MInGLE: An Efficient Framework for Domain Acceleration using Low-Power Specialized Functional Units

Cecilia González-Álvarez, Ghent University & Universitat Politècnica de Catalunya
Jennifer B. Sartor, Ghent University & Vrije Universiteit Brussel
Carlos Álvarez, Universitat Politècnica de Catalunya
Daniel Jiménez-González, Universitat Politècnica de Catalunya
Lieven Eeckhout, Ghent University

The end of Dennard scaling leads to new research directions that try to cope with the utilization wall in modern chips, like the design of specialized architectures. Processor customization utilizes transistors more efficiently, optimizing not only for performance, but also for power. However, hardware specialization for each application is costly and impractical due to time-to-market constraints. Domain-specific specialization is an alternative that can increase hardware reutilization across applications that share similar computations. This paper explores the specialization of low-power processors with custom instructions (CIs) that run on a specialized functional unit. We are the first, to our knowledge, to design CIs for an application domain *and* across basic blocks, selecting CIs that maximize both performance and energy efficiency improvements.

We present MInGLE (Merged Instructions Generator for Large Efficiency), an automated framework that identifies and selects CIs. Our framework analyzes large sequences of code (across basic blocks) to maximize acceleration potential, while also performing partial matching across applications to optimize for reuse of the specialized hardware. In order to do this, we convert the code into a new canonical representation, the *Merging Diagram*, which represents the code's functionality instead of its structure. This is key to being able to find similarities across such large code sequences from different applications with different coding styles. Groups of potential CIs are clustered depending on their similarity score to effectively reduce the search space. Additionally, we create new CIs that cover not only whole-body loops, but also fragments of the code, in order to optimize hardware reutilization further. For a set of eleven applications from the media domain, our framework generates CIs that significantly improve the energy-delay product and performance speed-up. CIs with the highest utilization opportunities achieve an average energy-delay product improvement of $3.8\times$ compared to a baseline processor modeled after an Intel Atom. We demonstrate that we can efficiently accelerate a domain with partially-matched CIs, and that their design time, from identification to selection, stays within tractable bounds.

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, Domain-specific, Canonical representation, Clustering

1. INTRODUCTION

Lately power efficiency has become a factor as important as performance in processor design. Fixed power budgets and the end of Dennard scaling [Dennard et al. 1974] have

This article is an extension of "Automatic Design of Domain-Specific Instructions for Low-Power Processors", by C. González-Álvarez, J. B. Sartor, C. Álvarez, D. Jiménez-González and L. Eeckhout, presented at the 26th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2015). David Whalley served as Editor-in-Chief for this submission.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM. 1544-3566/2015/12-ARTXXXX $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

challenged the sustainable growth in computer performance of the last decades, resulting in the *utilization wall* [Goulding-Hotta et al. 2011], which in turn has stimulated the adoption of custom computing. Specialized co-processors and datapath accelerators can take advantage of the under-utilized transistors of modern chips, implementing energy-efficient acceleration hardware that complements the main processor.

Extensible processors, also known as Application-Specific Instruction Processors (ASIPs) [Keutzer et al. 2002], are a solution that balances performance and flexibility and that still maintains the energy efficiency benefits of specialization. The design of ASIPs involves augmenting a general-purpose processor with instructions that are customized for a particular application, and that execute on specialized functional units. This design process can be simplified with automated techniques that implement those *custom instructions (CIs)*. Using CIs is less complex than specializing a complete processor and they are easier to program than bigger off-core accelerators. However, if CIs are not frequently executed, the acceleration benefits will not compensate for the overall energy consumption. Usually, the automated design of application-specific CIs is divided into the *identification* of CI candidates, and the *selection* of the best CI configuration. Additionally, we improve the design with *optimizations*.

In this paper, we explore the design space of potential CIs that accelerate sequences of operations across a domain of applications. We target domain-specific acceleration because applications often perform similar computations, and this can improve hardware reusability and acceleration opportunities, as opposed to prior research that has targeted isolated applications [Bauer et al. 2008; Govindaraju et al. 2012]. While previous work has also explored specializing hardware across a domain, they have either been targeting a much larger accelerator [Gupta et al. 2011; Venkatesh et al. 2011], targeting hardware that is dynamically reconfigured while the application is running [Bauer et al. 2008], or also targeting CIs that run on a specialized functional unit [Clark et al. 2005; González-Álvarez et al. 2013], but analyzing smaller sequences of code (basic blocks) to accelerate. We instead find CIs across basic blocks, which has the potential to accelerate larger sections of the applications; however, it is more challenging to reuse hardware as the larger code sequences analyzed are more likely to differ across applications. To tackle this problem, we transform the code into a new representation, called the *Merging Diagram*, that 1) is canonical, which represents the functionality of the code instead of its structure, 2) supports predication, which is necessary to cover loop bodies, and 3) facilitates finding partial matches of code sequences across CIs to make the most effective use of the hardware. We aim to share common operations of sequences of instructions in order to cover more code while taking up less hardware area. For instance, the functions $F1 = a + b + c$ and $F2 = a * b + c$ can be collapsed into a single instruction that shares the circuit of one addition, and selects between an addition and a multiplication. We take this a step further by matching selected parts of CIs (fragments) with fruitful full CIs, in order to accelerate even more code while adding minimal additional area.

We implement our contributions within MInGLE (Merged Instructions Generator for Large Efficiency), an automated methodological framework to design CIs that execute on a Domain-Specific Functional Unit (DSFU) (Section 2). We adapted the application-specific CI design process to suit an application domain. Our framework has identification and selection steps (modified for our goals), and two optimization steps for better use of the implementation area. Because we are solving the optimization problem of maximizing performance speedup *and* minimizing hardware area and energy usage for an entire domain of applications, designing our framework to generate the most fruitful CIs in a reasonable amount of time was a key goal. To identify CIs, we first profile the applications to extract hot loops that are implemented as po-

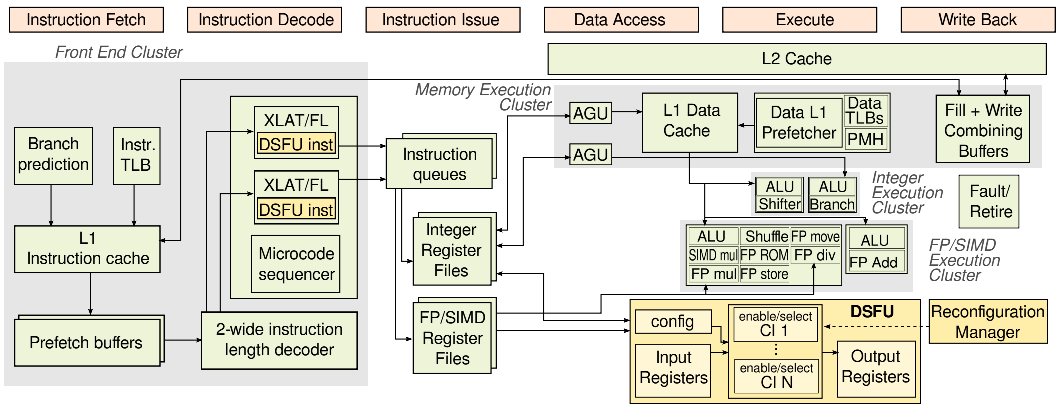


Fig. 1. Block diagram of a modified Intel Atom processor pipeline that includes a DSFU.

tential CIs using high-level synthesis. We also consider different implementations of code sequences, such as with different loop unrolling factors. We transform the CIs into a novel canonical *Merging Diagram* that facilitates similarity identification (Section 3). Then, as an optimization, the framework clusters these candidates to match not only CIs with exact functional similarity but also those with partial similarities (Section 4). CIs with overlapping functionality are merged such that the overlapping sections will be accelerated by the same DSFU pipeline, reducing specialization area usage. While datapath merging is a known approach for area savings, our method is suited for domain-specific CIs across basic blocks. We also introduce a new optimization that matches fragments of CIs with previously generated full CIs (Section 5). Finally, our framework selects a set of CIs that fit into a particular hardware area, maximizing energy efficiency and performance speedup across the applications (Section 6). For the evaluation (Section 8), we compare the effectiveness of exact, partial and fragment matching across and within basic blocks. We also evaluate different design parameters of the framework across a range of hardware areas. We demonstrate the validity of the framework using 11 media benchmarks in the context of a simulated superscalar in-order processor that is modeled after the Intel Atom. For the technique with fragments, we report average speed-up improvements across applications of up to $2.1\times$ for performance and $3.8\times$ for energy-delay product (EDP) improvement compared to the baseline processor. To summarize, the main contribution of this paper is an automated framework that:

— identifies CIs *across basic blocks* and *across the domain*, with *different implementations* of code sequences;
— selects the best CI configuration that is fair *across the domain*;
— optimizes using *partial matching* to create a reusable circuit;
— optimizes for both performance and energy efficiency at the smallest implementation areas with *CI fragments*.

2. SYSTEM OVERVIEW

We aim to accelerate a domain of applications for the embedded market, where both performance speedup and energy consumption are important. Thus, the baseline processor is a low power Intel Atom [Halfhill 2008], described in Section 2.1. The framework that we introduce in Section 2.2 extends the basic ISA with instructions that accelerate the programs by executing a bundle of predicated operations.

2.1. Target Architecture Model

We identify CIs that execute intermittently at different points of different programs. We consider a loop body, made up of one or several basic blocks, to be the basic portion of code that defines our CIs. CIs are multi-cycle, have variable latency, and use few inputs to produce few outputs. They access data through the processor's register files; therefore, input and output data sizes are always within some established limits. CIs are calculation intensive, and if they support branches, their execution can be predicated, computing if/else paths of branches in parallel. They exploit sub-word parallelism as SIMD instructions do, but they also exploit instruction-level parallelism, executing different operations in parallel. We achieve performance speedup because of this additional parallelism. Additionally, we can reduce resource contention in several pipeline stages due to collapsing several instructions into one, due to a minimization of branch penalty through predication, and to a reduction of cache instruction misses.

The target architecture is a low-power processor with a tightly coupled Domain-Specific Functional Unit (DSFU) that executes the CIs. We assume a superscalar in-order Intel Atom [Halfhill 2008] as our baseline processor, modified accordingly to the model in Figure 1 with an embedded DSFU. The DSFU architecture template is reconfigurable, and implements several CIs that share two fixed arrays of input and output registers, which are directly connected to the processor's register files. Those private registers are disjoint in order to overlap load and store operations. If the number of inputs or outputs of a CI exceeds the capacity of the register file's read and write ports, its execution requires pre/post execute stages for extra data transfers. We count on the same data bandwidth as for other processor instructions using the processor's memory hierarchy. Despite the fact that there exist CIs with memory support [Haaßet al. 2014], our CIs read and write data from and to the processor's register file to simplify the design and to not greatly increase power consumption beyond the processor's baseline. We also do not consider parallel execution of the DSFU with the processor's functional units because the performance improvement is not significant enough [Carrillo and Chow 2001]. Thus, when the DSFU executes, the rest of the pipeline stalls.

The *config* signal controls the CIs' connections to those registers and selects the CI datapath to execute. The reconfiguration manager is connected to memory, where it can read a new configuration with a different implementation of CIs and modify the whole reconfigurable area. Depending on the chosen reconfigurable hardware technology, CIs on the DSFU execute at a different frequency than on the main processor. The DSFU's period should be a multiple of the core's period to simplify changing the clock domain and sharing data across both clock domains. For instance, there is a 1:4 ratio between an Intel Atom core operating at 1.6 GHz and a DSFU operating at 400 MHz.

We extend the x86-64 based ISA with the following Atom-compatible instructions:

— DSFU_exec config, i/r: to execute the custom instruction specified by the *config* argument. The second operand may be used to transfer data to a determined input register. Latency: $4 \times C$, with C the number of internal cycles that the CI takes in the DSFU, scaled to the core's clock.
— RF2DSFU dsfu_in, r and DSFU2RF r, dsfu_out: to move data from/to the core register file r to/from the DSFU input registers. Latency: 1 cycle.
— DSFU_config m, i: i number of bytes are read from m memory location to reconfigure the CI implementation space of the DSFU. Latency: $f(i)$, as it depends on the number of bytes to read and the memory bandwidth.

2.2. MInGLE Framework

Figure 2 shows a high-level representation of our automated framework MInGLE, composed of five modules. The section where each module is explained is annotated in

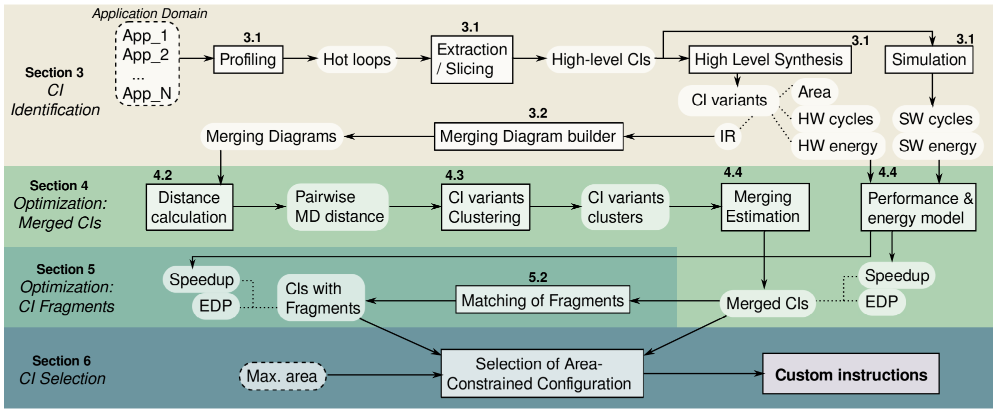


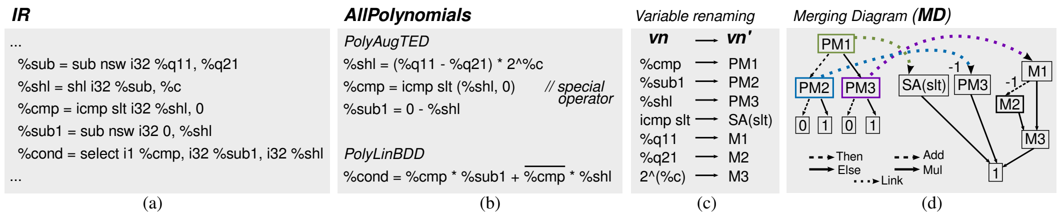
Fig. 2. MInGLE automated framework for the implementation and generation of partially-merged CIs.

the figure. Starting with a set of applications from a domain, the *Candidates Extraction* module (at the top, Section 3.1) identifies and generates potential CI candidates based on profile information. This paper's core work is implemented in subsequent modules, with the compound objective of generating energy-efficient CIs across a domain of applications, while making the exponential search space tractable. The *Canonicalization* module (Section 3.2) transforms CIs expressed in the compiler's Intermediate Representation (IR) into a new canonical representation: Merging Diagrams. The next module, *Merged CIs Generation*, in Section 4, first calculates the pairwise distances used in the identification of similarities between CIs (Section 4.2). Because we use a canonical representation and create a global ordering of variables, that step is computed quickly and efficiently. Then, the clustering (Section 4.3) allows the framework to do both exact and partial matching of CIs, with the latter an optimization that enhances the CIs' reutilization across applications. The *Merging Estimation*, together with the *Performance and Energy models* (Section 4.4), quantifies the advantages of the generated CIs, estimating the new area, energy and speedup of each clustered group of CIs. Afterwards, the *CI Fragments Generation* (Section 5) module implements a new optimization technique to obtain larger improvements in performance and energy efficiency with very limited hardware area. Finally, module *CI Selection* (Section 6) solves the optimization problem of fitting the best group of candidates that save the most energy across the domain, into a limited area.

3. IDENTIFICATION OF DOMAIN-SPECIFIC CUSTOM INSTRUCTIONS

3.1. Candidates Extraction: From Application Code to Hardware Acceleration

In the *Candidates Extraction* module of MInGLE (upper side of Figure 2), we first profile each of the input applications, identifying their hot loops in the *Profiling* step. We extract those hot loops' bodies as isolated code that we can execute as new CIs in the *Extraction/Slicing* step. As our target CIs operate on data transferred from and to the register file, memory operations are sliced and placed before and after the loop body computation. Although we only consider inner-most loop bodies for acceleration, if the number of iterations of the loop is known at compile time, then the inner loop is completely unrolled. Then, the next inner-most loop may also be considered for acceleration. All function calls in a loop are inlined and considered for acceleration if their functionality is known at compile time (i.e., mathematical functions such as *pow()*). In the *Simulation* step, we simulate the applications with the identified high-level CIs to measure cycles and energy consumption in the baseline processor. In the *High Level*



```
IR
...
  %sub = sub nsw i32 %q11, %q21
  %shl = shl i32 %sub, %c
  %cmp = icmp slt i32 %shl, 0
  %sub1 = sub nsw i32 0, %shl
  %cond = select i1 %cmp, i32 %sub1, i32 %shl
...
```

```
AllPolynomials
PolyAugTED
%shl = (%q11 - %q21) * 2^%c
%cmp = icmp slt (%shl, 0)      // special
                                 operator
%sub1 = 0 - %shl

PolyLinBDD
%cond = %cmp * %sub1 + %cmp̅ * %shl
```

Variable renaming

vn		vn'
%cmp	⟶	PM1
%sub1	⟶	PM2
%shl	⟶	PM3
icmp slt	⟶	SA(slt)
%q11	⟶	M1
%q21	⟶	M2
2^(%c)	⟶	M3

(a) (b) (c) (d)

Fig. 3. From an IR (a), polynomials are extracted (b), variables renamed (c), and an MD is created (d).

Synthesis (HLS) step, we implement CIs in hardware, obtaining their area occupancy, communication and execution cycles and energy, and, in parallel, their Intermediate Representation used in subsequent modules. At this step, we evaluate the performance of every CI in hardware compared to its software version, discarding those with worse performance in hardware. This could be the case with regions of code that have a higher communication latency than execution latency. In the HLS transformation, we apply different vectorization factors, and unroll the outer-most loop of the hot region. Therefore, besides the implicit instruction-level parallelism of the CIs, we also have potential data-level parallelism from the HLS optimizations. From now on, we talk about a **CI** as the high-level representation of a loop body or inline function that can be accelerated in hardware, and we talk about **CI variants** or only **variants** to specify distinct implementations of a CI. Different implementations include different unrolling factors and, potentially, also vectorization. Thus, depending on the compiler optimizations applied, we obtain several application-specific variants with their implementation details.

3.2. Canonicalization of Custom Instructions with Merging Diagrams

Identifying similarity between CI variants in a non-unified representation is difficult due to the amount of unnecessary information a modern compiler's Intermediate Representation (IR) includes. Also, a representation such as a data-flow graph, which expresses structural relations between operators, does not expose functional similarities, since different coding styles among applications may hide them. Therefore, in the *Canonicalization* step in Figure 2, we transform 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) [Ciesielski et al. 2006] and Binary Decision Diagrams (BDDs) [Bryant 1986]. We have successfully used TEDs for CI similarity detection within a basic block in our previous work [González-Álvarez et al. 2013], but extending CIs beyond the basic block level needs a new representation that includes predication. Also, the codes we process include operations that cannot be expressed as polynomial functions, which are the base of the TED representation. The following definitions explain the details of our new representation, which include both modified versions of TEDs and BDDs.

Definition 3.1. 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 1. 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 a function that cannot be represented by a Taylor expansion.

```
Algorithm 1: Merging Diagrams construction
input  : Array of CIs' IR codes $AllIRs$
output: Merging Diagrams $AllMDs$
Array $AllPolynomials, RewrittenPolynomials \longleftarrow \emptyset$
2D array $RenamedMap \longleftarrow \emptyset$
for $IR \in AllIRs$ do
    $PolyAugTED \leftarrow ComputationPolynomials(IR)$
    $PolyLinBDD \leftarrow PredicationPolynomials(IR)$
    add ($PolyAugTED \cup PolyLinBDD$) to $AllPolynomials$
end
for $p \in AllPolynomials$ do
    $M \leftarrow GetMonomials(p)$
    for $m \in M$ do
        $MonomialType \leftarrow GetMonomialType(m)$
        $VarNames \leftarrow GetVariablesNames(m)$
        for $vn \in VarNames$ do
            if $vn \notin RenamedMap$ then
                $vn' \leftarrow renameVar(vn, MonomialType)$
                add $< vn, vn' >$ to $RenamedMap$
            end
        end
    end
    $p' \leftarrow replaceVars(p, RenamedMap)$
    add $p'$ to $RewrittenPolynomials$
end
$VarsOccurrences \leftarrow countOccurrencesVars(RewrittenPolynomials)$
$OrderedVars \leftarrow ascendingOrderVars(VarsOccurrences)$
$s \leftarrow$ size of $OrderedVars + 1$
Array $AllMDs \longleftarrow \emptyset$
for $p' \in RewrittenPolynomials$ do
    $MD \leftarrow < Diagram$: $s \times s$ array, $Link$: 2D array$>$
    $MD.Link \leftarrow linkToAugTEDVars(p', RenamedMap)$
    $diagramExpansions(p', MD.Diagram, OrderedVars)$ add $MD$ to $AllMDs$
end
return AllMDs
```

Definition 3.2. 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 nodes have BDDs' 0-1 decision edges, and additionally a third edge $Link$ that references an outside diagram, namely 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 3.3. 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 computation and a set L of LinBDDs that represent control flow execution. $Link$ edges from the nodes in each member of L references a member in A.

Figure 3(d) 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 via $Link$ edges. There is a special label ($SA(slt)$) that stands for a relational operator that cannot be expressed by Taylor expansions. Details on the construction of the MD, with explanation of the example of Figure 3, follow.

3.2.1. Merging diagrams construction. To build all the canonical MDs of a group of CI variants, we follow the steps of Algorithm 1. We start processing in lines $3 - 7$ the set

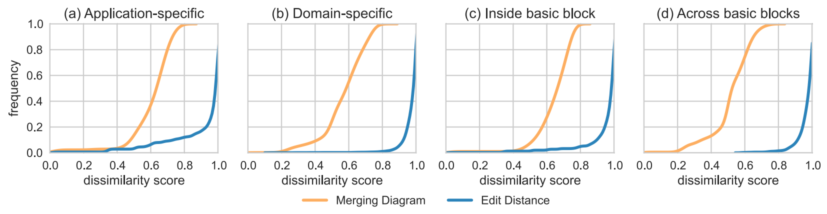


Fig. 4. Cumulative distribution of the similarity score obtained with *Edit Distance* and *Merging Diagram*, for pairs of CIs that (a) accelerate one application, (b) accelerate several applications, (c) cover code inside one basic block and (d) cover code across basic blocks.

of IRs of all the CI variants' code regions. Figure 3(a) shows an extract of an IR example with arithmetic, relational and conditional selection instructions. For each one of the IRs, we extract the polynomial representation of the computations ($PolyAugTED$) and the branch predication ($PolyLinBDD$) of the code, as illustrated in Figure 3(b). With those base polynomials, we establish a precise variable renaming that unifies the variable name space in lines $8-22$, which facilitates fast similarity identification in Section 4.2. To do so, we decompose each polynomial into its monomials (line 9), and we rename each variable based on the type of monomial where it is found (lines $10-19$). We find primarily adding and multiplying types of monomials, but also cover floating-point and predicated types. For instance, in Figure 3(c), variables are renamed as A (adding) and M (multiplying) preceded by P (predicated) or S (special).

Then, in lines $23-25$ we define a strict variable ordering for the expansions. As we have multiple polynomials that expand with the same group of variables, we first set 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.

Finally, in lines $27-31$, for each rewritten polynomial, we create an MD structure with a condensed matrix $Diagram$ that contains all the nodes and edges from the AugTEDs and LinBDD; it is thus of size $s\times s$, with s the precalculated size of all the variables involved. $Link$ edges are though kept apart in a two-dimensional array. Following the variable ordering, we build the MD expanding each term recursively as it is done regularly with TEDs and BDDs. We show in Figure 3(d) the resulting representation, which is still canonical for the variable order, as it is the case for regular TEDs and BDDs.

3.2.2. Global diagram of variants. In order to cut down computation costs in later steps, it is required to have a diagram that represents the entire design space of CI variants. To do so, 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 information, we produce a global variable ordering that is fixed for the design space, and produced MDs for each variant with the global ordering.

4. OPTIMIZATION: MERGING DOMAIN-SPECIFIC CUSTOM INSTRUCTIONS

4.1. Motivation and Comparison with the State-of-the-art

To show how the canonical Merging Diagram is better at finding similarities, we compare it to the structural graph-based techniques that state-of-the-art specialization research has used [Clark et al. 2005; Govindaraju et al. 2012; Huang et al. 2014; Venkatesh et al. 2011]. We quantitatively compare the ability of the different representations, a Directed Acyclic Graph (DAG) versus our canonical MD, to facilitate partial merging of CIs. We present each technique's ability to match CIs in the form of similarity scores between different CIs to assess how much hardware could be shared, or alternatively, how much code could be accelerated.

Performing partial matching of different code graphs is a difficult problem. Partial merging of CIs has been studied for small patterns of code, with a few inputs and outputs and no control code (inside basic blocks) [Clark et al. 2005]. In that case, the problem could be modeled as subgraph isomorphism detection, which means matching predefined small patterns against a bigger graph. However, in our case, i.e., using the Merging Diagram, the subgraphs to match are unknown, and enumerating them is computationally unfeasible due to the sizes of the code across basic blocks that we try to accelerate. A better model for our problem is maximum common subgraph identification between two graphs, where we allow disconnected maximum common subgraphs. Although this is an NP-complete problem, the distance calculation that we detail in the next section can solve the matching identification in $\mathcal{O}(n \log n)$ for each pair of CIs, with n being the number of AugTED and LinBDD nodes of the biggest CI MD.

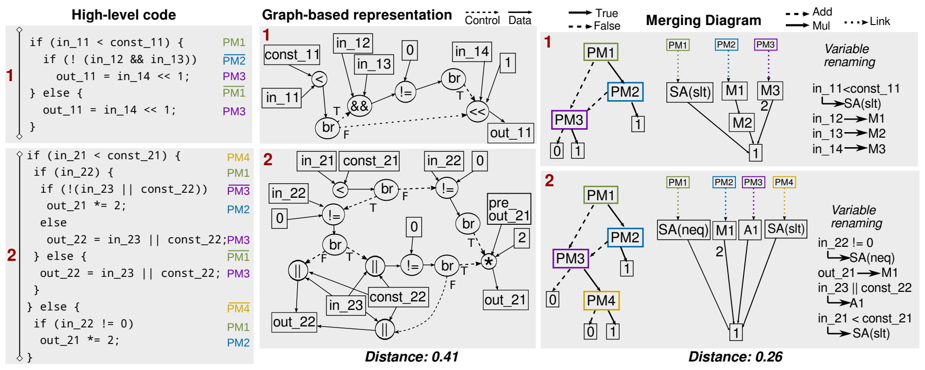
We compare our MD's similarity detection results against those of a state-of-the-art method [Huang et al. 2014] that finds partial similarities using the Edit Distance concept: the similarity is measured based on the edit operations to convert one graph into the other one. The time complexity of this approach is $\mathcal{O}(|G_1| \times |G_2|)$, with $|G_1|$ and $|G_2|$ the number of nodes of each graph. Although the original work does not cover code across basic blocks, we have adapted it to the loop-body level, adjusting this Edit Distance to be comparable to our own similarity score. The score of both methods indicates how similar two CIs are, with 0 being exactly similar, and 1 not similar at all.

We compare Edit Distance and Merging Diagram matching (EDM and MDM, respectively) in Figure 4, where we show the cumulative distribution of the scores obtained with both methods. The x-axis represents the distance, or dissimilarity score, and the y-axis represents the actual frequency percentage of the distribution. On average, EDM has a score of 0.96 and MDM has a score of 0.57, which means that our Merging Diagram is able to match many more code sequences than a regular DAG approach. Figure 4 breaks down the results into (a) application-specific CIs and (b) domain-specific CIs. While MDM finds many more similarities in both cases, EDM sees its best results when matching application-specific CIs. This is the reason why previous work that focuses on application-specific acceleration found DAG-matching techniques to be sufficient; the code within an application has a similar coding style, and thus can use traditional graph matching techniques. When analyzing code across a domain, however, it is much more beneficial to first convert the code to a canonical representation that encodes the functionality before matching. Figure 4 also compares the matching of CIs (c) inside a basic block and (d) across basic blocks. In both cases, MDM is able to find more similar CIs than EDM, especially with the matching across basic blocks, which is a benefit of the new canonical representation.

Being able to find a high degree of similarities using our new Merging Diagram means that more code sequences can be mapped into the same CI, shrinking the implementation area while accelerating more code. We then have more space to implement other CIs of the domain, and thus can expect overall higher speedups and energy savings than with an application-specific solution.

4.2. Distance Calculation

In the *Distance Calculation* step, we need to establish a concrete metric that measures similarities among CIs to guide the subsequent clustering step of the MInGLE framework. Thus, we develop a new way to measure how different two variants are in terms of their functionality, using the MD. 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


High-level code

```
1   if (in_11 < const_11) {                PM1
      if (! (in_12 && in_13))              PM2
        out_11 = in_14 << 1;               PM3
    } else {                               PM1
      out_11 = in_14 << 1;                 PM3
    }

2   if (in_21 < const_21) {                PM4
     if (in_22) {                          PM1
      if (!(in_23 || const_22))            PM3
       out_21 *= 2;                        PM2
      else
       out_22 = in_23 || const_22;         PM3
     } else {                              PM1
      out_22 = in_23 || const_22;          PM3
     }
    } else {                               PM4
     if (in_22 != 0)                       PM1
      out_21 *= 2;                         PM2
    }
```

Fig. 5. Distance between codes 1 and 2 in the left-most column, first with a graph-based representation in the middle column, and then with Merging Diagrams in the right-most column.

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 3.2.2, we would have to compute a local variable ordering 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 in CI_X that do not match with those in CI_Y, namely nM_X, and vice versa, nM_Y. We identify three types of matches with MDs: perfect, hidden and with overhead. An MD subdiagram S with nodes $< v_1, \ldots, v_n >$ and edges $< e_1, \ldots, e_n >$ has a perfect match with another MD subdiagram S' with nodes $< v'_1, \ldots, v'_n >$ and edges $< e'_1, \ldots, e'_n >$ if their labels and edges types match exactly. Afterwards, we identify a hidden match if the types of the outgoing edges of nodes v_z and v'_z match and are connected to subdiagrams with a perfect match. Finally, a match with overhead identifies only nodes that represent the same operations, but that do not share the same computational structure and would need a multiplexer to be shared. Mo_X and Mo_Y are then the number of nodes of CI_X and CI_Y, respectively, with the same operations but with that extra overhead. As those matches with overhead incur in area costs, we count them also for the dissimilarity metric. The matching information is kept for the merging step explained below in Section 4.4. 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:

$$\delta\,(CI_X, CI_Y) = average\,((Mo_X/2 + nM_X)/\,Tot_X, (Mo_Y/2 + nM_Y)/\,Tot_Y) \qquad (1)$$

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

Figure 5 shows a simple example of distance calculation using the graph-based Edit Distance from the previous section, and our method. On the left are two pieces of code, and the middle column shows their directed acyclic graph representations obtained from the LLVM intermediate representation. Using the Edit Distance technique on these graphs, we obtain a distance of 0.41 using a scale of 0 to 1, with 0 representing the highest similarity degree. The last column shows the code's MD representations, with LinBDDs on the left and AugTEDs on the right. We show the variable renaming to correlate MD nodes with the original variables, and denote the predicate nodes (PMX) next to the high-level code on the left. In comparison with DAGs, MDs obtain a higher matching degree (0.26). The Merging Diagrams are clearly very similar due to their canonicalized form. For instance, the polynomial that expresses the control flow of code 1 is: $P_1 = PM1 \cdot \overline{PM2} \cdot PM3 + \overline{PM1} \cdot PM3$, and the control flow of code 2: $P_2 = \overline{PM4} \cdot PM1 \cdot PM2 + PM4\,(\overline{PM1} \cdot PM3 + (PM1 \cdot \overline{PM3} \cdot PM2 + PM3))$. With a different variable order, P_2 is written as: $P'_2 = PM1\,(PM2 + \overline{PM2} \cdot PM3 \cdot PM4) + \overline{PM1} \cdot PM3 \cdot$

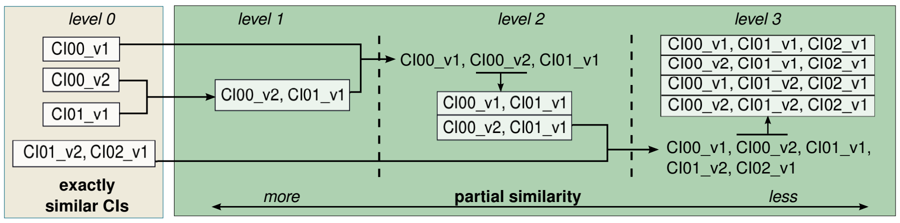

level 0
CI00_v1
CI00_v2
CI01_v1
CI01_v2, CI02_v1
exactly similar CIs
level 1
CI00_v2, CI01_v1
level 2
CI00_v1, CI00_v2, CI01_v1
CI00_v1, CI01_v1
CI00_v2, CI01_v1
level 3
CI00_v1, CI01_v1, CI02_v1
CI00_v2, CI01_v1, CI02_v1
CI00_v1, CI01_v2, CI02_v1
CI00_v2, CI01_v2, CI02_v1
CI00_v1, CI00_v2, CI01_v1, CI01_v2, CI02_v1
more
partial similarity
less

Fig. 6. Hierarchical clustering of CIs. Exact matching instructions are at the bottom, while nodes closer to the root are increasingly less similar CIs. CIXX_vy: CI with identifier XX and implementation variant y.

$PM4$, which corresponds to the LinBDD of the second MD. Canonicalization enables renaming variables to obtain a simplified polynomial which matches better with the first code's MD. Therefore, a canonical representation of the code's functionality leads to matching more code than with a structural representation, and therefore accelerates more code while saving more energy.

4.3. Custom Instruction Variants Clustering

For domain-specific acceleration, merging CIs reduces energy consumption because we need less implementation area, or alternatively improves performance since we can allocate 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, in the *CI variants Clustering* step, 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. Using the distance matrix computed in the previous step, we create clusters of CI variants. We perform hierarchical clustering of CI variants, obtaining a dendrogram, a tree-like structure, as shown in Figure 6, 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 maximum 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 6, 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 merged versions that do not duplicate the code covered ($CI00$ 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 case of many cluster versions, we select a reduced group chosen heuristically based on the expected EDP improvement, modeled using the equations in the next section.

4.4. Merging Estimation and Modeling

With the clustering formation, we obtain a bigger set of CI variants, some of which are merged to save area. In the *Merging Estimation* step, we estimate the new hardware area occupancy, performance and energy gains of merged variants to run the selection step with accurate information.

Based on the information from the distance calculation (Section 4.2) of non-common matches between each pair of variants, we obtain the area consumption of operators that are shared ($shared$) and of those that are not (non_shared). For sharing logic, we need to introduce multiplexers that will induce an extra area cost, $overhead$. Thus, we calculate the area a_i of a merged CI variant i as:

$$a_i = overhead_i + shared_i + \sum_{j=1}^{N} non\_shared_{ij}. \tag{2}$$

Then, in the *Performance and Energy Model* step, we first model the performance of an accelerated application. We start by obtaining the cycles c_l_SW that a hot loop iteration takes to execute in the baseline processor, excluding memory operations; this is obtained through 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, adjusted to the core's clock frequency domain. The cycles c_T to transfer data to the DSFU local memory are a function of the input data size. Then, we obtain the cycles we save executing a variant as:

$$c\_saved = (c\_l\_SW - (c\_HW + c\_T)) \times N\_it. \tag{3}$$

We calculate the new number of application cycles as:

$$App\_cycles = c\_total\_SW - c\_saved, \tag{4}$$

with c_total_SW as the application cycles without CIs.

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

$$E_{app} = E_{baseline} + E_{CI}, \tag{5}$$

with $E_{baseline}$ the 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:

$$E_{CI} = P_{dynamic} \times T_{CI} + P_{static} \times T_{total}, \tag{6}$$

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.

5. OPTIMIZATION: CUSTOM INSTRUCTIONS FRAGMENTS GENERATION

We call *CI fragments* a variation of partially matched CIs that will not include the full original CI, but parts (fragments) of it. This kind of matching is aimed to improve reutilization of hardware at the most limited areas. With CI fragments we can partially reuse an already merged CI cluster for CIs that were initially not included in that cluster, with minimum additional overhead. We obtain CI fragments in the *CI Fragments Generation Module* of the MInGLE framework.

Consider the clustering dendrogram of Figure 7 that organizes a hierarchy of CI similarities. Baseline CIs are located at level 0, while merged CIs start from level 1 and go forward from more to less similarity degree. At each new level, two CIs from lower levels are merged. In the merging process, the distance (*dist*) between CIs is evaluated to determine which pair of CIs to merge in the next level. Each one of the

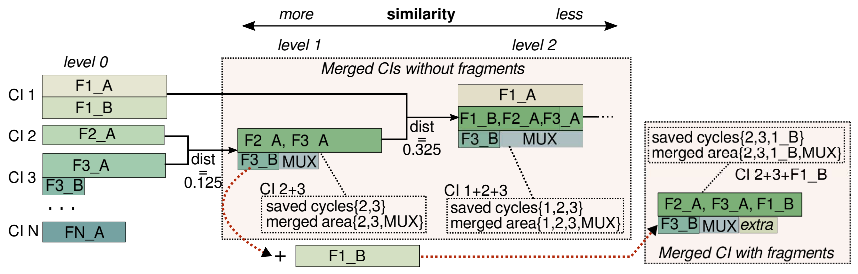


Fig. 7. Example of partial merging with CI fragments.

merged CIs has an overhead from multiplexer switches represented as *MUX*, and a number of saved cycles and new area placed in adjacent boxes.

Each one of the N baseline CIs is composed of one or several fragments, that we define as computation blocks that can be separated from the CI without structural problems. Although those computation blocks can overlap, in this example, the internal CI operations covered by each fragment are fixed for illustrative purposes. Also, to simplify the example, we consider that each CI has only one variant. For instance, the only variant of CI_1, at level 0, is composed of fragments $F1\_A$ and $F1\_B$. The partial merging explained in previous sections merges whole CIs, based on increasing distance. With that method, we first obtain a new merged CI_2+3 at level 1 and then CI_1+2+3 at level 2. With each new merged CI, we obtain a speedup based on the combined saved cycles, and a new area that includes non-common operations, merged common computations, and the switching logic overhead (*MUX*).

However, consider CI_1+2+3 and its fragments at level 2, product of merging CI_1 with CI_2+3. Fragment $F1\_B$ from CI_1 is completely merged with CI_2+3, avoiding a significant area increase. In contrast, fragment $F1\_A$, also from CI_1, is fully incorporated at a substantial area increase. Consequently, we can argue that if we merge only one of the fragments, we could obtain savings in cycles at a low area cost. This is what CI_2+3+F1_B on the right of the figure illustrates (*Merged CI with fragments*). If we merge only fragment $F1\_B$ from CI_1 with CI_2+3, the area increase from additional switching logic (*extra*) will be negligible, while performance will improve due to being able to accelerate more code ($F1\_B$).

5.1. Generation of Custom Instruction Fragments

There are some conditions to specify how suitable CI fragments are found:

— The size of a CI fragment is at most the same as the CI that matches, which is generally much bigger. Therefore, for a given merged CI, we can have several fragments from different applications matching.
— Operations included in a fragment do not depend on excluded ones, to avoid a *convexity violation* [Karuri and Leupers 2011], or circular dependency between operations that could result in wrong scheduling.
— CI fragments should not add logic to perform computations, but they can add some additional overhead for switching circuits. They may also have extra cycles to transfer data and the total number of saved cycles are probably less than if the full CI was included. All this additional overhead and reduced gains are carefully weighed to determine if a CI fragment is worth including.
— We can create CI fragments using CIs from any level of the dendrogram, either with exact or with partial similarities.

Algorithm 2: Fragment Matching

```
input  : Merged Diagrams $MDs$, merged clustering solutions $MS$, $threshold$
output: Solutions $MSF$
1  Array $MSF \longleftarrow MS$
2  for $Sol \in MS$ do
3      $Candidates \longleftarrow \emptyset$
4      for $md \in MDs$ do
5          if $CI(md) \notin Sol$ then
6              $FM \longleftarrow \emptyset$
7              for $WholeFragment \in md$ do
8                  $FM \leftarrow FM \cup GetMatchesOneWay(WholeFragment, Sol)$
9              end
10             if $matches(FM) > threshold$ then
11                 $EDPImprov \leftarrow GetEDPImprovement(FM)$
12                 $Candidates \leftarrow Candidates \cup < FM, EDPImprov >$
13             end
14         end
15     end
16     $BestCandidates \leftarrow FilterCIVariant(Candidates)$
17     $NewSols \leftarrow CreateSolutions(BestCandidates)$
18     $MSF \leftarrow MSF \cup NewSols$
19 end
20 return MSF
```

— Starting the fragment search from a CI configurationthat implements a set of CIs C, we will only consider adding fragments from variants not included in C.

Under those conditions, note that fragments of a given CI differ depending on the matching target, therefore their area coverage and saved cycles vary across solutions.

Algorithm 2 lists the pseudo-code that detects fruitful fragments to augment the initial set of solutions generated after the hierarchical clustering. We evaluate against each solution the possible matches of any CI variant, represented as an MD, that is not yet part of the solution. We start with the clustering solutions MS, that are the base to the new solution set plus fragments (*MSF*). For each MD evaluated, we obtain the fragment matches (FM) evaluating separately the sequence of solutions that lead to each output (lines $6-9$). Then, we can easily limit the fragmented matches to the boundaries of a certain output to control the convexity of the selected operations. With the function *GetMatchesOneWay* we perform a matching similar to that of Section 4.2. In this case, we are only interested in knowing the coverage of *WholeFragment* within *sol*. In lines $10-13$, we evaluate if the percentage of matches of the fragments found reach a user-defined *threshold*. If they do, an estimation of the expected EDP improvement is calculated, and the fragments of that CI variant are considered to be included if that expected EDP is better than the baseline (no CIs). As several variants of the same CI could be in the set of candidates, we filter them based on the best estimated EDP improvement in line 16. Finally, in the next line, we create a new solution structure with updated information over the area and the CI fragments that it includes, applying again the *Performance and Energy Model* step.

6. SELECTION OF DOMAIN-SPECIFIC CUSTOM INSTRUCTIONS

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 running application would consume less energy (assuming power consumption is constant). Therefore, in the last *CI 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 that 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, with fragments included and not included, 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, merged with other instructions, or partially added as a fragment – once we select one variant, all other variants of the same CI are removed from the following selection steps.

The constraints of our Mixed Integer Linear Programming (MILP) problem are:

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

with lb_i a loop body that can be implemented by (n) CI variants. Therefore, for a given loop body, only one of its CI variants will be selected. As our main goal is to accelerate execution and save energy, our objective function tries to maximize the EDP improvement. However, the total EDP value changes depending on the area occupancy, and thus, it cannot be deterministically precomputed before the selection starts. Though, for each potential CI we can calculate an approximated value of the EDP difference with respect to the baseline processor without the CI. Also, the static energy component of the EDP is subject to the known value of the maximum area A, which is an approximation for the value that we want to maximize. Thus, we define the objective function as:

$$\sum_{i=1}^{n} c_i \times \sigma\_EDP_i \rightarrow max. \tag{8}$$

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 follows:

$$\sigma\_EDP_i = \sum_{j}^{B} \|\sigma\_EDP_{ij}\|, \tag{9}$$

where B is the number of applications that the current variant targets, and $\|\sigma\_EDP_{ij}\|$ is the original application j's EDP minus the EDP with the variant, normalized to the observed maximum for that application in order to introduce fairness across the domain. We find that this metric selects more merged 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, both for partial matching and with fragments.

7. COMPLEXITY

While the overall complexity of the framework varies in each step, processing all the design possibilities would be impossible due to the exponentially growing search space. In order to cope with this problem, our methodology simplifies the representation of the search space, using the canonical Merging Diagram. This representation helps solving the NP-complete problem of CI partial matching in quasilinear time, as explained in Section 4.1. Selection is also a critical step that could be exponential in the worst case. Therefore, our methodology also reduces the search space to keep the exploration tractable and fast, by establishing bounds based on the number of total CI variants. We try to always keep a reduced number of CI variant candidates, while maintaining energy and performance efficiency, as explained below.

For each input application from the set of B benchmarks, we have a number of CIs C. 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 $CV = \sum_{i=1}^{B}\sum_{j=1}^{C_i} numVariants_j$. Establishing the global variable order is subject to the maximum number of input variables among all variants, that is $\mathcal{O}(maxVars \times CV)$. The complexity to build all the diagrams is determined by the number of individual variables and the established ordering. Following the ordering rules of Section 3.2.1, we can narrow the total number of expansions needed to build the diagram to $\theta(\sum_{k=1}^{CV} n_k^2)$. The complexity of calculating distances between pairs of MDs (Section 4.2) that contain at most n nodes is $\mathcal{O}(n \log n)$. This calculation is performed $(CV \times (CV - C - 1))$ times. The key design decision here is to have a global MD, which obviates the need for a new variable ordering to be computed to compare each pair of variants, speeding up the calculation. Finally, by performing the hierarchical clustering step explained in Section 4.3, and using a heuristic to limit the number of cluster versions per level, the final number of generated solutions that the selection of Section 6 processes is within the bounds of $\mathcal{O}(CV)$. Although the number of original benchmarks and CI variants influences the size of the final selection, the constant limit in the heuristic will always keep a non-exponential growth in generated solutions. 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.

8. EVALUATION

8.1. Experimental Setup

We now describe the setup and experimental evaluation of MInGLE.

The target architecture is an in-order Intel Atom with a tightly-coupled DSFU, as described in Section 2.1. The DSFU has private registers: 16 128-bit for input data and 32 64-bit for output. We determine the size of the register files with the maximum size needed among the benchmarks described later, which in this case is 2048 bits for both input and output data. Before starting any CI computation, data is moved into the input registers from the core's register files, and once the computation is completed the results are written back. Note that, for any CI, the extra cycles for reading and writing data are considered as part of the total latency for calculating speedup values.

In the *CI Identification* step of the framework, we first identify hot regions of code with the LLVM profiler [Lattner and Adve 2004] and extract the CI functionality in C code. We synthesize high-level CI descriptions with Vivado HLS 2013.3 [Xilinx 2014] to obtain the circuit design cycles and area consumption. We apply different unrolling factors to the CIs: none, 2, 4, and 8. We also apply vectorization whenever possible. The target FPGA is a Xilinx Virtex 7 (XC7VX690T) that runs at 400 MHz. DSFU power estimations are obtained with the Xilinx Power Estimator (XPE). We compile the target applications with LLVM-Clang with an unrolling factor of 8, automatic vectorization, and optimization $-O2$ as the baseline. Software cycles are measured with the Sniper simulator [Carlson et al. 2014], configured to simulate an Intel Atom processor running at 1.6 GHz. Thus, the DSFU runs $4\times$ slower than the baseline processor. Consequently, we adjust execution cycles on the DSFU to the core's clock domain. Power measurements on Sniper are obtained with McPAT [Li et al. 2009a]. We run two different versions of the code on Sniper: the original application for baseline comparison, and the application with the code accelerated by the CIs marked in assembly code for functional simulation. Unrolled, non-vectorized code sequences in the LLVM IR are analyzed to generate the polynomials for the Merging Diagrams, which are built with the support of the symbolic algebra and calculus part of Sage [Stein et al. 2013]. In step *Optimization: Merged CIs*, we use the Fastcluster library [Müllner 2013] for hierarchical clustering, and feed cycles and power data into the models of Section 4.4 to obtain results. The interface for the CPLEX optimizer [IBM 2014] in the *Selection*

Table I. For each benchmark: suite where it can be found, number of CIs and CI variants considered, the percentage of dynamic instructions covered by them, and the number of candidates found with partial matching and matching with fragments for regions across basic blocks.

Benchmark	Suite	Num. CIs	Num. variants	% dyn. ins.	Partial	Fragments
cjpeg	MediaBench II	4	16	81.6%	461	2890
djpeg	MediaBench II	3	12	45.3%	434	1756
gsmdec	MiBench	1	4	70.8%	399	2281
gsmenc	MiBench	2	7	56.5%	406	1788
mpeg2enc	MediaBench II	3	6	45.4%	364	2084
optflow	OpenCV	2	7	49.5%	440	2130
rawcaudio	MiBench	1	4	87.0%	402	2078
rawdaudio	MiBench	1	4	85.2%	410	2841
susan	MiBench	1	4	95.4%	427	2825
tmndec	MediaBench II	3	4	87.2%	401	2282
tmnenc	MediaBench II	2	6	50.6%	385	2632

step is OpenOpt [Kroshko 2015]. Although we use an FPGA as a testing platform, we do not consider run-time reconfiguration in this work.

8.2. Benchmarks

We evaluate the framework with eleven applications from the media domain, listed in Table I. The applications are extracted from the benchmark suites OpenCV [Bradski 2000], Mediabench II [Fritts et al. 2009] and MiBench [Guthaus et al. 2001], which are listed in the second column. For each one of the benchmarks in the first column, we show in the third column the number of critical CIs found across basic blocks. The fourth column lists the number of CI variants or distinct implementations, for several unrolling factors; only those implementations that yield some performance improvement are considered. The fifth column shows the percentage of dynamic instructions covered if all the CIs were selected, with all of them over 45%. Such a large code coverage is key for performance improvement, and better achieved with CIs that cover regions across basic blocks. Benchmark cjpeg has the highest number of CIs and variants; however, the highest coverage of dynamic instructions corresponds to susan. The two rightmost columns list the number of merged CIs generated with partial matching and matching with fragments, respectively. Note that both numbers include exact matching, and partial matching is a subset of matching with fragments. The threshold of matching with fragments is set at 50%, as discussed in detail in Section 8.3.2.

Although we evaluate our framework with only one domain, our technique can be applied to other domains as well. Applications that are CPU-bound could achieve significant speedup and energy efficiency improvements using our proposed acceleration framework. Some applications that could benefit include scientific applications with intensive use of algebra, machine learning applications that involve convolutions and matrix operations, or applications in the embedded domain that are compute-intensive, such as those in the fields of security or car electronics.

8.3. Results

We first compare different techniques implemented in the framework to identify CIs across and inside basic blocks to be accelerated by a DSFU in hardware, measuring both speedup and improvement in EDP across various area settings. We subsequently evaluate the effect of different threshold values on fragment matching. Finally, we present results of area characterization when we use different matching techniques.

8.3.1. Speedup and EDP Improvement. Figure 8 presents a comparison of different configurations that the framework generates for the benchmarks in Table I, with DSFU area on the x-axis expressed as a percentage of the Virtex 7's area, and *a)* the aver-

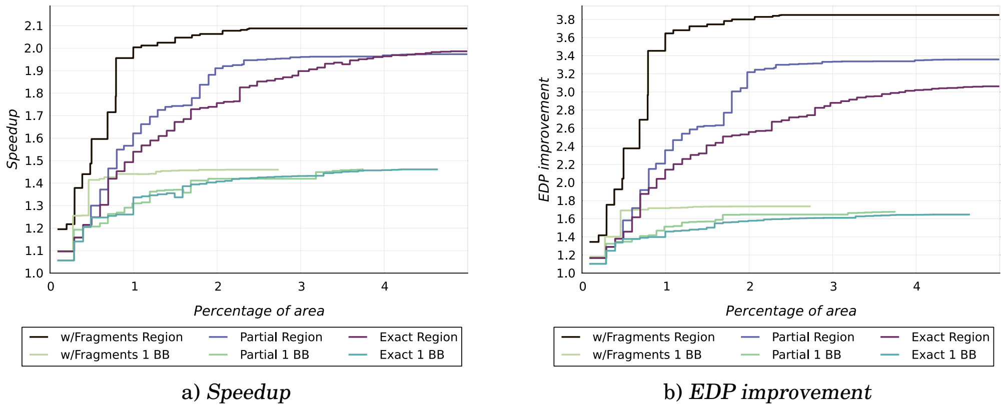


Fig. 8. Average speedup and EDP improvement (y-axis), against increasing area percentages (x-axis), for exact and partial matching, and matching with fragments, across and within the basic block level.

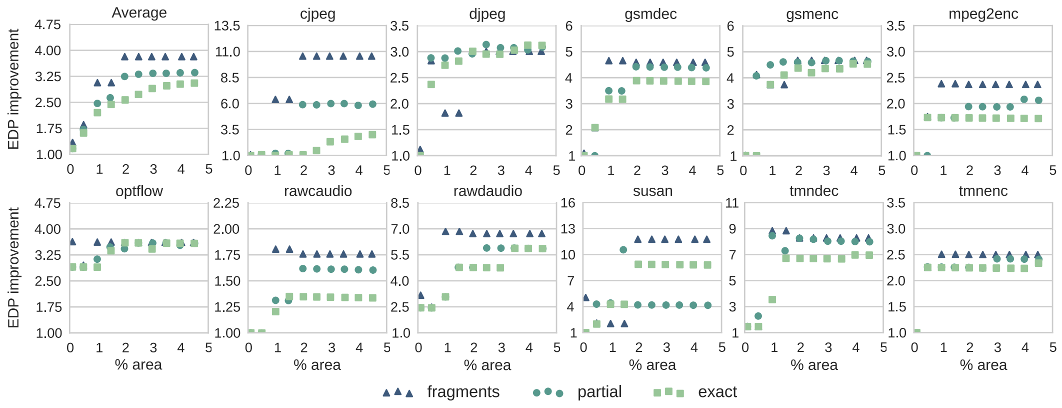


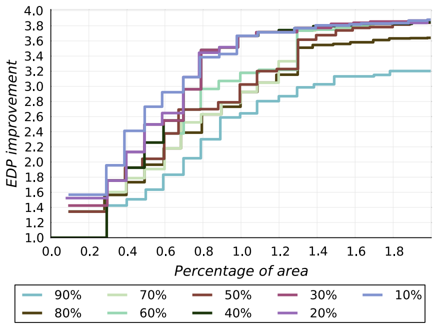
Fig. 9. EDP improvement for each benchmark, up to the 5% of the area, with CIs selected across basic blocks with fragments, partial matching and exact matching.

age performance speedup and *b)* the average EDP improvement across the domain on the y-axis. Speedup and EDP improvement are calculated with respect to the baseline processor, and are the average across all benchmarks. Lines marked with *1 BB* show improvements achieved when we use CIs targeting code within basic blocks. At the largest areas, performance improvement reaches a maximum of $1.48\times$ and EDP improvement goes up to $1.74\times$ the baseline. We compare this to the lines marked with *Region* in the figures, which target code regions across basic blocks. In this case, speedup reaches a maximum of $2.09\times$ and EDP improvement goes up to $3.84\times$. 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. CIs across basic blocks cover more code, expand the acceleration opportunities, and thus achieve higher energy efficiency and speedup.

In the same figures, we analyze the efficacy of exact matching, partial matching and matching with fragments by comparing those lines marked as *Region*. Note that partial matching choices include all those CIs matched with exact, and then additional CIs that could be partially matched. The same case applies for matching with fragments, with partial matching choices included among newly generated ones. In the case of partial matching, 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, for a given EDP improvement, partial matching saves area. For instance, 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. Matching with fragments, though, outperforms previous techniques from the beginning, at very limited areas. With only 1% of the Virtex 7, we have a speedup of $2\times$ and EDP improvement of $3.65\times$, clearly higher than the same values for partial matching, $1.63\times$ and $2.35\times$, respectively. Matching with fragments for CIs across basic blocks helps to reach the best speedup and energy efficiency at larger areas. However, the most important feature of matching with fragments is to enable maximum performance at smaller areas either within or across basic blocks. Hence, matching with fragments uses area more effectively; a speedup of $1.96\times$ is achieved with fragments at 0.75% of the area, in contrast with the 2.5% needed with partial matching. 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. As a rule of thumb, an Atom implementation took about 85% of the Virtex 5 LX330 that has roughly 25% of the capacity of the Virtex 7.

Figure 9 presents a graph for each benchmark with a range of area percentages dedicated to the CIs on the x-axis, and EDP improvement on the y-axis. Here, we only include CIs across basic blocks. Each point on a graph represents a group of CIs selected for *all applications* that uses that particular area. Furthermore, the group of CIs selected given a particular area is the same for all benchmarks. The graphs then present the EDP improvement each benchmark achieves given that CI grouping. Note that when a particular benchmark is not sped up by adding hardware area (because the additional hardware targets other applications), its EDP remains the same (horizontal dots). Results of the matching with fragments use a threshold of 50%, which we discuss in the next section. Only some area values are displayed, with a stride of 0.5%. Note that each benchmark has a different y-axis scale for legibility. The average of all applications is shown in the top left graph.

As we pointed out before, matching with fragments is, on average, the most effective technique at finding domain-specific CIs. This technique achieves higher EDP improvement at smaller areas, always increasing the speedup faster than the other two techniques. All but three benchmarks show the best efficiency with fragments regardless of the area. We can observe, though, that for djpeg, gsmenc and susan, between 0.5% and 1.5% of the area, solutions with fragments yield lower efficiency than with partial matching. In the case of djpeg, even at higher areas, the EDP improvement of the three methods overlaps. This is due to a great dependency of the benchmark on application-specific CIs, with very low sharing rates in all the CIs generated. Regarding gsmenc and susan, although the selected fragments at low area improve the EDP, they cannot reach the gains of CIs that cover the full body loop, and not only parts of it. However, for the other eight benchmarks, matching with fragments is clearly the best choice, since we are able to cover more CIs with less area. For instance, CIs that could give more than $10\times$ EDP improvement to cjpeg are not selected with partial matching because of unavailable area resources. With fragments, there is virtually more area available from the low overhead costs of including a new fragment, hence better performing CI variants can be selected.



a) *Percentage of area (x-axis) versus average EDP improvement (y-axis) for the matching with fragments for different thresholds.*

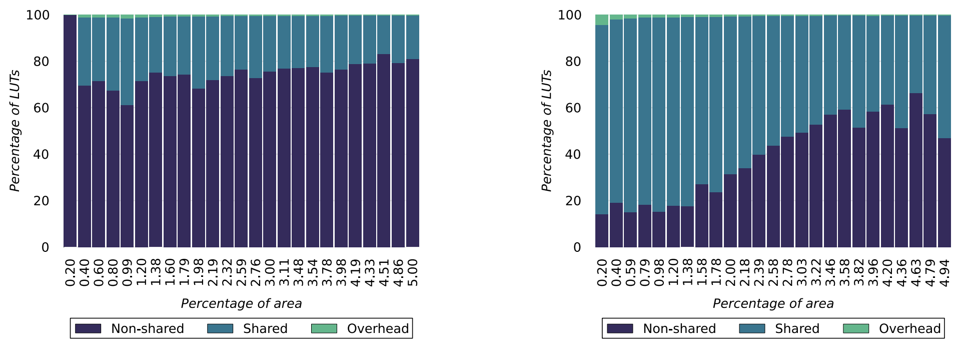
T	**Candidates**		**Solve time**		**EDP improv.**
	num.	*% inc.*	*secs.*	*% inc.*	
90	363	–	24.4	–	2.64×
80	390	+7.4	26.5	+8.6	2.94×
70	456	+16.9	27.4	+3.4	2.94×
60	581	+27.4	33.7	+22.9	3.19×
50	633	+8.9	35.9	+6.5	3.64×
40	923	+45.8	52.4	+45.9	3.67×
30	1056	+14.4	59.3	+13.2	3.67×
20	2117	+100.4	125.5	+111.6	3.67×
10	2263	+6.9	135.3	+7.8	3.67×

b) *Number of candidates in the selection step and time to solve the selection problem for different thresholds (T) using matching with fragments, for 1% of the area.*

Fig. 10. Effects of the threshold parameter on the selection of matching with fragments.

8.3.2. Threshold Analysis. We recall the user-defined threshold for fragment matching from Section 5 as the value that establishes the minimum percentage of matching operations of a fragment with respect to the evaluated CI, in order to generate a new CI that includes both the evaluated CI and the fragment. Figure 10 presents, on the left, a comparison of solutions with different threshold values, with area percentage on the x-axis up to 2% and the average EDP improvement across the domain on the y-axis. The legend shows the thresholds that go from 90% to 10% of matching. A higher threshold corresponds to a higher similarity. The CI candidates with a given threshold include all those CIs from higher thresholds. For instance, a threshold of 70% also includes the CIs of thresholds 80% and 90%. We observe that up to 0.8% of the area, a 10% threshold obtains the highest EDP improvement. However, from that area onwards, thresholds up to 50% yield the same EDP improvement and, from 1.3% of the area, the 60% and 70% thresholds join the efficiency ceiling. The EDP improvement with a threshold of 90% at 2% of the area equals the one achieved with partial matching (no fragments). At larger areas, we can choose bigger variants that provide the full CI acceleration instead of fragments that do not give the maximum efficiency. Also, fragments with 90% similarity matching are more difficult to find than those with lower thresholds.

The threshold level has a direct effect on the number of CI candidates in the selection pool and the runtime of the selection process, which is shown in the table on the right of Figure 10. Data in the table refer to the selection step for 1% of the area. For each threshold percentage (T), we list first the number of candidates considered for selection with the percentage increase with respect to the previous row. We list also the time in seconds to solve the selection problem with the pool of candidates and, again, percentage increases. In the last column we list the EDP improvement achieved. Note that, for different areas, the number of CI candidates varies because some CIs are pre-filtered by area occupancy. Also note that, as the number of candidates of a given threshold includes those of higher ones, the amount of candidates increases as the threshold value decreases. The time to solve increments linearly with the number of candidates; the largest difference in both the amount of CIs considered and seconds to solve happens relaxing the threshold from 30% to 20%, showing that smaller fragments are more frequent than larger ones. However, the EDP at those low thresholds is not better than thresholds of $40 - 50\%$ because larger fragments achieve better EDP, and the threshold is related to the size of the fragment. Thus, the increase in the problem complexity of the lowest thresholds weighed against the problem size and time to solve of the 50% threshold has no advantage because similar CIs are chosen.



a) *Partial matching* b) *Matching with fragments*

Fig. 11. Characterization of shared FPGA hardware for different area utilizations.

8.3.3. Sharing Characterization. In this last analysis, we evaluate how area is shared among CIs to understand the high gains that the matching with fragments provides. Figure 11 shows two graphs that display the increasing percentage of the Virtex 7 occupancy (up to 5%) of different CI configurations on the x-axis versus the normalized percentage of LUTs on the y-axis, broken down by non-shared operators, shared operators and multiplexer overhead for circuit sharing. The CI configurations target all the applications in Table I. The graph on the left *a)* shows the characterization for partial matching, whereas the graph on the right *b)* shows that of matching with fragments.

First, we observe that, for partial matching, the CI configuration with the smallest area (0.2%) does not share any of the CIs included. At that small area, the cost for merging CI variants is too high to compete against lighter application-specific CIs. In contrast, matching with fragments devotes 80% of the LUTs on the smallest configuration to shared resources. We find the maximum percentage of shared circuits at the smallest areas, which explains why the configurations with CI fragments are more efficient than those without. The percentage of overhead due to multiplexers is more noticeable at lower areas also, correlated with the amount of shared resources. Although at larger area utilizations the sharing levels decrease, they are steadily higher than those for partial matching, with sharing percentages around 30% on average.

9. RELATED WORK

There are many datapath specialization techniques that target different objectives and systems. A recent survey paper [Jówiak et al. 2010] presents a comprehensive overview of specialization methods for embedded devices. In this paper, we focus on a top-down approach for creating specialized hardware; we analyze a target application to automatically create CIs, also known in the literature as instruction set extensions. As early works established [Atasu et al. 2003; Yu and Mitra 2004], the design process analyzes the code in directed acyclic graph (DAG) form, and is separated into the *identification* and *selection* phases. First, the identification phase explores the application's bottlenecks and extracts subgraphs that preserve architectural constraints. Then, the selection phase chooses the best set of CIs that fulfill some given objectives. Additionally, *optimization* strategies try to make an efficient use of the available implementation area. We review prior research's approaches to these phases below.

Identification techniques. The scope of the code covered by an identified CI candidate varies from very fine-grained – inside a basic block – to across basic blocks. There are two types of very fine-grained CIs: MISO (Multiple Input Single Output) and MIMO (Multiple Input Multiple Output) CIs [Middha et al. 2002]. Several previous works [Atasu et al. 2003; Yu and Mitra 2004; Pozzi et al. 2006] extract these kinds of CIs, which are DAG subgraphs always found within a basic block. Also within a basic block, some authors attempt to extend the code coverage with the identification of maximal convex subgraphs [Atasu et al. 2008; Li et al. 2009b]. These subgraphs cover the maximum amount of code that we can get in a basic block without violating any constraint, such as the maximum number of inputs and outputs. The main problem with the identification inside a basic block is the limited performance improvement we can obtain. Therefore, other works focus on CIs that cover several basic blocks. DySER [Govindaraju et al. 2012] extracts CIs with basic control flow from hot regions following a *slicing* strategy similar to the one we present in Section 3.1. However, as they aim to specialize for a single application, their identified CIs are directly implemented to run on an accelerating functional unit network, tightly coupled with the processor of choice, such as OpenSPARC [Benson et al. 2012]. In contrast with these application-specific works, after the DAG-based identification, we transform our CIs to a functional representation that unifies coding styles across applications. Other authors [Arora et al. 2010] also propose a normalized representation of code functionality by applying a predefined set of rules, but they are limited to single-output code patterns (MISO). Another closely related work identified that a canonical representation, that captures the code's functionality, is better than a DAG for finding CIs across a domain, especially at limited areas [González-Álvarez et al. 2013]. This work is limited to accelerating maximal convex subraphs within a basic block, and matching CIs exactly. We, on the other hand, use a new canonical representation to facilitate generating CIs across basic blocks unified for a whole domain.

Selection techniques. Reducing the algorithmic complexity of the design method is a priority to make the CI selection process tractable. Some works rely on heuristics [Cong et al. 2004] to predict a CI's gain as a function of the instruction's frequency of execution and latency, while relying on dynamic programming to optimize area usage. Other authors [Pozzi et al. 2006] couple the identification and selection phases, which results in relaxed constraints such as an unlimited number of inputs and outputs. This opens up the possibility of approximate techniques and genetic algorithms that are computationally less expensive. Others [Verma et al. 2007] assume that the core processor must be a RISC, which implies a limited number of inputs and outputs. Consequently, the search space is pruned to minimize the number of registers that the CIs use. In this paper, we initially consider an unlimited number of inputs and outputs, although they are later pruned by architectural constraints due to the connections to memory. Other approaches to solve the selection problem include applying integer linear programming [Murray et al. 2009; Atasu et al. 2012] or constraint programming [Martin et al. 2012]. However, all these previous methods select application-specific CIs. In contrast, although we also use linear programming methods, our selection focuses on accelerating an application domain. There are also works that aim to select domain-specific CIs [Clark et al. 2005], but their heuristics do not reflect the potential reusability across the domain, focusing only on individual application performance gain. We define an objective function that optimizes for performance speedup and energy, that is fair across all the applications within a domain, and produces a set of CIs within a tractable time.

Optimization: area reduction. Implementation area is an expensive commodity when specializing hardware, especially for accelerators integrated into the processor core. This problem is notoriously more difficult when the accelerating hardware targets several applications. As CIs from multiple applications compete for limited space, circuit reusability becomes a key research topic. Beret [Gupta et al. 2011] presents a general-purpose specialized co-processor that extracts execution pipelines from the loop body based on application trace analysis. As they focus on reducing energy consumption in general-purpose computing, they try to fuse MISO/MIMO patterns within their accelerating engine. To do so, they match exact graphs of those small DAG patterns, using the implementation area for only the most repeated ones. Many works focus on accelerating several applications from a domain. One prior technique uses a DAG representation to identify small patterns of partially-matching subgraphs using heuristics to optimize performance speedup [Clark et al. 2005]. However, they work on small MISO/MIMO sequences (within a basic block), and are limited in that they identify CIs for one application and try to reuse those in only one other application. While this work notes that matching subgraphs is NP-complete [Clark et al. 2005], several other works also discuss the challenges of merging the data-flow graph representation of a CI [Huang et al. 2014; Zuluaga and Topham 2009; Stojilovic et al. 2013]. Qs-Cores [Venkatesh et al. 2011] create coarser acceleration units than previous works by partially merging similar code patterns. Their code representation is based on a kind of structural DAG, in contrast with our matching that takes advantage of a canonical representation across applications. Furthermore, they merge full CIs, while we are able to merge full CIs with small or medium-sized CI fragments, resulting in a better use of the available area.

Other optimizations. Although we do not consider runtime configuration issues, it is worth mentioning other works that focus on a dynamically reconfigurable specialization substrate. RISPP [Bauer et al. 2008; 2011] is an adaptable ASIP where instructions compete for area resources, with the goal of minimizing an application's total time. This work focuses more on how to efficiently schedule sections of code of a running application to specialized functional units while minimizing reconfiguration overhead. Another work to manage reconfigurable CIs at run time [Shafique et al. 2014] proposes a full energy model to choose the best set of CIs for the reconfigurable specialized area, and that is able to power gate an unused set of CIs.

10. CONCLUSIONS

Processor customization has gained attention over the last years due to the urgency of dealing with the increasing utilization wall in modern chips. Designing custom instructions (CIs) that are executed on specialized functional units is a relatively fast way of modestly extending a general-purpose processor with the potential to accelerate code sequences. If we can identify many code sequences that can reuse the same specialized hardware, we can reduce energy consumption while speeding up applications.

This paper presents MInGLE, an automated framework that identifies CIs at the loop body level from a domain of applications that are then executed on a domain-specific functional unit. We aim to select CIs that improve performance and energy efficiency fairly across all the target applications. The framework converts code sequences across basic blocks into CIs, considering several implementations for each of them. CIs are transformed into our new canonical representation, the Merging Diagram, which facilitates an optimization to reduce hardware area, namely partial matching of CIs based on their similarity. We have novelly added another optimization step that detects fragments of CIs that can use the existing merged clusters of CIs with minimal extra overhead. Our experimental results with eleven media benchmarks show that the new matching technique with fragments achieves a speedup of $2.1\times$ and an EDP improvement of $3.8\times$, on average, across basic blocks, while within a basic block we obtain a speedup of $1.5\times$ and EDP improvement of $1.7\times$. Compared to partially matched CIs, CIs with fragments are key for achieving larger performance ($2\times$ versus $1.6\times$) and EDP improvements ($3.6\times$ versus $2.4\times$) for a limited hardware area (1%). We also can achieve a particular energy efficiency with a greatly reduced specialized area. The presented work shows the applicability of introducing configurable accelerators with limited area inside simple in-order processors to accelerate a large number of applications from a domain, improving the system's performance and energy efficiency.

ACKNOWLEDGMENTS

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-2015-0493), the Spanish Ministry of Science and Technology (TIN2015-65316-P) and the Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272). We thank the Xilinx University Program for its hardware and software donations.

REFERENCES

N. Arora, K. Chandramohan, N. Pothineni, and A. Kumar. 2010. Instruction Selection in ASIP Synthesis Using Functional Matching. *Conference on VLSI Design* (2010), 146–151.

K. Atasu, W. Luk, O. Mencer, C. Özturan, and G. Dündar. 2012. FISH: Fast Instruction SyntHesis for Custom Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2012), 52–65.

K. Atasu, O. Mencer, W. Luk, C. Ozturan, and G. Dundar. 2008. Fast custom instruction identification by convex subgraph enumeration. In *ASAP*. IEEE Computer Society, 1–6.

K. Atasu, L. Pozzi, and P. Ienne. 2003. Automatic Application-Specific Instruction-Set Extensions Under Microarchitectural Constraints. *Int. J. of Parallel Programming* 31, 6 (Dec. 2003), 411–428.

L. Bauer, M. Shafique, and J. Henkel. 2008. Run-time instruction set selection in a transmutable embedded processor. In *Design Automation Conference (DAC)*. 56–61.

L. Bauer, M. Shafique, and J. Henkel. 2011. Concepts, architectures, and run-time systems for efficient and adaptive reconfigurable processors. In *Adaptive Hardware and Systems (AHS)*. 80–87.

J. Benson, R. Cofell, C. Frericks, C. Ho, V. Govindaraju, T. Nowatzki, and K. Sankaralingam. 2012. Design, Integration and Implementation of the DySER Hardware Accelerator into OpenSPARC. In *Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, 1–12.

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

R. E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* C-35, 8 (1986), 677–691.

T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models. *ACM TACO* (2014).

J. E. Carrillo and P. Chow. 2001. The Effect of Reconfigurable Units in Superscalar Processors. In *Field Programmable Gate Arrays (FPGA)*. ACM, 141–150.

M. Ciesielski, P. Kalla, and S. Askar. 2006. Taylor Expansion Diagrams: A Canonical Representation for Verification of Data Flow Designs. *IEEE Trans. Comput.* (2006), 1–11.

N. T. Clark, H. Zhong, and S. A. Mahlke. 2005. Automated Custom Instruction Generation for Domain-Specific Processor Acceleration. *IEEE Trans. Comput.* 54, 10 (2005).

J. Cong, Y. Fan, G. Han, and Z. Zhang. 2004. Application-specific instruction generation for configurable processor architectures. In *Field Programmable Gate Arrays (FPGA)*. ACM, 183–189.

R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and LeBlanc A. R. 1974. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits* 9 (Oct. 1974), 256–268. Issue 5.

J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf. 2009. MediaBench II video: expediting the next generation of video systems research. *Microprocessor and Microsystems* 33, 4 (June 2009), 301–318.

C. González-Álvarez, J. B. Sartor, C. Álvarez, D. Jiménez-González, and L. Eeckhout. 2013. Accelerating an Application Domain with Specialized Functional Units. *ACM TACO* 10, 4 (Dec. 2013).

N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. B. Taylor. 2011. The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future. *Micro, IEEE* 31, 2 (March 2011), 86–95.

V. Govindaraju, C. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. 2012. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *Micro, IEEE* 32, 5 (Sept 2012), 38–51.

S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August. 2011. Bundled Execution of Recurring Traces for Energy-efficient General Purpose Processing. In *Symposium on Microarchitecture (MICRO)*. 12–23.

M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Workshop on Workload Characterization (WWC)*. IEEE Computer Society, 3–14.

M. Haaß, L. Bauer, and J. Henkel. 2014. Automatic Custom Instruction Identification in Memory Streaming Algorithms. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. ACM, 6:1–6:9.

T. R. Halfhill. 2008. Intel's tiny Atom. *Microprocessor Report* 22 (2008).

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

IBM. 2014. ILOG CPLEX. www-01.ibm.com/software/integration/optimization/cplex-optimizer/. (2014).

L. Jówiak, N. Nedjah, and M. Figueroa. 2010. Modern Development Methods and Tools for Embedded Reconfigurable Systems: A Survey. *Integr. VLSI J.* 43, 1 (Jan. 2010), 1–33.

K. Karuri and R. Leupers. 2011. A Primer on ISA Customization. In *Application Analysis Tools for ASIP Design*. Springer, 93–109.

K. Keutzer, S. Malik, and A. R. Newton. 2002. From ASIC to ASIP: the next design discontinuity. In *Computer Design: VLSI in Computers and Processors*. 84–90.

D. Kroshko. 2007–2015. OpenOpt: Free scientific-engineering software for mathematical modeling and optimization. (2007–2015). http://www.openopt.org/

C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society.

S. Li, J. Ho Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. 2009a. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Symposium on Microarchitecture (MICRO)*. ACM, 469–480.

T. Li, Z. Sun, W. Jigang, and X. Lu. 2009b. Fast enumeration of maximal valid subgraphs for custom-instruction identification. In *Compilers, Architecture, and Synthesis for Embedded Systems*. ACM.

K. Martin, C. Wolinski, K. Kuchcinski, A. Floch, and F. Charot. 2012. Constraint Programming Approach to Reconfigurable Processor Extension Generation and Application Compilation. *ACM Transactions on Reconfigurable Technology and Systems* 5, 2 (June 2012), 1–38.

B. Middha, A. Kumar, V. Raj, M. Balakrishnan, P. Ienne, and A. Gangwar. 2002. A Trimaran Based Framework for Exploring the Design Space of VLIW ASIPs with Coarse Grain Functional Units. In *Symposium on System Synthesis*. 2–7.

Daniel Müllner. 2013. fastcluster: Fast Hierarchical, Agglomerative Clustering Routines for R and Python. *Journal of Statistical Software* 53, 9 (2013), 1–18. http://www.jstatsoft.org/v53/i09/

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

L. Pozzi, K. Atasu, and P. Ienne. 2006. Exact and approximate algorithms for the extension of embedded processor instruction sets. *CAD of Integrated Circuits and Systems* 25, 7 (July 2006), 1209–1229.

M. Shafique, L. Bauer, and J. Henkel. 2014. Adaptive energy management for dynamically reconfigurable processors. *Computer-Aided Design of Integrated Circuits and Systems* 33, 1 (2014), 50–63.

W. A. Stein and others. 2013. *Sage Mathematics Software (Version 5.8)*. The Sage Development Team. `http://www.sagemath.org`.

M. Stojilovic, D. Novo, L. Saranovac, P. Brisk, and P. Ienne. 2013. Selective flexibility: Creating domain-specific reconfigurable arrays. *CAD of Integrated Circuits and Systems* 32, 5 (2013), 681–694.

G. Venkatesh, J. Sampson, N. Goulding-Hotta, S. K. Venkata, M. B. Taylor, and S. Swanson. 2011. QsCores: Trading Dark Silicon for Scalable Energy Efficiency with Quasi-specific Cores. In *Symposium on Microarchitecture (MICRO)*. ACM, 163–174.

A. K. Verma, P. Brisk, and P. Ienne. 2007. Rethinking custom ISE identification: a new processor-agnostic method. In *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. ACM, 125–134.

Xilinx. 2014. Vivado HLS. www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. (2014).

P. Yu and T. Mitra. 2004. Scalable custom instructions identification for instruction-set extensible processors. *Compilers, Architecture and Synthesis for Embedded Systems* (2004).

M. Zuluaga and N. Topham. 2009. Design-space exploration of resource-sharing solutions for custom instruction set extensions. *Computer-Aided Design of Integrated Circuits and Systems* 28, 12 (2009), 1788–1801.