View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Ghent University Academic Bibliography

Automatic Design of Domain-Specific Instructions for Low-Power Processors

Cecilia González-Álvarez*†, Jennifer B. Sartor*, Carlos Álvarez†, Daniel Jiménez-González† and Lieven Eeckhout*

*ELIS department, Ghent University, Belgium
Email: cecilia.gonzalezalvarez, jennifer.sartor, lieven.eeckhout@elis.ugent.be
†DAC department, UPC - Barcelona Tech, Spain
Email: calvarez, djimenez@ac.upc.edu

***Abstract*—This paper explores hardware specialization of low-power processors to improve performance and energy efficiency. Our main contribution is an automated framework that analyzes instruction sequences of applications within a domain at the loop body level and identifies exactly and partially-matching sequences across applications that can become custom instructions. Our framework transforms sequences to a new code abstraction, a *Merging Diagram*, that improves similarity identification, clusters alike groups of potential custom instructions to effectively reduce the search space, and selects merged custom instructions to efficiently exploit the available customizable area. For a set of 11 media applications, our fast framework generates instructions that significantly improve the energy-delay product and speed-up, achieving more than double the savings as compared to a technique analyzing sequences within basic blocks. This paper shows that partially-matched custom instructions, which do not significantly increase design time, are crucial to achieving higher energy efficiency at limited hardware areas.**

I. Introduction

Hardware specialization has recently become a hot topic due to the end of Dennard scaling [1], which forces chip designers to focus on optimizing not only performance, but also power. Customizing hardware for each of the myriad of modern applications is infeasible. We instead explore specialization for a domain of applications, which are more likely to run on the same machine and perform similar tasks [2]. Moreover, using reconfigurable hardware to implement specialization facilitates adaptation for both new applications and different domains, extending the lifetime of the hardware.

In this paper, we identify common code sequences across applications, which can be transformed into custom instructions (CIs) that are accelerated in hardware in a domain-specialized functional unit (DSFU). We assume that CIs are executed in a low-power application-specific instruction-set processor (ASIP) [3], with an instruction-set architecture configurable either in the field (with an FPGA) or at design time.

There exists a wealth of prior work in CI design. However, this prior work is either limited to identifying acceleration opportunities within a single basic block [4], [5], and/or targeting isolated applications [6], [7]. In contrast, in this work we target CI acceleration across a domain of applications which was previously found to achieve larger speedups at small (realistic) area overheads [5]. Finding acceleration opportunities across applications, however, is challenged by the difficulty of finding exact matches of code sequences beyond the basic block level, which is why this work contributes by studying acceleration opportunities *across basic blocks* through *partial matching* of *different implementations* of code sequences.

The overarching contribution of this work is a complete and automatic methodological framework to identify fruitful CIs across a set of applications from a domain. While this search space can grow exponentially, we develop steps to tractably generate a set of potential CIs by preferably merging those with high similarity. We first use profiling to extract hot loops from the applications. We use high-level synthesis to gather execution time and hardware area measurements for several implementation versions of the potential CIs. Our framework then transforms the sequences into a new *Merging Diagram*, a canonical representation to facilitate similarity identification, and merges CIs that could be executed in the same DSFU pipeline to reduce specialized area. We cluster CIs to identify not only those that have exact functional similarity but also those with partial similarities that could cover more code while reducing the needed area for the DSFU. Finally, our framework selects a set of CIs that fit into a particular hardware area, maximizing energy efficiency and performance speedup across the applications. We demonstrate the effectiveness of the framework using 11 media benchmarks in the context of a superscalar in-order processor. We report average speed-up improvements of up to $1.98\times$ for performance and $3.35\times$ for EDP.

Overall, this paper presents the following key contributions:

- An automated framework to quickly and tractably explore the design space of accelerating a domain of applications, also exploring many code implementations of each custom instruction that will run on domain-specialized functional units.
- The Merging Diagram, a canonical representation of CIs across basic blocks, or at the loop body level, which facilitates similarity detection, which in turn achieves more than double the performance and energy improvements than for CIs within basic blocks.
- Clustering-based partial matching of code sequences to expand the opportunity for CIs to accelerate more computation within a limited area budget, which improves performance from $1.73\times$ to $1.88\times$ and energy-delay product (EDP) by $2.53\times$ to $3.04\times$ over exact matching for a limited area budget, or alternatively saves significant area for a given energy efficiency.
- A constraint-based selection mechanism that, with a novel objective function, solves the problem of choosing an energy-efficient set of specialized hardware to fit in limited area while accelerating a domain.

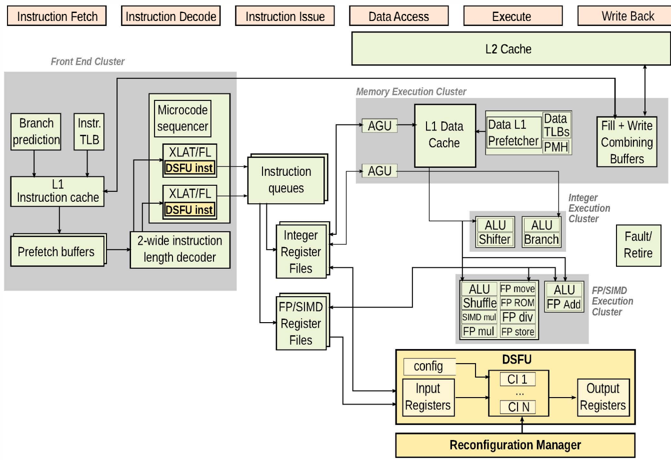

Fig. 1. Block diagram of a modified Atom processor pipeline that includes a DSFU.

II. Background and Motivation

The CIs we target in this paper aim to accelerate a domain of applications. They are executed on a domain-specialized functional unit (DSFU) integrated within the low-power processor core's datapath, as shown in Figure 1. This would be technically feasible with the last generation of FPGAs, connecting a processor core to a reconfigurable array seamlessly [8]. Deployment of DSFUs is more effective than specializing a complete processor and they are easier to program than bigger off-core accelerators. However, this kind of acceleration presents several challenges in existing design methodologies.

With a limited hardware area for implementation, we want to maximize the CIs' utilization. We can achieve this by targeting regions of code beyond basic blocks, although we must keep the number of data transfers from and to the DSFU limited to avoid high transfer overhead. In spite of the fact that there exist CIs with memory support [9], our CIs read and write data from and to the processor's register file to simplify the design and to not increase energy consumption significantly.

There are many techniques that select CIs targeting different objectives and systems. A recent survey on those methods can be found in [10]. Most known previous works extract patterns of code within a basic block [11], [12], [13]. Going beyond the basic block level is key to improve performance and justify the design effort of custom instructions, especially if the platform is an FPGA, which is reported to run a circuit implementation up to 4.6x slower than its ASIC equivalent [14]. Previous work reduces energy for general-purpose computing using a co-processor that extracts execution pipelines from the loop body [15]. Within the application-specific field, DySER [7] accelerates applications by extracting computation that runs on accelerated functional units.

Identification of CIs for a domain is challenging, because we must find similar code patterns that repeat across applications to improve hardware reusability. Inside the basic block, small patterns of partially-matched subgraphs have been identified via heuristics working on the data-flow graph [4]. Several works discuss the challenges of merging the data-flow graph representation of a CI [2], [16], [17]. We have

TABLE I. Percentages of area occupancy and EDP improvement for different CI implementations.

Benchmark	ID	Implementation	% area	% EDP improvement	
				cjpeg	gsmdec
cjpeg	ci1.1	no unroll	0.0020	+5.3	−1.0
	ci1.2	unroll 4	0.0080	+7.1	−1.0
gsmdec	ci2.1	unroll 4	0.0013	−1.0	+218.7
	ci2.2	unroll 8	0.0027	−1.0	+290.6
cjpeg+gsmdec	mci1	ci1.1 + ci2.1	0.0029	+4.5	+217.0
	mci2	ci1.2 + ci2.2	0.0087	+6.2	+227.0

previously proposed domain-specific acceleration, analyzing code sequences within the basic block, and doing exact-matching using a canonical representation [5]. While commonly used control and data flow graphs (CDFGs) hold the exact structure of a program, a canonical diagram represents the program's functionality, thus exposing common functions across applications that can become the same CI. In this paper, we also use a canonical representation, but extend the CI beyond the basic block and add partial matching.

Another issue less explored in the CI design literature is considering different circuit implementations for a CI representation as part of the CI exploration. We consider different implementations of each CI, i.e. several unrolling factors and vectorization, because they offer divergent tradeoffs and benefits. Consider, for instance, the CIs listed in Table I. For each CI, we show the benchmark where it was extracted, the ID, implementation details, the percentage of area it takes on a Virtex 7 FPGA and the EDP improvement (higher is better) of each application when that CI is implemented in the DSFU. The first four rows are application-specific CIs, while the last two ones merge the previous CIs into domain-specific ones. By exploring different implementations, we can vary the choice of which to include depending on the available area and potential EDP gains. Note that different implementations present the additional challenge of a bigger search space. We try to avoid exponential search algorithms, keeping the execution time of the framework linear with the search space size.

As we focus on low-power acceleration, the area budget is a key constraint that guides our design methodology. RISPP [6] is an adaptable ASIP where instructions also compete for area resources. Their selection objective is founded on minimizing a specific application's total time in a reconfigurable processor context, without optimizing energy. QsCores [18], although targeting coarser acceleration units, identifies and merges similar code patterns. Their selection heuristic relies on instruction coverage and area, only an approximation of our selection objective. In contrast, our design methods target not only the area budget, but try to optimize both energy and performance.

III. Custom Instructions Design Platform

We assume an in-order Intel Atom as our baseline processor, modified accordingly to the model in Figure 1. CIs execute on a DSFU that features several configurable pipelines and input and output registers of 16×128-bit entries. The DSFU reads and writes data from the processor's register files. Data is transferred into the DSFU at the beginning of the computation, and results are written back after it finishes. Loads and stores are therefore completely decoupled from CI execution. CIs that

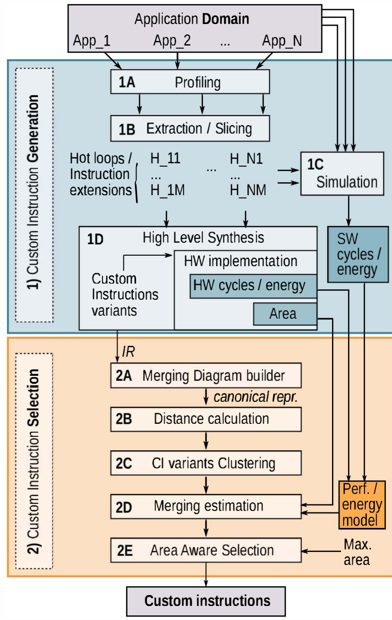

Fig. 2. Automatic framework for the implementation of merged custom instructions.

execute on the DSFU are multi-cycle and have variable latency. We do not consider parallel execution of the DSFU with the processor's functional units because it has been proven that the performance improvement is not significant enough [19]. Thus, when the DSFU executes, the rest of the pipeline stalls.

Figure 2 shows a high-level representation of our automatic framework that is a contribution of this paper. Starting with a set of applications from a domain, the *Custom Instruction Generation* module (on the top) will detect and generate CIs based on profile information. The subsequent *Custom Instruction Selection* module (on the bottom) merges those CIs to reduce area and selects the best subset for the target domain.

IV. Custom Instruction Generation

In our *Generation* module (upper half of Figure 2), we first profile each of the input applications, identifying their hot loops in step $1A$. We extract those hot loops' bodies in step $1B$. As our target CIs operate on data transferred from and to the register file, there is a transfer time before the execution starts and when it ends. Thus, memory operations are sliced and placed before and after the loop body computation. In step $1C$, we simulate the applications and their hot loops to measure cycles and energy consumption in the baseline processor. In step $1D$, we implement CIs in hardware with a High Level Synthesis (HLS) tool. We apply different unrolling and vectorization factors in the HLS transformation. Therefore, besides the implicit instruction-level parallelism of the CIs, we have also potential data-level parallelism from the HLS optimizations. From now on, we define **CI** as the high-level representation of a *loop body* that can be accelerated in hardware, and we talk about **CI variants** or only **variants** to specify distinct implementations of a CI (for example, with different unrolling factors). Thus, depending on the optimizations applied, we can obtain several variants of the same CI, as we saw in Table I. The *Generation* module produces application-specific CI variants with their implementation details.

V. Custom Instruction Selection

Most of our main contributions of this paper are implemented in the *Selection* module of the automatic framework (bottom half of Figure 2), which reduces the gigantic search space of identifying a good set of CIs across a domain of applications. We start with CI variants that are expressed in the compiler's Intermediate Representation (IR). Step $2A$ transforms them into a new canonical representation: *Merging Diagrams*. Because we use a canonical representation and create a global ordering of variables, the identification of similarities between CIs in step $2B$ is computed quickly and efficiently. This step quantifies similarity by the *distance* between pairs of CI variants. The clustering, in step $2C$, allows the framework to do both exact and *partial matching* of CI variants, the latter expanding the acceleration potential at smaller areas. In step $2D$, we perform an estimation of the new area, energy and speedup of each clustered group of variants. Finally, in step $2E$, our *area-aware selection* step solves the optimization problem of fitting the best group of candidates, that save the most energy, into a limited area.

A. Merging Diagram: Step 2A

Identifying similarity between CI variants in a non-unified representation is difficult due to the amount of unnecessary information a modern compiler IR includes. Also, a representation such as a CDFG, which expresses structural relations between operators, does not expose functional similarities, since different coding styles among applications may hide them. Therefore, we transform the codes of the CI variants expressed initially in a compiler IR, into an abstract, canonical representation: the Merging Diagram (MD).

The MD represents arithmetic and logic operations (within the basic block), and predicate information (at the loop level), both with unrestricted number of inputs and outputs. Its representation is partially based on Taylor Expansion Diagrams (TEDs) [20] and Binary Decision Diagrams (BDDs) [21]. We have used TEDs previously for exact similarity detection of CIs within a basic block [5]. However, this work improves upon the representation by including more types of computations and code sequences across basic blocks. In addition, MDs are built to facilitate the identification of partial matches in a reasonable computational time. The following definitions explain the details of our new representation, which include both a modified versions of TEDs and BDDs.

Definition 1: An Augmented TED (AugTED), is a directed acyclic graph based on linearized and reduced TEDs. It is composed of a labeled set of nodes V, a weighted set of edges E, and the terminal node. In normal TEDs, V represents variable names and E are additions/subtractions or multiplications. AugTEDs expand TED nodes to represent any kind of computation, using variable renaming. Here, labels in V can be integer, float or special. Integer and float labels represent variable types, and special labels denote a function that cannot be represented by a Taylor expansion.

Definition 2: A Linking BDD (LinBDD) is a directed acyclic graph based on reduced and ordered BDDs. It consists of a labeled set of nodes V', a set of edges E' and terminal nodes 0 and 1. LinBDDs have a third edge $Link$ to BDDs' 0-1 decision edges, which references an outside diagram, namely

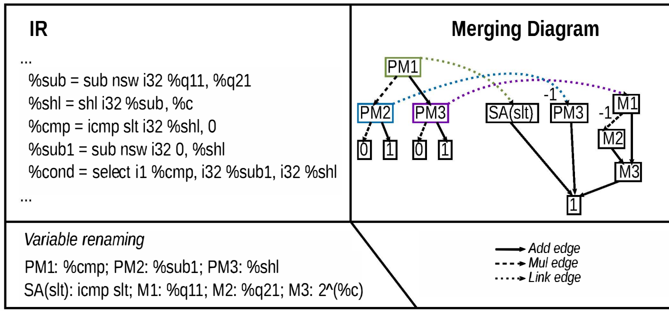

Fig. 3. Example of Merging Diagram for the IR on the left.

an AugTED. A LinBDD is constructed with the Shannon expansion of boolean functions created with the If-Then-Else (ITE) operator: $ITE\,(I, T, E) = I \cdot T + \bar{I} \cdot E$.

Definition 3: A Merging Diagram is a data structure that provides a canonical representation of a predicated code region. It consists of a set A of AugTEDs that represent computations and a set L of LinBDDs that represent control flow execution. Each $Link$ node from the members in L references a member in A.

Figure 3 shows an example of an MD for a given code sequence. The left part of the MD is a LinBDD and its nodes are linked to AugTEDs on the right by $Link$ edges. There is a special label ($SA(slt)$) that stands for a relational operator that cannot be expressed by Taylor expansions.

MDs have several advantages over CDFGs. They detect more functional similarities, and their node-labeling conventions and edge-node connections are standardized after construction. This results in a subgraph isomorphism detection of reduced complexity – linear instead of exponential for the general case – which is used in the distance calculation of Section V-B, improving the overall application performance.

1) Merging diagram construction: To build a canonical MD, we follow the steps of Algorithm 1, whose input consists of the IR of the region of code of a CI variant, which in the example of Figure 3 would be the code on the upper left. First, in lines $3-5$, we extract the polynomial representation of the computations and the branch predication of the code. With the base polynomials, we establish a precise variable renaming that unifies the variable name space in lines $6-20$, which facilitates fast similarity identification in step $2B$. We decompose each polynomial into its monomials, and we rename each variable based on the type of monomial where it is found. We find primarily adding and multiplying types of monomials, but also cover floating point and predicated types. For instance, in Figure 3 variables are renamed as A (adding) and M (multiplying) preceded by P (predicated) or S (special).

Then, in lines $21-22$ we define a strict variable ordering to perform the expansions, common to all variables implicated. As we have multiple polynomials that expand with the same set of variables, we first put variables in ascending order based on the number of times they occur. This ensures that we will have a minimum number of expansions, resulting in a more compacted MD. For the same reason, in the case of a tie in the number of instances between multiplying and adding variables, we prioritize the multiplying ones.

Algorithm 1: Merging Diagram construction

input : A region's IR code IR
output: A Merging Diagram MD

1. Array $P, P' \longleftarrow \emptyset$
2. 2D array $R \longleftarrow \emptyset$
3. $P1 \leftarrow ComputationPolynomials(IR)$
4. $P2 \leftarrow PredicationPolynomials(IR)$
5. $P \leftarrow P1 \cup P2$
6. **for** $p \in P$ **do**
7. $M \leftarrow GetMonomials(p)$
8.   **for** $m \in M$ **do**
9. $K \leftarrow GetMonomialType(m)$
10. $VM \leftarrow GetVariablesNames(m)$
11.   **for** $vm \in VM$ **do**
12.   **if** $vm \notin R$ **then**
13. $vm' \leftarrow RenameVar(vm, K)$
14.   add $< vm, vm' >$ to R
15.   **end**
16.   **end**
17.   **end**
18. $p' \leftarrow ReplaceVars(p, R)$
19.   add p' to P'
20. **end**
21. $Q \leftarrow CountOccurrencesVars(P')$
22. $O \leftarrow AscendingOrderVars(Q)$
23. $s \leftarrow$ size of $O + 1$
24. $MD \leftarrow$ < $Diagram$: $s \times s$ array, $Link$: 2D array>
25. $MD.Link \leftarrow LinkToAugTEDVars(P, R)$
26. **for** $p' \in P'$ **do**
27. $diagramExpansions(p', MD.Diagram, O)$
28. **end**
29. **return** MD

Finally, in lines $23-28$ we create an MD structure with a $Diagram$ that contains all the nodes and edges from the AugTEDs and LinBDD, except for the $Link$ edges that are kept apart. Following the variable ordering, for each rewritten polynomial we build the MD expanding each term recursively as it is done regularly with TEDs and BDDs. The resulting representation is still canonical for the assumed variable order, as is the case for regular TEDs and BDDs.

2) Global diagram of variants: To have a diagram that represents the entire design space of CI variants, thus cutting down on computation cost in later steps, we combine all the AugTED and LinBDD polynomials to obtain a global MD unified representation. For each variant, we locally rename its polynomial variables, saving the naming convention and number of instances in a global structure. Then, based on that locally collected information, we produce a global variable ordering that is fixed for the design space. Finally, MDs are produced individually for each variant with the global ordering.

B. Distance Calculation: Step 2B

We need to establish a concrete metric that measures similarities among CIs to guide the subsequent clustering step of the framework. Thus, we developed a new way to measure how dissimilar two CI variants are in terms of their functionality, using the MD.

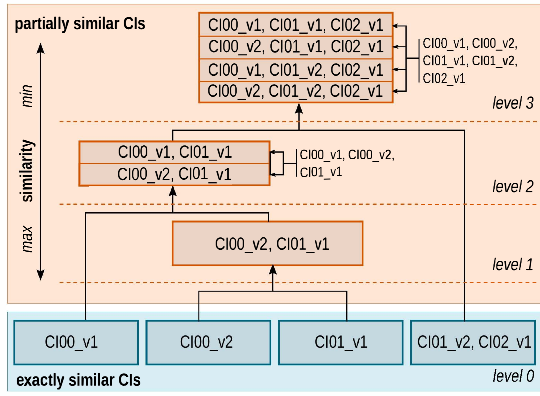

Fig. 4. Hierarchical clustering of custom instructions. Exact matching instructions are found at the bottom, while nodes closer to the root group are increasingly less similar CIs. CIXX_vy: CI with identifier XX and implementation variant y.

We perform a distance calculation for pairs of MDs of variants that do not implement the same loop body, CI_X and CI_Y. We use the previously built global diagrams to speed up this calculation. If we would not have the global, uniformed variable space that we obtained in Section V-A2, we would have to build a pair of diagrams for each pair of CIs being compared, which would be computationally very expensive. Thus, based on the pre-built global diagrams, we obtain the number of AugTED-operations and LinBDD-branches that in CI_X do not match with those in CI_Y, namely nM_X, and vice versa, nM_Y. An MD node v_x matches another MD node v_y if their labels and out edges also match. The matching information is kept for the merging step explained below in Section V-D. We also count the number of total AugTED and LinBDD nodes that each MD variant has – Tot_X and Tot_Y. Then, we compute the distance δ as:

$$\delta\left(CI_X, CI_Y\right) = average\left(nM_X/Tot_X, nM_Y/Tot_Y\right) \quad (1)$$

One-to-one distances are saved in a condensed distance matrix.

C. Hierarchical Clustering: Step 2C

For domain-specific acceleration, merging CIs together reduces energy consumption by shrinking the implementation area, or improves performance by allocating more CIs in the constrained area. We have to merge circuits of CIs that have more in common to maximize area reduction, as well as minimize the implementation overhead due to circuit multiplexing. However, with the huge set of CI variants that we obtain when we work with multiple applications, it is prohibitive to try all the possible combinations of CIs that could be grouped together. Therefore, we group CIs based on a hierarchical clustering that organizes groups by more to less functional similarity, cutting down the search space to avoid those groups that are not similar enough to be worth implementing together.

Distances between variants help to quickly decide which ones are better to merge together to reduce energy consumption. We perform hierarchical, agglomerative clustering of CI variants, obtaining a dendrogram, a tree-like structure, as shown in Figure 4, where tree leaves represent **exact** matches and internal nodes denote **partial** matches. Starting from the baseline CI variants, we form exact-matching clusters based on the distance matrix (leaves – level 0 in the figure). Then, distances between the newly formed clusters use the *complete* method to determine the agglomerative distance, that is, the maximal distance between any two variants in the cluster (levels 1 to 3, to the root). From leaves to root, we find different versions of merged variants, ordered from more to less similar.

Some of the obtained clusters may include variants that target the same CI. In Figure 4, level 0 includes two variants of the same CI: *CI00_v1* and *CI00_v2*; a variant of *CI01*, and {*CI01_v2, CI02_v1*}, that is the exact matching of two different implementations of two different CIs. Level 1 has the cluster {*CI00 v2, CI01 v1*}, which has the maximum similarity for partial matching. Variant *CI00_v1* from level 0 is clustered at level 2 with {*CI00_v2, CI01_v1*} from level 1. However, as a merged variant cannot implement a concrete CI more than once, we produce different versions that do not duplicate the loop body ($CI00$ or $CI01$) within the clusters where this problem occurs. Thus, at level 2 we generate two solutions: {*CI00_v1, CI01_v1*} and {*CI00_v2, CI01_v1*}. Since the latter already exists at level 1, we will eventually discard it, although its information is still used to generate the cluster at level 3. Note that this can induce an explosion in the number of solution clusters for a given level. In the case of a large number of cluster versions, we select a reduced group chosen heuristically by global performance speedup. Area and energy estimations at this point would slow down the generation of solutions, and by experimentation we find that for this particular task, performance speedup is a fair metric.

D. Merging Estimation: Step 2D

With the clustering formation, we obtain a bigger set of CI variants, some of which are merged to save area. We estimate the new area, performance and energy gains of merged variants in order to run the selection step with accurate information.

Based on the distance calculation information (Section V-B) of non-common matches between each pair of variants, we obtain the area of operators that are shared ($shared$) and of those that are not (non_shared). For sharing logic, we introduce multiplexers with an extra area cost, $overhead$. Thus, we calculate the area a_i of a merged CI variant i as:

$$a_i = overhead_i + shared_i + \sum_{j=1}^{N} non_shared_{ij} \quad (2)$$

To model the performance of an accelerated application, we first obtain the cycles c_l_SW that a hot loop iteration would take to execute in the baseline processor, excluding memory operations, from simulation. We also obtain the number of iterations N_it of that loop for a given execution of the benchmark. From hardware synthesis, we get the number of cycles c_HW that a CI variant takes. We calculate the cycles c_T to transfer data to the DSFU local memory as a function of the input data size. With the previous data we obtain the cycles we save executing a CI variant as:

$$c_saved = (c_l_SW - (c_HW + c_T)) \times N_it \quad (3)$$

We calculate the new number of application cycles as:

$$App_cycles = c_total_SW - c_saved \quad (4)$$

with c_total_SW as the application cycles without CIs.

Finally, the modeled energy consumption of an application that uses CIs is calculated as:

$$E_{app} = E_{baseline} + E_{CI} \quad (5)$$

with $E_{baseline}$ as baseline processor's energy model and E_{CI} the CI energy consumption. The latter is modeled as the sum of its dynamic and static components:

$$E_{CI} = P_{dynamic} \times T_{CI} + P_{static} \times T_{total} \quad (6)$$

where $P_{dynamic}$ and P_{static} are, respectively, the dynamic and static power of the hardware components that implement the CI variant, T_{CI} is the time that the CI is active, and T_{total} is the execution time of the application calculated from App_cycles.

E. Area-Aware Selection: Step 2E

Implementation area is an expensive commodity in our low-power target that largely influences the energy consumption of the final design. However, performance gains also play an important role, because a faster application would consume less energy. Therefore, in the final step of the *Selection* module, we address the performance and energy trade-off when choosing the best fitting set of CI variants for a given hardware area. We model this optimization as a Knapsack problem, in which one tries to fit a subset S of a collection of objects C – each object o_i with an intrinsic value v_i and weight w_i – within limited mass M so the sum of the values of the final subset is maximized and the sum of the weights does not exceed M. In our case, we try to fit the n CI variants, merged and not merged, within a limited hardware area A. Each c_i candidate has a value v_i that we describe later, and a hardware occupancy, hw_i. We have an additional requirement in our problem: as each CI can be selected only once, though it can be implemented by different variants – with distinct unrolling factors, or merged with other instructions – once we select one CI variant, all other variants of the same CI are invalidated for the following selection steps.

We model our problem with Mixed Integer Linear Programming (MILP). We define the constraints:

$$\sum_{i=0}^{n} c_i \times hw_i \leq A \quad ; \quad \sum_{i=0}^{n} lb_i \leq 1 \quad (7)$$

with lb_i a loop body that can be implemented by several CI variants. As our main goal is to accelerate execution and save energy, our objective function tries to maximize the energy-delay product (EDP) improvement. However, the total EDP value changes depending on the area occupancy, and thus, it cannot be deterministically precomputed before the selection starts. Therefore, to obtain consistent results, we define a specialized objective function:

$$\sum_{i=1}^{n} c_i \times \sigma_EDP_i \rightarrow max \quad (8)$$

The metric σ_EDP_i of a concrete CI variant is the value v_i in the original Knapsack problem and we calculate it as:

$$\sigma_EDP_i = \sum_{j}^{B} \|\sigma_EDP_{ij}\| \times (1 + \sigma_A_i \times A_i) \quad (9)$$

where B is the number of applications that the current variant targets; $\|\sigma_EDP_{ij}\|$ is the original application j's EDP minus the EDP with the variant, normalized to the observed maximum for that application; σ_A_i is applicable only to merged variants, since it is the percentage of area we save by merging and A_i is the percentage of the total area that the variant takes. We find that this metric selects more medium-sized variants that help to save area occupancy, and have lower overhead and lower static power than larger variants. From experimentation, we confirm that this objective gives stable results and maximizes EDP fairly among all applications.

F. Complexity

While the overall complexity of the framework varies in each step, our methodology reduces the search space to keep the exploration tractable and fast. We establish bounds based on the number of total CI variants. Selection is the most critical step and could be exponential in the worst-case. Therefore, we try to always keep a reduced number of CI variants candidates, while maintaining energy and performance efficiency.

For each input application from the set of B benchmarks we have a number of CIs C, and each CI is implemented as a variant $numVariants$ times. The total number of variants CV processed to build MDs by Algorithm 1 is determined as $CV = \sum_{i=1}^{B} \sum_{j=1}^{C_i} numVariants_j$. The complexity of calculating distances between pairs of MDs (Section V-B) is $O(CV \times (CV - C - 1))$. However, the key design decision here is to have a global MD, which obviates the need for a new MD to be computed to compare each pair of variants, speeding up the calculation. Finally, by performing the hierarchical clustering step explained in Section V-C, and using a heuristic to limit the number of cluster versions per level, the final number of generated solutions that the selection of Section V-E processes is within the bounds of $O(CV)$. We thus retain the most promising CI candidates, in terms of area, performance and energy efficiency, while making sure the selection step's complexity does not explode exponentially.

VI. Evaluation

A. Experimental Setup

We now describe the setup and experimental evaluation of our automated exploration framework. We evaluate the framework with eleven applications from the media domain: `cjpeg`, `djpeg`, `gsmdec`, `gsmenc`, `mpeg2enc`, `optflow`, `rawcaudio`, `rawdaudio`, `susan`, `tmndec` and `tmnenc`. We identify hot regions of code with the LLVM profiler [22] and compile all applications with LLVM-Clang with an unrolling factor of 8, automatic vectorization, and optimization $-O2$ as the baseline. Unrolled, non-vectorized code sequences in the LLVM IR are analyzed to generate the polynomials for the Merging Diagrams. Software cycles are measured with the Sniper simulator [23], with changes to accurately simulate an Intel Atom processor running at 1.6 GHz. Power measurements on Sniper were obtained with McPAT [24]. We synthesize the DSFU's description from C code with Vivado HLS 2013.3 [25] to obtain the circuit design cycles and area consumption for a target Xilinx Virtex 7 (XC7VX690T) FPGA that runs at 400 MHz – 4× slower than the baseline processor. DSFU power estimations are obtained with the Xilinx Power Estimator (XPE). Cycles and power data are fed into the models of Section V-D to obtain results. Although we use an FPGA as a testing platform, we do not consider run-time reconfiguration in this work. We use the Fastcluster library [26] for hierarchical clustering, and the interface for the CPLEX optimizer [27] in the selection module is OpenOpt [28].

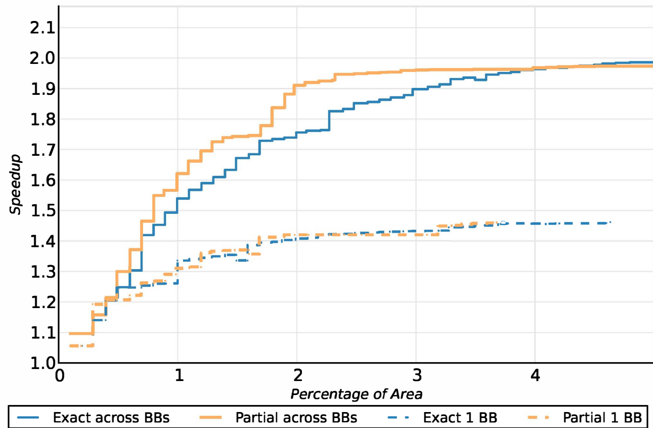

Fig. 5. Average speedup versus percentage of area occupancy of the DSFU for exact and partial matching methods, targeting one or many basic blocks.

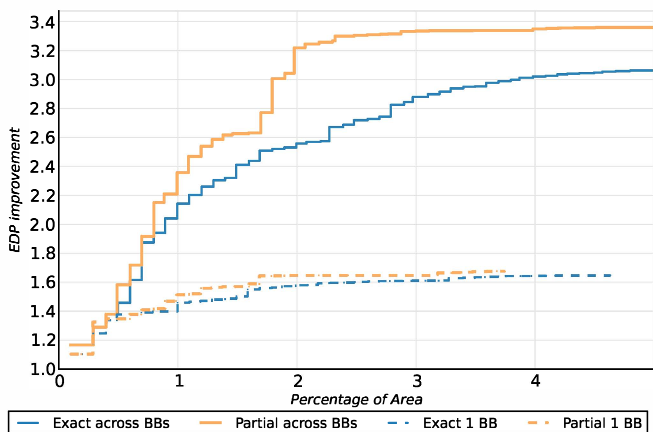

Fig. 6. Average EDP improvement versus percentage of area occupancy of the DSFU for exact and partial matching methods, targeting one or many basic blocks.

B. Results Discussion

We now present experiments and results to assess how well our framework can identify custom instructions to be accelerated by a DSFU in hardware, measuring both speedup and improvement in EDP across various areas.

Figure 5 presents a comparison of different configurations of our framework, with DSFU area on the x-axis expressed as a percentage of the Virtex 7's area, and the average performance speedup across the domain on the y-axis. Figure 6 shows the same comparison, but this time with average EDP improvement on the y-axis. Dashed lines show improvements achieved when we use CIs targeting code within basic blocks. At the larger areas, performance improvement reaches a maximum of $1.48\times$ and EDP improvement goes up to $1.67\times$ the baseline. We compare this to the solid lines in the figures, which target code regions across basic blocks. In this case, speedup reaches a maximum of $1.98\times$ and EDP improvement goes up to $3.35\times$. Considering regions with multiple basic blocks gives us a significant boost in both performance and energy efficiency, because we are able to accelerate 31% more statically counted body loops than with one basic block. Also, CIs across basic blocks cover 41% more dynamic instructions on average. Exploring CIs across basic blocks covers more code, expands the acceleration opportunities, and thus achieves higher speedups.

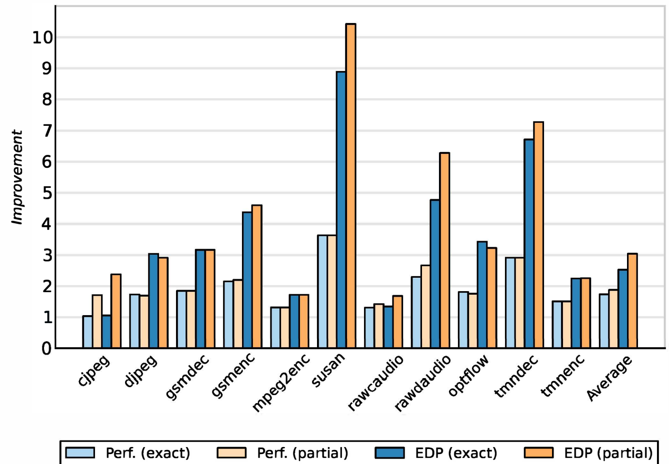

Fig. 7. Speedup (*Perf.*) and EDP improvement for each benchmark at a limited implementation area (1.8%) across basic blocks.

In the same figures, we analyze the efficacy of exact versus partial matching by comparing blue and orange lines, respectively. Note that partial matching choices include all those CIs matched with exact, and then additional CIs that could be partially matched. We start seeing a difference around 0.5% of the area across basic blocks, noting that partial matching achieves larger speedups and EDP improvements as compared to exact matching, given the same area. For instance, with a limited area budget (1.8%), we observe a speedup of $1.88\times$ and an EDP improvement of $3.04\times$ when using partially matched CIs, while with exact matching we obtain a speedup of $1.73\times$ and an EDP improvement of $2.53\times$. At 2.2% of the area, the EDP improvement difference is more noticeable, $2.57\times$ against $3.25\times$. Alternatively, we see that for a given EDP improvement, partial matching saves area. For an EDP improvement of $3\times$, exact matching takes 4% of the area, whereas partial matching takes only 1.8% of the area: a savings of 55% of the chip's reconfigurable area. This is important as the area available for the reconfigurable DSFU in a low-end processor like the one evaluated would be much less than the area available in a Virtex 7.

Figure 7 shows results for speedup and EDP improvement for each benchmark at the limited area (1.8%) discussed above, comparing exact and partial matching across basic blocks. As our selection optimizes for EDP, we see larger EDP gains than speedup gains, when going from exact to partial matching. The speedup difference is moderate because of our selection objective. A power-hungry CI with high speedup but low energy efficiency will not be selected. Looking at the EDP of particular benchmarks, only two benchmarks marginally suffer a speedup and energy efficiency reduction: `djpeg` and `optflow`. However, most benchmarks have a significant improvement in their performance and EDP. For instance, the energy efficiency of `cjpeg` improves from $1.06\times$ to $2.38\times$, for `susan` goes from $8.88\times$ to $10.42\times$, and `rawdaudio` gets $4.76\times$ with exact similarities and $6.28\times$ with partial ones. The average of all EDP improvements with partial matching is

positive and therefore fair to all applications. Partial similarities contribute to area shrinking, which is key to energy efficiency. For example, with partial similarities one of the selected CIs targets hot regions in seven different benchmarks, which results in an area reduction of 80% compared to exact ones.

VII. Conclusions

This paper presents a methodology and framework to automatically extract custom instructions from a domain of applications, ultimately selecting those that achieve the highest performance improvements and energy efficiency when accelerated. To do so, our proposal explores the design space of tightly-integrated configurable functional units of limited size that accelerate applications across a domain. The presented framework transforms code sequences at the loop body level into a canonical representation, which facilitates fast similarity detection, even considering several implementations of each custom instruction. We then cluster CIs to be able to find partially-matching sequences to minimize specialized area. Our experimental results with 11 media benchmarks show that looking across basic blocks achieves a speedup of $1.98\times$ and an EDP improvement of $3.35\times$, a significant gain over looking within a single basic block (speedup of $1.48\times$ and EDP improvement of $1.67\times$). Across basic blocks, partial matching compared against exact matching is crucial for achieving larger performance ($1.88\times$ versus $1.73\times$) and EDP improvements ($3.04\times$ versus $2.53\times$) for a limited hardware area (1.8%), or for a given energy efficiency, significantly reducing the needed hardware area. The presented work shows the applicability of introducing configurable accelerators with limited area inside simple processors to accelerate a large number of applications from a domain, improving the system's energy efficiency.

Acknowledgment

We thank the anonymous referees for their valuable feedback. This work is supported by the European Research Council under the European Communitys Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295, the Spanish Government under the Severo Ochoa program (SEV-2011-00067), the Spanish Ministry of Science and Technology (TIN2012-34557) and the Generalitat de Catalunya (MPEXPAR, 2014-SGR-1051). We thank the Xilinx University Program for its hardware and software donations.

References

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

[2] H. Huang, T. Kim, and Y. Hoskote, "Edit distance based instruction merging technique to improve flexibility of custom instructions toward flexible accelerator design," *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, pp. 219–224, 2014.

[3] K. Keutzer, S. Malik, and A. R. Newton, "From asic to asip: The next design discontinuity," in *Computer Design: VLSI in Computers and Processors*. IEEE, 2002, pp. 84–90.

[4] N. T. Clark, H. Zhong, and S. A. Mahlke, "Automated Custom Instruction Generation for Domain-Specific Processor Acceleration," *IEEE Transactions on Computers*, vol. 54, no. 10, 2005.

[5] C. González-Álvarez, J. B. Sartor, C. Álvarez, D. Jiménez-González, and L. Eeckhout, "Accelerating an application domain with specialized functional units," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 47:1–47:25, Dec. 2013.

[6] L. Bauer, M. Shafique, and J. Henkel, "Run-time instruction set selection in a transmutable embedded processor," in *Proceedings of the 45th annual Design Automation Conference*. ACM, 2008, pp. 56–61.

[7] V. Govindaraju, C. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "DySER: Unifying Functionality and Parallelism Specialization for Energy Efficient Computing," *Micro, IEEE*, vol. 32, no. 5, pp. 38–51, 2012.

[8] L. Bauer, M. Shafique, and J. Henkel, "Concepts, architectures, and run-time systems for efficient and adaptive reconfigurable processors," in *Adaptive Hardware and Systems (AHS)*. IEEE, 2011, pp. 80–87.

[9] M. Haaß, L. Bauer, and J. Henkel, "Automatic Custom Instruction Identification in Memory Streaming Algorithms," in *CASES*, 2014.

[10] L. Jówiak, N. Nedjah, and M. Figueroa, "Modern development methods and tools for embedded reconfigurable systems: A survey," *Integr. VLSI J.*, vol. 43, no. 1, pp. 1–33, Jan. 2010.

[11] K. Atasu, L. Pozzi, and P. Ienne, "Automatic Application-Specific Instruction-Set Extensions Under Microarchitectural Constraints," *Int. J. of Parallel Programming*, vol. 31, no. 6, pp. 411–428, Dec. 2003.

[12] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors," *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2004.

[13] L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1209–1229, Jul. 2006.

[14] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," in *FPGA'06*. New York, NY, USA: ACM, 2006, pp. 21–30.

[15] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44, 2011, pp. 12–23.

[16] M. Zuluaga and N. Topham, "Design-space exploration of resource-sharing solutions for custom instruction set extensions," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 12, pp. 1788–1801, 2009.

[17] M. Stojilovic, D. Novo, L. Saranovac, P. Brisk, and P. Ienne, "Selective flexibility: Creating domain-specific reconfigurable arrays," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 5, pp. 681–694, 2013.

[18] G. Venkatesh, J. Sampson, N. Goulding-Hotta, S. K. Venkata, M. B. Taylor, and S. Swanson, "QsCores: Trading Dark Silicon for Scalable Energy Efficiency with Quasi-Specific Cores," *MICRO-44*, p. 163, 2011.

[19] J. E. Carrillo and P. Chow, "The effect of reconfigurable units in superscalar processors," in *FPGA'01*. ACM Press, 2001, pp. 141–150.

[20] M. Ciesielski, P. Kalla, and S. Askar, "Taylor Expansion Diagrams: A Canonical Representation for Verification of Data Flow Designs," *IEEE Transactions on Computers*, pp. 1–11, 2006.

[21] R. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[22] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *CGO*, Mar 2004, pp. 75–88.

[23] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *SC '11*. ACM, 2011, pp. 52:1–52:12.

[24] S. Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO-42*, Dec 2009, pp. 469–480.

[25] Xilinx, "Vivado high-level synthesis," http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, 2014.

[26] D. Müllner, "fastcluster: Fast hierarchical, agglomerative clustering routines for R and Python," *Journal of Statistical Software*, vol. 53, no. 9, pp. 1–18, 2013.

[27] IBM, "ILOG CPLEX Optimizer," http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/, 2014.

[28] D. Kroshko, "OpenOpt: Free scientific-engineering software for mathematical modeling and optimization," 2007–2014. [Online]. Available: http://www.openopt.org/