CEDR-API: Productive, Performant Programming of Domain-Specific Embedded Systems

Joshua Mack ECE, The University of Arizona Tucson, Arizona, USA jmack2545@arizona.edu Serhan Gener ECE, The University of Arizona Tucson, Arizona, USA gener@arizona.edu Sahil Hassan ECE, The University of Arizona Tucson, Arizona, USA sahilhassan@arizona.edu

H. Umut Suluhan ECE, The University of Arizona Tucson, Arizona, USA suluhan@arizona.edu

Abstract-As the computing landscape evolves, system de-

Ali Akoglu ECE, The University of Arizona Tucson, Arizona, USA akoglu@arizona.edu

signers continue to explore design methodologies that leverage increased levels of heterogeneity to push performance within limited size, weight, power, and cost budgets. One such methodology is to build Domain-Specific System on Chips (DSSoCs) that promise increased productivity through narrowed scope of their target application domain. In previous works, we have proposed CEDR, an open source, unified compilation and runtime framework for DSSoC architectures that allows applications, scheduling heuristics, and accelerators to be co-designed in a cohesive manner that maximizes system performance. In this work, we present changes to the application development workflow that enable a more productive and expressive API-based programming methodology. These changes allow for more rapid integration of new applications without sacrificing application performance. Towards the design of heterogeneous SoCs with rich set of accelerators, in this study we experimentally study the impact of increase in workload complexity and growth in the pool of compute resources on execution time of dynamically arriving workloads composed of real-life applications executed over architectures emulated on Xilinx ZCU102 MPSoC and Nvidia Jetson AGX Xavier. We expand CEDR into the application domain of autonomous vehicles, and we find that API-based CEDR achieves a runtime overhead reduction of 19.5% with respect to the original CEDR.

Index Terms—heterogeneous programming models, runtime systems, resource management

I. INTRODUCTION

As technology scaling reaches its limits, system designers are exploring an increasingly diverse range of methodologies for building systems that can maximize their compute performance within limited size, weight, power, and cost budgets. One such methodology in the literature is the design and fabrication of Domain-Specific System on Chip (DSSoC) devices. The motivation of such devices is fairly simple: in a general-purpose computing context, heterogeneous computing systems are difficult to program and utilize effectively. The hypothesis with DSSoC devices, then, is that by restricting the applications used on a given system to a particular domain, it becomes more feasible to build productive software and programming abstractions for the finalized hardware.

We believe that such software and programming abstractions have a number of key requirements. These abstractions should support scenarios where multiple users can coexist and interleave their applications across the DSSoC's heterogeneous pool of processing elements (PEs) while utilizing compute resources effectively in a dynamic way. Many of the predominant heterogeneous compute frameworks such as CUDA [1], OpenCL [2], or SYCL [3] assume an environment where an application expert performs offline analysis across a number of possible implementations for a given application, determines the optimal static mapping for all computational kernels in that application, and produces a fixed binary that represents a single, expertly-tuned instance of the application. In an environment of widespread heterogeneous computation, these greedy, inflexible mappings ignore runtime resource contention and will clearly lead to system inefficiencies when such programs are required to share system resources with an arbitrary number of other heterogeneous applications. One might expect that resource contention issues would be solved by the operating system, but Roscoe argues [4] that the rate of architectural advancement for modern SoCs has tremendously outpaced the work in operating systems to meet them. Hence, to handle this, we believe that DSSoC software abstractions must be coupled with intelligent, intermediate runtime systems that are capable of arbitrating or scheduling requests from all applications to the PEs across the DSSoC. To enable this runtime to effectively arbitrate resources among applications, each application must also be capable of mapping its computational kernels to as wide of a number of the system's heterogeneous processing elements as possible. Namely, the programming

This material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7860. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusion contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defence Advanced Research Projects Agency (DARPA) or the U.S. Government.

abstractions provided to the user should be agnostic from the underlying hardware as this provides a number of key benefits: it allows for maximum flexibility when the runtime is performing its scheduling, and it allows the user to program for the DSSoC in a productive manner without needing to become a hardware expert in the process.

In prior work, we have explored the development of such a runtime through a framework called CEDR, a Compiler-Integrated, Extensible, DSSoC Runtime [5], [6]. CEDR has been designed to holistically target the joint challenges of choosing the composition and capabilities of accelerators on a DSSoC, determining the optimal scheduling heuristic for an application domain, and providing a productive set of software abstractions for programmers writing applications for DSSoCs. The primary contribution of CEDR thus far is its status as an open source ecosystem ¹ that integrates compile-time application analysis with a Linux userspace-based runtime system that addresses the aforementioned challenges. The programming model for CEDR relies on a Directed Acyclic Graph (DAG)-based program representation where each node in the graph represents schedulable unit of computation, and edges represent temporal dependencies. While this representation has proven itself to be a sufficiently powerful format to meet the aforementioned challenges with clear support for notions of concurrency and heterogeneity, it does have a number of limitations with regards to programmer productivity. First, despite a large amount of progress in compilation tooling, it is still comparatively difficult for many programmers to express applications in performant, DAG-based representations. Additionally, a DAG-based application format cannot accurately capture the control flow structures of many programs in a way that preserves scheduler flexibility without a mechanism to enable conditional DAG dependencies. In this work, we seek to address these limitations through the integration of a simplified API-based programming model that better aligns with traditional practices in programming for homogeneous computing systems. By incorporating both blocking and nonblocking APIs, we are able to make this change without impacting the ability to write performant applications and thus the utility that CEDR provides for hardware designers or scheduling heuristic developers seeking to stress their systems. The primary contributions of this work are as follows:

- We modify the programming model of CEDR to allow for a more productive API-based application workflow.
- We integrate non-blocking APIs that allow for performance programmers to maximally exploit opportunities for parallelism in their code.
- We expand CEDR to the domain of autonomous vehicles and incorporate lane detection.
- We perform hardware experiments with increasing heterogeneity and expose scenarios in which parallelism of heterogeneous accelerators is limited by the availability of homogeneous CPU cores.

¹Available at: https://github.com/UA-RCL/CEDR



Fig. 1: An overview of CEDR's integrated compiler and runtime system.

The rest of the paper is organized as follows: in Section II, we explore the details of the modified API-based CEDR. In Section III, we detail the workload and system configurations that will be leveraged in our experimental analysis. In Section IV, we analyze our modified CEDR runtime across a variety of workload and system configurations. In Section V, we include an updated analysis of CEDR's standing within the literature. Finally, in Section VI, we summarize our findings and conclude with avenues for future work.

II. CEDR-API

A. Background

For completeness, we begin with an overview of how the existing CEDR runtime operates. Shown in Fig. 1, CEDR is broadly composed of two components: a compilation workflow and a runtime workflow. The compilation workflow is used to transform user applications into applications that CEDR can execute. For details about this process in the baseline runtime. we refer readers to Section 2.2 of Mack et al. [6]. At the end of this process, a CEDR Application is produced that consists of a shared object binary and a JSON-based DAG. The shared object contains a number of functions representing the computational nodes within the DAG, but it does not contain application-level control flow information. Instead, the JSON captures temporal dependencies between nodes and high level control flow of the user's application. This pair of objects is submitted to the runtime, referred to as the "CEDR Daemon Process" in Fig. 1 via inter-process communication (IPC).

The CEDR Runtime consists of a few key components: a number of worker threads, a ready queue of tasks, and a main event loop that receives, parses, launches, and manages applications. Each PE in the system – accelerator or CPU core – is paired with a worker thread that manages executing tasks on that compute resource. In the case of CPU cores, each worker thread is assigned via its processor affinity to run on the corresponding resource. However, for accelerator cores, their respective worker thread is assigned via its affinity to some CPU core in the system, and that CPU core is responsible for coordinating any configuration updates or data transfers that accelerator requires. Within the main event loop, CEDR periodically pushes work to these threads by scheduling tasks to them from its ready queue of tasks according to a user-defined heuristic as part of the program's *Runtime* Configuration input. The Runtime Configuration also allows the user to enable or disable features such as PAPI-based performance counters [7]. As tasks are completed, the worker threads signal completion back to the main thread and JSON DAG dependencies of those tasks are then pushed to the back of the ready queue. To support heterogeneous execution, when a task is scheduled to a given resource, CEDR dynamically updates that task's function pointer such that its worker thread invokes a function that is compatible with that resource. As applications are submitted over the IPC channel, they are parsed and their executions are started by placing the head nodes of their DAGs into this same ready queue. This process continues indefinitely until an IPC command is received that tells CEDR to shutdown at which point it serializes all the logs it has collected relating to task execution, performance counter measurements, and so on for later offline analysis by the user.



Fig. 2: An example of an application structure that cannot be represented in the previous DAG-based representation without compressing it to a single node and losing the ability to schedule each kernel independently.

B. Limitations of the Existing Runtime

To motivate the contributions of this work, we first begin by discussing limitations in the types of applications that can be represented with a DAG-based format in the previous release of CEDR. Due to the acyclic nature of DAG-based applications, this format is unable to represent control flow concepts of iteration or conditional branching. This leads to issues when trying to schedule applications that have structures like that shown in the left half of Fig. 2. Kernel₁, Kernel₂, and Kernel₃ may all be individually compatible with accelerators on the system, but because a DAG-based program representation cannot allow for a sufficiently granular program representation like that shown in the right half of Fig. 2, this entire for-loop structure must be collapsed to a single DAG node and presented to CEDR as a single unit to be scheduled. Because it is unlikely that an accelerator exists on the system that can handle this specific sequence of iterated kernels, this single node is likely to only be supported on the CPU, and benefits of acceleration in this application are reduced.

C. Contributions of this Work

To address these limitations, in this work, we adjust CEDR's frontend tooling to consist of a new API-based development

workflow shown in Fig. 3. One of the core developments is a new libCEDR library and associated cedr.h. The structure of this library is shown in the left side of Fig. 3. APIs for use in application code are exposed to developers through the cedr.h header file. This header contains high level kernel declarations that do not contain any implementation details of the underlying operation with samples provided in Listing 1. As different DSSoC platforms develop different accelerator implementations of these kernels, they are incorporated through libCEDR Modules. As an example, for a platform with a Fast Fourier Transform (FFT) accelerator, libCEDR provides an fft module. This module is then responsible for providing physical implementations of the high-level APIs as desired. Finally, the platform.h header file provides global information about the platform in use such as base addresses for accelerators' AXI4 [8] interfaces to enable driverless memory-mapped I/O (MMIO) control and dispatch of tasks. It is expected that all APIs in this library provide, at a minimum, standard C/C++ implementations that can be leveraged across all the platforms that CEDR supports. As such, at compilation time, the user configures platform.h, chooses which libCEDR Modules to enable, and receives two outputs: a static libcedr.a archive containing only the CPU C/C++ implementations of all implemented APIs and a "runtime" libcedr-rt.so shared object containing both the CPU C/C++ implementations and all chosen accelerator implementations through their respective *libCEDR Modules*.

Listing 1: Sample libCEDR API Declarations

Once libCEDR is compiled, a user moves through the workflow shown in the right side of Fig. 3. One key benefit of this compilation approach is the way that it enables rapid application bring up and evaluation prior to testing in complex heterogeneous computing scenarios. In early stages of development integration, a user can begin by treating libCEDR like any other CPU-based library. Once they are convinced that their application is functionally correct, they can compile for CEDR-API by simply building as a shared object that avoids linking in implementations for their libCEDR API calls. This shared object application is then provided to the runtime using the same IPC submission flow shown in Fig. 1.

During startup, the CEDR Runtime is provided with the corresponding libCEDR-rt shared object containing all of the system's API implementations, and it builds a mapping from each API and resource type pairing to a physical implementation of that API on that resource if one exists. When an application is received by the CEDR Runtime's main event loop, the shared object is parsed and a new system thread is spawned that executes that application's



Fig. 3: Overview of the libCEDR and CEDR-API compilation methodology

main function. As this main function executes, it periodically encounters libCEDR API calls. These calls are linked during binary parsing against implementations in libCEDR-rt that themselves call an enqueue_kernel function inside the CEDR runtime. When enqueue_kernel is invoked, a task is placed into CEDR's ready queue, and from there, all of CEDR's existing heuristics are able to process it in the same fashion as the DAG-based methodology. As this process is multi-threaded in nature – involving the user application's thread, the main CEDR thread, and the eventual worker thread of the executing resource – there is a need for a synchronization methodology that the user application can rely on to acknowledge completion of the underlying API call.



Fig. 4: Synchronization methodology dispatching heterogeneous kernels from libCEDR to CEDR

An overview of this synchronization methodology is shown in Fig. 4. Before pushing the requested task to the ready queue, the user's application thread initializes a set of pthread_cond and pthread_mutex variables to use to receive updates on the progress of its task. After dispatching the new task with CEDR, the thread then reaches a pthread_cond_wait barrier and goes to sleep. As the task propagates through to the scheduler and eventual API implementation, the corresponding worker thread signals task completion back to the application thread via pthread_cond_signal. After this, the application thread wakes and resumes its computation.

While this methodology ensures that we preserve functional correctness relative to the baseline application's singlethreaded execution, it does leave performance to be desired for high-performance users as it does not allow them to parallelize execution of kernels. To support these needs, we have also developed non-blocking variations of all APIs that are present in libCEDR. These APIs allow the end user to have full control over the task synchronization primitives such that they can manually maximize parallelism in the underlying application. It has been designed such that it is still compatible with the standalone validation to shared object compilation workflow shown in Fig. 3. As we will see in Section IV-A, these nonblocking APIs allow users to extract equivalent performance to the DAG-based methodology without sacrificing productivity improvements and support for richer program representations.

III. EXPERIMENTAL SETUP

For our experiments, we use the Xilinx Zyng Ultrascale+ ZCU102 [9] and NVIDIA Jetson AGX Xavier [10] development boards. We use three representative real-world applications for our evaluations covering radar processing, communications system, and autonomous vehicle domains with Pulse Doppler (PD), WiFi TX (TX), and Lane Detection (LD). Pulse Doppler calculates velocity of an object, by measuring distance of the object using 256-point FFTs, and measuring the frequency shift between transmitted and emitted signals. WiFi TX generates packets of 64 bits and prepares for transmission over an arbitrary channel through scrambler, encoder, modulation, and forward error correction processes. WiFi TX relies on 128-point inverse FFT for each packet transmitted. Lane Detection is a convolution intensive routine from autonomous vehicles domain. In the literature, it has been shown that implementing convolution in the frequency domain rather than the spatial domain through a combination of FFT and pointwise product (ZIP) operations can reduce algorithmic complexity and inference time [11].

A workload composed of these three applications allow us to evaluate various scenarios where a heterogeneous SoC is shared by multiple applications in an interleaved manner. An example scenario could involve Lane Detection running as a continuous process where Pulse Doppler and WiFi TX applications arrive dynamically and are executed periodically. Such scenarios allow us to study the relationship between degree of SoC heterogeneity, scheduling overhead and quality of schedules achieved by various heuristics targeting autonomous vehicles domain. The Lane Detection application stresses the FFT accelerator on the emulated heterogeneous SoC with number of 1024-point FFTs and IFFTs scaling to 16384 and 8192 instances respectively for a 960x540 image. WiFi TX and Pulse Doppler are lower latency applications with number of FFTs scaling to 100 and 512 respectively. Driven by these three applications, we use FFT and ZIP as key functions that are supported with accelerator based execution. Each application is implemented via the hardware agnostic API calls for each function.

We compile all three applications using the CEDR compilation toolchain described in Section II and prepare binaries to be executed on heterogeneous SoC configurations that are emulated on both ZCU102 and Jetson development boards. The ZCU102 is formed of 4 ARM cores running at 1.2 GHz and programmable FPGA fabric where we invoke FFT accelerators running at 300 MHz. The FFT accelerator is implemented using Xilinx FFT IP supporting up to 2048-point FFTs. The FFT accelerator uses direct memory access (DMA) to manage data transfers between ARM cores and accelerator through AXI4-Stream [8]. The Jetson board is formed of 8 ARM cores running at 2.3 GHz and a Volta GPU running at 1.3 GHz, where we implement FFT and ZIP accelerators as CUDA kernels. The data transfers between ARM cores and the accelerators are handled with standard cudaMemcpy functions using the PCIe interface.

We compose heterogeneous SoCs by varying the number and types of processing elements on the ZCU102 platform from the pool of 3 ARM cores along with 8 FFT accelerators. We utilize the Jetson platform to demonstrate the portability of our compile and runtime system and conduct cross-platform comparisons in terms of factors that contribute to the runtime and scheduling overhead. The amount of data processed by an application is considered a frame, measured in Megabits (Mb). Injection rate is defined as the rate at which frame instances are generated per second and measured in Mbps. We use 29 injection rates between 10 and 2000 Mbps, where each injection rate defines a periodic rate of job along with its associated input data arrival for the given workload. We use Round Robin (RR), Earliest Finish Time (EFT), Earliest Task First (ETF), and the runtime variant of the Heterogeneous Earliest Finish Time (HEFT_{RT} [12]) scheduling heuristics executed along with the CEDR management thread using one of the ARM cores on the target SoC.

We use metrics of average scheduling overhead per application and average execution time per application for performance evaluation. The scheduling overhead captures the time spent by the runtime in making scheduling decisions. This time is proportional to the number of scheduling rounds made by the runtime as well as the complexity of the scheduling algorithm. The application execution time is the time difference between the beginning and completion of an application's execution, including the overhead of all scheduling decisions in between. Lower execution times indicate the scheduler's capability to manage the workload efficiently. To make these two metrics comparable across different runtime configura-



Fig. 5: Runtime overhead in API and DAG-based CEDR.

tions, we normalize them with respect to the number of applications and take their average over 25 trials to reduce the effect of noise. For brevity, throughout the article, we will take each metric (for instance "execution time") to refer to its corresponding averaged-per-application version ("average execution time/application").

IV. EXPERIMENTS

A. Runtime and Scheduling Overhead Analysis

We present the runtime overhead of API-based CEDR with respect to the DAG-based CEDR as illustrated in Fig. 5. We define the runtime overhead as the overall time spent by CEDR to receive, manage, and terminate all applications in a given workload. This overhead excludes the overhead of task scheduling, as we define it separately as scheduling overhead. For this experiment, we use 5 instances for each of the Pulse Doppler and WiFi TX applications as a workload and collect runtime overhead across the sweeping range of the injection rate on the ZCU102 platform with 3 ARM CPUs and 1 FFT accelerator. The X-axis shows the injection rates, and Yaxis shows the runtime overhead. As the injection rate grows, the runtime overhead reduces and then saturates at around injection rate of 200 Mbps for both API and DAG-based CEDR. The applications arrive in an increasingly overlapping manner to the runtime with the increase in injection rate and, in turn, ready queue size grows. This gives runtime the opportunity to manage multiple tasks concurrently rather than serially, which in turn enables the runtime to complete processing same number of applications in a shorter span of time, thereby reducing the runtime overhead. The saturation of the trend lines indicate that beyond a certain injection rate, the runtime becomes oversubscribed, where all the applications within the workload get executed by CEDR with maximum concurrency. We further observe from this plot that, throughout the saturated region, the API-based CEDR achieves a 19.52% reduction on average in runtime overhead with respect to the DAG-based CEDR. This reduction can be attributed to the simplification of runtime steps in API-based CEDR compared to DAG-based CEDR. For the DAG-based CEDR, the runtime overhead involves time required for receiving and parsing application DAG files via IPC to construct application DAG, parsing shared object, pushing tasks to the ready queue, popping completed tasks from the queue, and finally terminating



Fig. 6: Execution time with respect to injection rate for different schedulers with 3 CPUs, 1 FFT, and 1 MMULT.

the completed applications. For the API-based CEDR, two factors contribute to the reduced overhead. First, API-based CEDR does not need to parse DAG files when applications are submitted via IPC. Second, pushing tasks to the ready queue is eliminated as it is handled by the application thread.

Next we present an experiment aiming to validate the application execution time and scheduling overhead trends in APIbased CEDR against DAG-based performance trends presented in [6]. For this experiment, we borrow key parameters such as hardware configuration, workload composition, and scheduling heuristics from the DAG-based CEDR work, for fairness of comparison. The hardware is composed of 3 ARM CPUs, 1 FFT, and 1 MMULT accelerators on the ZCU102 platform. We use the same workload as utilized in [6] that consists of WiFi TX and Pulse Doppler applications with 5 instances each following the experimental procedure described in Section III. Fig. 6 (a) and (b) show average execution time per application for the DAG and API-based execution on CEDR respectively, with respect to injection rate, where individual line plots represent execution using different schedulers. Both Figures 6 (a) and (b) show similar saturation trends as injection rate increases where system becomes oversubscribed at around 200 Mbps. Furthermore, the mean magnitude of the saturated region of API-based execution deviates by 32% compared to the one of DAG-based execution. From the scheduler perspective, we notice ETF scheduler resulting in a significantly higher execution time in both plots while remaining schedulers perform similar to each other in each setup. We note that the ETF scheduler in the oversubscribed region shows average execution time as 700 ms in the DAG-based CEDR, while



Fig. 7: Scheduling overhead with respect to injection rate for different schedulers with 3 CPUs, 1 FFT, and 1 MMULT.

we observe 425 ms with the API-based CEDR. We attribute this execution time reduction for ETF to the ready queue size being smaller, as API-based CEDR only schedules libCEDR API calls/portions of the application, which have support for heterogeneous execution. In DAG-based CEDR, the whole application, including non-accelerated regions, is divided into tasks that are scheduled by CEDR scheduler.

We show the scheduling overhead with respect to injection rate and different schedulers in Figures 7 (a) and (b) for DAG and API-based CEDR executions respectively. In both plots we notice that with the exception of the ETF scheduler, the scheduling overhead is stable for the remaining schedulers across the injection rates with very close overhead values. The ETF scheduler however shows remarkably different trend in the API-based CEDR, where the scheduling overhead reduces to around 1.15 ms in the saturated region, from a scale of around 70 ms in DAG-based CEDR. This reduction is due to fewer number of tasks that need scheduling in the API-based CEDR. This further demonstrates that the ETF's execution overhead is more sensitive to the ready queue size than the remaining schedulers.

Referring back to Fig. 6 (a) and (b), while ETF is observing a reduction in average execution time with the APIbased CEDR, the remaining schedulers observe an increase in execution time from around 200 ms on the DAG-based CEDR to around 350 ms on the API-based CEDR in the oversubscribed region. This is primarily due to the way the worker and application threads are managed in API-based CEDR compared to DAG-based CEDR. In DAG-based CEDR, the whole application code is executed on the worker threads



Fig. 8: Execution time over injection rates for different schedulers on Jetson AGX Xavier with 3 CPUs and 1 GPU.

as DAG task nodes, hence the available CPU cores are only shared among worker threads. In API-CEDR however, both application and worker threads are launched on the available CPU resources, where only the worker threads execute the application portion with heterogeneity support. For the presented experiment on ZCU102 with 3 CPU cores, DAGbased CEDR spawns 4 worker threads while API-based CEDR launches an additional 10 application threads (5 instances of each application), leading to increased thread contention on the underlying CPUs.

We perform the same experiment on the Jetson with a configuration of 3 CPU cores and 1 GPU as shown in Fig. 8. With the availability of a total of 7 CPU cores, the 4 worker threads (3 CPU and 1 GPU) and 10 application threads have more resources to share between them. This reduces the thread contention compared to the ZCU102. Compared to DAG-based CEDR in Fig. 8(a) which spawns only 4 worker threads to execute the workload while under utilizing the available CPU cores, API-based CEDR better exploits the available resources through concurrent execution of worker and application threads, resulting in reduced execution time in Fig. 8 (b). We further study the thread contention in Section IV-C to better understand the root cause of this behavior.

For the case of the ETF scheduler on the ZCU102, we notice execution time reduction, because as shown in Fig. 7 (a) and (b), ETF is most sensitive to the heterogeneity with the highest scheduling overhead. The benefit of reduced ready queue size due to API-based execution results with reduction in scheduling time that is larger in magnitude than the increase in execution time due to thread contention.



Fig. 9: Execution time with respect to injection rate in API-CEDR using Lane Detection, Pulse Doppler and WiFi TX.

B. Versatility

In this section we expand our experimental evaluations to demonstrate the versatility of the CEDR framework by introducing Lane Detection as a new application to the workload, increasing the number of FFT accelerators on the ZCU102 to 8, and performing execution time performance analysis with respect to changes in injection rate using the same workload on the Jetson platform. As discussed in Section III, Lane Detection has a large number of FFT instances, and we expect it will stress both the runtime system and the schedulers as the ready queue size is expected to grow substantially. The autonomous vehicle workload includes a single instance of Lane Detection as a long latency job while lower latency WiFi TX and Pulse Doppler applications arrive dynamically. Fig. 9 (a) and (b) present the execution time trends of this workload with respect to injection rate, on the ZCU102 and Jetson platforms respectively.

In both execution scenarios, across all schedulers we notice similar saturation trends as observed in Fig. 6. However, even though number of accelerators has increased from one to eight FFTs, the runtime now approaches the saturation point earlier at 100 Mbps on the ZCU102 based emulation indicating the increase in the complexity of the workload with the inclusion of the LD that stresses both the runtime system and the schedulers. In Fig. 9(b), we observe that for all schedulers the execution time performance shows saturation after 500 Mbps, showing the Jetson platform's ability to cope with the workload better than the ZCU102. While in the best case, during the saturated region ZCU102 platform achieves execution time of 2000 ms, the Jetson based execution is in the range of 600 to 700 ms.



(b) Jetson 1-7 CPU, 1 GPU; 500 Mbps injection rate

Fig. 10: Average execution time per application with respect to varying PE configurations.

In both Fig. 9 (a) and (b), RR performs worse than the other schedulers. The performance of RR degrades relative to its performance shown in Fig. 6 (b) where the hardware composition is 3 CPUs and 1 FFT only. RR being a fair scheduler is not able to exploit the increased resource pool or heterogeneity. Whereas the sophisticated and heterogeneity aware schedulers ETF and HEFT_{RT} have more opportunity to make better decisions, resulting in lower execution time.

The comparative scheduler performance on the ZCU102 and Jetson platforms presented in Fig. 9 (a) and (b) show different trends. The performance gap between schedulers on the Jetson is much smaller than the gap we observe on the ZCU102. Similar to the conclusion presented in Section IV-A, we believe this is due to the Jetson platform offering richer set of CPU cores that makes it less prone to the thread contention. In the following section we will perform scalability analysis to expose this issue.

C. Scalability

In this section, we study the impact of increasing the number of processing elements on the application execution performance in CEDR. Fig. 10 (a) and (b) show the application execution time of the autonomous vehicle workload on the ZCU102 and Jetson platforms respectively. We fix the workload injection rate to 300 Mbps and 500 Mbps when executing applications on the ZCU102 and Jetson platforms respectively, which are the points where these two systems are in oversubscribed state as observed in Section IV-B. The X and Y-axes of both Fig. 10 (a) and (b) present the resource pool and corresponding average execution times respectively.

On the ZCU102 platform in Fig. 10 (a), the resource pool configurations consist of a fixed CPU count of 3 as there are a maximum of 3 ARM cores on the platform, and a varying FFT accelerator count from 0 to 8. For Jetson in Fig. 10 (b), we keep the GPU count fixed at 1 as there is one GPU on the platform, and vary the CPU count between 1 and 7, as there are 7 CPU cores available. One CPU core on both ZCU102 and Jetson is reserved for the execution of CEDR runtime.

The execution scenario on ZCU102 in Fig. 10 (a) shows that the least execution time is observed with 3 CPUs and no FFT accelerators, across all schedulers. Increasing the FFT accelerator count rapidly increases the execution time. This increase can be explained by the sharing of limited CPU cores by larger number of threads. The 3 CPU cores each accommodate the 3 CPU worker threads, the launched application non-kernel threads, as well as the accelerator management threads for the FFTs. As illustrated in Fig. 1, with the increasing PE count (e.g., FFT), the number of worker threads sharing each CPU core increases, leading to each thread waiting for longer periods to get access to the CPU core. This, in turn, increases the application execution time. We further notice a gap in the application execution times of different schedulers as FFT count scales up. The RR scheduler performs worse than other schedulers, as it tries to use all of the PEs equally. This maximizes the number of worker threads competing for the scarce CPU resources, creating contention and in turn results with poor execution time performance. The EFT scheduler performs better than RR, as it doesn't force the uniform use of all PEs, rather it focuses on assigning tasks to a subset of PEs that can finish the tasks earliest. The sophisticated heterogeneity aware schedulers ETF and HEFT_{RT} further reduce the execution time compared to EFT. ETF not only attempts to find the most optimal task-to-PE mapping during scheduling, but also tries to find the most optimal task to schedule first. This enables ETF to better choose the subset of available PEs that minimize the execution time. We find that, in this experiment, HEFT_{RT} narrowly achieves the best application execution time performance.

We observe an interesting trend in execution time for all the schedulers on the Jetson platform as we increase the number of CPU cores as shown in Fig. 10 (b). Here, as the CPU worker thread count increases from one to five, the execution time across all schedulers show a downward trend. This reduction is achieved due to the fact that the Jetson has 7 CPU cores available, out of which one is dedicated for GPU management, and remaining 6 CPU cores can accommodate up to 6 CPU worker threads without needing any processor sharing between multiple worker threads. Therefore increasing CPU thread count introduces opportunity for concurrent execution and reduces execution time. However, this trend is not linear downward, rather it is polynomial with a minimum execution time observed at 5 CPU and 1 GPU threads. This trend is caused by the fact that CEDR-API launches the application non-kernel threads on all 7 CPU cores regardless of the number of worker threads. As the number of worker threads increases from 1 CPU to 5 CPUs, more parallel worker threads get introduced, and sharing of the CPU resources between worker and application threads also increases. This causes the downward execution time to be polynomial rather than linear. Beyond the 5 CPU 1 GPU configuration, addition of worker threads increases CPU resource sharing between application and worker threads to the point where the overhead of thread sharing diminishes the execution time reduction. Therefore, the execution time experiences polynomial increase.

In our scheduling overhead analysis with respect to increase in number of FFTs on the ZCU102 and number of CPU cores on the Jetson platforms, we observe negligible overhead at a scale of 0.1% and 0.5% respectively for each hardware configuration relative to the application execution time. This supports our earlier analysis, that the increase in execution time with increasing resource count is primarily caused by the worker thread contention.

V. RELATED WORK

Many DSSoC design efforts start with simulation-based modeling in both high level simulation [13]–[15] and cycle accurate simulation [16]–[18]. In early design space exploration (DSE) scenarios, many of these options are highly effective at narrowing down the scope of designs that are worth exploring on hardware. Compared to CEDR, these works are complementary as the designs that are narrowed down via early DSE can then be modeled on commercial FPGA platforms and evaluated to a greater extent with CEDR in order to collect ground-truth hardware measurements that feed back into future cycles of chip design.

Focusing on application runtimes, we can segment the literature by works that target accelerator-rich heterogeneous platforms and those that do not. Prioritizing discussion of those targeting accelerator-rich heterogeneous platforms, there are many works of note. Bolchini et al. [19] propose a runtime controller for OpenCL-based applications on heterogeneous platforms. It offers many notable features including the ability to perform cluster-level mapping of tasks on Linux and monitor power or execution metrics. However, the authors do not discuss the ability to launch simultaneous applications or adjust their scheduling policy. Picos++ [20] proposes a hardware-based runtime that provides support for applications written with OmpSs or OpenMP. These applications are mapped with Nanos++ API calls [21] using the Mercurium Source-to-Source compiler. The current capabilities and status of this compiler is described in [22]. While the compilation tooling in this ecosystem is substantial, due to the hardwarebased design of their runtime, they are unable to support interchangeable and platform-independent scheduling policies. Kumar et al. [23] present DELTA, an approach for dynamically scheduling Function-as-a-Service (FaaS) computations to heterogeneous, network-connected computing resources. The adaptive, resilient nature of their scheduling heuristics are notable, but distinct from CEDR, their definition of heterogeneity primarily focuses networked collections of CPUonly or CPU-GPU systems. A large unsolved problem in that area would seem to be development of a FaaS-based

methodology for dispatching work to function-specific accelerators like FFTs. Hseih et al. have introduced SURF [24], a runtime built to allow efficient execution on heterogeneous SoCs. It follows a similar API-based programming approach to that discussed here, and each API can have a number of implementations on heterogeneous resources. However, their approach would appear to only support linear chains of kernels and lacks support for parallel dispatch through a non-blocking execution methodology. Pinto et al. present StarVZ [25], a performance evaluation framework built on StarPU [26], a task-based runtime that supports heterogeneous execution of OpenCL or CUDA-based accelerators. Vasiliadis, Tsirbas, and Ioannidis [27] propose a device-agnostic, adaptive scheduling approach for scheduling machine learning kernels on heterogeneous architectures. Kim et al. [28] present IRIS, a heterogeneous runtime system that incorporates a unique set of resource discovery, adaptive scheduling, data movement, and programming model capabilities.

While a comprehensive review of this domain is infeasible given page constraints, we believe that CEDR provides a unique set of functionality through its open source accessibility; its status as a highly portable, configurable runtime; and its ability to support rapid DSE for DSSoCs through support for arbitrary workload injection, scheduler integration, PAPI-level performance counters, and kernel-specific accelerators.

VI. CONCLUSION

In this work, we present CEDR-API, a new programming methodology for the CEDR framework that enables more productive programming of domain-specific architectures while supporting concurrent execution of heterogeneous kernels. We have expanded CEDR-API to the domain of autonomous vehicles, and we have conducted a number of experiments that explore the overhead, versatility, and scalability characteristics of this modified runtime. We have found that, while runtime overhead is reduced, as systems evolve into highly heterogeneous architectures with large numbers of distinct processing elements, runtime systems should be designed in ways that allow them to cope with that growth in heterogeneity.

One promising path to address the barrier of CPU availability is to leverage progress in big.LITTLE architectures and exchange a fraction of the heavyweight CPUs with a larger quantity of lightweight CPUs specialized for worker thread management. In future work, we will explore the development of such optimized, core-rich architectures to enable maximal parallelism across diverse configurations of heterogeneous accelerators while minimizing the energy and latency that such configurations otherwise introduce.

REFERENCES

 J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for?" *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008, ISSN: 1542-7730. DOI: 10.1145/1365490.1365500.

- [2] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66– 73, 2010. DOI: 10.1109/MCSE.2010.69.
- [3] R. Reyes, G. Brown, R. Burns, and M. Wong, "SYCL 2020: More than meets the eye," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '20, Munich, Germany: Association for Computing Machinery, 2020, ISBN: 9781450375313. DOI: 10.1145/3388333.3388649.
- [4] T. Roscoe, "It's time for operating systems to rediscover hardware," USENIX Association, Jul. 2021.
- [5] J. Mack, N. Kumbhare, A. NK, U. Y. Ogras, and A. Akoglu, "User-space emulation framework for domain-specific SoC design," in 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2020, pp. 44– 53. DOI: 10.1109/IPDPSW50202.2020.00016.
- [6] J. Mack, S. Hassan, N. Kumbhare, M. Castro Gonzalez, and A. Akoglu, "CEDR: A Compiler-Integrated, Extensible DSSoC Runtime," vol. 22, no. 2, Mar. 2023, ISSN: 1539-9087. DOI: 10.1145/3529257.
- [7] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-c," in *Tools for High Performance Computing 2009*, Berlin, Heidelberg: Springer, 2010, pp. 157– 173, ISBN: 978-3-642-11261-4. DOI: 10/b8rf67.
- [8] A. AMBA, "AXI4-stream protocol specification," *Volume IHI* 51A,
- ZCU102 evaluation board, Accessed: 2023-02-09. [Online]. Available: https://www.xilinx.com/support/documentation/ boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf.
- [10] Jetson AGX Xavier evaluation board, Accessed: 2023-02-09. [Online]. Available: https://www.nvidia.com/en-us/ autonomous-machines/embedded-systems/jetson-agx-xavier/.
- [11] T. Abtahi, C. Shea, A. Kulkarni, and T. Mohsenin, "Accelerating convolutional neural network with FFT on embedded hardware," *IEEE Transactions on Very Large Scale Integration* (*VLSI*) Systems, vol. 26, no. 9, pp. 1737–1749, 2018. DOI: 10.1109/TVLSI.2018.2825145.
- [12] J. Mack, S. E. Arda, U. Y. Ogras, and A. Akoglu, "Performant, multi-objective scheduling of highly interleaved task graphs on heterogeneous system on chip devices," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 9, pp. 2148– 2162, 2022. DOI: 10.1109/TPDS.2021.3135876.
- [13] S. E. Arda, A. Krishnakumar, A. A. Goksoy, N. Kumbhare, J. Mack, A. L. Sartor, A. Akoglu, R. Marculescu, and U. Y. Ogras, "DS3: A System-Level Domain-Specific System-on-Chip Simulation Framework," *IEEE Transactions on Comput*ers, vol. 69, no. 8, pp. 1248–1262, Aug. 2020, ISSN: 1557-9956. DOI: 10.1109/TC.2020.2986963.
- [14] O. Matthews, A. Manocha, D. Giri, M. Orenes-Vera, E. Tureci, T. Sorensen, T. J. Ham, J. L. Aragon, L. P. Carloni, and M. Martonosi, "MosaicSim: A Lightweight, Modular Simulator for Heterogeneous Systems," in 2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Aug. 2020, pp. 136–148. DOI: 10.1109/ ISPASS48437.2020.00029.
- [15] A. Vega, A. Amarnath, J.-D. Wellman, H. Kassa, S. Pal, H. Franke, A. Buyuktosunoglu, R. Dreslinski, and P. Bose, "STOMP: A tool for evaluation of scheduling policies in heterogeneous multi-processors," *arXiv:2007.14371 [cs]*, Jul. 28, 2020. arXiv: 2007.14371.
- [16] J. Cong, Z. Fang, M. Gill, and G. Reinman, "PA-RADE: A cycle-accurate full-system simulation Platform for Accelerator-Rich Architectural Design and Exploration," in 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Nov. 2015, pp. 380–387. DOI: 10.1109/ ICCAD.2015.7372595.

- [17] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-designing accelerators and SoC interfaces using gem5-Aladdin," in 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Oct. 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783751.
- [18] Y. Xiao, S. Nazarian, and P. Bogdan, "Self-Optimizing and Self-Programming Computing Systems: A Combined Compiler, Complex Networks, and Machine Learning Approach," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 6, pp. 1416–1427, Jun. 2019, ISSN: 1063-8210, 1557-9999. DOI: 10.1109/TVLSI.2019.2897650.
- [19] C. Bolchini, S. Cherubin, G. C. Durelli, S. Libutti, A. Miele, and M. D. Santambrogio, "A runtime controller for opencl applications on heterogeneous system architectures," *SIGBED Rev.*, vol. 15, no. 1, pp. 29–35, Mar. 2018. DOI: 10.1145/ 3199610.3199614.
- [20] X. Tan, J. Bosch, C. Álvarez, D. Jiménez-González, E. Ayguadé, and M. Valero, "A hardware runtime for task-based programming models," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 9, pp. 1932–1946, 2019. DOI: 10.1109/TPDS.2019.2907493.
- [21] Nanos++ C/C++ API, Accessed: 2023-02-09. [Online]. Available: https://pm.bsc.es/ftp/nanox/doxygen/group_capi.html.
- [22] J. M. de Haro, J. Bosch, A. Filgueras, M. Vidal, D. Jiménez-González, C. Álvarez, X. Martorell, E. Ayguadé, and J. Labarta, "Ompss@fpga framework for high performance fpga computing," *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2029–2042, 2021. DOI: 10.1109/TC.2021.3086106.
- [23] R. Kumar, M. Baughman, R. Chard, Z. Li, Y. Babuji, I. Foster, and K. Chard, "Coding the computing continuum: Fluid function execution in heterogeneous computing environments," in 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2021, pp. 66–75. DOI: 10. 1109/IPDPSW52791.2021.00018.
- [24] C. Hsieh, A. A. Sani, and N. Dutt, "SURF: Self-aware Unified Runtime Framework for Parallel Programs on Heterogeneous Mobile Architectures," in 2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC), ISSN: 2324-8432, Oct. 2019, pp. 136–141. DOI: 10.1109/ VLSI-SoC.2019.8920374.
- [25] V. G. Pinto, L. Leandro Nesi, M. C. Miletto, and L. Mello Schnorr, "Providing in-depth performance analysis for heterogeneous task-based applications with StarVZ," in 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2021, pp. 16–25. DOI: 10.1109 / IPDPSW52791.2021.00013.
- [26] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011. DOI: https://doi.org/10.1002/cpe.1631.
- [27] G. Vasiliadis, R. Tsirbas, and S. Ioannidis, "The best of many worlds: Scheduling machine learning inference on cpu-gpu integrated architectures," in 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2022, pp. 55–64. DOI: 10.1109/IPDPSW55747.2022.00017.
- [28] J. Kim, S. Lee, B. Johnston, and J. S. Vetter, "IRIS: A portable runtime system exploiting multiple heterogeneous programming systems," in 2021 IEEE High Performance Extreme Computing Conference (HPEC), 2021, pp. 1–8. DOI: 10.1109/ HPEC49654.2021.9622873.