
Yao Xiao, Shahin Nazarian, Member, IEEE, and Paul Bogdan, Senior Member, IEEE

Abstract—There exists an urgent need for determining the right amount and type of specialization while making a heterogeneous system as programmable and flexible as possible. Therefore, in this paper, we pioneer a self-optimizing and self-programming computing system (SOSPCS) design framework that achieves both programmability and flexibility and exploits computing heterogeneity [e.g., CPUs, GPUs, and hardware accelerators (HWAs)]. First, at compile time, we form a task pool consisting of hybrid tasks with different processing element (PE) affinities according to target applications. Tasks preferred to be executed on GPUs or accelerators are detected from target applications by neural networks. Tasks suitable to run on CPUs are formed by community detection to minimize data movement overhead. Next, a distributed reinforcement learning-based approach is used at runtime to allow agents to map the tasks onto the network-on-chip-based heterogeneous PEs by learning an optimal policy based on \( Q \) values in the environment. We have conducted experiments on a heterogeneous platform consisting of CPUs, GPUs, and HWAs with deep learning algorithms such as matrix multiplication, ReLU, and sigmoid functions. We concluded that SOSPCS provides performance improvement up to \( 4.12 \times \) and energy reduction up to \( 3.24 \times \) compared to the state-of-the-art approaches.

Index Terms—Distributed Q-learning, domain-specific system-on-chip (DSSoC), heterogeneous systems, network-on-chip (NoC), neural networks (NNs), self-optimizing, self-programming, software-defined hardware (SDH).

I. INTRODUCTION

TODAY, multicore platforms can solve general types of problems with the help of programmers. Programming flexibility is the primary reason why general-purpose machines have been adopted. However, programmers would not be offered with necessary capabilities to run algorithms efficiently, as they have to trade off efficiency of their algorithms with the available hardware components [29]. With the advent of tensor processing units (TPUs) [17] dedicated to accelerate deep learning algorithms and some hardware accelerators (HWAs) for image processing, domain-specific accelerators [5] can typically improve performance with less energy. Nevertheless, to exploit the high degree of parallelism available on multicore platforms, programmers write multithreaded applications where multiple threads are spawned and mapped onto cores by the operating system (OS). These threads share data by storing one copy of the data into the data cache. If one thread updates the shared variable, the same new copy has to be broadcast to the rest of cores in order to keep information consistent. Network-on-Chip (NoC) has been proven effective for data transfer among cores. For each core, there exists a router to transfer data from one node to another and the router keeps track of the next node to which flits should be sent. However, there are drawbacks associated with each approach:

A. General-Purpose Machines Are Not as Efficient as Specialized Accelerators

The compute engines for these machines are designed to be simple and general such as floating-point adders, multipliers, and dividers. For compute-intensive applications like signal processing consisting of millions of operations, running these applications on general-purpose machines is inefficient compared to specialized HWAs tailored to specific computations.

B. Domain-Specific Accelerators Lack Programmability and Add Complexity to OS Design

Domain-specific accelerators [5] are specific to one domain. For example, TPU [17] dedicates a large area for matrix multiplication (MM) unit to speed up deep neural network (NN) algorithms consisting of a huge amount of MM operations. However, applications without these operations cannot be executed in TPU. Furthermore, with available on-chip accelerators, the OS should decide the following: 1) the components of each task (e.g., if OS decides a task consisting of fast Fourier transform (FFT) plus some sequential code,
the FFT accelerator cannot execute it due to the inflexibility of execution of general programs) and 2) the mapping of tasks onto processing elements (PEs) such as CPUs, GPUs, and accelerators with low network latency and high performance. Tasks with FFT are required to be mapped onto the FFT accelerator and highly intercommunicating tasks are mapped to nearby PEs for the purpose of high performance.

C. NoC Consumes a Significant Portion of System Power Due to Data Movement

Several NoC prototypes demonstrated that approximately 30% of the total energy are spent on NoC in the Intel 80-core Terascale chip [12] and 40% in the MIT RAW chip [37]. Furthermore, [19] reported that data movement consumes, on average, 25% of the total energy.

Within the cyber-physical systems era, future hardware platforms will perform streaming and steering computations [10]. Streaming applications require inputs as unbounded sequences of various types of events to be sequentially applied to each application (without random accesses) which needs near real-time processing and can have multiple consumers and producers. Steering is defined as the ability to dynamically control the progression of each computational process to support decision-making such as the mapping of tasks and the control of autonomous vehicles. For example, in autonomous cars, data streams are unbounded and heterogeneous as the vehicles require the real-time vision of roads, streets, and cars in order to react on the direction and speed. In addition, the autonomous vehicles allow for dynamic steering (turn left/right or stop) in case of a real-time event (the change of traffic lights). Therefore, simple heuristics are difficult to cover all possible cases under heterogeneous data streaming and steering. Reinforcement learning (RL)-based distributed intelligent scheduling has proven to be very promising in reacting in real time and allows the dynamic control of events such as traffic congestion in the interconnect and task scheduling in cloud computing [6].

In this paper, we propose a self-optimizing and self-programming computing system (SOSPCS) framework, which overcomes the aforementioned drawbacks. SOSPCS provides flexibility, programmability, and performance/energy efficiency for heterogeneous systems. Self-optimization and self-programmability are significant features in computing systems. It, at a low level, understands the dynamic nature of heterogeneous systems and, at a high level, creates and maps heterogeneous tasks to hardware [8], [14] without the needs of programmers to write domain-specific languages to fully utilize the hardware components. It also finds the optimal number and type of cores and HWAs in order to run applications in a domain such as machine learning. Therefore, in SOSPCS, at compile time, SOSPCS first transforms target applications from low-level virtual machine (LLVM) intermediate representation (IR) instructions into instruction dependency graphs (IDGs) where nodes denote LLVM IR instructions; edges denote data dependencies; and edge weights denote the amount of data to be transferred between two dependent instructions. Second, we train in offline the sliding window-based NN classifiers with three hidden layers (HLs) by applying some standard applications in a domain such as MM, neurons, and ReLU activation function to make sure that the NNs can learn these special graph structures. The first NN determines the feature type, e.g., MM, neurons, or ReLU; on the other hand, the second decides the coordinates of each feature. Third, when new applications arrive, SOSPCS transforms them into IDGs and applies NNs to locate special structures and their specific types. Fourth, SOSPCS applies community detection (CD) to partition the remaining graph into interconnected tasks with minimal data movement overhead. At runtime, we combine RL with task scheduling to implement distributed intelligent schedulers to map tasks onto either general CPUs/GPUs or domain-specific PEs. These schedulers obtain feedback from the hardware environment such as the next state and the immediate reward formulated in terms of the types of tasks and availability of hardware components.

In general, SOSPCS integrates the concepts of software-defined hardware (SDH) and domain-specific system-on-chip (DSSoC). Our system has the ability to reconfigure the hardware as specified in application requirements and translate high-level programs into tasks with application-specific hardware configurations by LLVM code that has the well-established interface with the existing OS. On the other hand, SOSPCS satisfies the requirements of DSSoC by having RL-based distributed intelligent schedulers to manage the set of domain resources required by applications and map each task created at compile time onto the appropriate PE [29].

Toward this end, our main contributions are as follows.
1) SOSPCS provides flexible programmability by developing a compiler-based approach to profile high-level programs into universal machine codes and constructing the corresponding dynamic IDG from LLVM IR instructions.
2) SOSPCS provides hardware efficiency and reusability through an optimization framework for automatically identifying the optimal number and type of hardware resources to speed up the execution of applications (e.g., deep learning).
3) SOSPCS provides energy efficiency by minimizing the amount of messages among hardware resources to balance computation and communication.
4) SOSPCS allows multiple applications to be scheduled and run in parallel. Its learning-based engines are designed to easily incorporate any scheduling algorithm and learn application/task priority features. The hardware reconfigurability of our underlying DSSoC facilitates the consideration of application priority.

The rest of this paper is organized as follows. Section II presents the related work on task mapping and RL. Sections III and IV provide background and mathematical models for this paper. Section V describes the proposed SOSPCS framework, including compile-time resource allocation and runtime resource management. Section VI provides experimental results on performance improvement of SOSPCS against the state-of-the-art scheduling policy on heterogeneous systems. Section V concludes this paper.
II. RELATED WORK

The complex software applications with fixed deadlines and parallel programming require high-performance heterogeneous computing platforms. Efficient application mapping methods are studied to improve performance or energy consumption on complex computing systems. Design-time mapping methods [4, 7, 13, 15, 25, 33, 39, 40] are suitable for static workloads where computations and communications are well-known. Runtime mapping methods [1, 34, 35, 38] allow dynamic mapping of tasks considering the inefficiency of static mapping. Hybrid mapping [20, 22, 30, 32] combines design-time mapping with runtime mapping to overcome the drawbacks.

A. Design-Time Mapping

Thiele et al. [39] proposed a distributed operation layer mapping framework for efficient parallel execution on multicore systems. Their heuristics called group-based mapping to assign processes and channels of KPN applications onto heterogeneous multiprocessor systems-on-chips (MPSoCs). Choi et al. [7] proposed an evolutionary algorithm-based scheduling technique to address a limited size of scratchpad memory (SPM) on a multicore platform for synchronous dataflow applications. To reduce memory latency, prefetching is performed to fetch data from SPM concurrently with computation.

B. Runtime Mapping

Ahmed et al. [1] proposed a novel scheme for adaptive runtime resource management in reconfigurable multicore processors. At compile time, it analyzes the criticality of each task to constrain the functional blocks in applications. At runtime, it dynamically allocates the fabric to tasks in terms of their performance and time constraints. Huang et al. [13] introduced a self-adaptive mapping approach. Their heuristic, called the group-based mapping technique, addresses a limited size of scratchpad memory (SPM) on a multicore platform for synchronous dataflow applications. To reduce memory latency, prefetching is performed to fetch data from SPM concurrently with computation.

C. Hybrid Mapping

Schranzhofer et al. [32] proposed a novel methodology to compute a task-to-PE mapping based on the execution modes of target applications to minimize power consumption. At design time, this methodology precomputes the static mapping of tasks onto PEs for all possible execution modes. At runtime, it applies the mapping in order to adapt to the environmental variations. Piscitelli and Pimentel [30] presented a hybrid technique to help system-level design space exploration (DSE) to prune the number of states needed during DSE. It analyzes the expected throughput estimation of applications in terms of application-to-architecture mapping. In the meantime, simulation is also applied to examine the design points in order to make sure DSE process is correct.

In contrast, in this paper, we make the following novel contributions: 1) to address programming flexibility and fully exploit the underlying heterogeneous PEs (CPUs, GPUs, and HWAs), SOSPCS automatically partitions target applications in one domain into interdependent tasks with different PE affinities. Some tasks such as FFT prefer executing on accelerators, some such as for-loops prefer executing on GPUs, whereas the rest (e.g., general-purpose code) execute on CPUs and 2) our runtime mapping is based on distributed RL to self-optimize the optimal policy under the environment variations.

III. BACKGROUND

A. Low-Level Virtual Machine

It is a compiler framework that makes program analysis lifelong and transparent by introducing IR as a common model for analysis, transformation, and synthesis [21], [31]. LLVM is a load/store architecture in which data transferred among registers, caches, and memory banks use only load and store instructions. There are several advantages using IR: 1) it is machine independent (portable) to avoid low-level actions such as register spilling and function prolog/epilog insertion and 2) it is language independent to represent high-level languages including C/C++ and build retargetable compilers.

B. Community Detection and Modularity

Graph partitioning is an approach to divide the computations among processors, which is vital in parallel computing [11]. We are inspired from complex network theory where there is an increasing need to find community structures [28], clusters of densely connected nodes with sparse intercommunity edges. The most popular method to detect communities is based on modularity [9]. Modularity can be described as a quality function $Q$ for good partitioning of graphs into communities by comparing edge weights falling within communities with the corresponding weights in a random graph with the equivalent number of nodes [27]. The higher the modularity is, the better is the partitioning. Mathematically, modularity [9] is expressed as

$$Q = \frac{1}{2m} \sum_{ij} (A_{ij} - P_{ij}) \delta(C_i, C_j)$$

where $m$ is the total number of edges in a network; $A$ is the adjacency matrix; $C_i$ is the community index for node $i$; $\delta(C_i, C_j)$ equals 1 if nodes $i$ and $j$ fall within the same community and 0 otherwise; $P$ denotes the expected number of edges in a random graph with the same degree distribution as the graph to be analyzed. Therefore, CD is commonly used in complex networks to find communities or clusters, e.g., tweeters with similar interests in social networks. In our case,
we modify the CD to account for the computational and communication requirements and so the communities or clusters obtained by our algorithm will detect the high computation intensity, i.e., computations mostly done within a community without the need to fetch from another community.

IV. GRAPH THEORETICAL MODELS

In this section, we present four definitions for SOSPCS.

Definition 1: An IDG is defined as a directed acyclic graph (DAG) $TDG(n_I, e_{ij}, w_{ij}, i, j \in \{1, \ldots, N\})$ where nodes $n_I$ represent LLVM IR instructions, edges $e_{ij}$ represent data dependencies between two instructions, and edge weights $w_{ij}$ represent the amount of data to be transferred.

Definition 2: A task is defined as a DAG $T(n_I, e_{ij}, w_{ij}, typeT, i, j \in \{1, \ldots, N\})$ where nodes $n_I$ represent IR instructions, edges $e_{ij}$ represent data dependencies, and edge weights $w_{ij}$ represent the amount of data to be transferred. Note that $T \subseteq TDG$. In addition, each task has its own attribute type representing the execution affinity. For example, a task with loops (type $T$ = par) is more beneficial to be executed in GPUs rather than CPUs. A task with MM (type $T$ = mm) can be executed in HWAs to gain the best performance.

Definition 3: A task pool is defined as a DAG $TP(n_T, e_{ij}, w_{ij}, typeTP, i, j \in \{1, \ldots, N\})$ where nodes represent tasks $T_i$ and edges represent messages to be transferred among tasks. Note that $TP = T_1 + \ldots + T_N = TDG$.

Definition 4: An architecture graph with a heterogeneous NoC of dimensions $X$ and $Y$ is defined as an undirected graph $NoC(p_{xy}, busy_{xy}, typeNoC, l_{ij}, b_{ij})$ where each node $p_{xy}$ represents a PE at position $(x, y)$ with types of CPU, GPU, or various specialized HWAs such as FFT and MM. $busy_{xy}$ models the availability of a given PE $p_{xy}$ with 1 being not available. Furthermore, each communication link $l_{ij}$ has a maximum bandwidth $b_{ij}$.

Note that each task has preferred execution affinity. Therefore, in order to improve performance, tasks should be mapped to their most beneficial PEs. For example, a task with loops (type $T$ = par) should be mapped to GPU (type $NoC$ = GPU) in order to improve performance whereas a task with a convolutional layer (type $T$ = conv) cannot be mapped to special HWAs for MM (type $NoC$ = mm).

The SOSPCS framework solves the following problem: Given an architecture graph $NoC$ and target applications, 1) find a partition $P$ of $TDG$ to detect tasks with common features: $P(TDG) = TP$ (type = fft, mm, par, ser, …) and 2) find a mapping $M$ of $TP$ to PEs such that performance is maximized: $M(TP) = NoC$.

V. SOSPCS OPTIMIZATION FRAMEWORK

In this section, we describe the software implementation of SOSPCS in four steps as illustrated in Fig. 1, namely, application transformation, NN classifiers, CD (all three of which performed at compile time), and runtime resource management. At compile time, SOSPCS transforms applications into a complex graph and determines the types and locations of special features existed in the graph by NN classifiers. During training, sample applications are driven into the networks to learn specific features. Machine learning applications contain code regions that may be sequential or have loops, vector/MM, different kinds of activation functions (sigmoid or ReLU), and gradient descent. After training, these networks have the capability to mine hidden structures in new fairly complicated applications with at least two mixed features. Next, SOSPCS applies the CD approach to partition the rest of the graph into interdependent clusters/tasks with minimized data communication. At runtime, we manage resources at by using RL-based distributed intelligent schedulers to explore the best
locations onto which heterogeneous tasks should be mapped (i.e., tasks with loops are mapped to GPUs, whereas tasks with FFT or MM features should be mapped to specialized HWAs).

A. Compile Time: Resource Allocation

The objective is to find a partition function \( P \) that separates one application into tasks with various functionalities such as loops, FFT, and MM to exploit the underlying heterogeneous NoC-based hardware platform. Therefore, we first apply NNs to find and detect the special features existed deep in applications. Next, SOSPCS applies a modularity-based CD inspired approach to partition the remaining graph into clusters or tasks with minimal data movements. Therefore, similar to SDH, SOSPCS has the ability to translate high-level programs into LLVM IR and transform the IR trace into a graph representing the hardware constraints and resources.

Although modularity-based CD can partition the graph into several tasks with minimal intertask data transfers, we rely on NN to further guide the partitioning of the graph into tasks with the common features (that are learned offline such as loops and MM) because the CD may not find such patterns.

1) Application Transformation: We model target applications by constructing the corresponding IDG [41]. First, we execute the applications to collect dynamic LLVM IR traces. Second, we analyze the traces to figure out whether source registers of the current instruction depend on destination registers of the previous instructions. If data dependencies exist, we insert edges in between. Third, these instructions are profiled to get the precise data size as edge weights in IDG. Therefore, our approach combines static and dynamic program analysis. Static analysis means that we perform program data dependency analysis to construct the graph. Dynamic analysis means that we run applications with representative inputs to get the dynamic IR traces because memory dependencies cannot be resolved statically. For example, a part of dynamic traces have been listed as follows.

Listing 1. LLVM trace example.

\[
\begin{align*}
\%5 &= \text{or} \ %2, \ %3; \\
\%6 &= \text{add} \ %2, \ %5; \\
\%7 &= \text{mul} \ %5, \ %6;
\end{align*}
\]

1) The number of input neurons cannot be dynamically adjusted to accommodate to graphs with the varying number of nodes. Note that input data have an impact on the number of iterations in applications, leading to a different number of nodes in graphs IDG. Therefore, we apply the sliding window in NNs to make sure that the number of input neurons is static and equal to the size of the sliding window.

2) It is possible that due to the enormous number of iterations in features such as MM, the sliding window with the fixed size cannot detect the exact features. Hence, we try to remove some unnecessary dimensions to detect the features with high probability \( \beta \).
Principal component analysis (PCA) is an approach to compress a set of observations with \( n \) dimensions to a set of linearly uncorrelated variables with \( k \) dimensions \((k < n)\), which are called principal components.

3) Inputs such as the number of edges and average path length have different scales. For example, the number of edges may vary from hundreds to millions, whereas the average path length may be in the range of tens to hundreds. We scale inputs via mean normalization:

\[
\mu_j = \left(\frac{1}{m}\right) \sum_{i=1}^{m} x_j^{(i)}, \quad \text{where } x_j^{(i)} \text{ is the } i\text{th vector with the } j\text{th input to the NNs.}
\]

Note that we only normalize the properties associated with the graph such as clustering coefficient, not the vectorized adjacency matrix representing the structure of the application graph.

c) Inputs to NNs: After applying PCA to the dimension and preserve the main components, SOSPCS drives the vectorized adjacency matrix, along with the normalized number of edges, average path length, clustering coefficient, average degree, and average betweenness centrality into NNs.

d) Outputs to NNs: NNs return feature index and coordinates. This feature index determines the task attribute in Definition 2 and runtime intelligent schedulers require this attribute to decide the mapping and possible reconfiguration of the hardware components. It is determined by the output of the first NN. The coordinates indicate the location and size of each special feature such as FFT and MM. For example, identifying an object in an image requires coordinates to determine the location and size of each object. Similarly, coordinates are necessary to locate features in a large graph. Fig. 2 shows an example of coordinates. The location of the blue box is [6, 6, (1, 1)], which means that the height and width of the box are 6 and 6, and the location of its upper left corner is (1, 1). Its type, suggested by the first NN, is MM.

e) Training data preparation: During training, we collect a group of applications with different features. These are written as follows: 1) without loops for CPUs; 2) with loops for GPUs; or 3) with one feature such as neurons, different types of activation functions (sigmoid and ReLU), matrix/vector multiplication, gradient descent, various for-loops in a machine learning domain. These applications are written in C in separate files in order to be ready for SOSPCS to be transformed into an interacting graph. Since these applications are written by us, we know in advance the task attribute and assign the correct feature index during data preparation.

f) New arriving applications: During testing and simulation, we use unknown applications but from the same domain as those in the training testbenches, e.g., machine learning applications. These applications are listed in Table I.

These applications have at least two mixed features mentioned above to be able to test the validity of SOSPCS. When new applications arrive, we first transform them into IDGs. Based on the graphs and their metrics, we prepare input data for NNs, which is then driven into NNs. The first trained NN figures out the types of special features. For example, if the target domain is deep learning, special features contain MM, ReLU, neurons, and so on. The reason why we need to identify these features is that we could reconfigure them onto a field-programmable gate array or map them onto application-specified integrated circuit instead of executing on CPUs. The second NN determines the exact locations and types of features in a graph if they exist.

g) Prediction accuracy: In Section VI, we demonstrate the accuracy of the prediction and application performance while varying the number of HLs and the impact of misclassification. Based on the results, we decide to implement NNs with three HLs.

3) Graph Partitioning: After detecting special features in an IDG, SOSPCS partitions the remaining graph into tasks (type = seq) to: 1) obtain better parallelism in CPUs and 2) minimize data communication overhead to reduce NoC latency and improve performance. Hence, SOSPCS applies CD [41] to partition the remaining graph \( IDG_{\text{NN}} \) produced by application transformation into densely interconnected tasks such that data communications among tasks are minimal. Given a weighted graph \( G(n_i, e_{ij}, w_{ij}, i, j \in [1, \ldots, |N|]) \), modularity \( Q \) with adjustable number of tasks is defined as

\[
Q = \sum_{i=1}^{n_c} \left[ \frac{W(i)}{W} - \frac{S(i)^2}{2W} \right] - \frac{\gamma}{n_c} [n_c - N] \tag{1}
\]

where \( n_c \) is the number of partitioned communities; \( N \) is the number of available CPUs; \( \gamma \) represents the regularization parameter; \( W(i) \) represents the sum of edge weights connected within community \( i \) \( (W(i) = \sum_{j \in c} \sum_{k \in c} w_{kj}) \); \( W \) denotes the sum of weights of all edges \( \left( W = \sum_{ij} w_{ij} \right) \); and \( S(i) \) denotes the sum of all weights adjacent to community \( i \).

We then define the partitioning problem as follows: given an IDG \( IDG \), find nonoverlapping communities such that the quality function \( Q \) in (1) is maximized.

According to (1), \( (W(i)/W) \) represents the ratio of the total edge weights within a community \( i \), whereas \( (S(i)^2/2W) \) measures the ratio of the weights adjacent to the community representing the amount of data to be communicated to other communities. Therefore, SOSPCS tries to maximize the intracommunity weights and minimize the intercommunity weights. For the second term, we constrain that the number of clusters generated from the model is approximately the same as the number of CPUs.

Inspired by the algorithm discussed in [2] from complex network literature, we propose Algorithm 1 to detect communities. The basic idea is that a pass is repeated until there are no more modularity gains (MGs) in the network. In one pass, we randomly choose a cluster \( i \) as a base case, calculate MGs when adding the neighbors into cluster \( i \) one at a time, select the clusters with the highest MG, and combine them into one cluster until there are no MGs.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>#Nodes</th>
<th>#Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>BP</td>
<td>Backpropagation</td>
<td>476,012</td>
<td>893,471</td>
</tr>
<tr>
<td>CNN</td>
<td>Convolutional Neural Nets</td>
<td>1,017,434</td>
<td>1,728,723</td>
</tr>
<tr>
<td>KM</td>
<td>K-Means Clustering</td>
<td>787,645</td>
<td>925,336</td>
</tr>
<tr>
<td>SVM</td>
<td>Support Vector Machine</td>
<td>723,559</td>
<td>923,128</td>
</tr>
<tr>
<td>NB</td>
<td>Naive Bayes</td>
<td>348,664</td>
<td>803,207</td>
</tr>
<tr>
<td>LR</td>
<td>Logistic Regression</td>
<td>925,733</td>
<td>1,371,252</td>
</tr>
</tbody>
</table>
Algorithm 1 Iterative Approximated CD Algorithm

**Input:** A remaining dependency graph $TDg \setminus TDg^N$

**Output:** A task pool $TP$

1. Start with each node as a singleton cluster
2. repeat
3. $bestMG = 0$, $currMG = 0$, $bestC = \emptyset$
4. Randomly choose a cluster $i$
5. for $C \in \text{Neighbor}(i)$ do
6. if $(currMG = \text{ModGain}(i, C)) > bestMG$ then
7. $bestMG = currMG$, $bestC = C$
8. end if
9. end for
10. Combine clusters $bestC$ and $i$ into one cluster
11. until There are no modularity gains

Fig 3. IDGs. (a) FFT. (b) Signal processing application.

Fig. 3(b) shows the structure of the graph for a signal processing application, including the FFT functionality and a sequential program with loops. Nodes of the same color represent computations within a task. Therefore, we can see that SOSPCS has created the task pool to be consumed by the runtime system.

**B. Runtime: Resource Management**

In this section, the objective is to find a mapping $\mathcal{M}$ to schedule tasks onto PEs such that performance is maximized. However, in order to gain the maximum speedup, ideally, the following criteria have to be satisfied: 1) the types of tasks should match with the types of PEs to which the tasks are mapped: $type^T = type^x\_xy$ and 2) data communications among PEs have to be minimal by putting PEs that directly communicate with each other close enough.

Traditional static task mapping ignores the dynamic nature of the underlying hardware platforms such as traffic in NoC and availability of cores. For example, in Fig. 4, two dependent tasks could be mapped to the blue region if there exists traffic congestion in the red region. Learning is a better approach to deal with these uncertain situations in multicore platforms. In addition, streaming and steering computations have to work with the heterogeneous data streams, making simple heuristics difficult to solve all possible cases [10]. Therefore, we design distributed RL-based intelligent schedulers to map tasks to the best suitable PEs. There are two advantages: 1) the technique learns and self-optimizes the mapping of tasks onto heterogeneous systems in order to gain the optimal performance and 2) it is dynamic, meaning that at runtime, based on the states of the environment and immediate rewards, agents decide the best policies for task mapping. Therefore, SOSPCS satisfies the requirements of SDH and DSSoC by having distributed intelligent schedulers to manage the set of domain-specific tasks and enable reconfiguration of hardware components.

1) Task Scheduling: We model the scheduling of tasks into agents as a balls-into-bins problem and provide a theoretical upper bound for loads in each agent to make sure loads are somewhat balanced among agents. A balls-into-bins problem can be described as the processing throwing $n$ balls into $n$ bins. Each ball is thrown into a uniformly random bin, regardless of the other balls. Therefore, the probability that a ball falls into any given bin is $(1/n)$. It is proven that the maximum loaded bin has $(3 \log n / \log \log n)$ balls with high probability, i.e., $(1 - 1/n)$. However, we can make loads among bins even more balanced by the power of two choices: randomly choose two bins and throw a ball into the bin with the least number of balls. With probability of at least $(1 - O(\log^2 n/n))$, it gives a maximum load of $(\ln n/\ln 2 + O(1))$.

However, in our settings, agents constantly remove tasks (balls) from the input queues (bins) and map them onto NoC, which is different from the balls-into-bins problem. In order to balance the load among agents, we need to measure the rate of task scheduling instead of the current number of tasks in agents. Therefore, in our implementation, we randomly pick two agents and calculate the rates of task scheduling $(N^\prime_{\text{task}} - N^i_{\text{task}})$, where $N^i_{\text{task}}$, $N^\prime_{\text{task}}$ represent the number of task measured last time and this time in the agent $i$, respectively. If the agent $i$ has the higher rate, we schedule the current task to it.

2) Distributed RL: RL is an area of machine learning, developed to find optimal solution in different areas including control theory. It is inspired by human trial-and-error experiments in order to learn the optimal policy under...
all circumstances. In RL, the agent takes actions and interacts with an environment in order to maximize cumulative results. The basic environment is modeled as a Markov decision process (MDP).

1) A set of environment states $S$ observed by the agent. In this paper, states represent the availability and type of PEs in NoC.

2) A set of actions $A$ of the agent. Actions, in this paper, represent mapping of tasks onto one type of PEs in task pools $T_P$. The probability of selecting an action under a state is called policy $\pi(s, a) = P(a_t = a|s_t = s)$.

3) The probability of a transition from the state $s$ to another state $s'$ under an action $a$: $P(st_{a+1} = s'|s_t = s, a_t = a) = P_a(s, s')$. Agents give a large negative reward to prevent the action of assigning a task to a busy PE. However, during exploration, these assignments are not forbidden. If the target PE to which a task is mapped is busy in serving the current request, we perform local search [3] by migrating the task to the nearby PE with a cost of network congestion and latency.

4) The immediate reward $r_a(s, s')$ after a transition from the state $s$ to another state $s'$ with action $a$.

The immediate reward is a technique to inform an agent whether the action $a$ from $s$ to $s'$ is perfect. We give high rewards when mapping tasks onto PE with the same type and penalize the mapping of tasks onto occupied PEs. Therefore, our reward function is formulated as follows:

$$r^a = \begin{cases} r(\text{type}_T, \text{type}_{\text{NoC}}), & \text{if } \text{busy}_{xy} = 0 \\ \theta l_{(\text{src, dest})} - b_{u_{\max}}^{\text{curr}}_{xy}, & \text{otherwise} \end{cases}$$

$$r(t_1, t_2) = \begin{cases} \text{negative reward, if } t_1 \text{ dependency exists} \\ \text{high reward, if } t_1 \text{ matches } t_2 \\ \text{low reward, if } t_1 \text{ executes in } t_2 \end{cases}$$

where $r^a$ returns a high reward (100) when type matches type$_{\text{NoC}}$, a low but still positive reward (10) when the task with type matches can be executed in the PE with type$_{\text{NoC}}$, and a negative reward proportional to communication costs if dependency exists. For example, it is tolerable to have a mapping of for-loops onto CPUs instead of GPUs (in this case, the reward is 10), whereas a mapping of for-loops onto HWAs is strictly forbidden. Otherwise, the task requires a certain bandwidth to be migrated to a nearby available PE. Therefore, it returns a large negative reward if the task cannot be executed, depending on the current bandwidth $b_{u_{\max}}^{\text{curr}}_{xy}$, the maximum bandwidth $b_{u_{\max}}^{\text{max}}_{xy}$, and the $l_1$ distance due to migration. If network congestion occurs, $b_{u_{\max}}^{\text{curr}}_{xy} \geq b_{u_{\max}}^{\text{max}}_{xy}$ and the reward is negative. busy$_{xy}$ equals 1 if the PE at location $(x, y)$ is occupied and 0 otherwise. If busy$_{xy}$ equals 0, then NoC communication has to be considered to reduce the number of hops for messages. Therefore, in (3), the existence of task dependencies indicates NoC communication. The amount of communication messages are relied on our graph partitioning algorithm. $\theta$ is a parameter to show how much we want to penalize the incorrect mappings. $l_{(p1, p2)} = |p1_x - p2_x| + |p1_y - p2_y|$ is the $l_1$ distance between two points p1 and p2.

The long-term reward starting from time $t$ is defined as $R_t = r^a + \gamma r^a_{t+1} + \cdots = \sum_{i=0}^{\infty} \gamma^i r^a_{t+i}$, where $r^a_t$ is an immediate reward after action $a$ at time $t$ and $\gamma$ is a discount factor to represent that the future reward decays at the factor of $\gamma$. If $\gamma = 1$, the future reward is as important as the current reward.

Q-learning is used to find an optimal policy which the agent follows in selecting actions for any MDP. $Q(s, a)$, the expected utility of taking an action $a$ in a state $s$, can be learned by following policies $\pi$ given the state. It is one of the most important algorithms in RL because it does not require building explicit representations of the transition $P_a(s, s')$ and the expected reward in MDP and it converges to the optimal solution of the Bellman equation which is formulated as

$$Q(s, a) = R(s, a) + \gamma \sum_{s'} P_a(s, s') \max_{a'} Q(s', a').$$

At times, the system may not be aware of the transition $P_a(s, s')$ and the expected reward $R(s, a)$. Therefore, it is of great importance to apply incremental updates with the current state $s$, the next state $s'$, the action $a$, and the immediate reward $r_a(s, s')$ to approximate (3) as

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

where $\alpha \in [0, 1]$ is the learning rate. Once $Q$ value is updated, we may follow (4) to get the optimal action $a^* = \arg\max_a Q(s, a)$. However, keeping choosing the optimal action $a^*$ for updated $Q$ value can cause overexploitation, which might lead to local optimum. Therefore, agents should keep exploitation and exploration balanced in order to find a globally optimal action. If agents only explore the environment, they just randomly try to perform actions without learning the optimal $Q$ value. If agents only exploit the environment based on the current Q-learning, they have a chance to be stuck in a local optimum without trying to explore the global optimum. In this paper, to strike a balance between exploitation and exploration, we implement an effective $\epsilon$-greedy action selection algorithm [36]. The agent with a small probability $\epsilon$ randomly picks a PE to which the task is mapped (line 8 in Algorithm 2) to explore the possible mapping. With high probability $1 - \epsilon$, the agent chooses a PE based on the $Q$ value. In our implementation, we set $\epsilon$ to be 0.1, which guarantees that agents keep trying new actions for each state (exploration) while following the optimal action according to the $Q$ values (exploitation).

In this paper, we present a distributed Q-learning algorithm where multiple agents interact with environment independently (line 2 in Algorithm 2) by mapping tasks assigned by the task schedulers onto suitable PEs. If one PE is occupied by another task, we perform local search by selecting the available PE nearby with the least number of flits (lines 15–17 in Algorithm 2). At the same time, all agents can get access to the shared $Q$ values and update them accordingly after receiving rewards from the environment (line 21 in Algorithm 2). However, since $Q$ values are shared among all agents, we should make sure that they are up-to-date when agents are about to modify $Q$ values. Therefore, $Q$ value updates should be protected by a critical section, which means that only one
Algorithm 2 Distributed Q-Learning of the Best Mapping

Input: Task queues in agents and the architectural graph \((N/oC)\)

1: Initialize \(Q\)
2: parallel for each agent \(i\) do
3: Initialize time \(t_i = 0\) and states \(s_{i,0}\)
4: repeat
5: \(\text{task} = \text{pop}(	ext{InputTaskQueue}[i])\)
6: /* e-greedy algorithm */
7: \(\text{rand} = \text{randnum}(0, 1)\)
8: if \(\text{rand} < e\) do randomly choose a PE
9: else choose a PE based on \(Q\) values
10: endif
11: /* local search */
12: if \((\text{busy} = 1)\)
13: search a nearby PE with the least number of flits
14: endif
15: \(s_{i,t_{i+1}} = \text{mapping(task, PE)}\)
16: calculate \(r\) using the equation (2)
17: critical section do
18: \(Q(s_{i,t_i}, a_{i,t_i}) = Q(s_{i,t_i}, a_{i,t_i}) + \alpha[r + \gamma \max_{a'}Q(s_{i,t_{i+1}}, a') - Q(s_{i,t_i}, a_{i,t_i})]\)
19: end
20: \(t_i = t_i + 1\)
21: until \(Q\) values converge
22: endfor

Fig. 5. Simulation flow of SOSPCS.

agent can get access at a time. Performance may suffer but it guarantees a better convergence of \(Q\) values.

VI. SIMULATION RESULTS

In this section, we evaluate the benefits of SOSPCS with a set of machine learning algorithms (see Table I) to demonstrate the performance improvement. These algorithms written without using Pthreads or OpenMP\(^1\) are processed by SOSPCS to obtain their IDGs. We compare our experimental results with state-of-the-art HETS scheduling [26] and METIS graph partitioning [18] approaches. The baseline for our results is to run applications with parallel execution using METIS in the simulator. We also show that SOSPCS is scalable to NoC-based heterogeneous systems with hundreds of cores.

Fig. 5 shows the simulation flow. We use Contech [31] to generate dynamic LLVM IR traces from target applications and apply SOSPCS to generate IR traces of tasks and PE indices to which these tasks are mapped. Trace-driven timing simulators MacSim [23] and BookSim2 [16] execute these instructions and return the next state and rewards back to SOSPCS to help agents optimize themselves. As shown in Table II, we model a heterogeneous platform consisting of 32 CPUs and 32 GPUs using MacSim plus special HWAs with NoC substrate using BookSim2. HWAs are specific in each domain, e.g., FFT and MM in signal processing or ReLU and neurons in NNs. To validate our software implementation of SOSPCS, we used the configuration parameters of Table II. Note that in addition to mesh, SOSPCS is applicable to other topologies as well. Also, as mentioned earlier, our RL-based scheduler can incorporate any scheduling algorithm such as priority-based schedulers; however, for simplicity, first come first serve was chosen to handle new applications.

A. Impact of \(\gamma\) on Community Count and Communication Overhead

The parameter \(\gamma\) in (1) controls the number of communities partitioned from \(IDG\). Small community count indicates low data communication overhead but the performance suffers as no more fine-grained parallelism exists. For simplicity, we set \(\gamma\) to 1. As demonstrated in Fig. 6(a), support vector machine (SVM), Naive Bayes (NB), and logistic regression (LR) have 106, 97, and 112 communities, respectively. However, \(\gamma = 1\) may not be the best solution as we can see in Fig. 6(b). We can increase \(\gamma\) to reduce the number of communities, leading to low data movement overhead. However, there is a diminishing return when the number of communities exceeds a threshold, which is different across different applications. Therefore, we employ “elbow method” to determine the best number of communities \(N\). We augment the objective function of SOSPCS in (1) with a data movement overhead [see Fig. 6(b)] to minimize communications cost. There are diminishing returns when an increase in the value of \(N\) leads to a decline in the additional quantity of data movement overhead. Mathematically, we need to find an elbow at \(N^*\) subject to \((df/dN)|_{N^*} - (df/dN)|_{N^{+}} > \xi\). When \(\xi\) becomes larger and larger, in the end, we will find the point which is called “elbow.”

B. Effects of the Number of Hidden Layers on Inference Time, Accuracy, and Application Speedup

Fig. 6(c) illustrates inference time and accuracy in terms of the number of HLs. With only three HLs, good accuracy

---

\(^1\)SOSPCS automatically parallelizes the sequential programs without the need to insert pragmas.

---

TABLE II

<table>
<thead>
<tr>
<th>Configuration Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
</tr>
<tr>
<td>Clock frequency</td>
</tr>
<tr>
<td>L1 private cache</td>
</tr>
<tr>
<td>L2 shared cache</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>Technology node</td>
</tr>
<tr>
<td><strong>GPU</strong></td>
</tr>
<tr>
<td>Clock frequency</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td><strong>HWA</strong></td>
</tr>
<tr>
<td>Types</td>
</tr>
<tr>
<td><strong>Network</strong></td>
</tr>
<tr>
<td>Topology</td>
</tr>
<tr>
<td>Routing algorithm</td>
</tr>
<tr>
<td>Flow control</td>
</tr>
</tbody>
</table>
Fig. 6. Performance evaluation of SOSPCS and comparison with state-of-the-art approaches.

(around 90%) in terms of the type of computations in each community while the inference time is tolerable (around 6.83 ms). We obtain these accuracy numbers from examining the feature index and coordinates of NNs are identical to the ground truth. Although accuracy gets better with more number of HLs, it takes much more time to inference whether the \( IDG \) contains certain patterns learned during training. Since performance is critical at runtime, we choose NNs with three HLs to classify the graphs. Accuracy of prediction of the task attributes influences the application speedup. Therefore, Fig. 6(d) illustrates the correlation between average application speedup and accuracy. With three HLs, average application speedup is 3.27\( \times \) while the most time-consuming NN in our experiments with six HLs could only achieve 3.35\( \times \). Therefore, in order to trade off inference time and application speedup, we decide to implement NNs with three HLs. However, task misclassification has to be handled properly as the accuracy of the NN pipeline is approximately 90%. If a PE cannot evaluate a given task (e.g., an FFT task mapped to an MM accelerator), the task is migrated to general-purpose CPUs to be executed. However, we allow a task to be executed to a PE if they are compatible (e.g., an MM task scheduled to GPUs). The accuracy of NNs has a significant impact on the task attribute and the ability to efficiently use heterogeneous resources.

C. Performance Comparison and Improvement

Fig. 6(e) shows the performance comparison among the baseline, HETS + METIS, and our approach with and without a machine learning strategy [41] for discovering the special patterns and mapping tasks. Our approach achieves the best performance improvement from 2.78\( \times \) to 4.12\( \times \) including the learning overhead for all the considered applications in Table I because our approach partitions the graph into hybrid tasks to exploit the underlying NoC-based heterogeneous PEs. For example, after we detect MM, maximum pooling, and NN inference in the convolutional neural net (CNN) application graph, these special tasks are mapped onto accelerators instead of CPUs, which could potentially improve the system performance. We also compare SOSPCS with the heuristic mapping to quantitatively see the improvements of SOSPCS. Most applications demonstrate at least 20% performance improvement over a heuristic mapping. We also note that occasionally tasks that are labeled as \( \text{par} \) are mapped onto CPUs. We believe it is mainly caused by the traffic congestion in NoC.

Fig. 6(f) shows the performance improvement of \( K \)-means (KM) clustering and backpropagation (BP) over time using SOSPCS. In the beginning, agents make some random actions (mapping of tasks) to explore the environment (NoC-based platform). It may not be the best action according to the speedup. However, when they make bad actions, the environment returns back rewards such that agents can change the next action based on these rewards and the states to learn how to improve the long-term rewards. When agents have explored enough states, they are capable to select the best actions for the current states based on the \( Q \) values as indicated by the plateau in Fig. 6(f).

D. Energy Comparison

For our energy consumption evaluation, MacSim develops power/energy model using McPAT [24]. Fig. 6(g) illustrates the energy reduction comparison among different approaches.
SOSPACS outweighs METIS by at least 2× energy reduction because SOSPACS tries to minimize data movement across different tasks and fully exploit the heterogeneous platform by identifying the specialized features existed in target applications. However, our approach has only 1.15× energy improvement over the same approach without machine learning. The reason may be that at the beginning of RL, agents need to explore the possible outcomes in the underlying heterogeneous platform to decide the best action. Therefore, it consumes some energy when a task is mapped to the wrong PE and migrates to the nearby PE suitable for the task. Nevertheless, our approach achieves the best performance and energy consumption reduction compared to the state-of-the-art approaches.

E. Scalability Analysis

Fig. 6(h) summarizes the scalability analysis when considering several NoC configuration sizes and varying the number of communities by modifying the γ value such that there are enough tasks to be mapped onto NoC. If the size of NoC is doubled, γ is reduced by half to generate much more tasks than the number of PEs, which could increase the amount of data transferred among PEs. Nevertheless, when the size of NoC changes from 4 × 4 to 16 × 16, the speedup increases drastically. For example, when experimenting KM, we set γ to be 0.2 for the 16 × 16 NoC, which could generate at least 300 tasks. Then SOSPACS tries to map these tasks onto NoC. The results show that the speedup varies from 2.4× to 10.23× when the size of NoC changes from 4 × 4 to 16 × 16.

VII. Conclusion

With CPUs, GPUs, and specialized HWAs coexisting on manycores, there is an increasing tension between programmability of CPUs and efficiency of accelerators. Therefore, we presented the SOSPACS framework that incorporates the necessary SDH and DSSoC capabilities to tackle issues such as performance inefficiency of general-purpose machines, inadequate programmability of domain-specific accelerators, and energy inefficiency of NoC by relying on NNs to classify task attributes and intelligent schedulers to manage the set of domain-specific tasks and enable reconfiguration of hardware components. First, we transform target applications from LLVM IR into IDGs where nodes represent LLVM IR instructions, edges represent dependencies among instructions, and edge weights represent the amount of data to be transferred between two nodes. Based on IDGs, we propose sliding window-based NN classifiers to detect existing patterns (MM/SGD/ReLU/parallel for-loop structures) in IDGs. NN classifiers are trained offline with representative IDGs for various special patterns. We then partition the rest of the graph into interconnected communities (tasks) with minimal data communication overhead to reduce energy consumption. This forms the task pool consisting of heterogeneous tasks: tasks are more suitable for either CPUs with sequential execution or GPUs with parallel execution, or accelerators with special patterns. Next, the task allocator distributes tasks to agents following a load balancing goal. Distributed intelligent schedulers map tasks to PEs and reconfigure the hardware platform as required. The environment returns back the rewards representing whether the mapping is perfect. Based on these values, agents can learn the best mapping of hybrid tasks. We conducted experiments on NoC-based heterogeneous PEs consisting of 32 CPUs, 32 GPUs, and HWAs such as MM and stochastic gradient descent in a machine learning domain. Results indicate that SOSPACS compared to state-of-the-art application scheduling algorithm provides performance and energy improvements as high as 4.12× and 3.24×, respectively. Future work will focus on further optimizations that will consider moving most of this framework analysis to hardware and minimizing the software requirements due to long delays.

REFERENCES


Yao Xiao is currently working toward the Ph.D. degree at the Ming Hsieh Department of Electrical and Computer Engineering, University of Southern California, Los Angeles, CA, USA. His current research interests include VLSI, parallel computer architecture and programming, Network-on-Chip (NoC), complex networks, and machine learning, modeling applications as complex networks, mapping communities detected from complex networks onto NoCs, and designing high-performance NoCs.

Shahin Nazarian (S’02–M’07) received the Ph.D. degree in electrical engineering from the USC Viterbi School of Engineering, Los Angeles, CA, USA. He was a Senior Research and Development Software Engineer at Magma Design Automation (now part of Synopsys), where he was involved in timing and noise solution development as part of Talus and Tekton tools. He has numerous industrial experiences, as a Technical Expert, Software and Hardware Design Engineer, in a wide range of areas including computer-aided design, architecture, and embedded system design. He is the Founder and President of Vervecode, Inc., and serves as a Consultant to several technical companies. He is currently an Associate Professor of Engineering Practice at the Ming Hsieh Electrical and Computer Engineering Department, USC Viterbi School of Engineering. His current research interests include machine learning, system timing and power optimization of VLSI circuits, and emerging technologies such as superconducting devices. Dr. Nazarian was a recipient of the USC Dean’s Award For Teaching Excellence and the USC Center for Excellence in Teaching Innovative Undergraduate Teaching Award.

Paul Bogdan (M’13–SM’18) received the Ph.D. degree from Carnegie Mellon University, Pittsburgh, PA, USA. He is currently an Assistant Professor at the Ming Hsieh Department of Electrical and Computer Engineering, University of Southern California, Los Angeles, CA, USA. His current research interests include performance analysis and design methodologies for manycore systems, theoretical foundations of cyber-physical systems, control of complex time-varying interdependent networks, modeling and analysis of biological systems and swarms, new control algorithms for dynamical systems exhibiting multifractal characteristics, modeling biological/molecular communication, fractal mean field games to model and analyze biological, social and technological system-of-systems. Dr. Bogdan was a recipient of a number of distinctions, including the 2017 Defense Advanced Research Projects Agency Young Faculty Award, the 2017 Okawa Foundation Award, the 2015 National Science Foundation CAREER Award, the 2013 Best Paper Award from the 18th Asia and South Pacific Design Automation Conference, the 2012 A.G. Jordan Award from Carnegie Mellon University for an outstanding Ph.D. thesis and service, the 2012 Best Paper Award from the Networks-on-Chip Symposium, the 2012 D.O. Pederson Best Paper Award from the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, the 2012 Best Paper Award from the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), and the 2009 Roberto Rocca Ph.D. Fellowship.