have developed an automatic flow to generate custom instructions or instruction combinations that maximize the performance improvement for a given program, under constraints on the overhead due to the additional hardware. We have implemented this flow using a combination of commercial and public-domain tools, and our own in-house tools. Our experiments thus far have demonstrated promising results, indicating that automatic generation of custom instructions can result in large improvements in performance, energy, and energy-delay product, while significantly reducing design turnaround time.

References

[1] G. De Micheli, R. Ernst, and W. Wolf, *Readings in Hardware/Software Codesign*, Morgan Kaufmann Publishers, San Mateo, CA, 2001.

[2] A. Wang, E. Killian, D. Maydan, and C. Rowen, "Hardware/software instruction set configurability for system-on-chip processors," in *Proc. Design Automation Conf.*, June 2001, pp. 184–188.

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

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

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

[6] *SP-5flex DSP core*, 3DSP Corp. (`http://www.3dsp.com`).

[7] R. Sucher, "Carmel: A configurable long instruction word DSP core," in *Microprocessor Forum*, Oct. 1998.

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

[9] R. Cloutier and D. E. Thomas, "Synthesis of pipelined instruction set processors," in *Proc. Design Automation Conf.*, June 1993.

[10] I.-J. Huang and A. M. Despain, "Generating instruction sets and microarchitectures from applications," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1994, pp. 391–396.

[11] A. Kitajima, M. Itoh, J. Sato, A. Shiomi, Y. Takeuchi, and M. Imai, "Effectiveness of the ASIP design system PEAS-III in design of pipelined processors," in *Proc. Asia South Pacific Design Automation Conf.*, Jan. 2001, pp. 649–654.

[12] 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.

[13] A. Pyttel, A. Sedlmeier, and C. Veith, "PSCP: A scalable parallel ASIP architecture for reactive systems," in *Proc. Design Automation & Test Europe Conf.*, Mar. 1998.

[14] V. S. Lapinski, *Algorithms for Compiler-assisted Design Space Exploration of Clustered VLIW ASIP Datapaths*, Ph.D. thesis, University of Texas at Austin, May 2001.

[15] S. Aditya, B. R. Rau, and V. Kathail, "Automatic architectural synthesis of VLIW and EPIC processors," in *Proc. Int. Symp. System-Level Synthesis*, Nov. 1999, pp. 107–113.

[16] W. Zhao and C. A. Papachristou, "An evolution programming approach on multiple behaviors for the design of application specific programmable processors," in *Proc. European Design & Test Conf.*, Mar. 1996, pp. 144–150.

[17] K. Kim, R. Karri, and M. Potkonjak, "Synthesis of application specific programmable processors," in *Proc. Design Automation Conf.*, June 1997, pp. 353–358.

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

[19] 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.

[20] I.-J. Huang and A. M. Despain, "Synthesis of instruction sets for pipelined microprocessors," in *Proc. Design Automation Conf.*, June 1994.

[21] J. van Praet, G. Goossens, D. Lanneer, and H. De Man, "Instruction set definition and instruction set selection for ASIPs," in *Proc. Int. Symp. High-Level Synthesis*, 1994, pp. 11–16.

[22] A. Y. Alomary, T. Nakata, Y. Honma, M. Imai, and N. Hikichi, "An ASIP instruction set optimization algorithm with functional module sharing constraint," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1993, pp. 526–532.

[23] M. Gschwind, "Instruction set selection for ASIP design," in *Proc. Int. Symp. HW/SW Codesign*, May 1999, pp. 7–11.

[24] M. Arnold and H. Corporaal, "Automatic detection of recurring operation patterns," in *Proc. Int. Symp. HW/SW Codesign*, May 1999, pp. 22–26.

[25] C. Liem and P. Paulin, *Compilation Techniques and Tools for Embedded Processor Architectures, in Hardware/Software Co-design: Principles and Practice, J. Straunstrup and W. Wolf (Editors)*, Kluwer Academic Publishers, Norwell, MA, 1997.

[26] W. E. Dougherty, D. J. Pursley, and D. E. Thomas, "Subsetting behavioral intellectual property for low power ASIP design," *J. VLSI Signal Processing*, vol. 21, no. 3, pp. 209–218, July 1999.

[27] J.-G. Cousin, O. Sentieys, and D. Chillet, "Multi-algorithm ASIP synthesis and power estimation for DSP applications," in *Proc. Int. Symp. Circuits & Systems*, May 2000, pp. II/261–II/264.

[28] C. Lee, J. Kin, M. Potkonjak, and W. H. Mangione-Smith, "Designing power efficient hypermedia processors," in *Proc. Int. Symp. Low Power Electronics & Design*, Aug. 1999, pp. 276–278.

[29] A. Abnous and J. Rabaey, "Ultra-low-power domain-specific multimedia processors," in *Proc. VLSI Signal Processing IX*, Oct. 1996.

[30] C. Glokler and H. Meyr, "Power reduction for ASIPs: A case study," in *Proc. Wkshp. Signal Processing Systems (SIPS)*, Sept. 2001.

[31] B. W. Kernighan and R. Pike, *The Practice of Programming*, Addison-Wesley, Inc., Menlo Park, CA, 1999.

[32] Aristotle research group, `http://www.cc.gatech.edu/aristotle/`, *Aristotle manual.*

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

[34] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman Publishers, San Mateo, CA, 1989.

[35] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, New York, NY, 1994.

[36] S. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, San Mateo, CA, 1997.

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

[38] Sente Inc., `http://www.senteinc.com`, *Wattwatcher manual.*