Accelerating an Application Domain with Specialized Functional Units

CECILIA GONZÁLEZ-ÁLVAREZ, Ghent University & UPC
JENNIFER B. SARTOR, Ghent University
CARLOS ÁLVAREZ and DANIEL JIMÉNEZ-GONZÁLEZ, UPC
LIEVEN EECKHOUT, Ghent University

Hardware specialization has received renewed interest recently as chips are hitting power limits. Chip designers of traditional processor architectures have primarily focused on general-purpose computing, partially due to time-to-market pressure and simpler design processes. But new power limits require some chip specialization. Although hardware configured for a specific application yields large speedups for low-power dissipation, its design is more complex and less reusable. We instead explore domain-based specialization, a scalable approach that balances hardware's reusability and performance efficiency. We focus on specialization using customized compute units that accelerate particular operations. In this article, we develop automatic techniques to identify code sequences from different applications within a domain that can be targeted to a new custom instruction that will be run inside a configurable specialized functional unit (SFU). We demonstrate that using a canonical representation of computations finds more common code sequences among applications that can be mapped to the same custom instruction, leading to larger speedups while specializing a smaller core area than previous pattern-matching techniques. We also propose new heuristics to narrow the search space of domain-specific custom instructions, finding those that achieve the best performance across applications. We estimate the overall performance achieved with our automatic techniques using hardware models on a set of nine media benchmarks, showing that when limiting the core area devoted to specialization, the SFU customization with the largest speedups includes both application- and domain-specific custom instructions. We demonstrate that exploring domain-specific hardware acceleration is key to continued computing system performance improvements.

Categories and Subject Descriptors: Computer systems organization [**Other Architectures**]: Special purpose systems

General Terms: Design, Performance, Measurement, Experimentation

Additional Key Words and Phrases: Customization, acceleration, specialized functional unit, domain-specific, application-specific, canonical representation

ACM Reference Format:
González-Álvarez, C., Sartor, J. B., Álvarez, C., Jiménez-González, D., and Eeckhout, L. 2013. Accelerating an application domain with specialized functional units. ACM Trans. Architec. Code Optim. 10, 4, Article 47 (December 2013), 25 pages.
DOI: http://dx.doi.org/10.1145/2555289.2555303

Authors' addresses: C. González-Álvarez, J. B. Sartor and L. Eeckhout, ELIS department, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium; emails: cecilia.gonzalezalvarez@elis.ugent.be, jennifer.sartor@elis.ugent.be and lieven.eeckhout@elis.ugent.be. C. Álvarez and D. Jiménez-González, DAC department, UPC - Barcelona Tech, Campus Nord, D6 building, Jordi Girona 1-3, 08034 Barcelona, Spain; emails: calvarez@ac.upc.edu and djimenez@ac.upc.edu.

DOI: http://dx.doi.org/10.1145/2555289.2555303

1. INTRODUCTION

Since G. Estrin proposed the first model of a specialized computer over 50 years ago [Estrin 1960], computer engineers have extensively studied the implementation of specific compute units. Specialization can offer many benefits over traditional, general-purpose architectures, and now, specialization is viewed as a viable way to combat the end of Dennard scaling [Dennard et al. 1974], or chips hitting a power wall because of slowed supply voltage scaling [Esmaeilzadeh et al. 2011; Hameed et al. 2010; Venkatesh et al. 2010]. Computing systems are moving away from general-purpose designs out of necessity, but more specific designs add complexity and limit flexibility. Application-specific architectures have been proposed to improve performance and power efficiency for both research [Vassiliadis et al. 2004] and commercial [Gonzalez 2000; Altera Corporation 2013] purposes. However, time to market is a major issue with these customized designs, which are more complex, are costlier, and have shorter lifetimes. Application-specific specialization is economically feasible only for a few very important applications in big-volume markets.

In the middle of the spectrum between general-purpose and application-specific processors, we have Application-Specific Instruction-set Processors (ASIPs). An ASIP tailors its instruction-set architecture, providing a tradeoff between the flexibility of a general-purpose processor and the performance and energy efficiency of an application-specific design. The instruction-set architecture of an ASIP can be configurable, either in the field (in a fashion similar to an FPGA) or at design time. Optimizing an ASIP for a given application domain not only may be more economically viable but also can deliver better system performance when multiple applications run on the device. Although we focus on the media domain, the concept can be applied to tune an otherwise general-purpose processor for other domains such as image and audio processing, medical imaging, and so forth.

In this article, we focus on identifying potential custom instructions that extend the instruction-set architecture of a base architecture and accelerate a sequence of operations in an application. We explore the design space of custom instructions that are implemented in a configurable Specialized Functional Unit (SFU) in hardware, from those designed for a particular application versus those applicable to many applications within a domain. Previous research has used automatic tools to identify repeated patterns of instructions and propose them as extensions to the ISA. Initial developments established the grounds for the field using exhaustive identification of patterns [Atasu et al. 2008] and approximate techniques [Pozzi et al. 2006]. Other works [Arnold and Corporaal 2001; Clark et al. 2005] have used pattern-matching-based approaches on the data flow of programs, represented as Directed Acyclic Graphs (DAGs), to identify custom instructions across a domain. However, pattern matching cannot always find similarities between sequences of code in order to map different functionality to the same custom instruction, inherently limiting specialized hardware opportunities.

We introduce a new technique to extract common sequences of computations from several applications within a domain, which become custom instructions implemented within an SFU, which is tightly integrated with a processor core's data path. We use Taylor Expansion Diagrams (TEDs), which are canonical representations of polynomial computations [Ciesielski et al. 2006], to identify common computations. Thus far, TEDs have only been used in the areas of compiler optimization and design verification, and we novelly use them to identify common sections of code that can be accelerated by specialized hardware. We compare the effectiveness of DAG, TED, and a new Hybrid DAG/TED technique in finding common code sequences to target for acceleration in hardware. Our study shows that the canonical representation is key to identifying sequences that are mapped to the same custom instruction across applications. We also evaluate four new scoring heuristics that prune the huge search space of the potential

custom instructions without a detailed evaluation, selecting those that maximize the speedup of our application domain.

We build an exploration framework to estimate the speedup of new custom instructions across the spectrum of application-specific and domain-specific acceleration in hardware. We use nine media benchmarks and extend the LLVM compiler framework to identify code sequences amenable for acceleration in the SFU. We extract sets of reusable custom instructions, both within and across benchmarks, which we subsequently analyze and rank using our scoring heuristics. We then use the Xilinx design software to synthesize a hardware implementation of a potential custom instruction. Given an instruction's hardware data path, we use estimation models to approximate its core area and number of cycles, and thus speedup. We show that while DAG, TED, and Hybrid perform similarly when finding custom instructions for a particular application, using the TED and Hybrid techniques to identify custom instructions across a domain leads to much higher speedups than when using the DAG technique alone. Our analysis reveals that when the SFU occupies a small, realistic core area, it obtains the highest speedups when including both custom instructions designed across all applications in a domain and some specific to one application. Using only application-specific custom instructions performs best at large, unbounded core areas. We study a few machine design points in detail: Given a particular area, we present the characteristics of the SFU that obtains the highest speedup. Finally, we study how well custom instructions identified for a set of benchmarks perform for other, previously unseen workloads.

Overall, we make the following major contributions in this article:

—We propose TEDs for identifying hardware acceleration opportunities. We find that their canonical representation allows them to identify more sequences across applications that are mapped to the same custom instruction, thus achieving higher speedups for a lower area than the traditionally used DAGs.
—We propose and evaluate four scoring heuristics to quickly and effectively cull the huge specialized functional unit design space and rank potential domain-specific custom instructions. The best scoring heuristic is random-scaled sharing, which takes into account sharing custom instructions across applications as well as introducing some controlled randomness to smooth out unaccounted factors.
—Our exploration study reveals that while using only application-specific custom instructions results in the highest possible speedups at large or unbounded core areas, it is suboptimal and ineffective at small areas. Instead, considering domain-specific custom instructions along with application-specific custom instructions yields the highest possible speedup at small, more realistic core areas. This underlines the importance of identifying custom instructions that can be shared across applications.
—We demonstrate that new applications inside a domain can substantially benefit from an SFU already designed for that domain. This suggests that processors with domain-specific functional units can extend their lifetime and utility by being applicable to other applications.

2. PROBLEM STATEMENT

We assume that the custom instructions execute on an SFU that is tightly integrated in the data path of the general-purpose processor, as in Figure 1. Our target architecture is a single-issue in-order processor with a configurable pipeline to execute custom instructions. Our hardware exploration focuses on identifying sequences of code that can be mapped to the same custom instruction, which runs inside one Specialized Execution (SE) pipeline of the SFU and takes a variable number of cycles (c). We assume that SE pipelines can be configured at system boot time. All custom instructions are implemented in the SFU, which works as a multicycle functional unit and reads and writes

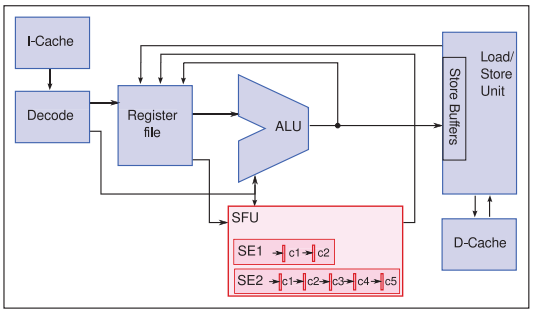


Fig. 1. Target architecture. The Specialized Functional unit (SFU) is part of the execution pipeline of an in-order processor core.

data from and to the register file of the core. When analyzing code sequences to identify custom instructions, we disallow control or memory operations. We do not focus on creating a new specialized processor, but on accelerating a general-purpose processor using a small amount of its area. Benefits of such a design include a system that maintains precise interrupts, the reduction of instructions in the execution pipeline of the processor core, and the increment of operational and data-level parallelism in the SFU.

In this article, we explore the tradeoff between application-specific versus domain-specific hardware specialization. Given a defined set of applications, our main objective is to design the hardware to maximize the platform's efficiency. We focus on maximizing speedup, or boosting system performance and application execution time, given a particular core area dedicated to the SFU. Exploring the application-specific versus domain-specific specialization tradeoff involves a number of challenges. For one, we need a framework to identify code sequences within and across applications that are amenable to hardware acceleration. Finding common code sequences across applications is particularly challenging because of the huge search space; that is, one needs to keep track of all code sequences of all applications to be able to find commonalities, and one needs to find the best way to represent these code sequences to maximize the likelihood of finding commonalities both within and across applications. Further, to be able to quickly explore the custom instruction design space and keep exploration time reasonable, we need heuristics to rank the effectiveness of potential specialized hardware without relying on detailed evaluation of each possible custom instruction. We have to use tools to estimate the speedup an application would achieve when using a particular set of custom instructions and optimize not only for speedup across the domain of applications but also for minimizing the SFU's area. In order to perform this study, we have built an accelerator exploration framework, which we describe next and which includes several novel contributions over prior work to identify and rank potential specialized functional units that accelerate computation.

3. CUSTOM INSTRUCTION SELECTION AND EVALUATION

Figure 2 shows an outline of our custom instruction selection and evaluation framework, which we detail in the following sections. We first analyze application code to identify potential code sequences for custom instruction design (Step 1). We then take steps to find commonalities among these identified code sequences, both within and across applications (Step 2), and then evaluate which custom instructions are most effective using newly proposed scoring heuristics (Step 3). Using these heuristics, we plug our chosen custom instructions into a low-level model that estimates both the speedup and the area of each (Step 4), so we can evaluate the potential of new computer designs with hardware acceleration.

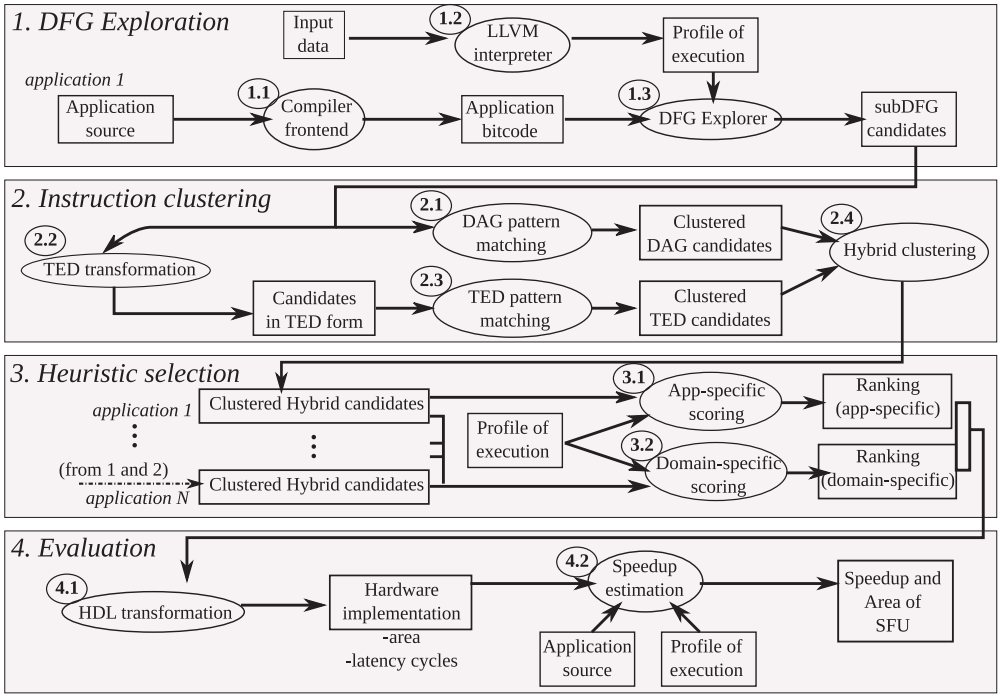


Fig. 2. Schematic overview of our custom instruction selection and evaluation framework.

3.1. DFG Exploration

Step 1 of Figure 2 shows how we identify code sequences amenable for acceleration in hardware. We use the compiler (label 1.1 in the figure) to transform the source code of the application into its Intermediate Representation (IR) to expose the Data Flow Graph (DFG) and Control Flow Graph (CFG) of the program. We use an IR representation close to the assembly language to find sequences of code that could be turned into specific custom instructions in hardware. Because identifying sequences of code to accelerate could blow up to a huge state space search, we apply certain constraints to lower the space exploration.

Static program analysis, implemented in the DFG Explorer (label 1.3), identifies a list of candidates that could be implemented as custom instructions. Each candidate must be a maximal convex subgraph [Atasu et al. 2008] of a data-flow graph for a given basic block, that is, the biggest disconnected subgraph of a basic block that preserves the convexity constraint [Pozzi et al. 2006]. These subDFGs exclude *invalid* instructions that cannot be executed in the SFU. In this article, we assume that the SFU executes neither memory nor branch instructions to keep the unit highly integrated in the processor's pipeline. Instead, they are executed in the core's ALU; thus, we mark them as *invalid* in the exploration step. However, to support other kinds of acceleration hardware that target code beyond the basic block level and include memory instructions, we could extend this step of the framework as well as Step 2, which clusters instructions using TEDs. Therefore, our exploration framework was built to be general and broad enough to study a variety of acceleration designs.

The DFG exploration is done with a fast implementation of the algorithm presented by Li et al. [2009] using binary structures. The algorithm performs a binary search for each basic block in the application, first enumerating the *invalid* instructions of the

graphs, which turn into the cutting nodes of the subtrees to be explored recursively in the search. The exploration result is a list of candidate code sequences, represented as subDFGs, that satisfy the previously mentioned criteria in nonexponential asymptotic time complexity (bounded by the number of *invalid* instructions, as they define the number of recursive calls).

In order to cut down on the number of candidates, we define a few rules to limit subDFG candidates. Groups of instructions are selected to preserve the consistency of scheduling, which means that all the inputs of the set are ready at issue time. In our exploration, we allow unlimited inputs and outputs to the custom instruction, because more complex custom instructions will potentially achieve a higher speedup. We also limit the exploration space by only considering executed parts of the code, using a previously gathered execution profile of the application (label 1.2 in Figure 2). At the end of Step 1, we have a list of candidates that are then passed to the next step, which clusters the potential code sequences to help select custom instructions.

3.2. Instruction Clustering

In Step 2 of Figure 2, we analyze the code sequences found in Step 1 in order to cluster them to propose custom instructions that apply to several different sequences of code. This clustering step can be performed on code sequences identified from the same application (targeting application-specific custom instructions) and/or sequences from different applications (targeting domain-specific custom instructions). Clustering serves several functions: to enhance reusability, to minimize implementation area in hardware, and to reduce the search space in the selection step.

In the following sections, we describe three methodologies for the clustering: *DAG*, *TED,* and *Hybrid*.

3.2.1. Clustering with DAG Isomorphism. The first technique clusters code sequences using DAGs. For each pair of subDFGs obtained in Step 1, we perform a one-to-one isomorphism detection (label 2.1 in Figure 2). Those graphs that are isomorphically exact are clustered under the same label, to be potentially transformed into a single custom instruction candidate.

Previous works [Arnold and Corporaal 2001; Clark et al. 2005] approached the problem by starting from small graphs, building them up to arrive at relatively large-sized accelerators—a bottom-up approach. In our work, we employ a top-down approach and start from maximal subgraphs extracted from a basic block, ideally covering as large code sequences as possible, and exploit as much instruction-level parallelism as possible.

Relatively larger custom instructions are more likely to yield better overall performance, but the identification of big patterns of functionally identical computation is a complex problem. Consider the three examples of subDFGs in Figure 3, identified in different benchmarks and their equivalent algebraic expressions. Example 1 shows two portions of code of the aacenc application from different basic blocks in their DAG representations. They differ in the number and types of instructions they contain. Simple DAG pattern matching would not cluster these two DAGs, although their algebraic functions are equivalent. In Example 2, we extend the problem to a domain of applications. We show DAGs of basic blocks from different benchmarks (mpeg2dec, aacenc, mpeg2enc, and face_detect) that perform the same computation, but with different operators. The DAGs of two of them (mpeg2dec, mpeg2enc) are isomorphically the same; therefore, they could be clustered with DAG pattern matching. However, DAG pattern matching is not able to cluster all four of them. In Example 3, we show two DAGs of face_detect and tmndec with multiple outputs. In this case, although we can have a partial match with DAGs for outputs 2 and 3, the full match for identical computation

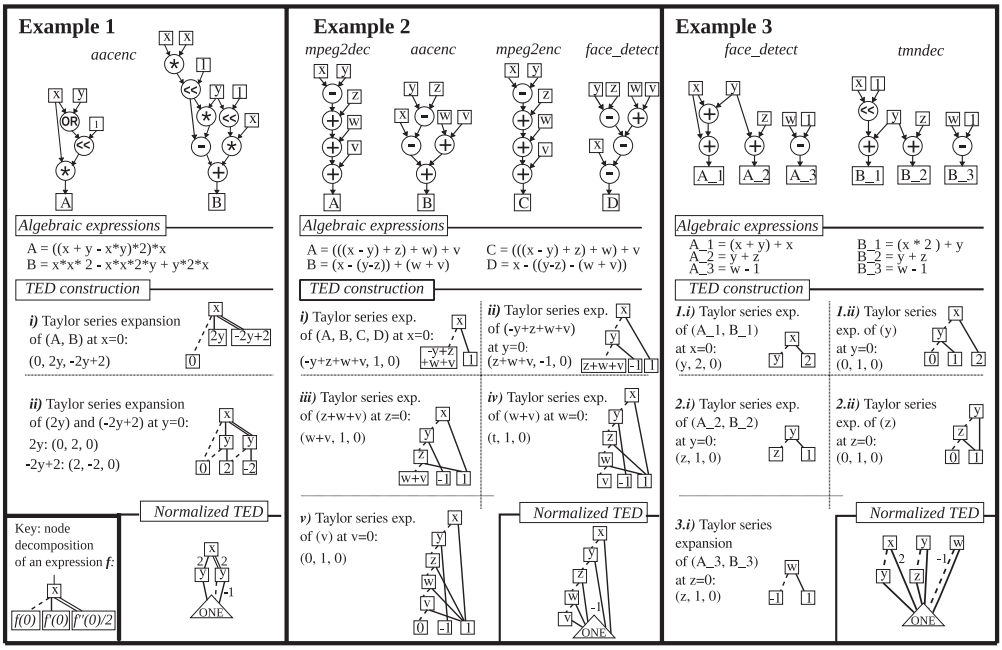


Fig. 3. Three examples of the usage of TEDs for instruction clustering. From top to bottom: DAGs, algebraic expressions, TED construction process, and final normalized TEDs.

cannot be found. Summarizing, in the three motivational examples, pattern matching using DAGs is missing opportunities to find commonalities among code sequences.

3.2.2. Clustering with TED Isomorphism. Because of the limitations of using DAG pattern matching, we introduce a second clustering technique based on a canonical representation of portions of the application's code. We gather insights from works on TEDs [Ciesielski et al. 2006], commonly used for circuit verification. We use these TEDs for another purpose: to find common parts of the code that cannot be found with a simple pattern-matching technique using DAGs. We match code from applications using TEDs at compile time (at an intermediate code level), and thus the shape of a TED does not influence the final implementation of a custom instruction at the circuit level.

In order to understand how the TED technique works for cases such as the one depicted in the examples of Figure 3, we first describe the basics of the representation. Taylor series expansion defines the representation of a multivariate algebraic expression $f(x, y, \dots)$ as:

$$f(x, y, \dots) = f(0, y, z, \dots) + xf'(0, y, z, \dots) + \frac{1}{2}x^2 f''(0, y, z, \dots) + \dots,$$

where the origin is set in $x = 0$ and with $f'(x = 0)$ and $f''(x = 0)$ as the successive derivatives of $f(x = 0)$. This decomposition, applied recursively to algebraic functions, is stored into a directed acyclic graph, the Taylor Expansion Diagram (label 2.2 in Figure 2). Each node of the graph represents an input variable, and three different types of edges can be linked to a node: constant Taylor expansion is represented with a dashed edge, the expansion on the first derivative is a plain lined edge, and the expansion on the second derivative is a double-lined edge. On the bottom left of Figure 3, we can find a key of that representation. Following a set of rules, we obtain a normalized and canonical representation of the TED from the starting algebraic expression.

In our concrete case, we start with the computations expressed as subDFGs or DAGs from Step 1 in Figure 3. Then, in order to build a TED, we execute the following steps:

(1) Convert the subDFG into an algebraic expression. Note that boolean logic can be expressed as an algebraic expression as well: for example, the logical "or" operation can be represented as $x \vee y = x + y - xy$ [Ciesielski et al. 2006].
(2) Decide the order in which the variables will be expanded, as it affects the size and shape of final canonical representation. We followed the recommendations of Gomez-Prado et al. [2004] to keep optimized TEDs.
(3) Recursively calculate the values of the Taylor expansion for the constant, first, and second derivative for every term in the algebraic expression.
(4) Apply reduction and normalization rules to arrive at and ensure canonicity as explained by Ciesielski et al. [2006].

We explain the TED construction with the examples in Figure 3. In Example 1, the first step converts the DAGs into the algebraic expressions A and B written under the graphs. Note the expansion of the "or" operation into its counterpart algebraic expression. In the second step, we decide the ordering of the variables, which is important to arrive at a canonical representation. In this case, the order is x, y. In the third step, we construct the TED, which will be unique for both A and B, as their Taylor series expansions yield the same values. Step *i* in the TED construction builds a partial TED performing the Taylor series expansion first on variable x. Then, step *ii* expands on variable y. The resulting TED, after applying normalization and reduction, leads to the reduced version in the bottom of the example. For Example 2, the four algebraic expressions are expanded in the same way, as shown in steps *i* to *v*. In Example 3, with multiple output DAGs, we will have an algebraic expression for each one of the outputs. Each expression is transformed into the corresponding TED, with as many steps as input variables. At the end, the generated TEDs, separately, are reduced and normalized, but also merged into a single normalized TED.

Finally, as TEDs are also directed acyclic graphs, we perform a one-to-one isomorphism detection with the normalized TED—like the ones at the bottom of Figure 3—as we do with the DAG representation (label 2.3 in Figure 2).

3.2.3. Hybrid TED-DAG Clustering. The final clustering technique is the Hybrid TED-DAG technique. Not all computations in their directed acyclic graphs can be converted to a polynomial expression, and only polynomials with a finite Taylor expansion can be modeled as TEDs. This excludes modular arithmetic, relational operations, and exponentiation of constants as a base, whereas a DAG can represent all types of computations as they are expressed in the DFG. Due to these restrictions, we propose a hybrid technique that uses the TED representation when it can be created, and otherwise uses the DAG representation of subDFGs to cluster computation (label 2.4 in Figure 2). Using this hybrid approach, we should be able to cluster more code sequences to target the same hardware, identifying the most efficient custom instructions for our set of applications.

3.3. Heuristic Selection

After clustering code sequences, we have identified many different possible custom instructions. In order to select the most promising ones for our applications, we introduce four novel scoring heuristics in Step 3 of Figure 2. Our scoring techniques use dynamic execution data from the applications in order to prioritize custom instructions, either focusing on application-specific or domain-specific custom instructions, that maximize speedup. Our scoring techniques do not currently take hardware implementation area into account. They score based on the number of regular instructions covered by each

custom instruction, the frequency of execution of the basic blocks that contain the subDFG that maps to that custom instruction, and (for domain-specific) the number of applications that can use each custom instruction.

3.3.1. Application-Specific Scoring. We first focus on a scoring heuristic that prioritizes custom instructions targeted at just one application (label 3.1 in Figure 2). Our heuristic ranks custom instructions based on the potential speedup they can offer, using the following terms: K is a custom instruction for which n code sequences are found in an application; that is, n code sequences can be accelerated using custom instruction K. $ninst_i$ is the number of regular instructions and $freq_i$ is the frequency of execution of the code sequence amenable to the custom instruction. The latter is gathered through profiling (label 1.2 in Figure 2).

Our application-specific scoring heuristic for custom instruction K is then defined as:

$$scoring_K = \sum_{i=1}^{n} ninst_i \times freq_i,$$

and essentially weights all code sequences with their instruction counts and execution frequencies to have a measure of the speedup of the application as a whole.

3.3.2. Domain-Specific Scoring. To identify custom instructions that are most efficient across a domain of applications, we must use different heuristics that take into account the reusability of the hardware (label 3.2 in Figure 2). We still take into account a custom instruction's execution frequency, but with a slight change. Because we are considering different applications, we must normalize the execution frequencies to the application's total dynamic instruction count. For any given application, the normalization is done by scaling the frequency of execution to the percentage of the application's total number of instructions executed.

We first define the following variables:

—K is a custom instruction with n code sequences found across all applications ($1 \leq n$).
—$ninst$ is the number of regular instructions of a given code sequence amenable to the given custom instruction.
—$nfreq$ is the normalized frequency of execution of the given code sequence.
—$napp$ is the number of applications that can use the custom instruction.
—Each of these $napp$ applications can use the custom instruction at m different points in the code ($1 \leq m \leq n$), and thus ($n = \sum_{i=1}^{napp} m_i$).

We now detail four new scoring heuristics that each prioritize custom instructions differently, and we compare them later in the experimental results section.

Scoring #1: Normalized application specific.

$$scoring_K = \sum_{i=1}^{n} ninst_i \times nfreq_i$$

This first scoring is similar to the application-specific scoring, though it uses normalized frequency values. It maximizes the ranking of frequently used custom instructions targeting high numbers of instructions. A custom instruction's sharing across applications is not taken into account with this scoring heuristic.

Scoring #2: Scaled by sharing.

$$scoring_K = \left(\sum_{i=1}^{n} ninst_i \times nfreq_i \right) \times napp$$

Our second scoring technique does take into consideration a custom instruction's ability to be reused or shared across applications. The *napp* factor prioritizes custom instructions that have a high sharing factor, when the scoring has to discriminate among custom instructions with similar numbers of normalized dynamic instructions. Application-specific custom instructions that are very frequently used are still highly ranked, since $nfreq_i \gg napp$.

Scoring #3: Geometric mean of sharing.

$$scoring_K = \sqrt[napp]{\prod_{i=1}^{napp} \left(\sum_{j=1}^{m_i} ninst_j \times nfreq_j \right)}$$

Our third scoring heuristic calculates the geometric mean of the m_i application-specific scores, where i is an index that iterates over the applications involved. Since application-specific scores for a given custom instruction can vary by several orders of magnitude, we propose this scoring to smooth out the spikes in the scores due to a single application (when $napp > 1$). Custom instructions that benefit many applications but get a high score from only one application are penalized. This heuristic thus introduces fairness for custom instructions targeting several applications. However, custom instructions used by one application are not penalized.

Scoring #4: Random-scaled sharing.

$$scoring_K = \sum_{i=0}^{napp-1} \left(\sum_{j=1}^{m_i} ninst_j \times nfreq_j \right) \times \frac{napp}{napp - i}$$

In the final scoring heuristic, we introduce a randomness factor controlled by the number of applications that the custom instruction targets. The application-specific scoring is weighted by $\frac{napp}{napp-i}$. The assignment of i is random, but *napp* still influences the final result; thus, the higher the sharing factor, the higher the score. Note that the value of i assigned to a particular application is nondeterministic, so the applications are weighted differently for each code sequence. The reason for introducing some controlled randomness is to distribute scores in a more flexible way, since there are other factors that we do not consider in our current heuristics.

3.4. Evaluation: Estimating Performance and Area

Finally, in Step 4 from Figure 2, we evaluate the effectiveness of the custom instructions identified by the previous three steps. Informed by the prioritization of custom instructions by the scoring heuristics in Step 3, we feed top custom instructions into a hardware description language conversion tool that creates a preliminary hardware implementation (label 4.1 in Figure 2). This implementation verifies that the identified sequences of code can be implemented as hardware structures and double-checks the scoring techniques. The hardware implementation, using information from the application profile, is fed into a model that estimates the achievable speedup and area occupied by each custom instruction (label 4.2 in Figure 2). Area estimates are obtained through hardware synthesis as we will explain in Section 4.1.

We estimate the speedup each custom instruction can achieve for each identified sequence of code as follows. Consider a custom instruction that would be invoked at n different locations in the code of a particular application, that covers *ninst* normal instructions, and that is executed *nfreq* times at a particular location. Further assume that hardware synthesis estimates the custom instruction to take *hw_cycles* to execute.

Consider also a cost of Cin cycles to move input data from the register file to the SFU before the custom instruction starts and a $Cout$ cost to move outputs back to the register file at the end of the accelerated execution. Both costs depend on the number of input and output parameters of a particular custom instruction and the available register ports in the baseline processor. We first estimate the execution time in cycles of all uses of the custom instruction (on the SFU) as: $T_{w/ci} = \sum_{i=1}^{n} nfreq_i \times (hw\_cycles + Cin_i + Cout_i)$, or the number of times the custom instruction is invoked multiplied by its execution time in cycles. Then, we estimate the number of cycles that the same sequences of code would take on the uncustomized processor (without using the custom instruction): $T_{w/o\,ci} = \sum_{i=1}^{n} ninst_i \times nfreq_i \times CPI$, with CPI as the cycles per instruction of the application on the target processor.

We define T as the total application execution time in cycles on the target processor (without using the custom instruction). We then can find the difference between the number of cycles our candidate sequences take on the uncustomized processor versus using custom instructions, and subtract this from T to approximate the accelerated performance. Formally, the estimated total application time when using custom instructions is $T - (T_{w/o\,ci} - T_{w/ci})$. We then divide that estimated time by T to calculate the SFU's achievable speedup. This is a conservative estimate since we do not take into account the potential instruction-level parallelism between regular and custom instruction execution, which would result in higher speedups.

With this evaluation step, we are able to compare the potential performance improvements that a set of custom instructions, whether including just application-specific custom instructions, domain-specific instructions, or both, can provide to an application or set of applications.

4. EXPERIMENTAL SETUP

We briefly detail the implementation details of our specialized functional unit design exploration framework, including the software and hardware tools used, and our benchmarks.

4.1. Framework

We use the LLVM compiler infrastructure [Lattner and Adve 2004] as the front-end to our custom instruction design exploration framework. We modify the LLVM code generation module to find maximum valid subDFGs for DFG exploration (Step 1 in our framework). We perform graph isomorphism detection using the NetworkX library [Hagberg et al. 2008] and construct the TED representations using the variable algebra analysis part of Sage [Stein et al. 2013]. We obtain an execution profile for each of our applications using the LLVM binary interpreter. The profile indicates the frequency of execution for each basic block and is used in Steps 2 to 4 of our framework.

We assume that the target architecture has a spare core area tightly coupled to the processor core to implement the configurable SFU, as shown in Figure 1. We consider a single-core single-thread OpenSPARC T1 as the baseline architecture, which has been adapted previously for research on embedded applications [SRISC 2012; González-Álvarez et al. 2011]. The register file that both the ALU and the SFU access consists of thirty-two 64-bit registers with three read, two write, and one transport ports. The instruction encoding allows moving two input operands to the SFU with no additional cost. Any extra inputs are sent in groups of three, with a cost of one cycle per transfer, before the custom instruction execution starts. When the instruction ends, outputs are packed together in groups of two and moved back to the register file, with a cost of one cycle per transfer.

Table I. Description of the Evaluated Application Benchmarks and Their Input Files

Benchmark	*Description*	*Input*
aacenc	AAC audio compression format encoder	33.9MB WAV
cjpeg	JPEG image format compressor	1.2MB PPM (Mediabench)
djpeg	JPEG image format decoder	12.8kB JPEG (Mediabench)
face	Face detection on bitmap files	734.5kB bitmap
tmndec	H263 video format decoder (TMN impl.)	114kB H263 (Mediabench)
tmnenc	H263 video format encoder (TMN impl.)	5.5MB YUV (Mediabench)
mpeg2dec	MPEG2 video format decoder	34.9kB (Mediabench)
mpeg2enc	MPEG2 video format encoder	506.9kB (Mediabench)
opt_flow	Optical flow for motion estimation	884kB images

To evaluate the selected custom instructions, we first translate their functionality to C code. For a given application, custom instructions that are functionally equivalent are translated to one common piece of code. Across applications, for a given set of sections of code identified as functionally equivalent, we provide an implementation of the custom instruction execution path for each application involved. Later, we choose the best among them for the performance model. We use the Vivado HLS suite to perform C to HDL conversion on those C-code segments. For feasibility reasons, our automatic toolchain uses the default optimizations of Vivado HLS [Xilinx 2012]. Any further improvements to the hardware implementation with specifically set optimizations would result in better overall speedups. The Xilinx ISE tool performs the synthesis of the design, using the Virtex 5 FPGA as a target, which estimates the new hardware's area (per custom instruction) as a number of look-up tables (LUTs) and slices. We report area estimates relative to the OpenSPARC T1 core area, which is also mapped onto a Xilinx Virtex 5 FPGA for apples-to-apples comparison. Although this work currently targets an ASIP for which the instruction-set architecture is configured at boot time, we use an FPGA model to keep open the option of exploring ASIPs with runtime programmable ISAs in the future. We also use the Xilinx ISE reports to estimate the number of cycles per custom instruction, which we use to estimate performance speedup through acceleration as previously explained.

4.2. Benchmarks

Table I shows the list of benchmarks that we use for our experiments, with their descriptions and input files. All of the applications belong to the media domain. The optical flow kernel and the face detection benchmark are part of the OpenCV library [Bradski 2000]. The AAC (audio compression) encoder is based on a program provided by Renesas Technology and Hitachi Ltd. The rest of the applications and their input files belong to the Mediabench benchmark suite [Fritts et al. 2009].

5. ESTIMATED PERFORMANCE RESULTS

In this section, we present the experimental results obtained using the custom instruction design exploration framework presented in Section 3. We first compare the speedup that we can achieve using the DAG, TED, and Hybrid clustering techniques described in Section 3.2, showing in Section 5.1 that the TED and Hybrid techniques by far outperform DAG for identifying custom instructions across a domain. We then show differences between our four new scoring heuristics (from Section 3.3) across benchmarks, demonstrating in Section 5.2 that, on average, the random-scaled sharing heuristic works best for our applications. In contrast to Sections 5.1 and 5.2, focusing only on domain-specific custom instructions, we then evaluate the differences in speedup that can be achieved using only domain-specific, only application-specific,

Table II. Number of Code Sequences and Custom Instructions Found in Each Application with DAG, TED, and Hybrid Methods, and the Percentage of Dynamic Instructions Covered by Them
These results use the random-scaled sharing heuristic and are for unlimited core area.

	Num. code sequences			Num. custom instr.			% dynamic instr.		
% Benchmark	DAG	TED	Hybrid	DAG	TED	Hybrid	DAG	TED	Hybrid
aacenc	81	73	72	29	32	27	10.5	6.1	4.9
cjpeg	126	138	140	53	41	41	3.5	10.8	10.9
djpeg	115	119	119	52	43	43	2.0	16.9	16.9
face	165	211	211	45	66	66	0.9	9.3	9.4
tmnenc	89	116	121	29	37	38	0.5	0.9	0.8
tmndec	51	68	70	31	43	45	2.8	6.6	6.6
mpeg2dec	75	83	86	44	40	43	24.1	16.6	21.2
mpeg2enc	106	164	172	51	68	72	2.1	9.0	9.7
optflow	1	7	7	1	6	6	0.0	27.2	27.2

or a mix of both kinds of custom instructions in Section 5.3. With the whole core area at our disposal, application-specific custom instructions achieve the highest speedup; however, at lower core areas, domain-specific custom instructions perform well, but always benefit from the addition of application-specific custom instructions. Using both kinds of custom instructions, we achieve the highest speedups. In Section 5.4, we perform a detailed analysis of the custom instructions included at particular percentages of the core area for application-specific, domain-specific, and mixed configurations. We reveal insights about the number of small, medium, and large custom instructions; the average number of inputs and outputs; and the number of applications each configuration can target. Finally, in Section 5.5, we evaluate a more realistic setting using cross-validation, evaluating how a set of custom instructions identified as useful for a group of applications perform for another, previously unseen, application.

5.1. DAG versus TED versus Hybrid

We first evaluate the effectiveness of using a directed-acyclic graph to guide pattern matching between code sequences (DAG) versus using a canonical approach to cluster code sequences (TED). We compare their effectiveness considering all applications from the domain. Table II compares the three techniques for each benchmark in the number of code sequences they identified, number of custom instructions selected, and percent of total dynamic instructions that can be converted to custom instructions. These numbers were gathered using the random-scaled sharing heuristic to rank candidates and devoting an unlimited core area to the SFU. We select a custom instruction if it can accelerate two or more code sequences from different benchmarks. For all but one benchmark (aacenc), the TED and Hybrid techniques find a larger number of code sequences than DAG. For all but two benchmarks (cjpeg and djpeg), TED and Hybrid also select about the same or a larger number of custom instructions. Even with cjpeg and djpeg, we see TED and Hybrid cover significantly more dynamic instructions than DAG, which is also the case for all other benchmarks except aacenc and mpeg2dec. Because the selection heuristic discards instructions that might cover more execution time, TED and Hybrid perform slightly worse for aacenc and mpeg2dec.

Figure 4 presents a graph for each benchmark with a range of core areas dedicated to the SFU on the x-axis and speedup on the y-axis. Here, we only include domain-specific custom instructions, or those that accelerate more than one application. These results use the best-performing scoring heuristic (random-scaled sharing), which we discuss in detail in the next section. Each point on the graph represents a group of domain-specific custom instructions that can be used by that benchmark and that fit inside that core area (x-axis), which together can achieve that speedup (y-axis) for a

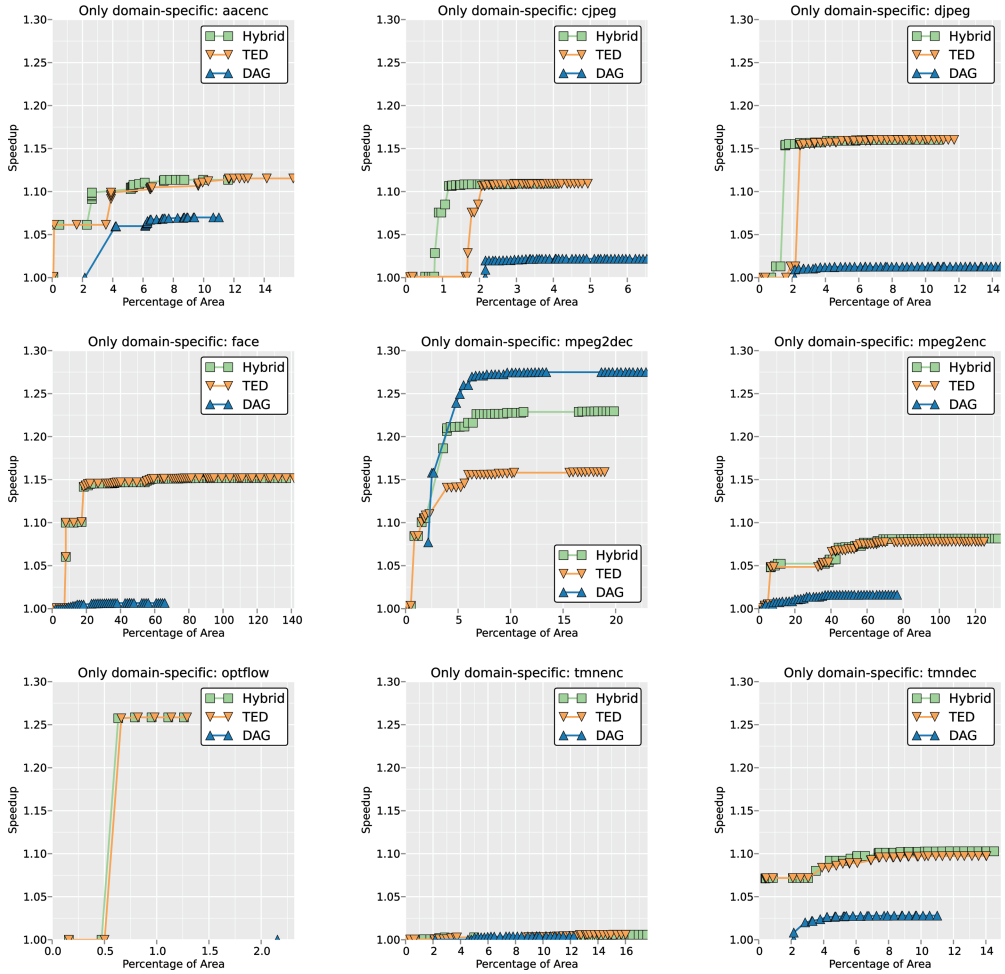
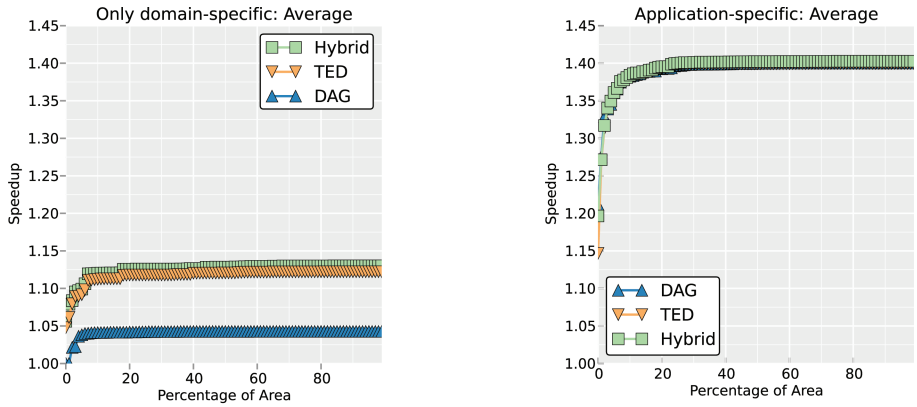


Fig. 4. Results of benchmark speedup versus custom instruction area for DAG, TED, and Hybrid methods, with domain-specific custom instructions using random-scaled sharing scoring.

given benchmark. Note that each benchmark has a different x-axis scale because these are the area percentages used per benchmark, not for the entire SFU. In all following sections, we consider the entire SFU design when discussing area. The average of all applications (using total SFU area) is shown in Figure 5(a).

On average, the Hybrid technique, which uses the TED representation when it is able and otherwise uses DAG, is the most effective technique at finding domain-specific custom instructions (see Figure 5(a)). The Hybrid technique achieves higher speedups at smaller areas (left-hand side on the graphs in Figure 4), always increasing the speedup faster than the other two techniques. All but two benchmarks show the best speedups with TED and Hybrid techniques regardless of area, and for `tmnenc`, DAG performs best only between 6% and 12% of the core area. When given an unbounded core area, only one benchmark, mpeg2dec, performs better with the DAG clustering technique than with Hybrid. This happens because the Hybrid technique first tries to identify custom instructions using TED, and when it cannot find any more, it complements with DAG. If part of an application's code is represented by TEDs and creates a less



a) *Domain-specific results* b) *Application-specific results*

Fig. 5. Average over all applications for DAG, TED, and Hybrid methods, using random-scaled sharing scoring, for domain-specific (a) and application-specific (b) custom instructions.

efficient custom instruction than a DAG design would, then the Hybrid technique would not be able to take advantage of the better DAG implementation. We also see that for most benchmarks, Hybrid and TED techniques perform very similarly. However, for mpeg2dec, which reveals a large opportunity with the DAG technique, Hybrid can achieve higher speedups than the TED technique alone because it can benefit from the code sequences that can only be represented in a DAG.

Figure 5(a) shows that on average across our benchmarks, TED and Hybrid achieve around 12% and 13% speedup, respectively, when using only 20% of the core area for domain-specific custom instructions, while DAG obtains only 4% speedup. We contrast this with Figure 5(b), which shows the average area and speedup numbers across our benchmarks for the three clustering techniques when we only include application-specific custom instructions. (We further compare application-specific versus domain-specific designs in Section 5.3.) While TED's canonical representation does not make a large difference when clustering code sequences within the same application, we see that it is very important to achieve higher speedups when generating domain-specific custom instructions. The key insight here is that individual applications are coded following the same style, so the benefit of a canonical representation is not so clear. However, as we move across applications, we find different code styles and a canonical representation is key to identifying acceleration opportunities.

5.2. Domain-Specific Scoring

We next compare the four new scoring heuristics that we explain in Section 3.3. Figure 6 presents a graph for each benchmark of the speedup that each heuristic predicts for a given SFU area. For these graphs, we use the Hybrid clustering technique and include only domain-specific custom instructions. Note that in these and all following sections, we consider the entire SFU design and its area, not only those custom instructions useful per application. Thus, area always ranges between 0% and 100% of the core. The average across all benchmarks is presented in Figure 7 for 100% of the area, and on the right we zoom in on smaller, more realistic areas of 0% to 20%.

Across all benchmarks, we see that the fourth scoring technique, or random-scaled sharing, performs best on average. In Figure 7, it achieves higher speedups quicker at lower areas, and at an unlimited area, it performs the best. At 20% area, shown in Figure 7(b), this technique achieves similar speedups to scaled-by-sharing. There are some variations across benchmarks in Figure 6. For face, the geometric mean

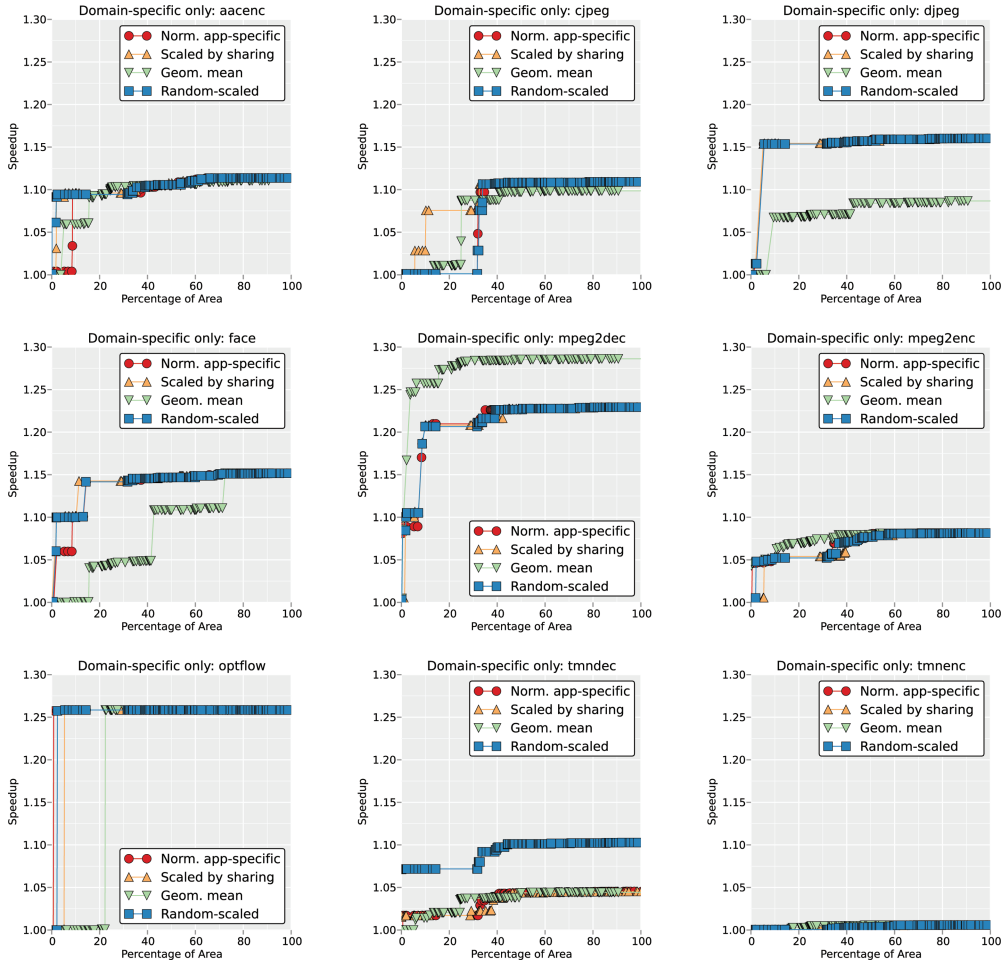
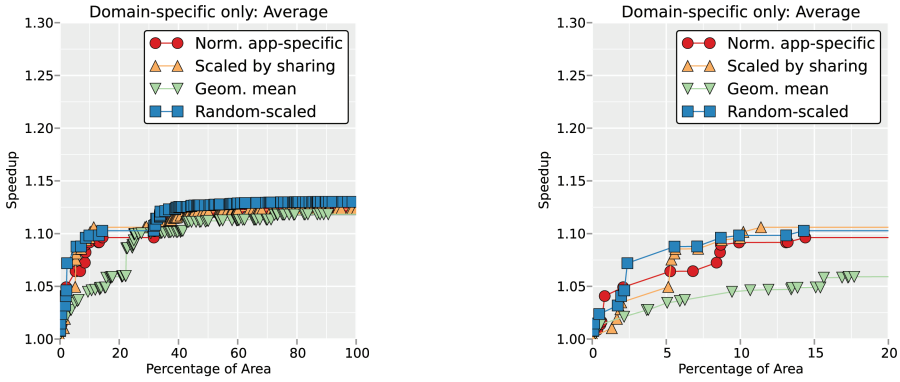


Fig. 6. Results of benchmark speedup versus SFU area for scoring techniques, with domain-specific custom instructions created with the Hybrid technique.

scoring takes more area to achieve similar speedups, probably because it dampens the importance of a domain-specific custom instruction that only performs well for one application. For djpeg, the geometric scoring heuristic cannot achieve the speedups the other three techniques achieve, and for tmndec, we see random-scaled sharing more than doubling the speedup of any other heuristic at any given area. For mpeg2dec, and to a lesser extent, mpeg2enc and tmnenc, the geometric mean heuristic that averages the benefit each application can receive does rise to higher speedups at lower areas. Only for mpeg2dec does the geometric mean technique get larger speedups than the random-scaled sharing heuristic at high areas. In this particular case, the geometric mean heuristic ranks a pair of custom instructions with low reutilization higher compared to the other scoring heuristics. The other heuristics did not rank these custom instructions as high because of previously identified, partially overlapping custom instructions. For aacenc, random-scaled maximizes the speedup at smaller areas. In particular, a custom instruction that causes a 6% speedup improvement is selected with that scoring three positions earlier than with scaled-by-sharing. However, for cjpeg, the scaled-by-sharing heuristic is the one that raises to high speedup values



a) *100% of SPARC core area* b) *20% of SPARC core area*

Fig. 7. Average over all applications for scoring techniques, with domain-specific custom instructions created with the Hybrid technique.

at lower areas. We find here a counterexample: scaled-by-sharing selects a custom instruction that contributes 5% to the speedup improvement five positions earlier than random-scaled. A closer look at the groups of code sequences that are clustered into those custom instructions tell us that in both cases the coverage across applications is maximized. However, random-scaled prioritizes less aggressively, and custom instructions with a medium number of applications but good overall performance will still rank high. Therefore, we use that scoring as our default in the other experiments reported in the article.

5.3. Application-Specific Versus Domain-Specific Configurations

Up until now, we have analyzed the potential of only domain-specific custom instructions. But our framework allows us to compare the performance of potential application-specific custom instructions as well. In this section, we compare the speedups that can be achieved using a part of the core area dedicated to only application-specific, only domain-specific, or a mixture of both kinds of custom instructions. Our goal here is to understand how to best configure an SFU to optimize full-system performance across applications subject to area constraints. Or in other words, for a given core area, are we better off choosing application-specific only, domain-specific only, or both application- and domain-specific custom instructions for the SFU?

Figure 8 presents the speedup for each benchmark across a range of areas, including only application-specific, only domain-specific, and both kinds of custom instructions. We analyze performance when the SFU takes 0% to 100% of the core area. Figure 9 shows the averages across all benchmarks, using up to 100% of the core's area, and zooming in on small, more realistic areas from 0% to 20%. For all of these graphs, we use the Hybrid clustering technique, and we use the application-specific scoring for application-specific custom instructions, and the random-scaled sharing scoring for domain-specific.

Our results reveal that, if given an unlimited area, using only application-specific custom instructions can achieve the maximum speedup (34%, on average) for our benchmarks. However, a potentially surprising result is that using both application- and domain-specific custom instructions together approaches the performance of using only application-specific custom instructions (29%) and obtains higher speedups at lower areas as compared to only application-specific. While using only domain-specific custom instructions limits maximal speedup to around 13%, we see that this

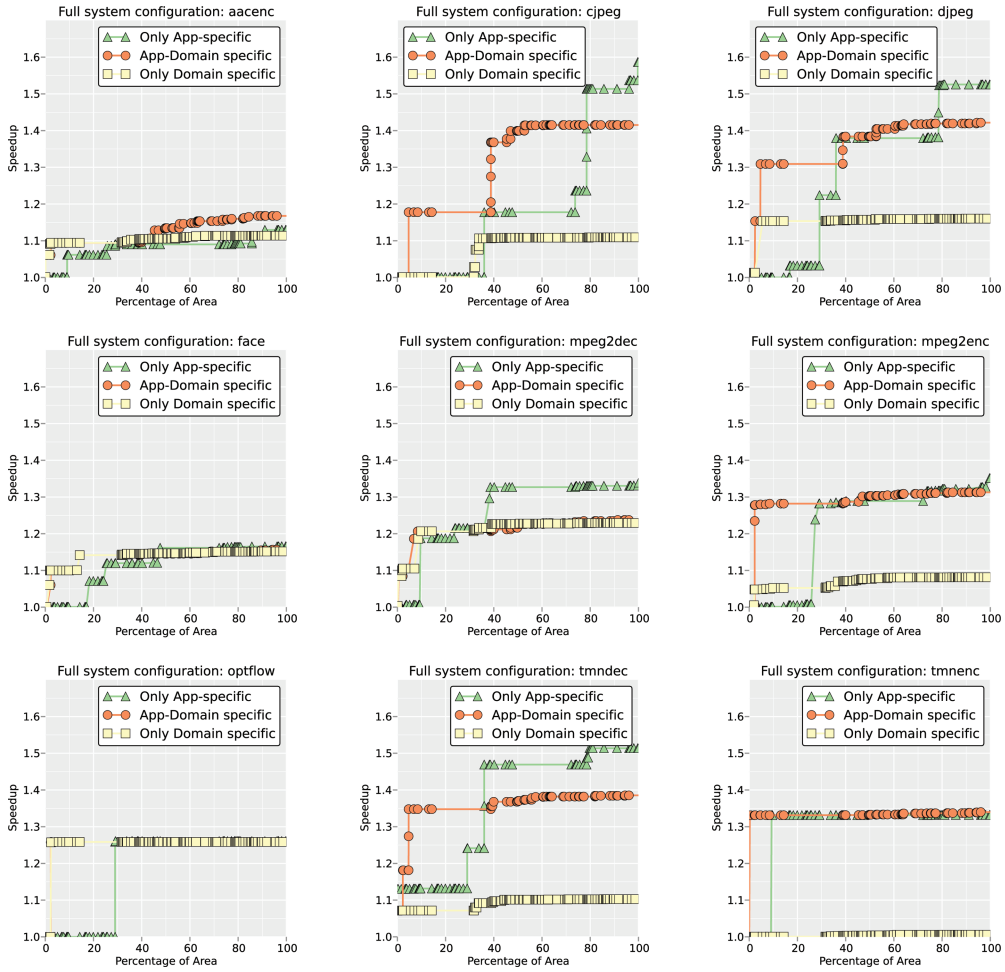
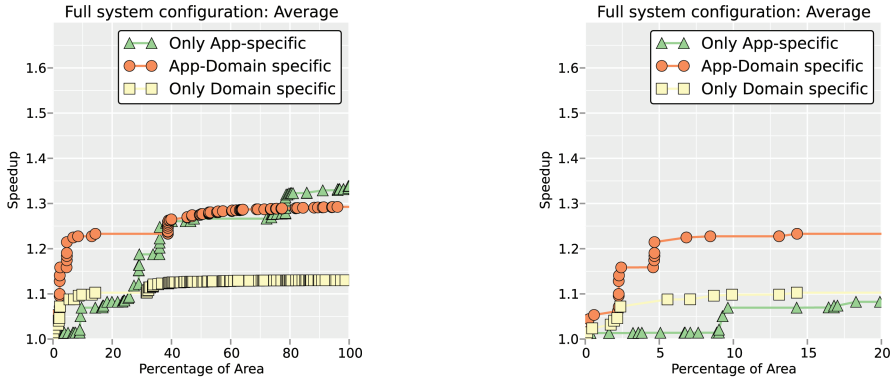


Fig. 8. Results of benchmark speedup versus SFU area using only application-specific, application- and domain-specific, or only domain-specific custom instructions. Results gathered using the Hybrid technique.

technique is more effective than application-specific at obtaining speedups at very small areas. Given 20% area, application-specific achieves 8% speedup, while domain-specific achieves 10% and both together achieve 23%. Furthermore, for several benchmarks, namely, aacenc, face, optflow, and mpeg2dec, using only domain-specific custom instructions performs close to the best of the other two techniques.

The key insight here is that, while using only application-specific custom instructions results in the highest possible speedups at large or unbounded core areas, considering domain-specific custom instructions next to application-specific custom instructions yields the highest possible speedup at realistic, smaller core areas. The reason is that the domain-specific custom instructions benefit several applications, which are more area efficient compared to application-specific custom instructions, which benefit a single application only, and therefore have limited contribution to overall system performance. A corollary of this finding is that, in order for hardware acceleration to deliver substantial speedups, some notion of application-specific hardware acceleration is needed (even at small areas). This requires knowing the target domain and



a) 100% *of SPARC core area.* b) 20% *of SPARC core area.*

Fig. 9. Average over all applications using only application-specific, application- and domain-specific, or only domain-specific custom instructions. Results gathered using the Hybrid technique.

Table III. Classification of Custom Instructions (CI) in a Full-System Configuration of 5%, 10%, and 15% of the SPARC Area
AS = application-specific, DS = domain-specific. Small = 1–5 regular instructions; Medium = 6–15 instructions; Large = >15 instructions.

%area	Config	Small sized			Medium sized			Large sized			#app	Spdup
		# CI	in	out	# CI	in	out	# CI	in	out		
5%	only AS	2	2.5	2	0	–	–	2	38	2.5	4	1.07×
	AS/DS	6(0/6)	5.3	2.2	2(0/2)	10	5	6(6/0)	26.5	8.2	9	1.22×
	only DS	7	4.8	2	1	9	5	0	–	–	9	1.07×
10%	only AS	4	2.7	1.5	0	–	–	2	38	2.5	6	1.07×
	AS/DS	8(0/8)	5.4	2.3	4(2/2)	11.25	5.25	6(6/0)	26.5	8.2	9	1.24×
	only DS	11	4.6	1.8	3	11.33	5.33	0	–	–	9	1.10×
15%	only AS	15	4.9	2.3	1	9	5	3	31.6	7	9	1.13×
	AS/DS	9(0/9)	4.7	1.8	4(2/2)	11.25	5.25	6(6/0)	26.5	8.2	9	1.24×
	only DS	13	4.8	2	4	12	6.5	0	–	–	9	1.10×

its applications at SFU configuration time so that some application-specific custom instructions can be included. Alternatively, one could devote a fraction of the SFU die area to domain-specific and application-specific custom instructions that are known to perform well given the applications known at design time.

5.4. Custom Instruction Analysis

In order to reveal further insights about how to build future specialized computing units, and which custom instructions offer the most benefit inside an application domain, we present an analysis of the custom instructions identified as the most effective at a few particular core areas. We compare the details of the SFU for designs with application-specific, domain-specific, and a mixture of both kinds of custom instructions. We show custom instruction statistics for core area percentages 5%, 10%, and 15% in Table III, taking the best configurations as shown in Figure 8.

Table III shows three configurations: using only application-specific custom instructions (*only AS*), using only domain-specific custom instructions (*only DS*), and using both (*AS/DS*, with the specific AS and DS portions in parentheses). We define three sizes of custom instructions, depending on the number of instruction primitives that each custom instruction implements. A small-sized custom instruction has one to five instructions, a medium-sized one has six to 15, and a large-sized one has more than 15.

We also present the average number of inputs and outputs for each size class; however, these do not affect the size class (i.e., small custom instructions could have a large number of inputs or outputs). Finally, we show the number of applications that each configuration can cover in the second-to-last column and the speedup it achieves.

We can draw a few interesting conclusions from the best-performing custom instruction configuration statistics. First, using both application- and domain-specific custom instructions already achieves 22% speedup using only 5% of the SPARC core's area. At the same area, using only application-specific custom instructions targets only four applications and can get only 7% speedup, which raises to 13% when using 15% of the core (while covering all nine applications). Interestingly, application-specific custom instruction configurations usually include small- and large-sized custom instructions but few medium-sized ones; in comparison, domain-specific custom instruction configurations include no large-sized custom instructions, instead prioritizing custom instructions with fewer than 15 base ISA instructions. Using both kinds of custom instructions (AS/DS), we find more domain-specific small-sized custom instructions, but more application-specific ones of the large size. We also see that, though the average input and output sizes are independent of the number of regular instructions per custom instruction, in general, the numbers of inputs and outputs grow as we go from small- to medium- to large-sized custom instructions. Interestingly, the mixed application and domain configurations include custom instructions from each size class and achieve the highest speedup for our applications. This suggests that the best-performing machine should include both application- and domain-specific custom instructions.

5.5. Cross-Validation

In all previous experiments, we generated candidate domain-specific custom instructions from code sequences using the entire set of benchmarks. In this final section, we evaluate a realistic setting where the machine is configured with a set of custom instructions for a particular application domain, but then an as-yet-unseen application runs upon it and tries to take advantage of the flexibility of the domain-specific custom instructions (generally known as cross-validation). In Step 3 of our methodology, shown in Figure 2, we cluster code sequences from $N-1$ of our benchmarks, prioritizing using our random-scaled scoring heuristic, and then in Step 4, we evaluate the effectiveness of those custom instructions on a different, the Nth, application.

Figures 10 and 11 show our cross-validation results for each benchmark and the average across benchmarks, respectively. When given the total core area, all but two benchmarks can reach the maximal speedup (obtained using domain-specific custom instructions identified over *all* benchmarks, as in Section 5.3, when given unlimited area). Benchmarks optflow and tmnenc cannot achieve their maximum speedup using our cross-validation approach. optflow achieves its speedup when using only one custom instruction; in addition, as shown in Figure 8, optflow does achieve its maximum speedup when we include domain-specific custom instructions identified from all benchmarks, whereas tmnenc can only benefit from application-specific custom instructions (achieving very limited speedup overall). The other seven benchmarks can take advantage of custom instructions deemed useful for the domain, and especially aacenc, face, mpeg2dec, tmndec, and djpeg achieve high speedups at very low core area percentages. At only 20% of the core area (Figure 11), our applications achieve over 7% speedup on average, which is a significant percentage of the maximum of 10%.

6. RELATED WORK

Here, we first survey work on application-specific custom instruction design, then detail domain-specific techniques, and finally describe a few holistic system designs.

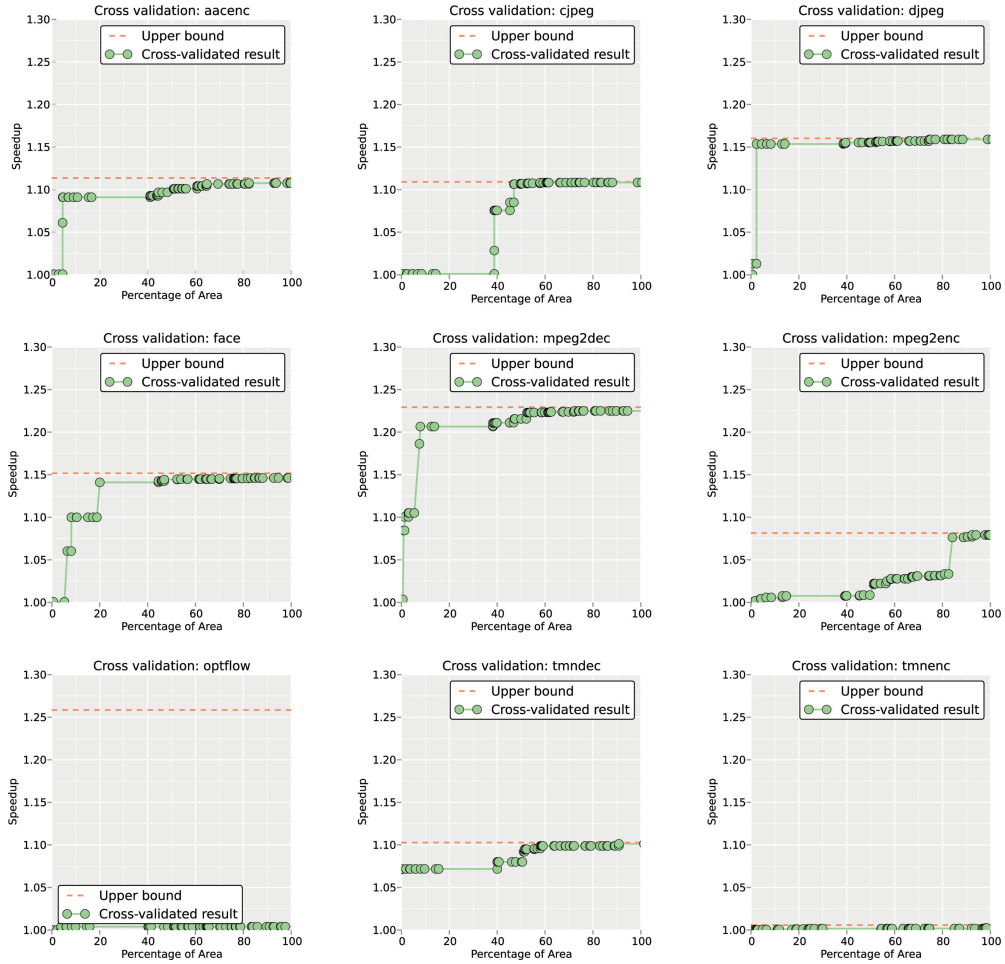


Fig. 10. Results of benchmark speedup versus SFU area for cross-validation per application using domain-specific custom instructions. Results gathered using the random-scaled sharing scoring and the Hybrid technique.

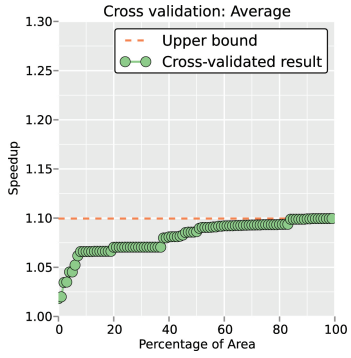


Fig. 11. Average over all applications for cross-validation results using domain-specific custom instructions. Results gathered using the random-scaled sharing scoring and the Hybrid technique.

Application-specific acceleration. Some research identifies custom instructions for particular applications, for performance and/or power reasons. Early works [Yu and Mitra 2004, 2007] established the baseline of the analysis using Data Flow Graphs (DFG), and showed the importance of preserving graph convexity. They differentiated the search process into *identification* and *selection* phases. Constraints such as the number of input and output nodes of the DFG help to prune the search space during identification. Later work coupled the identification and selection phases [Pozzi et al. 2006], which resulted in relaxing the constraints and opening up the possibility of approximate techniques that are less computationally expensive. They use heuristics to generate instruction patterns, maximizing instruction coverage, but do not explicitly rank the instructions as in our scoring methodology. Others, such as Verma et al. [2007], assume that the core processor must be an RISC, which also relaxes constraints. This implies a limited number of inputs and outputs, which prunes the results, in order to minimize the number of registers used. In our exploration, we accept any number of inputs and outputs for the custom instruction generation to maximize acceleration.

Symbolic algebra helps to identify and minimize the size of custom instructions [Peymandoust and Pozzi 2003]. However, this work did not use polynomials in a canonical form, as we do using TEDs. In addition, we use symbolic algebra for a different purpose, namely, to find code commonalities. We follow a previously proposed fast enumeration algorithm [Li et al. 2009] that we extend beyond their only application-specific applicability. Other authors [Arora et al. 2010] apply a predefined set of rules, in a specific order, to obtain a DAG representation of code functionality. This work, in contrast to ours, does not consider TEDs or domain-specific custom instructions.

In contrast with some later works [Murray et al. 2009; Atasu et al. 2012] that rely on integer linear programming, our final selection of custom instructions is based on a heuristic-based search. Other works with heuristics [Cong et al. 2004] forecast the gain of an instruction as a function of the instruction's frequency of execution and latency. They also use a dynamic programming algorithm to optimize for area, while our scoring focuses on coverage of the critical path, potential reutilization, and equality in the custom instruction's sharing across applications. Heuristics of application-centered works [Pothineni et al. 2007; Verma et al. 2007, 2010] maximize speedup with software and hardware latency estimations, which we use for modeling purposes.

Domain-Specific Acceleration. Previous works on domain-specific processors [Arnold and Corporaal 2001] or custom units [Clark et al. 2005] build their new instructions from small subDAGs extracted from the DFG. The former [Arnold and Corporaal 2001] limits the instruction patterns to three-node DAGs to limit the search space. The latter [Clark et al. 2005] uses a pattern-matching approach on DAGs that are developed in a bottom-up fashion using heuristics. They define guide functions that prune the DFG exploration space, using the criticality of the data path, latency, and area as metrics. In contrast to these prior works, we propose and use TEDs as a generalized representation to improve custom instruction coverage across applications. In addition, we propose scoring heuristics specifically designed to select domain-specific custom instructions, with the benefit of preserving maximal subgraphs. We do not consider area in our heuristics, but we take area into account to study application-specific versus domain-specific specialization, which reveals the importance of domain-specific custom units at small areas.

System Design. A few previous hardware acceleration design papers have been more holistic in nature, addressing the entire execution stack from the programming language to the compiler and the target platform. Almer and Bennett [2009] introduce support for application-specific instruction set extensions into a complete framework built on top of GCC. Our work also presents custom instruction generation as part

of a framework based on the (LLVM) compiler but targets domain-specific custom instruction designs. Another work targets health care applications [Cong et al. 2011] but requires programmer support, while our methodology requires no user input.

7. SUMMARY

Hardware specialization is a promising paradigm to improve performance and energy efficiency in the absence of Dennard scaling. However, a customized processor tailored to a specific application delivers high performance for that specific application only and is costly to manufacture. In contrast, a customized processor targeting an entire application domain, while being less effective for an individual application, may deliver better overall system performance when different applications run on the device and may be more economically viable by targeting a larger market.

This article explores this tradeoff between application-specific versus domain-specific hardware specialization and makes a number of contributions with respect to accelerating an application domain by identifying custom instructions to add to an existing ISA. We propose the use of Taylor Expansion Diagrams (TEDs), canonical representations of code sequences, previously used for circuit verification, to identify custom instruction opportunities. We find TEDs to be substantially more effective at identifying functionally equivalent code sequences across applications than the previously used directed acyclic graph (DAG) representation; combining TEDs with DAGs is even more effective at accelerating applications. To be able to quickly compare and rank potential domain-specific custom instructions during exploration, we propose scoring heuristics that take into account the frequency of custom instruction use both within and across applications. We use both TEDs and our scoring heuristics in our custom instruction exploration framework, along with performance and area estimation. We find that while application-specific custom instructions result in the highest possible performance at large or unbounded core areas, including domain-specific custom instructions yields the highest possible speedup at small, more realistic core areas. This finding underlines the need for domain-specific instructions for practical and flexible hardware specialization. In addition, we demonstrate that the identified custom instructions using our exploration framework are effective for previously unseen applications within the same domain, making specialization more generally applicable.

ACKNOWLEDGMENTS

We thank the anonymous referees and the associate editor for their valuable feedback and suggestions. This work is supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2012-34557, by the Generalitat de Catalunya (contract 2009-SGR-980), and by the HiPEAC3 Network of Excellence (FP7/ICT 287759). Additional support is provided by the FWO project G.0179.10N, the UGent-BOF project 01Z04109, and the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013)/ERC Grant agreement no. 259295. We would also like to thank the Xilinx University Program for its hardware and software donations.

REFERENCES

Almer, O. and Bennett, R. 2009. An end-to-end design flow for automated instruction set extension and complex instruction selection based on GCC. In *Proceedings 1st International Workshop on GCC Research Opportunities (GROW'09)*.

Altera Corporation. 2013. Altera Nios II. Retrieved November 26, 2013 from http://www.altera.com/devices/processor/nios2/ni2-index.html.

Arnold, M. and Corporaal, H. 2001. Designing domain-specific processors. In *Proceedings of the 9th International Symposium on Hardware/Software Codesign*. ACM, New York, NY, 61–66.

Arora, N., Chandramohan, K., Pothineni, N., and Kumar, A. 2010. Instruction selection in asip synthesis using functional matching. In *Proceedings of the International Conference on*. 146–151.

Atasu, K., Luk, W., Mencer, O., Ozturan, C., and Dundar, G. 2012. FISH: Fast Instruction SyntHesis for Custom Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 20,* 99, 1–1.

Atasu, K., Mencer, O., Luk, W., Ozturan, C., and Dundar, G. 2008. Fast custom instruction identification by convex subgraph enumeration. In *Proceedings of the 2008 International Conference on Application-Specific Systems, Architectures and Processors (ASAP'08)*. IEEE Computer Society, Washington, DC, 1–6.

Bradski, G. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.

Ciesielski, M., Kalla, P., and Askar, S. 2006. Expansion diagrams: A canonical representation for verification of data flow designs. *IEEE Transactions on Computers 55*, 1188–1201.

Clark, N. T., Zhong, H., and Mahlke, S. A. 2005. Automated custom instruction generation for domain-specific processor acceleration. *IEEE Transactions on Computers 54*, 2005.

Cong, J., Fan, Y., Han, G., and Zhang, Z. 2004. Application-specific instruction generation for configurable processor architectures. In *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays (FPGA'04)*. ACM, New York, NY, 183–189.

Cong, J., Sarkar, V., Reinman, G., and Bui, A. 2011. Customizable domain-specific computing. *IEEE Design & Test of Computers 28,* 2, 6–15.

Dennard, R. H., Gaensslen, F. H., Yu, H., Rideout, V. L., Bassous, E., and LeBlanc, A. R. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits 9*, 256–268.

Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. ACM, New York, NY, 365–376.

Estrin, G. 1960. Organization of computer systems. In *Proceedings of the Western Joint IRE-AIEE-ACM Computer Conference (Western'60)*. ACM Press, New York, 33.

Fritts, J. E., Steiling, F. W., Tucek, J. A., and Wolf, W. 2009. MediaBench II video: Expediting the next generation of video systems research. *Microprocess. Microsyst. 33,* 4, 301–318.

Gomez-Prado, D., Ren, Q., Askar, S., Ciesielski, M., and Boutillon, E. 2004. Variable ordering for taylor expansion diagrams. In *Proceedings of the 9th IEEE International High-Level Design Validation and Test Workshop (HLDVT'04)*. IEEE Computer Society, Washington, DC, 55–59.

Gonzalez, R. 2000. Xtensa: A configurable and extensible processor. *IEEE Micro 20,* 2, 60–70.

González-Álvarez, C., Fernández, M., Jiménez-González, D., Alvarez, C., and Martorell, X. 2011. Automatic generation and testing of application specific hardware accelerators on a new reconfigurable OpenSPARC platform. In *Proceedings of the Workshop in Reconfigurable Computing (HiPEAC'11)*. 85–94.

Hagberg, A. A., Schult, D. A., and Swart, P. J. 2008. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy'08)*. 11–15.

Hameed, R., Qadeer, W., Wachs, M., Azizi, O., Solomatnikov, A., Lee, B. C., Richardson, S., Kozyrakis, C., and Horowitz, M. 2010. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 37–47.

Lattner, C. and Adve, V. 2004. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization(CGO'04)*. IEEE Computer Society, Washington, DC, 75.

Li, T., Sun, Z., Jigang, W., and Lu, X. 2009. Fast enumeration of maximal valid subgraphs for custom-instruction identification. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'09)*. ACM, New York, NY, 29–36.

Murray, A. C., Bennett, R. V., Franke, B., and Topham, N. 2009. Code transformation and instruction set extension. *ACM Transactions on Embedded Computing Systems 8,* 4, 1–31.

Peymandoust, A. and Pozzi, L. 2003. Automatic instruction set extension and utilization for embedded processors. In *Proceedings of the 14th International Conference on ASAP, Application-Specific Systems*.

Pothineni, N., Kumar, A., and Paul, K. 2007. Application specific datapath extension with distributed i/o functional units. In *Proceedings of the 20th International Conference on VLSI Design Held Jointly with 6th International Conference: Embedded Systems (VLSID'07)*. IEEE Computer Society, Washington, DC, 551–558.

Pozzi, L., Atasu, K., and Ienne, P. 2006. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 25,* 7, 1209–1229.

SRISC. 2012. Simply risc s1 core.

STEIN, W. ET AL. 2013. *Sage Mathematics Software (Version x.y.z)*. The Sage Development Team. Retreived from http://www.sagemath.org.

VASSILIADIS, S., WONG, S., GAYDADJIEV, G., BERTELS, K., KUZMANOV, G., AND PANAINTE, E. 2004. The MOLEN polymorphic processor. *IEEE Transactions on Computers 53,* 11, 1363–1375.

VENKATESH, G., SAMPSON, J., GOULDING, N., GARCIA, S., BRYKSIN, V., LUGO-MARTINEZ, J., SWANSON, S., AND TAYLOR, M. B. 2010. Conservation cores: reducing the energy of mature computations. *SIGARCH Comput. Archit. News 38,* 1, 205–218.

VERMA, A. K., BRISK, P., AND IENNE, P. 2007. Rethinking custom ISE identification: A new processor-agnostic method. In *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'07).* ACM, New York, NY, 125–134.

VERMA, A. K., BRISK, P., AND IENNE, P. 2010. Fast, nearly optimal ise identification with I/O serialization through maximal clique enumeration. *Trans. Comp.-Aided Des. Integ. Cir. Syst. 29,* 3, 341–354.

XILINX. 2012. *Vivado Design Suite User Guide*.

YU, P. AND MITRA, T. 2004. Scalable custom instructions identification for instruction-set extensible processors. In *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'04).* ACM, New York, NY, 69–78.

YU, P. AND MITRA, T. 2007. Disjoint pattern enumeration for custom instructions identification. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'07).* 273–278.

Received June 2013; revised November 2013; accepted November 2013