DS-DSE: Domain-Specific Design Space Exploration for Streaming Applications

Jinghan Zhang
Department of Electrical and Computer Engineering
Northeastern University
Boston (MA), USA
Email: zhangjinghan@ece.neu.edu

Hamed Tabkhi
Department of Electrical and Computer Engineering
University of North Carolina at Charlotte
Charlotte (NC), USA
Email: htabkhiv@uncc.edu

Gunar Schirner
Department of Electrical and Computer Engineering
Northeastern University
Boston (MA), USA
Email: schirner@ece.neu.edu

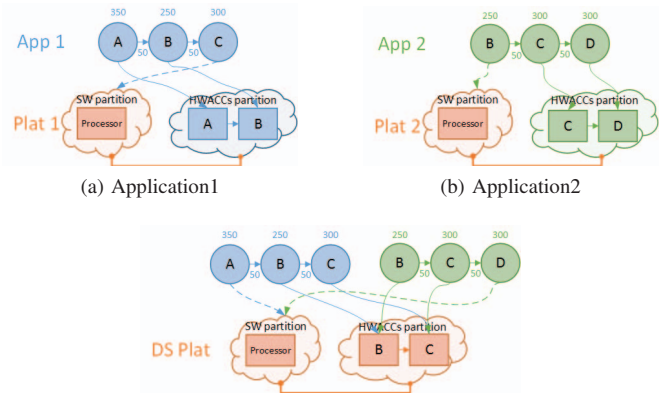
***Abstract*—Domain-specific computing is promising for high-performance low-power execution of applications with similar functionality. In particular, streaming applications with significant functional and structural similarities can tremendously benefit. However, current Design Space Exploration (DSE) focuses on individual applications in isolation. Hence, much of the domain optimization opportunities are missed. DSE methodologies need to broaden the scope from individual applications in isolation to optimizing across applications within a domain.**

This paper introduces a novel Domain-Specific DSE (DS-DSE) approach for domain-specific computing with a focus on streaming applications. Key contributions are: (1) a formalized method to extract the functional and structural similarities of domain applications, (2) a novel algorithm for hardware/software partitioning of a domain-specific platform to maximize the throughput across domain applications (under certain constraints) and (3) a methodology to evaluate a domain platform. This paper demonstrates the benefits using 4 domains: OpenVX (vision processing), and 3 synthetic domains (with greater complexity). Our experiments demonstrate a performance improvement (average throughput) of 36.8% for OpenVX and 46.2% for synthetic domains of the DS-DSE generated platform compared to an application-specific platform.

I. Introduction

Using heterogeneous platforms [1] that combine general-purpose processor(s) (SW) and custom HWACCs (HW) is the primary approach for efficient, high-performance stream computing. These platforms are application-specific, targeting only a narrow set. Designing one platform for each application is prohibitively expensive. In contrast, domain-specific platforms (e.g. FLP[2], TPU[3]), that utilize functional and structural similarities among applications, are promising to achieve flexibility to execute a set of applications efficiently. However, designing these platforms is a tremendous effort taking a vast amount of empiric knowledge into account. There is only little methodological support as current Design Space Exploration (DSE) mostly focuses on individual applications in isolation. With moving toward domain-specific platforms, novel domain-specific design methodologies and tools are needed to streamline architecting for a domain.

Fig. 1 illustrates the lost opportunities due to the limited DSE scope for a domain with two apps. Fig. 1a and 1b select ACCs with application scope (A,B and C,D respectively) yielding efficient execution on the own platform. However, executing an app on the foreign platform (i.e. $app1$ on $plat2$, $app2$ on $plat1$) results in significant penalties as either ACC A or D is not used. In result, the overall domain performance (all apps execute on same platform) is low.



(a) Application1 (b) Application2

(c) Domain-Specific Platform

Figure 1: Penalty of Application Scope

Domain DSE can dramatically improve domain performance. Its aim is to detect and exploit common used kernels (e.g. function B and C) and composition (e.g. B-C in Fig. 1) across apps. When analyzing both applications, the common domain architecture (Fig. 1c) executes both applications efficiently. However, current DSE methodologies only have an application scope and are oblivious to common computation requirements across applications in a domain. New design methodologies and tools required to shift the DSE focus from a single application to a group of applications (domain-specific DSE). To achieve domain-specific DSE, a formalization of domain features and similarities are also needed to aid design of domain-specific platforms.

This paper introduces a domain-specific design flow to aid HW/SW partitioning when designing a platform for a domain of applications. The key insight is to identify both behavioral (functional) and structural similarities across applications in a domain, and then evaluate benefits of the commonalities in a novel Domain DSE for HW/SW partitioning to optimize performance across all domain applications. To this end, the contributions of this paper are: (1) definition of quantifiable domain features to express the similarity among applications;

978-3-9819263-0-9/DATE18/©2018 EDAA

(2) introduction of a domain analyzer; (3) Dynamic Score Selection (DSS) for domain-specific architecture HW/SW partitioning; and (4) a methodology to evaluate a domain platforms. The DSS algorithm dynamically evaluates the importance and benefit of each common feature according to domain performance bottleneck and chooses the most promising one to accelerate. It improves domain performance and balances performance among applications. The DSS generated architectures for four domains (OpenVX for vision processing, and 3 synthetic domains with greater complexity), achieve significant performance improvement: 36.8%-50.6% on average compared to application-specific architecture designs.

The paper continues with Section II summarizing the related research work. Following that, Section III identifies domain features metrics. Section IV presents domain features analyzer. Section V introduces the DSS algorithm for domain HW/SW partitioning. Section VI evaluates the domain-specific architectures and compares them with application-specific architectures, and finally Section VII concludes this paper.

II. Related Work

Since the large and complex design space, heuristics methods have widespread use in DSE. The heuristics methods are comprised of an algorithm to explore design space and a fast evaluation/estimation to judge the performance/benefit of each iteration/selection. To traverse the design space, genetic algorithms [4], simulated annealing [5], tabu search [6], and greedy algorithms [7] have been used. A wide range of performance estimation approaches exist. Examples include analytical/statistical models [8] with static/dynamic scheduling. However, these DSE approaches focus in essence on a single application in isolation (although being able to iterate over applications). There is no clear path to broaden their scope to domain DSE.

Multi-variant-based [9] and multi-mode DSE [10] design one overall architecture for multiple applications in different domains. However, they ignore the significant functional and structural similarities among applications within the same domain, and cannot obtain the domain-specific architecture. Some promising architectures for domain-specific platforms have been proposed (e.g. [2]). However, the domain analysis and domain DSE have not been tackled.

Overall, despite the emergence of domain platforms, domain exploration methodologies and tools are not defined well. Missing fundamental aspects are (1) a formalization to capture similarities among domain applications, (2) exploration algorithms that optimize across applications in a domain, and (3) metrics to evaluate the efficiency of a domain platform.

III. Domain and Domain Features

The foundation of any systematic approach for domain-specific-DSE is the formalization of a domain and its features (metrics) to quantitatively reason about the domain. This section defines the domain scope and its features (metrics). They capture the behavioral and structural features of a domain showing what functions are commonly used and how they are composed. Defining these features and metrics lays the foundation for automatic domain analysis and exploration.

A. Domain Definition

Domain is a set of applications, which share common functions and common patterns[11]. To allow reasoning about the domain[1], we formally define it as a set of graphs.

$$\begin{aligned} &G = \{g_0, g_1, ..., g_N\}, \quad g_i = (A, E) \\ &A = \{a_0, a_1, ..., a_n\}, \quad E = \{e_0, e_1, ..., e_m\} \\ &a_i(t, d_P), \quad e_i((a_{src}, a_{dst}), d_C) \end{aligned} \tag{1}$$

In Eq. 1, domain G is a set of streaming applications (g_0 .. g_N), each captured as a dataflow graph [12]. Each application g_i contains a set A of processing actors (a_0 .. a_n) and a set of E edges (e_0 .. e_m) representing the communication between actors. Each actor a_i is an instance of a function type t with an instance-specific processing demand d_P (# of operations). Multiple instances of the same function type t may exist within and across applications within the domain. Each edge e_i is the directed communication between its a_{src} and a_{dst} with a communication demand of d_C as a measure of the transferred volume (bytes). E.g., in application $g0$ of Fig. 2, the first actor $A0$ is an instance of t_A and its $d_P = 350$, and the edge between $A0$ and $B0$ contains $d_C = 50$.

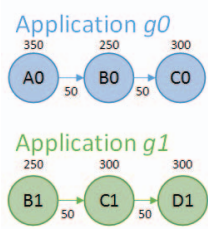


Figure 2: Example: Domain Applications

Composition	P	D_P	D_C
$\{t_A\}$	50%	350	–
$\{t_B\}$	100%	500	–
$\{t_C\}$	100%	600	–
$\{t_D\}$	50%	300	–
$\{t_A, t_B\}$	50%	–	50
$\{t_B, t_C\}$	100%	–	100
$\{t_C, t_D\}$	50%	–	50
$\{t_A, t_B, t_C\}$	50%	–	–
$\{t_B, t_C, t_D\}$	50%	–	–

Table I: Example: Domain Features

B. Domain Features

Domain features lay the foundation for automatic exploration. It is essential to capture behavioral similarity to assess processing needs, and structural similarities to assess communication / topology requirements. Aligned with the domain definition, we identify commonly used functions and function patterns. We express common functions as the function types of actors that repeat within and across applications. Function patterns are repeating compositions these function types (i.e. identical subgraphs). Different lengths of compositions are considered. For example, application $g0$ in Fig. 2, has three, two and one composition of a degree one, two and three, respectively. The function type composition $\{t_B, t_C\}$ (which is of degree 2) appears in application $g0$ (actors $B0$,$C0$) and $g1$ ($B1$,$C1$). Table I lists the compositions of the domain.

A composition of degree one ($|C| = 1$) contains a single function type. All compositions $|C| = 1$, represent the function types shared across the domain. Since compositions C can

[1]Defining the scope of a domain, i.e. assessing domain membership, is an additional research topic.

capture both common functions (behavioral similarity) and function patterns (structural similarity), this paper chooses function type compositions C to express domain features. In addition to the subgraph of function types, we focus on three aspects to characterize compositions: probability of appearance, processing demand and communication demand.

Compositions that appear more often across applications are more important for the domain than infrequent compositions. To quantify the importance, we define appearance probability P as per Eq. 2.

$$P(C_i) = (\sum\nolimits_{g_j \in G} |\{Instance(C_i) \in g_j\}|) / |G| \quad (2)$$

P is probability that an instance of composition C_i appears in a domain application. The composition $\{t_B, t_C\}$ in Fig. 2 and Table I appears in all applications (P = 100%). Whereas $\{t_A, t_B\}$ and $\{t_C, t_D\}$ each appear only in one (P =50%).

A key challenge to enable domain DSE is how to identify processing and communication demands across applications which are necessary to guide resource allocation. However, considering each individual application is impractical. To obtain a domain-level view, we aggregate the demands for each composition as defined in Eq. 3.

$$\begin{aligned} D_P(C_i = \{t_j\}) &= \sum\nolimits_{g_k \in G} \sum\nolimits_{a_l = Instance(t_j) \in g_k} a_l.d_P \\ D_C(C_i = \{t_i, t_j\}) &= \sum\nolimits_{g_k \in G} \sum\nolimits_{e_l \in g_k, e_l = \{t_i, t_j\}} e_l.d_C \end{aligned} \quad (3)$$

The processing demand D_P is calculated for each composition C_i with a degree of one as the sum over all applications and all instances of the type t_j (e.g. $D_P(t_B)$ = 500). Communication demand is computed for each pair of actor types (i.e. compositions of degree two). It is the sum of the communication demand for each edge of that type over all applications ($D_C(\{t_B, t_C\})$ = 100).

$$\begin{aligned} &CS = \{C_0, C_1, ..., C_z\} \\ &C_i(P, D_P, D_C) \end{aligned} \quad (4)$$

In summary, the domin features are captured as a set of compositions, where each composition C_i is defined by its appearance probability P, processing demand D_P (for $|C_i| = 1$), and communication demand D_C (for $|C_i| = 2$). Table II shows the general view of the selected domain features.

Table II: Domain Features

Composition		Appearance Probability	Processing Demand	Communication Demand
C_0	$\{t_A\}$	$P(C_0)$	$D_P(C_0)$	–
C_1	$\{t_B\}$	$P(C_1)$	$D_P(C_1)$	–
...	...	...	...	–
C_x	$\{t_N\}$	$P(C_x)$	$D_P(C_x)$	–
C_{x+1}	$\{t_A, t_B\}$	$P(C_{x+1})$	–	$D_C(C_{x+1})$
C_{x+2}	$\{t_A, t_C\}$	$P(C_{x+2})$	–	$D_C(C_{x+2})$
...	...	...	–	...
C_y	$\{t_I, t_N\}$	$P(C_y)$	–	$D_C(C_y)$
C_{y+1}	$\{t_A, t_B, t_C\}$	$P(C_{y+1})$	–	–
...	...	...	–	–
C_z	$\{t_I, t_J, ..., t_N\}$	$P(C_z)$	–	–

IV. Domain Analyzer

Domain analyzer extracts the behavioral and structural similarities from the applications within a domain. It expresses these similarities using the domain feature definitions of Section III. The computed domain features then feed into our domain-specific DSE described in Section V.

Algorithm 1 overviews the domain analyzer. The analyzer is comprised of domain analysis and application analysis. The domain analysis (lines 1-11) first calls application analysis for each application to obtain the list of function type compositions ($CList$ in line3). Then the domain analysis merges compositions from all applications, counts their appearance frequency (lines 4-8), and aggregates their processing and communication demand (D_P, D_C in line 9). Finally, the domain analysis calculates each composition appearance probability (P in line 11) and returns the the domain features as a set of function type compositions.

Algorithm 1 Domain Analyzer

```
1:  function DMAnalysis(G)
2:      for each g ∈ G do
3:          CList = appAnalysis(g)
4:          for each C ∈ CList do
5:              if C ∈ CS then
6:                  Freq(C)++
7:              else
8:                  CS = CS ∪ {C}; Freq(C) = 1
9:              D_P(C) += C.D_P; D_C(C) += C.D_C
10:     for each C ∈ CS do
11:         P(C) = Freq(C)/|G|
        return CS

12: function appAnalysis(g)
13:     for each a ∈ g.A do
14:         cList.add({a})
15:     for k ∈ {1, 2, ...} do                    ▷ k is composition degree
16:         for each c ∈ {cList ∩ |c| = k} do
17:             for each a_next ← c.a_tail().a_outNeighbor() do
18:                 if a_next ∉ c then
19:                     cList.add(c ∪ {a_next})
20:         if {c|c ∈ cList ∩ |c| = k+1} = ∅ then Break
21:     for each c ∈ cList do
22:         C = {a.t|a ∈ c}
23:         if |c| = 1 then C.D_P = a.d_P, which a ∈ c
24:         if |c| = 2 then C.D_C = e.d_C, which e.a_src,dst ∈ c
25:         if C ∈ CList then
26:             CList[C].updateD(C.D_P, C.D_C)
27:         else
28:             CList.add(C)
        return CList
```

The application analysis (lines 12-28) creates a list of actor compositions (lines 12-20) and then abstracts from actor instances into function type compositions (lines 21-28). The analysis starts with each actor from the application as 1-degree composition (lines 13-14). Then for each k-degree composition, it adds one more actor to it to build a $k+1$ degree composition (lines 15-19). When no actor can be added anymore, actor compositions detection stops (line 20). After that, the analysis obtains a function type composition from each actor composition(lines 21-22). It merges duplicated function type compositions and aggregates their processing/communication demand (lines 23-28). It returns the

list of compositions of this application, with their aggregated processing and communication demands.

V. Domain DSE

This section proposes a novel domain-specific DSE for HW/SW partitioning for a domain of applications. The key insight is to broaden the scope of the exploration heuristic from a single app in isolation to a group of applications. In general, different optimization goals are possible, e.g. minimize power consumption or maximize throughput with various constraints (e.g. cost, area). In this work, we focus on throughput improvement for all applications within a certain HW budget. This section introduces a new algorithm: dynamic score selection (DSS). It starts with a pure SW platform and then iterates through greedily selecting a composition candidate from domain features for HW realization. Our greedy approach uses the Dynamic Composition Score (DCS) to estimate which composition is the most promising to improve domain performance. This section, first introduces the dynamic composition score and then the DSS algorithm.

A. Dynamic Composition Score (DCS)

Given the amount of applications a domain and the myriad domain platform design options, a detailed simulations or detailed analytical approaches that include scheduling effects are prohibitively expensive to execute. Instead, our approach focuses on overall processing demand (i.e. how many operations) of the domain and processing supply by the architecture (symmetric for communication). The dynamic composition score quantifies the benefits moving a composition C from processor to hardware mapping. DCS estimates the benefits stemming from processing and communication mapping, scales them using dynamic weights (tuned to domain properties) and includes the additional cost of hardware implementation.

$$\begin{aligned} Score &= (B_P * w_P + B_C * w_C)/cost \\ cost &= |C \cap SW| \end{aligned} \tag{5}$$

In Eq. 5, B_P and B_C are the benefits of processing and communication if the composition is mapped on HW, respectively. They are calculated considering both aggregated benefits and appearances (Sec.V-A1 and V-A2) . w_P and w_C are the weights to scale both benefits. The weights are dynamically adjusted tracking performance bottlenecks, where processing or communication might be more important. Obtaining these weights will be discussed in Sec. V-A3. The *cost* is defined the number of additional function types (i.e. HWACCs) to realize the composition in HW.

1) Processing Benefit: Eq. 6 estimates the processing benefit B_P as execution time reduction of the domain when mapping to HW only taking computation demand and supply into account.

$$B_P = \sum\nolimits_{t_i \in (C \cap SW)} \left(\frac{D_P(\{t_i\})}{S_{SW}} - \frac{D_P(\{t_i\})}{S_{HW}}\right) * P(\{t_i\}) \tag{6}$$

In Eq. 6, S_{SW} and S_{HW} are the estimated execution speed of SW and HW, respectively. For each function type t in C that is not mapped to HW, the overall difference of execution time between SW and and HW processing is calculated. This difference is then multiplied by the appearance probability P of t to prioritize frequently appearing compositions. The sum of the scaled processing benefits of these function types yields the processing benefit B_P of this composition given the existing mapping.

2) Communication Benefit: The communication benefit is defined as the change in bus usage (transfer delay) due to mapping to HW. It considers three aspects: (1) intra-composition communication, (2) communication with other HW-mapped functions, and (3) additionally exposed HW/SW communication. Communication between SW-mapped functions is assumed to be hidden in the memory hierarchy.

$$\begin{aligned} B_C = &\sum\nolimits_{t_i,t_j \in (C \cap SW)} D_C(\{t_i,t_j\})/S_C * P(\{t_i,t_j\}) \\ &+ \sum\nolimits_{t_i \in (C \cap SW), t_j \in HW} D_C(\{t_i,t_j\})/S_C * P(\{t_i,t_j\}) \\ &- \sum\nolimits_{t_i \in (C \cap SW), t_j \in (SW \setminus C)} D_C(\{t_i,t_j\})/S_C * P(\{t_i,t_j\}) \end{aligned} \tag{7}$$

Eq. 6 captures the communication benefits, assuming S_C is the communication speed of buses. It computes the saved versus exposed data transfer volume D_C on bus divided by the communication speed S_C to yield the change transfer delay. The change in delay scaling with appearance probability and its accumulation across all applications yields the communication benefit B_C, which is the change in transfer delay for the whole domain.

3) Dynamic Weights: Communication and processing demands vary by domains. In result, e.g. for a processing dominated domains (e.g. computer vision) targeting processing has priority resulting in the biggest improvements. While the benefits computed before already capture some of it, a global domain scope is needed to weigh the importance of communication versus processing of the whole domain. To capture this, we define the weights w_C and w_P. However, each mapping to HW alters the effective communication vs. processing split and may change the properties and bottlenecks of the remaining compositions in a domain. For example, once processing intensive compositions are mapped to HW, the remaining compositions are communication-intensive. Therefore, the weights cannot be static but must be dynamically updated in each iteration to adjust for past mapping decisions.

$$\begin{aligned} w_P &= \frac{\sum_{t_i \in SW} D_P(\{t_i\})}{\sum_{t_i \in SW} D_P(\{t_i\}) + \sum_{\{t_i,t_j\} \notin HW} D_C(\{t_i,t_j\})} \\ w_C &= 1 - w_P \end{aligned} \tag{8}$$

Eq. 8 illustrates the dynamic weight calculation. It uses the total remaining domain processing and communication demand (not mapped to HW) to determine the current performance bottleneck. A higher total demand of non-mapped (i.e. SW) processing raises the processing weight w_P. Conversely, if communication dominates the remaining domain, the communication demand w_C will increase.

B. DS-DSE with Dynamic Score Selection (DSS)

Dynamic Score Selection (DSS) implements Domain DSE based on DCS (Sec. V-A). Algorithm 2 illustrates the flow. DSS starts with a pure SW mapping (line 1-2) and then greedily select the best candidate for HW mapping. Candidates are all compositions (and their contained function types) of the domain (line 3). DSS evaluates their benefits using the DCS (line 5-8) given current mapping and selects highest scored candidate. With a positive score, ie. performance improvement (line9), it maps all its SW-mapped function types to HW (lines 10-12). Subsequently it updates candidates by removing HW-mapped compositions (line 13). The loop (starting at line 4) repeats until the HW budget is exhausted (line 4) or no further improvement is possible (all negative score) (line 15).

Algorithm 2 DSS: Dynamic Score Selection

```
1:  SW ← T
2:  HW ← ∅
3:  Cand ← {C|C ∈ CS}                                  ▷ Candidates
4:  while |HW| < HWbudget do
5:      ▷ Processing/Communication weight calculation
6:      (w_P, w_C) = weightCal(CS, SW, HW)
7:      ▷ Calculate each candidate C score, return the highest
8:      C = scoreHighest(Cand, CS, SW, HW, w_P, w_C)
9:      if C.Score > 0 then
10:         for each t ∈ C ∩ SW do
11:             SW = SW \ {t}
12:             HW = HW ∪ {t}
13:         Cand = Cand \ {C.t|C ⊆ HW}
14:     else
15:         Break                                      ▷ No improvement
```

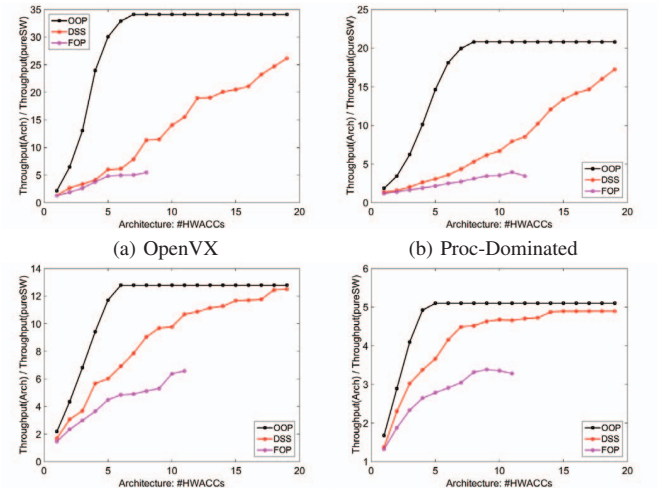
VI. Experimental Results

This section evaluates the benefit of executing on a DSS generated domain-specific platform. To understand the range of performance and to compare with application-specific platforms, we define an OPTimal application-specific architecture (OPT). Each application has an own OPT that maximizes the application throughput. Each applications OPT is obtained through (time consuming) exhaustive search and evaluation. The applications performance upper bound can be obtained by executing it on its own OPT. The domain's upper performance is obtained by executing each application on its own OPT. The aggregated throughput across all apps in the domain is reported as **Own OPT Platform (OOP)**. One application's OOP becomes one foreign platform for all other apps in the same domain. To understand domain effects, we also run all applications in a domain on all the OPTs (each of which is optimal for one, but foreign for all other applications). The aggregated performance is called **Foreign OPT Platform (FOP)**. The paper evaluates a pure SW platform (pureSW) as a lower bound.

Table III: Example Domains: OpenVX and Synthetic

Domain	OpenVX	Synthetic ProcDom	Synthetic Balanced	Synthetic CommDom
$\|G\|$, #app	24	100	100	100
$\|T\|$, #func types	28	50	50	50
$\|A\|$, #actors in apps	2-12	5-15	4-12	5-14
D_P:D_C (#op:bytes)	19.4	8.74	3.32	1.17

We evaluate 4 domains. The computer vision domain OpenVX, contains 24 real applications from Intel[13] and AMD[14]. The other three domains are synthetically generated with different processing/communication ratios. Table III lists their properties: number of applications, function types, application processing kernel range and the ratio between domain processing and communication demand. Performance is evaluated on automatically generated virtual platforms (SCE[15]) capturing the abstract data flow model (SDF3[12]) and architecture resources. The platform is a single processor with N ACCs. Communication occurs on a 3R3W layer bus. Processing on HW is 20x faster than SW. ACCs can communicate directly with each other[16].

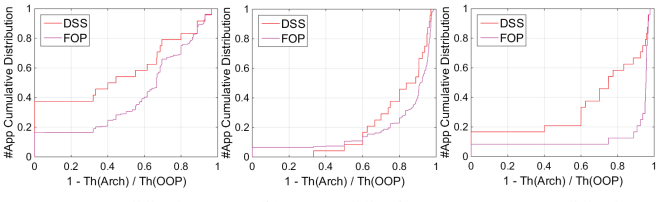


(a) OpenVX (b) Proc-Dominated

(c) Balanced (d) Comm-Dominated

Figure 3: Average Throughput Improvement

Fig. 3a plots the average performance improvement of OpenVX over increasing HW budget. Performance improvement is the ratio between each application throughput on the test architecture over the lower bound (pureSW): $throughput(arch)/throughput(pureSW)$. OOP yields the absolute the upper bound given the HW constraints. The highest throughput improvement is achieved at HWACCs=8, when all function types are mapped to HW in its own OPT architecture. DSS is below that, but keeps improving with HW budget. DSS is always much better than than application-specific FOP. The FOP plateaus with 8 HWACCs as already all possible ACCs have been selected for each app and it cannot find other function types outside this application to accelerate. With a budget of 1 through 8, DSS has improves throughput by 36.8% over FOP.



(a) #HWACCs=2 (b) #HWACCs=4 (c) #HWACCs=8

Figure 4: OpenVX: Throughput Loss Cumulative Distribution

To better understand performance variations in OpenVX's performance when using DSS and FOP, Fig. 4 shows the cumulative distribution of applications throughput loss over varying number of HWACCs (2, 4 and 8). The throughput loss is in comparison execution on OPT (each app's optimal platform), defined as $1 - throughput(arch)/throughput(OOP)$ for each application. The performance loss on DSS is much less than FOP. For example in Fig. 4a, 40% of applications lose with DSS only up to 30%, whereas up to 60% with FOP. The DSS and FOP have similar average performance with a budget of 4 (see Fig. 3a). The cumulative distribution in Fig. 4b reveals that less than 10% applications have a lower performance on DSS compared to FOP. Further analysis showed that the better performance for the 10% in FOP stems from each application running on its own OPT architecture. With increased budget, Fig. 4c, the DSS has much less performance loss, e.g. 20% of applications have loss of 40% or less, 90% or less for FOP.

Fig. 3b, 3c, and 3d show the average throughput improvement of the three synthetic domains. In these three domains, DSS has a significantly better performance, which achieves a 39.2%, 48.8% and 50.7% higher improvement than FOP. Improvement is influenced by the domains ratio of processing/communication demand and the number of HWACCs. In the processing dominated domain (Fig. 3b), DSS cannot achieve the highest performance even with 19 HWACCs, as all function types have high processing demand. In contrast, in a communication dominated domain (Fig. 3d) and the balanced domain (Fig. 3c), DSS almost achieves the highest possible performance ($>$90%) at HWACCs=9 and HWACCs=15, respectively. After mapping processing intense functions mapped to HW, communication time on the bus becomes the bottleneck. By avoiding communication exposed on the bus, the DSS algorithm can achieve performance very close to the OOP.

Domain analyzer and domain exploration have a negligible run-time due to the high level of abstraction. Analyzing the balanced synthetic domain with 100 apps (with up to 12 nodes each) on Intel i5-3450 with 3.10GHz takes 22.1s. Exploring the domain through DSS for 10 accelerators finishes in 0.07s. As the graph traversal in the analyzer already identifies the essential features, evaluation for exploration is rapid. The validation through VP simulation occupies dramatically more time. As the current synthetic domains are sized to match OpenVX, larger domains are needed investigate scalability.

VII. Conclusion

This introduces a Domain-specific Design Space Exploration (DS-DSE) methodology to broaden the scope of existing DSE from individual apps in isolation to a whole domain. This paper lays the foundations by defining a domain and quantifiable features (metrics) that can guide exploration. The features take both behavioral (processing) and structural (communication) aspects into account, as well as consider the distribution over the domain. Based on these definitions, the paper has introduced the Dynamic Score Selection (DSS) algorithm for domain exploration. It maximizes domain throughput creates a domain-specific architecture that has more flexibility to execute domain applications. Our results on 4 domains (OpenVX and 3 synthetic domains) demonstrate a significant performance improvement (36.8%-50.7%) executing on the respective domain architecture compared to application-specific architectures.

Acknowledgment

This material is based upon work partially supported by the National Science Foundation under Grant No. 1319501.

References

[1] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit, "Platform 2012, a many-core computing accelerator for embedded socs: performance evaluation of visual analytics applications," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 1137–1142.

[2] H. Tabkhi, R. Bushey, and G. Schirner, "Function-Level Processor (FLP): A Novel Processor Class for Efficient Processing of Streaming Applications," *Journal of Signal Processing Systems*, vol. 85, no. 3, pp. 287–306, 2016.

[3] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," *arXiv preprint arXiv:1704.04760*, 2017.

[4] H. Abdeen, D. Varró, H. Sahraoui, A. S. Nagy, C. Debreceni, Á. Hegedüs, and Á. Horváth, "Multi-objective optimization in rule-based design space exploration," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 289–300.

[5] Z. Liang, X. Cheng, T. Zheng, and L. Tao, "Hardware/software partitioning based on greedy algorithm and simulated annealing algorithm," *J. Comput. Appl*, vol. 7, p. 030, 2013.

[6] J. Wu, P. Wang, S.-K. Lam, and T. Srikanthan, "Efficient heuristic and tabu search for hardware/software partitioning," *The Journal of Supercomputing*, vol. 66, no. 1, pp. 118–134, 2013.

[7] J. W. Tang, Y. W. Hau, and M. N. Marsono, "Hardware/software partitioning of embedded system-on-chip applications," in *Very Large Scale Integration (VLSI-SoC), 2015 IFIP/IEEE International Conference on*. IEEE, 2015, pp. 331–336.

[8] M. Flasskamp, G. Sievers, J. Ax, C. Klarhorst, T. Jungeblut, W. Kelly, M. Thies, and M. Porrmann, "Performance estimation of streaming applications for hierarchical mpsocs," in *Proceedings of the 2016 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. ACM, 2016, p. 3.

[9] S. Graf, M. Glaß, J. Teich, and C. Lauer, "Multi-variant-based design space exploration for automotive embedded systems," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. IEEE, 2014, pp. 1–6.

[10] S. Wildermann, F. Reimann, J. Teich, and Z. Salcic, "Operational mode exploration for reconfigurable systems with multiple applications," in *Field-Programmable Technology (FPT), 2011 International Conference on*. IEEE, 2011, pp. 1–8.

[11] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, Tech. Rep., 1990.

[12] S. Stuijk, M. Geilen, and T. Basten, "Sdf^ 3: Sdf for free," in *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*. IEEE, 2006, pp. 276–278.

[13] "Beta for intel computer vision sdk (intel cv sdk) support," https://software.intel.com/en-us/computer-vision-sdk-support/code-samples, accessed: 2017-09-12.

[14] "Amd openvx (amdovx)," http://gpuopen.com/compute-product/amd-openvx/, accessed: 2017-09-12.

[15] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. D. Gajski, "System-on-chip environment: A specc-based framework for heterogeneous mpsoc design," *EURASIP Journal on Embedded Systems*, vol. 2008, p. 5, 2008.

[16] N. Teimouri, H. Tabkhi, and G. Schirner, "Improving scalability of cmps with dense accs coverage," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*. IEEE, 2016, pp. 1610–1615.