Tutorial: A Novel Runtime Environment for Accelerator-Rich Heterogeneous Architectures

JOSHUA MACK, Electrical and Computer Engineering, The University of Arizona, Tucson, United States
ANISH KRISHNAKUMAR, Electrical and Computer Engineering, University of Wisconsin-Madison, Madison, United States
UMIT OGRAS, Electrical and Computer Engineering, University of Wisconsin-Madison, Madison, United States
ALI AKOGLU, Electrical and Computer Engineering, The University of Arizona, Tucson, United States

As the landscape of computing advances, system designers are increasingly exploring methodologies that leverage higher levels of heterogeneity to enhance performance within constrained size, weight, power, and cost parameters. CEDR stands as an ecosystem facilitating productive and efficient application development and deployment across heterogeneous computing systems. It fosters the co-design of applications, scheduling heuristics, and accelerators within a unified framework. Our goal is to present CEDR as a promising environment for lifting the barriers to research on heterogeneous systems and addressing the broader challenges within domain specific architectures. We introduce CEDR and discuss the evolutionary design decisions underlying its programming model. Subsequently, we explore its utility for broad range of users through design sweeps on off-the-shelf heterogeneous platforms across scheduling heuristics, hardware compositions, and workload scenarios.

CCS Concepts: • **Computer systems organization → System on a chip**; **Heterogeneous (hybrid) systems**; *Multicore architectures*; • **Software and its engineering → Runtime environments**; • **Hardware →** *Analysis and design of emerging devices and systems.*

Additional Key Words and Phrases: Domain-Specific SoCs, Heterogeneous application runtimes

1 Introduction

The continued stagnation of transistor scaling is leading to a resurgence of research into foundational assumptions in computer architecture design, leading to what Hennesey and Patterson term "a new golden age for computer architecture" [1]. While techniques such as pipelining, prefetching, and out-of-order execution have yielded performance gains for general-purpose processors, their efficacy is reaching a plateau. Hence, the quest for a solution prompts an inquiry into the potential of heterogeneity. Heterogeneous computing, a concept well-recognized in the field, offers diverse architectural strategies, including big.LITTLE CPU designs and GPU/FPGA accelerators, each exploiting SIMD and pipelining to varying degrees. Nevertheless, these approaches often fall short when benchmarked against application-specific integrated circuit (ASIC) designs. Moreover, achieving generalized heterogeneity across diverse computational tasks remains a formidable challenge.

Domain-Specific Architectures (DSAs) are one promising avenue by which this architectural innovation is manifesting to meet tomorrow's computational demands. The motivation of such devices is fairly simple: in

Authors' Contact Information: Joshua Mack, Electrical and Computer Engineering, The University of Arizona, Tucson, Arizona, United States; e-mail: jmack2545@email.arizona.edu; Anish Krishnakumar, Electrical and Computer Engineering, University of Wisconsin-Madison, Madison, Wisconsin, United States; e-mail: anish.n.krishnakumar@wisc.edu; Umit Ogras, Electrical and Computer Engineering, University of Wisconsin-Madison, Madison, Wisconsin, United States; e-mail: uogras@wisc.edu; Ali Akoglu, Electrical and Computer Engineering, The University of Arizona, Tucson, Arizona, United States; e-mail: akoglu@ece.arizona.edu.

ACM 1558-3465/2024/8-ART
https://doi.org/10.1145/3687463

a general-purpose computing context, heterogeneous computing systems are difficult to program and utilize effectively. The hypothesis with DSAs, 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. Domain-specific System-on-Chips (DSSoCs) represent a specialized subset tailored for embedded systems applications. The overarching objectives of DSAs encompass synergizing general-purpose processors with specialized accelerators within constrained computational domains. This strategic alignment aims to augment programmability beyond the confines of traditional heterogeneous systems, facilitating energy-efficient execution and affording flexibility within the designated domain. The architectural challenges [2] inherent in developing DSAs span multiple layers, encompassing hardware design, resource management, and programming paradigms [3–9]. At the programming layer, the integration and representation of new applications, alongside debugging and insight-gathering methodologies, pose significant considerations. Effective resource management mechanisms are imperative to optimize shared compute platforms, ensuring efficient utilization of available resources.

System designers are continuously exploring design methodologies that harness increased levels of heterogeneity towards pushing the boundaries of achievable performance gains. Optimizing a heterogeneous computing system is a multi-dimensional and complex design space exploration problem particularly in scenarios where algorithms, data types, and processing resources and their availability vary dynamically. We have developed CEDR (Compiler-integrated Extensible DSSoC Runtime) [10–12], an open-source [1], unified compilation, and runtime framework designed for heterogeneous systems. This framework empowers users to develop, compile, and deploy applications on off-the-shelf heterogeneous computing platforms seamlessly, eliminating the need for users to possess specialized hardware expertise throughout the process. CEDR is scalable as it allows execution of thousands of jobs, flexible as it is able to execute arbitrary interleaved workloads across pools of arbitrary accelerators. CEDR supports the integration of a wide variety of different scheduling heuristics. All these features together allow performing comprehensive design space exploration and co-design of application schedulers and accelerators in one unified environment. Importantly, this framework is portable across a wide range of Linux-based systems, ensuring that effort to migrate across systems is minimal for all developers involved. We have conducted evaluations of this runtime system across a diverse array of platforms, including various FPGA systems such as the Xilinx ZCU-102 and GPU systems such as the NVIDIA Jetson AGX. Furthermore, we have integrated other compute frameworks like GNU Radio and PyTorch, alongside preliminary work with custom RISC-V architectures emulated on FPGAs. Its utility has been successfully tested and independently evaluated by several industry and academic partners with their applications.

This paper presents the lecture titled "CEDR: A Novel Runtime Environment for Accelerator-Rich Heterogeneous Architectures," which was delivered during the "Education Classes" session of the "2023 Embedded Systems Week" [13]. Our lecture caters to audiences with diverse backgrounds and varying levels of expertise, providing an opportunity for exploration and study of computing in heterogeneous context. We organize the paper as follows. Before delving into the technical details of our runtime, we start our discussions by establishing background on the broader challenges and approaches within DSA research in Section 2. We then give an overview of CEDR in Section 3 and discuss the design decisions made throughout the evolution of the framework [10–12]. We then present in-depth exploration of its productive programming and deployment methodology tailored for three distinct user types in Section 4. First and foremost, we guide the naive application developer, aiming to harness acceleration within heterogeneous environments, through the process of utilizing CEDR to adapt their reference C/C++ applications for execution on heterogeneous computing systems [14, 15]. Moving on, the lecture transitions to the perspective of the system designer, illustrating how performance evaluations can be conducted through design sweeps encompassing various hardware compositions while accommodating dynamically arriving

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

workload scenarios. Lastly, we delve into the realm of the resource management developer, demonstrating how to integrate a new scheduler and evaluate its performance concerning a specified workload and hardware composition [16, 17]. Throughout these scenarios that target application developers, system designers and resource management heuristic developers, the common thread is lifting the barriers to research and enabling productive application development and deployment on heterogeneous systems. Finally we present our conclusions and future work in Section 5.

2 Overview of Domain-Specific Architecture Research

Hardware components integral to domain-specific architecture (DSA) construction include fixed-function accelerators, specialized processors, and general-purpose processors. Fixed-function accelerators are dedicated hardware units optimized for specific kernels, such as neural network inference or signal processing algorithms. Specialized processors offer a more generalized approach, capable of accelerating a library of kernels within a specific domain. Examples include coarse-grained reconfigurable architectures (CGRAs) and systolic array processors [18]. These processors typically comprise clusters of processing elements interconnected with configurable networks. General-purpose processors, such as x86, Arch64, or RISC-V cores, remain indispensable in DSAs for executing non-accelerated tasks or novel kernels not optimized for specialized accelerators. An example is the Boom RISC-V core [19], featuring out-of-order execution capabilities alongside baseline RISC-V support. An often-overlooked but critical aspect of DSAs is on-chip interconnects, facilitating efficient data movement among interconnected processors [20]. Various interconnect types, including network-on-chip systems and packet-switched buses, play a vital role in optimizing data transfer within the system. Resource management is another crucial aspect of DSA design, necessitating supervisory scheduling to address resource contention. While traditional operating systems excel in homogeneous systems, they often fall short in heterogeneous SoC environments [21]. Poor scheduling decisions can negate the performance gains achieved through kernel acceleration, underscoring the importance of effective resource management strategies. Resource management in DSAs presents several challenges that include include heterogeneity, the need to handle continuous data flows with regular deadlines, and dynamically interleaving applications. Heterogeneity within DSAs complicates scheduling by enlarging the search space and removing the symmetry present in homogeneous contexts. Unlike standard homogeneous CPUs, where tasks exhibit consistent execution times regardless of the resource, heterogeneous architectures require more nuanced scheduling approaches. Many DSA applications operate on continuous data flows with strict deadlines, necessitating schedulers capable of managing such streams efficiently while accommodating sporadically arriving tasks. Dynamic interleaving of applications in DSAs complicates offline expert analysis and necessitates the development of schedulers capable of producing execution traces comparable in quality to those crafted by experts. Scheduling in DSA architectures is inherently an NP-complete problem [22], posing challenges in identifying quality solutions. Approaches to address this problem fall into three categories: optimization-based methods [23, 24], heuristic methods [25–30], and machine learning-based methods [31, 32]. It's also essential to recognize that resource management in DSAs extends beyond task scheduling. Preserving energy and prolonging battery life are crucial considerations, necessitating the inclusion of dynamic thermal and power management (DTPM) and dynamic voltage and frequency scaling (DVFS) policies. These policies enable processing elements to regulate their thermal and power requirements by adjusting voltage and frequency based on current system conditions. Additionally, ensuring reliability in DSAs remains an open area of research, highlighting the multifaceted nature of resource management in these systems. When considering programming for DSAs, it becomes apparent that building fast processors alone does not suffice if the programming environment is overly complex and inaccessible to prospective programmers. Typically, such architectures come with extensive data sheets, often spanning thousands of pages, making it challenging for programmers to comprehend the architecture, let alone utilize it effectively. In overall, programming environments for DSAs should possess certain key characteristics:

- Accessibility: Programming environments should enable effective utilization of architecture resources by users with limited hardware knowledge. They should strive to simplify the programming process, ensuring accessibility to a broader range of users.
- Performance and Efficiency: These environments should prioritize performant and energy-efficient execution, guiding users towards optimal resource allocation decisions rather than forcing them into suboptimal choices.
- Portability: Ideally, programming environments should support portably performant code, allowing the same code to run seamlessly across multiple heterogeneous platforms without degradation in performance.

When designing programming approaches for DSAs, it's essential to consider four key user categories:

- Application Programmers: These users possess programming knowledge and understand the algorithms they are trying to map. They are willing to invest effort in modifying their code to suit the architecture, making them early adopters of the technology.
- Application Users: Unlike programmers, these users lack background knowledge in algorithms but have a pool of previously developed applications they wish to deploy. They seek relatively performant solutions that leverage the heterogeneity of the system.
- Performance Programmers: Users in this category have strict power or execution constraints and are adept at modifying their code to optimize performance for the architecture. They are willing to invest time in reading architecture manuals to maximize their performance.
- Performance Users: Similar to performance programmers, these users have strict requirements but lack the background or ability to modify their code. They heavily rely on the programming methodology and compiler to meet their performance goals.

An ideal programming environment for DSAs would cater to the needs of all these user categories. In the case of CEDR, efforts have been directed towards building a runtime system that accommodates the diverse requirements of users, aiming to strike a balance between accessibility, performance, and efficiency across various user profiles.

3 Design Overview of CEDR

As illustrated in Figure 1, the CEDR ecosystem includes a compiler frontend where users prepare their applications using API-based programming model using C or C++ and the compiler flow prepares application binaries for execution on the target heterogeneous system. The back-end runtime system receives user applications through a dedicated job submission process that goes through a shared memory channel to the runtime that executes as a background job in the Linux user-space on the target hardware platform. CEDR offers several key features for application developers and designers of heterogeneous computing systems as listed below:

- Portable across a wide range of Linux systems and commercially off the shelf heterogeneous SoC systems relying primarily on POSIX APIs for platform-specific code implementation and has been deployed and validated on rich set of platforms such as Zynq UltraScale+ MPSoC ZCU102, Virtex UltraScale+ VCU128, Virtex 7 FPGA VC707, NVIDIA Jetson AGX Xavier, Synopsys HAPS100, OKdo ROCK 5, Genesys 2 Kintex-7 and Odroid XU4.
- Scalable and flexible to support executing arbitrary interleaved workloads scaling to thousands of applications across pool of CPU cores and different accelerators.
- Plug-and-play interfaces to support integration of a wide variety of scheduling heuristics, enabling comprehensive design space exploration and co-design of application schedulers and accelerators within a unified environment.

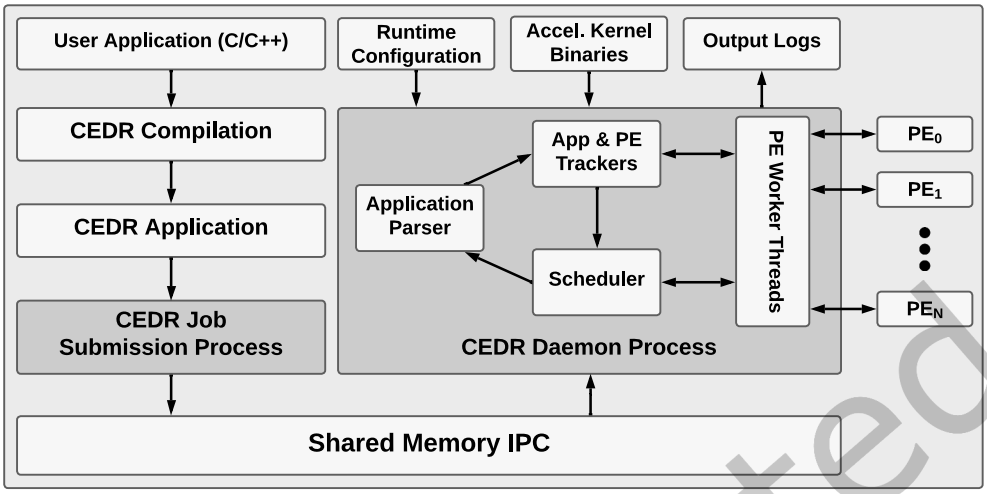

Fig. 1. Job submission process triggers daemon process

3.1 Programming Approach

In the programming environment, users provide their applications in C or C++, and various compilation approaches are employed to produce CEDR applications. These applications are then submitted through a dedicated job submission process to the CEDR runtime, which operates as a background job on the managed system. The runtime configuration specifies the scheduling heuristic and available accelerator binaries for dispatching accelerated tasks. The CEDR runtime executes applications in a runtime loop on various processing elements and generates multiple types of output logs to facilitate analysis of the execution flow and performance metrics.

One programming methodology supported by CEDR is DAG-based execution. In this approach, applications are represented as directed acyclic graphs (DAGs), where each node corresponds to a computational task within the application. Nodes supporting accelerator dispatch may have multiple implementations. The role of the scheduler is to select the appropriate implementation for execution based on the system's current state. Since dependencies are captured within the DAG, the scheduler can determine the order of task execution without extensive program analysis, as precedence of tasks are encoded directly in the metadata of the binary format. In CEDR's JSON format, crucial sections include a variables section utilized to enable management of application memory. Variables, along with their memory allocation and possibly initial values, can be declared. This allows CEDR to handle memory management for DAG nodes' arguments, potentially facilitating future memory optimization strategies. Another component in the JSON format is the DAG node section, where each node contains a list of arguments required by its implementation functions, predecessors, successors, and platforms with associated functions for executing the task on specific accelerators or CPUs.

Building DAGs can be time-consuming. CEDR offers two main approaches for constructing these applications as illustrated with Figure 2. The first approach is an automated compilation tool flow, which relies on dynamic instrumentation and tracing of the baseline application. This methodology involves behavioral analysis based on the observed execution characteristics of the application. CEDR utilizes a framework called TraceAtlas [33] for this purpose. TraceAtlas instruments the binary to understand transitions between LLVM basic blocks, recording the sequence of these transitions. It then clusters portions of the program into kernels based on this data, providing identified kernels and basic memory analysis. Using LLVM, the original application can be partitioned into

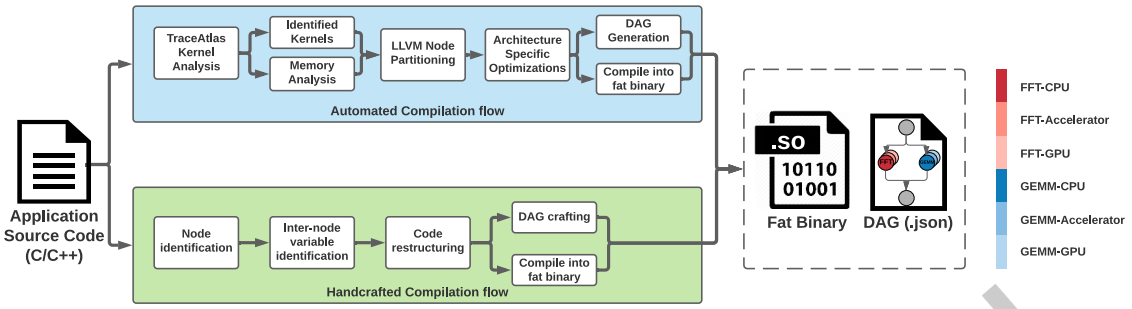

Fig. 2. Programming Methodologies

DAG nodes according to the boundaries of these kernels. Architecture-specific optimizations, such as enabling accelerators and implementing patterns, can be applied at this stage. Finally, the JSON DAG and the binary object of the application are generated for use with CEDR. The second approach to preparing DAG-based applications involves handcrafted compilation, where expert tuning and transformation are employed to optimize the application for CEDR's DAG-based representation. While this approach offers significant performance benefits by allowing explicit definition of parallelism and heterogeneity, it is a labor-intensive process that CEDR aims to minimize in the future. As the development of DAG-based methodologies progressed, limitations became apparent. Neither automated transformation nor handcrafted DAGs are perfect solutions. Automated transformation struggles with generalizing across various use cases, while handcrafting for realizing optimized DAGs is inherently complex. Moreover, the strict adherence to a DAG model imposes limitations, particularly regarding cyclic program structures. Many desirable program structures, such as loops, involve cycles, which cannot be represented in a strict DAG model. For instance, a simple for loop iterating over input data and processing it through a sequence of kernels, where some kernels depend on the results of previous ones, cannot be accurately captured without allowing cycles in the graph representation.

Due to these challenges, we introduced and developed an API-based programming methodology focusing on user productivity and allow users to develop and deploy their applications without having to follow hand-crafting based optimization methodology. The core idea is to define a library of APIs, such as FFT , matrix multiplication, FIR kernels, allowing users to invoke those functions in their implementations and pass arguments in a standard manner and programming environment they are familiar with. Conceptually, the program is segmented into non-kernel portions, where application reaches a barrier and API call is dispatched to CEDR to manage that task and support execution of heterogeneous tasks on domain-specific resources. Upon completion of these tasks, indicated by a barrier, the non-kernel portion of the application continues execution as illustrated in Figure 3. Practically, this approach involves writing code where API calls, like FFT on input and output buffers, serve as barriers. These calls are akin to initializing a barrier and invoking an asynchronous task dispatch function, where the task type and input parameters are passed to CEDR. Subsequently, the program waits for the completion of the dispatched task on the designated resource.

All implementations of CEDR APIs are housed within a library called `libCEDR` as illustrated with Figure 4. This library comprises hardware-agnostic API calls coupled with a pool of platform-specific implementations that can be dynamically enabled or disabled based on the available hardware on the target system. For instance, the platform-agnostic FFT API call can be linked to CPU or accelerator implementations internally. `libCEDR` is designed to allow the use of the same source code for verification outside of CEDR's runtime environment. By linking in CPU implementations of tasks, users can validate their applications independently. Transitioning

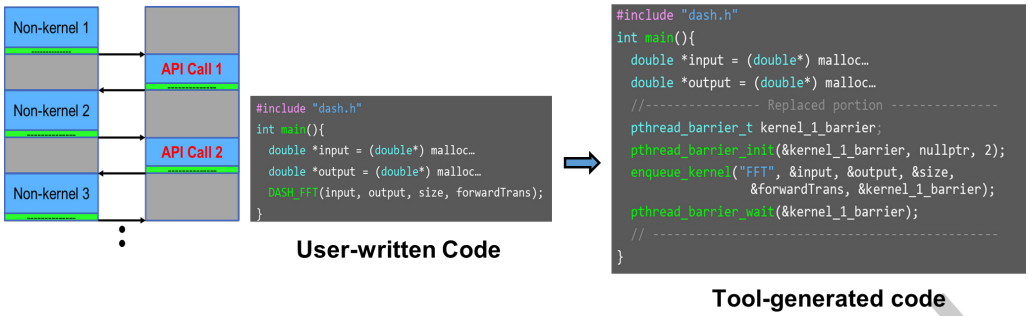

Fig. 3. API-based programming model

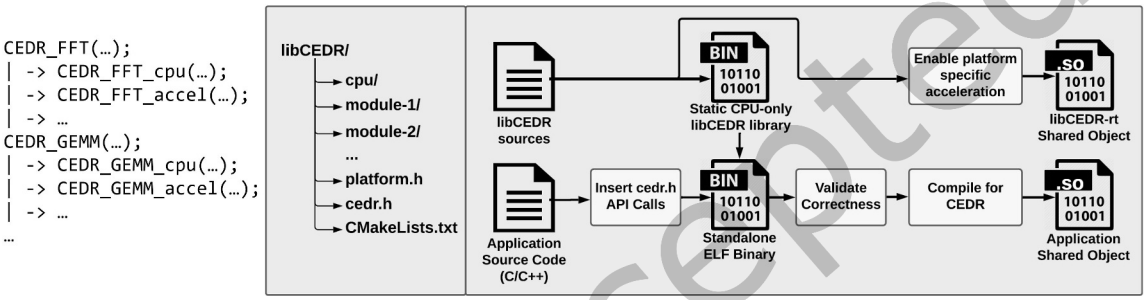

Fig. 4. CEDR-API implementation and support

to execution within CEDR's runtime environment involves building the application as a shared object and adjusting compilation flags. This enables execution on various platforms supported by CEDR by simply adjusting the shared object containing backend accelerator implementations. The directory structure typically includes `libCEDR`, with `CEDR.h` containing platform-agnostic headers for all APIs, and several modules containing CPU and hardware accelerator backend implementations. The usage flow involves inserting CEDR-specific API calls into the application source code, compiling `libCEDR` sources into a static CPU-only library, and combining them to produce a standalone ELF binary for validation purposes during development. Developers can iterate on their implementations, check for functional correct execution in each design iteration and converge to a final form. Once functional correctness is achieved, the code can be compiled for execution within CEDR's runtime.

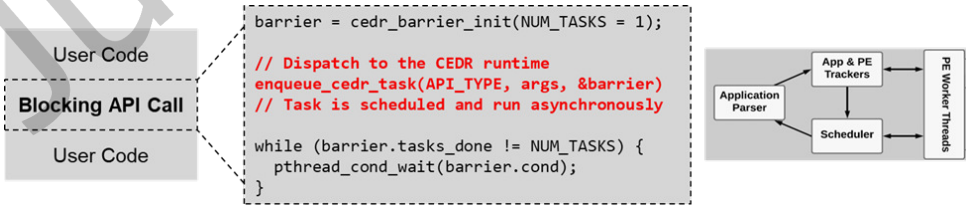

Fig. 5. Non-blocking API execution in CEDR

The compilation process generates a shared object that is handed off to the CEDR runtime along with the `libCEDR` library containing all backend implementations. Task synchronization is crucial to maintaining functional

Table 1. List of built-in applications in CEDR.

Application	Description	Number of APIs
Radar Correlator	Compares radar echoes with reference signals to identify targets or patterns of interest	3 FFTs
Pulse Doppler	Enables simultaneous target detection and velocity measurement	512 FFTs
WiFi TX	Involves transmitting data over WiFi networks to establish wireless connectivity	100 FFTs
Temporal Mitigation	Minimizes interference caused by overlapping or conflicting temporal signals	4 GeMMs
Synthetic Aperture Radar	Uses radar antenna motion to create high-resolution images of the radar scene	1,537 FFTs and 768 ZIPs
Lane Detection	Identifies and tracks the lanes on a road, aiding in autonomous driving systems	24580 FFTs

correctness in the user application. The CEDR APIs, as mentioned earlier, allow for initializing barriers and asynchronously enqueuing tasks to CEDR. These tasks encode the API type, arguments, and the associated barrier. While waiting for task completion, the application synchronously waits for the barrier. This methodology ensures that kernel executions within CEDR's runtime loop occur asynchronously in different threads relative to the baseline user application. Without this synchronization methodology, there would be a risk of the code expecting results before they are available. CEDR provides two types of APIs to accommodate different user needs. For users focused on maximizing parallelism and performance, non-blocking APIs are available as illustrated with Figure 5. These APIs allow for the asynchronous submission of tasks, enabling parallel scheduling and dispatching within CEDR. On the other hand, blocking APIs are provided for baseline users who prioritize leveraging heterogeneity without necessarily maximizing parallelism. These users prefer synchronous execution and do not require extensive parallelism optimization. In summary, CEDR offers two programming methodologies to cater to different user preferences and application requirements.

CEDR provides several built-in applications summarized in Table 1 spanning signal processing, communications, and autonomous vehicles domains. Radar Correlator detects targets or patterns of interest by correlating received echoes with transmitted radar pulses. Pulse Doppler Radar measures not only distance but also velocity of the object using radar technology. Wi-Fi transmit chain (WiFiTX) handles processing for transmitting Wi-Fi frames. Temporal Mitigation mitigates interference caused by overlapping signals in time, enabling extraction and processing of specific signals. Synthetic Capture Radar utilizes radar to generate images. Lane Detection identifies and tracks lane lines on roads. All these applications vary in terms of their control flow structures, demand for accelerators, and degree of concurrency during their execution. Therefore, when coupled together, they enable users to generate workloads to stress the system resources. While CEDR includes these pre-built applications, users can also develop and integrate their own applications. Additionally, existing applications can be ported to CEDR by following one of the two programming methodologies discussed earlier.

3.2 Runtime Flow

Below, we outline the main runtime loop within CEDR and discuss how tasks are managed by CEDR through queues formed at worker thread level as illustrated with Figure 6. Applications are submitted to the CEDR runtime via a helper process, where they are parsed and launched. In the case of API-based applications, a new thread is spawned on the system to execute the main function of the application. As the application progresses, API functions are called, pushing tasks into CEDR management threads' ready queue. The scheduler then schedules these tasks to available resources. Each resource in the system, whether it's a CPU or an accelerator, is broadly

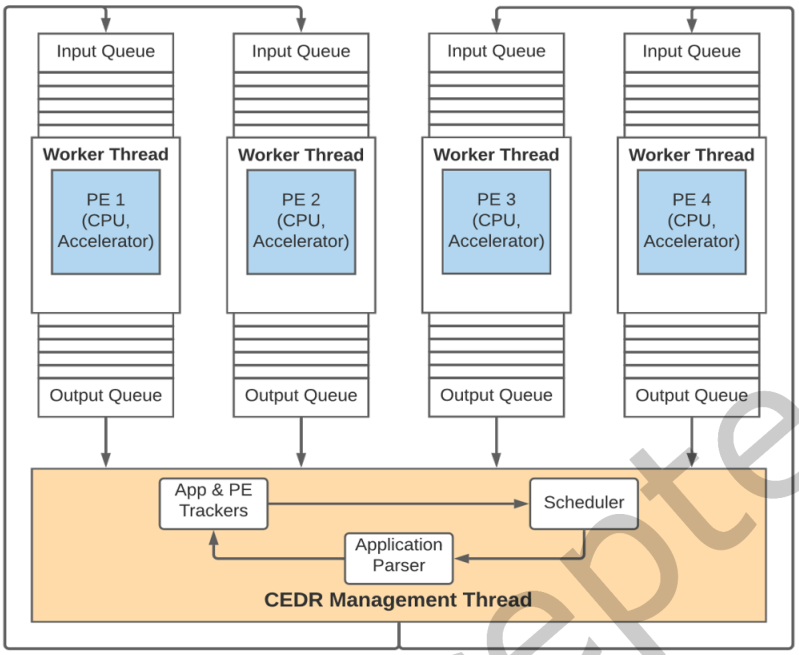

Fig. 6. Worker thread-level management of tasks in the user application.

represented by a worker thread. When a worker thread assigned to a CPU receives a task, it directly executes it. For instance, if the task is to run an FFT using the CPU implementation, the CPU worker thread executes it, records execution metrics, and pushes the completed task back onto the output queue. Conversely, if the resource is an accelerator, the worker thread acts as a management CPU core for that accelerator. It configures the accelerator, manages data movement, and waits for completion notices. Once completed, it transfers necessary data back and pushes the task onto the output queue. This workflow ensures efficient scheduling and execution of tasks across available resources within the CEDR runtime environment.

The task scheduling mechanism within CEDR is supported by a library of schedulers listed in Table 2, each offering distinct strategies. Among these schedulers we note the Heterogeneous Earliest Finish Time (HEFT) [25] algorithm that is used as a reference algorithm for heterogeneous environments to this date. We implemented runtime version of this method [16] that is referred to as $HEFT_{RT}$. In order to offer rapid decision making capability in real time resource management scenarios, we also implemented the hardware-based $HEFT_{RT}$ [17] and integrated with CEDR. CEDR offers distinct plug and play interfaces for integrating new scheduling heuristics and methodologies. One such use case is the integration of machine learning based schedulers [31], where sophisticated scheduling heuristics such as HEFT, ETF or EFT can be used to generate data in CEDR framework for a given workload scenario and train an Imitation Learning (IL) model. These schedulers leverage decision trees and deep neural network implementations. They learn from past scheduling decisions and adaptively schedule tasks based on learned patterns, aiming to optimize resource utilization and performance. These schedulers collectively contribute to enhancing CEDR's task scheduling capabilities, catering to various application requirements and system configurations.

Table 2. List of built-in schedulers in CEDR.

Algorithms	Class
Round-Robin	Software
Random	Software
Earliest Task First	Software
Earliest Finish Time	Software
Minimum Execution Time	Software
Real-time Heterogeneous Earliest Finish Time	Software and Hardware
Imitation Learning (IL) based Decision Trees	Machine Learning
Imitation Learning (IL) based Deep Neural Networks	Machine Learning

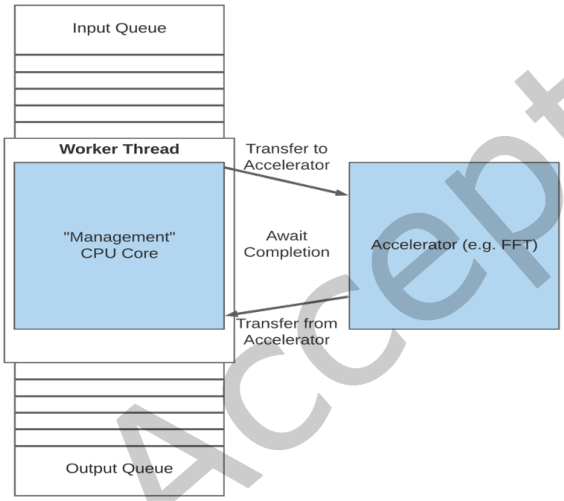

Fig. 7. Accelerator management and integration

Accelerators within the system are managed by dedicated threads due to inherent limitations—most accelerators lack the capability to independently execute threads. Moreover, operating systems lack standardized abstractions for representing arbitrary accelerators, which complicates their management. While efforts have been made to define interfaces for GPUs and PCI Express devices, the landscape remains sparse, especially in embedded systems housing fixed-function accelerators like FFT units.

Our approach has been to associate accelerators with supporting CPU threads to circumvent these limitations. However, the data transfer methodology varies depending on the accelerator's requirements. In our experimentation, particularly on FPGA-based platforms, we've predominantly utilized userspace DMA buffers via a library called UDMA-buf, along with DMA engines embedded in FPGA programmable fabric. DMA buffers serve a crucial role in facilitating data transfer to accelerators. Since many accelerators are incompatible with virtual memory, data must be staged in contiguous memory. This involves defining a physical base address in DDR and specifying a length for data transfer. However, challenges arise when considering virtual memory segmentation by operating system pages, often leading to fragmented data representation from the accelerator's perspective. To address this, DMA buffers enable the declaration of physically contiguous memory buffers larger than the

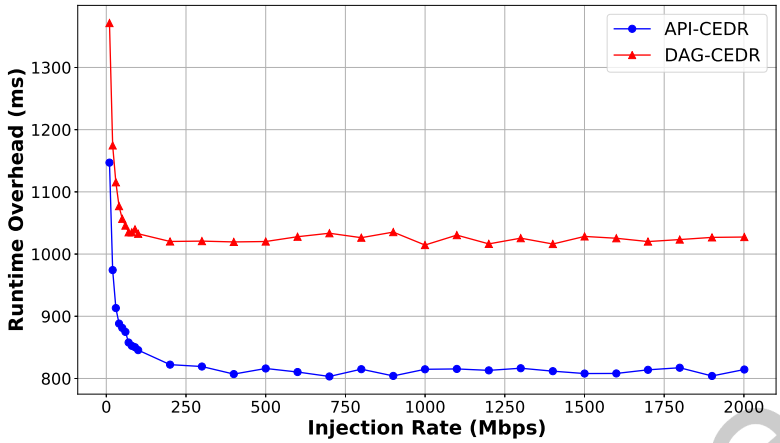

Fig. 8. Runtime overhead comparison between DAG and API-based CEDR with respect to injection rate

OS page size, facilitating efficient data staging for transfer into accelerators by DMA engines. For example, as illustrated with Figure 7, in the context of an FFT IP core, data is copied from a virtual memory-based array into a DMA buffer. Subsequently, the DMA engine, coupled with the FFT accelerator, orchestrates data transfer and processes the input. Finally, the output is copied back from the UDMA buffer to the output array. Despite these advancements, optimizing data movement remains a focal point for future exploration. Zero-copy transfers to accelerators or u-dma-buffers could mitigate data transfer overhead. This may entail developing heap algorithms atop u-dma-buffers to enable direct memory allocation, eliminating the need for intermediate data movement. Additionally, optimizing worker threads awaiting accelerator completion is imperative to minimize CPU resource consumption. Techniques such as efficient thread yielding mechanisms and leveraging signaling mechanisms like interrupts could enhance overall system efficiency.

CEDR offers robust support for platform performance monitoring and workload profiling, essential for design space exploration and application development. This capability is facilitated through the integration of PAPI (Performance Application Programming Interface), enabling the reading of hardware-level performance counters directly within CEDR. With PAPI, users can conduct low-level profiling and workload characterization for every task within an application, all without necessitating code modifications. Whether employing the DAG-based or API-based methodology, users submit task nodes to CEDR. Within the worker threads responsible for task execution, performance counter measurements commence before the dispatch function is invoked for either the CPU or accelerator. These performance counters encompass a wide array of metrics, tailored to suit diverse profiling needs. For instance, on platforms like the ZCU-102 FPGA from Xilinx, users can select from a comprehensive list of 113 counters. Metrics may include instructions executed, branch operations, cache hits, cache misses, and more. Data collected from these performance counters can be aggregated at both the application and individual task levels. At the application level, counter values are summed across all tasks, providing a holistic view of performance. Conversely, at the task level, specific counters can be tailored to each task, enabling fine-grained profiling tailored to the task's requirements. This profiling capability proves invaluable in various scenarios, such as designing memory management techniques where metrics like cache loads and cache misses serve as key optimization indicators. Overall, CEDR's support for performance monitoring and workload profiling empowers users with comprehensive insights into system behavior and performance characteristics.

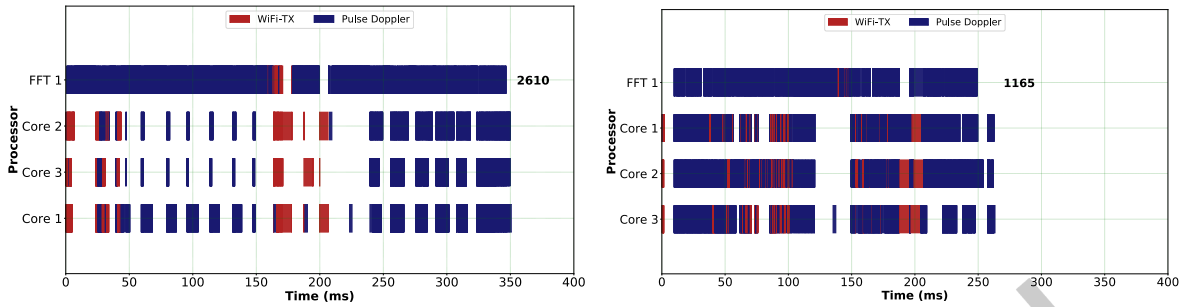

Fig. 9. High latency workload composed of WiFi TX and Pulse Doppler applications arriving dynamically on a system composed of 3 CPUs and 1 FFT , where system is oversubscribed (injection rate 2000 Mbps). Workload involves total of 2610 FFT tasks scheduled with the (a) MET scheduler and (b) EFT scheduler.

In our experiments, we compared the runtime overhead between the API-based and the DAG-based CEDR as illustrated in Figure 8. We conducted these experiments on a platform featuring the ZCU-102 FPGA, equipped with three CPU cores, one FFT accelerator, and one MMULT or GEMM accelerator, all managed by a simple round-robin scheduler. Our workload setup involved running batches of five Wi-Fi TX applications and five Pulse Doppler applications repetitively, with varying injection rates to simulate different data processing speeds. As the injection rate increased, reflecting a higher data processing load, we observed a saturation trend in the runtime overhead. Specifically, our findings revealed a notable reduction in runtime overhead, approximately 20%, with the API-based CEDR compared to its DAG-based counterpart. This reduction in overhead signifies the efficiency gained by the API-based approach, attributable to factors such as reduced dependency tracking and streamlined memory management during application initialization. To quantify the runtime overhead, we measured the makespan of the overall workload, calculated as the duration between the end of the last workload batch and the start of the first batch. During idle periods, when the system is not actively computing tasks, we accounted for the runtime overhead incurred. Our analysis considered the ratio of time spent by the runtime performing tasks against the total computation time, which is inherently lower at lower injection rates due to fewer tasks being processed. It's important to note that the y-axis in our data represents the overhead amortized among a larger number of jobs, rather than the duration of each individual task or job. As the injection rate increases, more tasks are processed concurrently, leading to a gradual increase in overhead, which is spread across a greater number of jobs, hence the observed trend. In essence, our experiments highlight the efficacy of the API-based CEDR in minimizing runtime overhead, particularly under varying data processing loads, showcasing its suitability for diverse application scenarios.

4 Use-cases of CEDR for DSA Research

4.1 Is acceleration always the best choice?

In this experiment, we aimed to evaluate the performance impact of different scheduling strategies on a workload consisting of Wi-Fi TX and Pulse Doppler applications. Specifically, we compared the performance of two schedulers: Minimum Execution Time (MET) and another scheduler (let's call it EFT), which prioritizes task dispatch based on the processing element with the lowest estimated execution time for each task, regardless of the processing element's current workload. Under the MET scheduler, tasks are dispatched to the processing element with the lowest estimated execution time, favoring accelerators whenever possible. For instance, if a

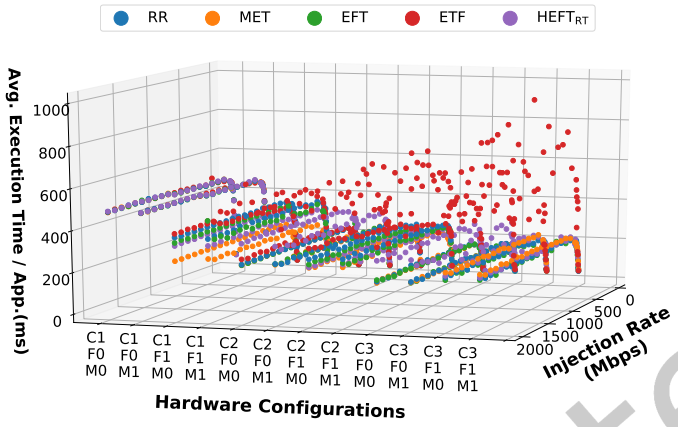

Fig. 10. Design space exploration across scheduling heuristics, workload complexity variations and hardware configurations

CPU core has an estimated execution time of 100 milliseconds, while an accelerator can complete the same task in 10 milliseconds, the scheduler will always choose the accelerator. Our experiment involved running the workload at an injection rate of 2000 megabits per second, representing a relatively high data processing load. The performance comparison illustrated in Figure 9 revealed significant insights. The overall makespan of the workload under the MET scheduler was approximately 350 milliseconds. However, when employing the EFT-based approach, which distributes tasks more evenly across all available cores, even if locally suboptimal decisions are made for some tasks, we observed a notable improvement in execution time. Specifically, the number of FFT tasks launched on the FFT accelerator decreased from 2610 to 1165, indicating a more balanced distribution of tasks across ARM cores. This balanced distribution led to an almost 100-millisecond reduction in execution time compared to the MET scheduler. These findings underscore the importance of considering workload distribution strategies in heterogeneous systems. While prioritizing accelerators may seem advantageous initially, a more balanced approach can often lead to improved overall performance, especially under varying workload conditions.

4.2 Large Scale Design Space Exploration

In large-scale design space explorations facilitated by runtime environments like CEDR, researchers and developers can conduct comprehensive evaluations of various scheduling heuristics and hardware configurations. Such experiments serve multiple purposes, including assessing the performance of new scheduling algorithms, comparing different heuristics under identical conditions, and evaluating the impact of hardware accelerators on diverse workloads. One key application of these experiments is for researchers developing new scheduling heuristics. By running simulations on the same hardware configurations and workloads, developers can gauge the effectiveness of their algorithms relative to existing heuristics. They can also assess how their algorithms perform across different platforms, identifying scenarios where they excel or fall short. Another use case involves hardware accelerator developers seeking to evaluate the impact of their accelerators on workload performance. By testing their accelerators with a variety of scheduling heuristics or application workloads, developers can gain insights into the effectiveness of their hardware in speeding up different types of tasks. In a specific experiment conducted with CEDR, as illustrated with Figure 10, a design space sweep was performed, encompassing 3,480

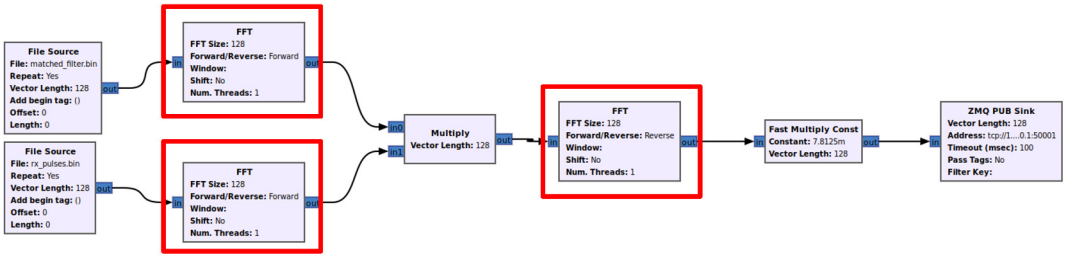

Fig. 11. GNURadio workflow implementing radar correlator with FFT function calls

configurations and executing over 10 million tasks on a Xilinx ZCU-102 FPGA within three hours. Such scale of experiment is not feasible with cycle accurate simulators. Furthermore, the execution time trends and performance characteristics are measured based on deployment of real life applications on commercial off the shelf platforms. This particular experiment involved varying parameters in terms of scheduling heuristic, application injection rate, and hardware configuration (including the number of CPU cores and enabled accelerators). The color-coding in the resulting 3D plot indicates the scheduler used for each configuration. A notable finding from the experiment was the observed behavior of the ETF scheduler. Despite being theoretically promising, this scheduler exhibited significant overhead as the system became highly oversubscribed. As the number of hardware resources increased, the execution time scaled upwards due to the scheduler needing to choose between more resources. Moreover, the time per scheduling decision increased with the data rate, eventually saturating at approximately 80 milliseconds of scheduling overhead per application. A detailed analysis of the scheduling overhead for the most heterogeneous hardware configuration revealed the challenges faced by certain schedulers, particularly in environments with a high task count and diverse accelerators. This analysis provided valuable insights into the scalability and performance characteristics of different scheduling heuristics, shedding light on their suitability for various workload scenarios.

4.3 GNURadio and CEDR

A user group that could benefit from exploring CEDR is developers working with the GNURadio framework. GNURadio is an open-source framework used for building radio and communications applications. With CEDR, developers can enhance the capabilities of GNURadio by incorporating heterogeneous computing resources seamlessly. An out-of-tree module called GR-CEDR has been developed [14] for GNURadio, enabling integration with CEDR functionalities. Out-of-tree modules are add-ons to GNURadio that are compiled separately from the main source code and loaded dynamically. GR-CEDR introduces custom flow graph blocks that utilize CEDR APIs to implement various signal processing tasks. These flow graphs can be constructed graphically using GNURadio's visual representation, where different blocks represent signal processing kernels connected together as illustrated with Figure 11. Alternatively, developers can write code directly in C/C++ or Python and convert it using Cython. CEDR dynamically schedules and dispatches these blocks onto heterogeneous resources, providing transparent heterogeneity to GNU Radio users. An important experiment conducted with GR-CEDR involved running a GNURadio application alongside other CEDR applications (WiFi TX and SAR) as illustrated with Figure 12. The experiment demonstrated effective resource sharing among different applications without impacting the behavior of GNURadio application. Even though GNURadio is unaware of being executed in a heterogeneous environment, CEDR optimally distributes resources, ensuring cooperative system utilization. As shown in the experiment output, GNURadio applications can produce meaningful results, such as estimating the range in a radar correlator application. This seamless integration of CEDR with GNU Radio empowers developers to leverage heterogeneous

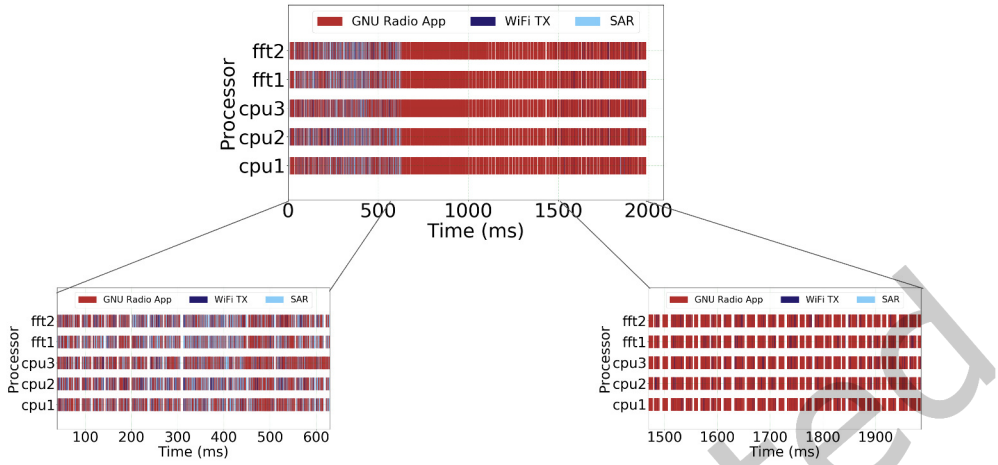

Fig. 12. Deployment of GNURadio workflow on heterogenous systems with WiFi TX and SAR applications on a system composed of 3 CPUs and 2 FFT accelerators. In the early phase of the experiment two applications are arriving to the system (WiFi TX and SAR) along with the GNU radio application (Radar Correlator). System is able to schedule all tasks of the three applications. Non GNU-radio applications are injected later in the experiment to stress the system and CEDR is able to cope with the dynamically changing workload.

Table 3. PyTorch application CEDR-API map

Application	**CEDR-API**			
	ZIP	CONV_2D	CONV_1D	GEMM
OD	54,496	54,496		
VGG	1,634,496	1,634,496		3
Speech	8,224		8,224	1

computing resources without needing to understand platform specifics. Moreover, CEDR facilitates portability across multiple platforms, requiring only a library update for backend implementations. This capability opens new possibilities for enhancing the performance and versatility of GNURadio applications.

4.4 PyTorch and CEDR

Another user group that could benefit from CEDR integration is PyTorch developers. There are two primary use cases where PyTorch developers might find CEDR valuable. Firstly, while GPU acceleration is widespread among different neural network frameworks, PyTorch developers may have scenarios where they need to run multiple simultaneous networks concurrently. For example, a system involving object detection, speech recognition, and object localization simultaneously. In such cases, dynamically distributing the workload among all available processing elements, rather than greedily mapping all tasks to the GPU, could result in better overall performance. CEDR's dynamic scheduling capabilities make it suitable for efficiently utilizing various processing elements in such scenarios. Secondly, while GPU adoption is prevalent, FPGA deployment for PyTorch models is less mature. PyTorch developers interested in exploring energy-efficient FPGA execution for embedded systems could leverage CEDR integration as CEDR offers a pathway for PyTorch models to be executed on FPGAs efficiently.

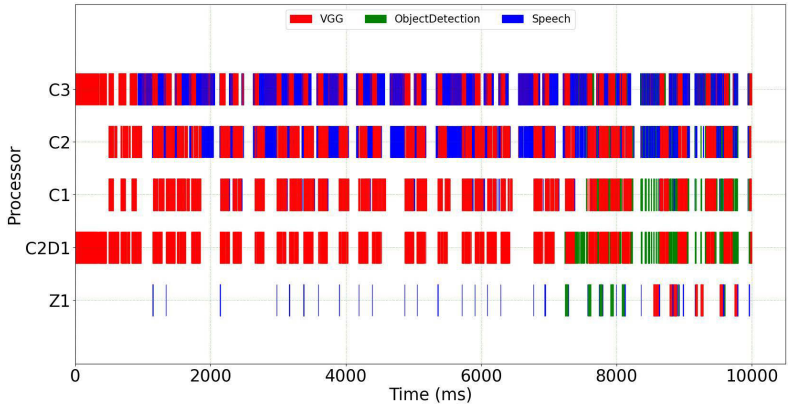

Fig. 13. Multiple PyTorch models running simultaneously (VGG, OD, and Speech) using 3 CPU, 1 C2D, and 1 ZIP accelerators with EFT scheduler

A framework was developed to integrate PyTorch models into CEDR. An example workload was demonstrated on a system with three CPU cores, one 2D convolution accelerator, and one zip accelerator for vector addition and multiplication. The workload consisted of three different neural networks, each performing various inference tasks. Object Detection (OD) is a modified version of the U-Net network [34] with a pre-trained autoencoder. It identifies positions of cars within an input image and generates an output image with marked car locations. Visual Geometry Group (VGG) VGG model recognizes objects within an input image. Specifically, it utilizes the Imagenette Dataset [35], which is a subset of the larger ImageNet Dataset [36]. Speech Classification (SC) model detects a spoken word within the audio, and generates the corresponding word as text output. It is trained and evaluated using the Speech Commands Dataset [37]. Characteristics of these applications are listed in Table 3.

The framework allowed these networks to cooperatively share system resources in a way that was not previously possible. Figure 13 displays a chart illustrating the assignment of tasks to the system resources during the initial 10-second interval of the experiment. We observe that between around 6.5-second and 10-second time frames, all three models (OD, VGG, and Speech) are running simultaneously and CEDR is able to manage pairing the system resources with the tasks of each application. The integration has been tested on the ZCU-102 FPGA and the NVIDIA Jetson platforms, showcasing its versatility across different hardware environments.

4.5 RISC-V and CEDR

Another use case involves exploring RISC-V architecture and composing heterogeneous systems tailored for specific applications as shown in Figure 14. In this scenario, the user is interested in leveraging accelerators and CPUs customized for the RISC-V architecture to build efficient systems. RISC-V presents opportunities for creating domain-tailored CPU architectures, offering avenues for exciting advancements. Given the absence of a general OS-level abstraction for accelerators, management of accelerators by CPUs remains a necessity. Therefore, customizing the CPUs on the system becomes crucial. This customization may involve incorporating big RISC-V cores for compute-intensive tasks and small RISC-V cores optimized for managing accelerators. By minimizing the hardware units in the smaller cores, resources and power budgets can be efficiently allocated to implement more potent accelerators.

Preliminary experiments have been conducted with CEDR on an emulated RISC-V system using the Rocket Core CPU [38] pipelines as shown in Figure 15. The system comprises two compute cores and varying numbers of FFT accelerators, each managed by smaller RISC-V cores. Results indicate that compared to previous experiments,

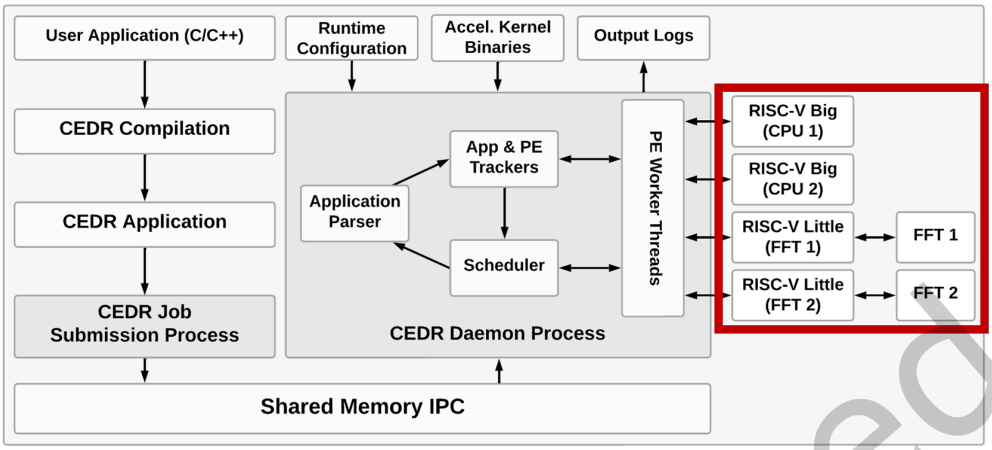

Fig. 14. System composed of heterogeneous set of RISC-V cores

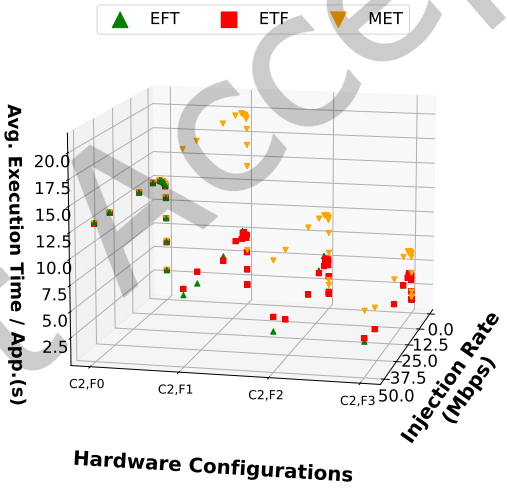

Fig. 15. RISC-V based CPU cores coupled with a pool of accelerators emulated on the ZCU102 FPGA running the WiFi TX and Pulse Doppler applications with respect to various injection rate and scheduling policies

where increasing heterogeneous resources led to contention issues and negative impacts on application execution time, the introduction of customized accelerator management cores has yielded positive outcomes. Specifically, there has been a decrease in average execution time for tasks, indicating improved parallelism and leveraging of system heterogeneity. In summary, exploring RISC-V architecture and customizing CPUs for specific applications

can lead to more efficient and powerful heterogeneous systems. CEDR provides a platform for conducting such experiments and optimizing system configurations for optimal performance.

4.6 Kernel Development and Integration with CEDR

We have developed a rich set of tutorials [39] with code-based demonstrations for different user types who wish to utilize CEDR for application development and deployment on heterogeneous SoCs. As the FPGAs are being embedded in all layers of computing infrastructure from edge to HPC scale, system designers continue to explore design methodologies that leverage increased levels of heterogeneity to push performance within the target performance goals or constraints. Recently we presented a tutorial [40] catering to audiences with diverse backgrounds and varying levels of expertise, providing an opportunity for exploration and study of FPGA-based computing in heterogeneous contexts. The tutorial starts with an overview of CEDR, followed by an in-depth exploration of its productive programming and deployment methodology tailored for three distinct user types. First and foremost, we guide the naive application developer, aiming to harness FPGA-based acceleration within heterogeneous environments, through the process of utilizing CEDR to adapt their reference C/C++ applications for execution on FPGA-integrated systems. Moving on, the tutorial transitions to the perspective of the system designer, illustrating how performance evaluations can be conducted through design sweeps encompassing various hardware compositions while accommodating dynamically arriving workload scenarios. Lastly, we delve into the realm of the resource management developer, demonstrating how to integrate a new scheduler and evaluate its performance concerning a specified workload and hardware composition. Throughout these scenarios that target application developers, system designers and resource management heuristic developers, the common thread is lifting the barriers to research and enabling productive application development and deployment on FPGA-integrated heterogeneous systems. The tutorial adopts a hands-on approach utilizing the Xilinx ZCU-102 platform, enabling participants to gain practical experience on systems that amalgamate ARM CPU cores with FPGA accelerators. The tutorial set [39] includes step by step flow with running examples for setting up CEDR and utilizing it for the following scenarios:

- Introducing an API call to a baseline C++ application.
- Design space exploration by varying number of compute resources across different scheduling heuristics in dynamically arriving workload scenarios.
- Integrating and evaluating scheduling heuristic with CEDR and conducting performance evaluation with dynamically arriving workload scenarios.
- Hands-on exercises for experimenting with a new application that relies on key computation kernels such as FFT, GEMM, Convolution, Vector addition or Vector multiplication on GPU and FPGA based SoCs.
- Integrating a new accelerator to the SoC, introducing the new API and integration with CEDR, and finally application development with the new API to utilize the accelerator.

4.7 Comparative Analysis

We group runtime systems into three classes covering homogeneous or single-ISA heterogeneous environments [41–44], multi-ISA or accelerator-rich environments [3, 6, 7, 45, 46], and compiler integrated ecosystems [11, 42, 43, 47]. Particularly for accelerator rich and heterogenous architectures, runtime design approaches [6, 45, 48] are notable but they lack ability to support scheduling policy development and non-blocking execution. Kim et al. [49] present IRIS, a heterogeneous runtime system that offers resource discovery, adaptive scheduling, data movement, and programming model capabilities. Hardware agnostic and adaptive scheduling approach presented in [50] targets machine learning kernels on heterogeneous architectures. In summary, CEDR offers an integrated environment that uniquely combines application programming, scheduling, and execution

into a single extensible framework that is portable and flexible. Through a hardware agnostic application development and deployment experience on any Linux based heterogeneous system it is designed to lift the barriers to research on heterogeneous computing for application developers, system designers and resource management heuristic developers.

5 Future Directions & Conclusions

In conclusion, CEDR represents a significant advancement in domain-specific architecture research, offering a unique set of capabilities. However, numerous challenges remain to be addressed across various fronts. In resource management, the quest for optimal policies tailored to specific use cases will continue, recognizing that a one-size-fits-all solution is unlikely. Efforts will focus on designing more lightweight and resource-optimal techniques. For accelerators, ongoing work will involve automating the generation of software support for new hardware accelerators, thereby reducing development time and enhancing efficiency. On the programming side, there is a pressing need to make heterogeneity more accessible, ensuring that developers can leverage diverse hardware resources seamlessly. In data flow management, efforts will be directed towards enabling dependent tasks to route data flows directly between accelerators, bypassing complex memory hierarchies such as DRAM to minimize latency.

5.1 Memory Management

In many heterogeneous devices, the availability and composition of memory resources can be as heterogeneous, if not more so, than the compute units themselves. To extract the maximal performance on such systems, heterogeneous runtimes and their corresponding programming interfaces should be designed with comprehensive methods of representing and allocating memory to different regions within the device in order to minimize communication overheads and maximize compute intensity. Following an approach like that taken by the CEDR-API runtime, a possible path here would be the design of a set of "CEDR Malloc" capabilities. Such capabilities could be made aware of the underlying set of scratchpads and memories on a given compute platform and allow data to be allocated via heap-based allocation techniques directly onto the most relevant memories for their computation. If coupled with intelligent heterogeneity-aware heap algorithms, such an approach has the potential to enable a wide range of performance gains within and across accelerator boundaries without excessively burdening the programmer with understanding the details of their platforms' memory architecture.

5.2 Streaming Dataflow Management

Beyond the static memory management approaches of a "CEDR Malloc" methodology, many DSSoC systems are additionally intended to operate on continuous streams of data (for instance, RF systems processing incoming data off a variety of analog to digital converters (ADCs)). Such systems would see little benefits from a static memory allocation methodology as the specific data being processed is rather short-lived. Instead, allocation must be performed by considering the full life cycles of the running applications' dataflows. Such studies are likely made easier through graph-based representations such as those leveraged in DAG CEDR, but if coupled with program analysis frameworks or extended API methodologies that allow for submitting application subgraphs with each invocation of CEDR rather than single tasks at a time, there is also strong potential for such studies to be built on productive baselines like CEDR-API.

5.3 Extensible Software Representations

The most productive software framework is the one a programmer is most familiar with. Consequently, there is a large amount of future work that can be explored in the vein of enabling extensible software representations that allow users to bring their code to heterogeneous systems with as few modifications as possible. One path here is

to build extensible hooks from CEDR into the programming frameworks that users are familiar with (such as PyTorch [51], GNURadio [52], or Taskflow [53]) such that users can port their code with little effort by building bridges towards them. Another more ambitious path is to continue developing and integrating cutting-edge application analysis and partitioning tools – such as next-generation iterations of TraceAtlas [54] – that enable machine understanding of code.

5.4 Heterogeneity-Aware Compiler Design

As the needs of heterogeneous hardware have applied pressure on runtime designers to adjust for the complexities of heterogeneous resource management, runtimes now are now applying pressure back towards compiler designers to include all relevant information – such as data flow and dependency analysis or hardware-specific representations of application tasks – in their binaries to enable resource management policies to arbitrate effectively. Advancements in machine understanding of code will be critical in enabling progress here as they can be leveraged to perform tasks such as automatic kernel detection, runtime dependency analysis, or heterogeneity-aware compiler transformations of user applications. This pressure from runtimes to compilers calls for new approaches that can generalize platform-agnostic intermediate representation (IR) representations into representations that are tailored for the needs of the underlying heterogeneous management policies and hardware. Such advancements will enable closing the loop on the design of heterogeneous compilers, runtimes, and hardware. Only then will users of such systems be able to rapidly harness the computational efficiency afforded through heterogeneous architectures with the same ease as general purpose platforms.

5.5 Security of Heterogenous Systems

Runtime systems managing heterogeneous architectures creates a complex challenge in understanding and learning the underlying and representative characteristics of the entire system comprehensively. Therefore, security is a crucial consideration for architectures like DSAs, where the integration of diverse accelerators and specialized processors may introduce new vulnerabilities and attack surfaces that need to be addressed as discussed in a recent survey [55]. Differentiating normal and abnormal application behavior is particularly challenging in heterogeneous computing systems given the diverse execution characteristics influenced by factors such as varying programming models, diversity of Processing Elements (PEs), and dynamically changing resource allocation decisions [56]. CEDR offers ability to capture features systematically across the hardware, application, runtime, and scheduler layers. Although not explored yet, we believe that CEDR framework can be extended to expose the correlations among the system layers in a unified ecosystem. Furthermore, mechanisms, such as secure boot processes, data encryption, access control, and secure communication protocols should be carefully designed and implemented in DSAs to protect sensitive data and ensure system integrity.

5.6 Scalability Challenges

The runtime workflow executes continuously as CEDR management thread and for each application task a worker thread is spawned during the execution. The centralized worker-thread model poses as a limitation for CEDR for systems with large number of PEs. In this section we expose the scalability challenges and discuss a solution strategy.

The management thread parses application binaries, tracks the state of PEs, and monitors the execution of application tasks [12]. All tasks whose dependencies are resolved get placed in a ready queue as illustrated with Figure 16. When multiple tasks appear in the ready queue at the same time either due to arrival of multiple applications or concurrent tasks within an application, management of the ready queue is handled by mutexes labeled as M in Figure 16. For the tasks that are in the ready queue, the integrated scheduler determines the PE to invoke each task and places those tasks to the to-do queue of the assigned PE. Similarly, when multiple

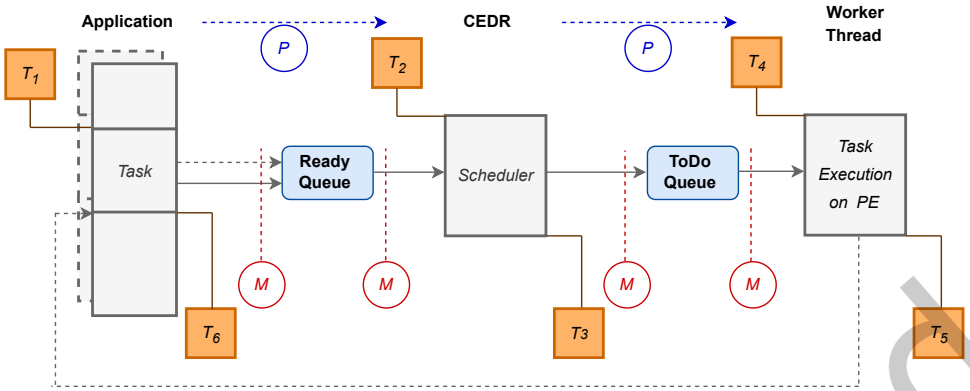

Fig. 16. CEDR worker thread execution flow.

tasks are scheduled, mutexes are required for the to-do queues as well. These mutexes serialize the execution. When a worker thread is spawned, it handles the execution and data flow management of that specific task on the designated PE. CEDR utilizes direct memory access (DMA) blocks and AXI4-Stream protocol [57]. The udmabuf [58] is used to setup contiguous buffers and handle data transfers between the host CPUs and PEs. In a CPU-core limited environment with large number of PEs and application tasks, the worker-thread model results with increased number of context switches to manage each spawned worker thread. Incorporating light-weight CPU cores tailored for thread management tasks only can reduce the burden on CPU cores that are designated as PEs [59]. However, the tradeoff between reduction in context switching and hardware overhead associated with the lightweight CPU cores is an open research problem.

Overall, while CEDR has made significant strides, the journey towards fully realizing the potential of domain-specific architectures continues, with ongoing research and innovation driving progress in overcoming these challenges.

Acknowledgments

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.

We are thankful for the continuous and generous support from the AMD University Program, including the donation of FPGA prototyping board used in this work.

References

[1] John L. Hennessy and David A. Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (1 2019), 48–60. DOI: http://dx.doi.org/10.1145/3282307

[2] Anish Krishnakumar, Umit Ogras, Radu Marculescu, Mike Kishinevsky, and Trevor Mudge. 2023. Domain-Specific Architectures: Research Problems and Promising Approaches. ACM Trans. Embed. Comput. Syst. 22, 2, Article 28 (jan 2023), 26 pages. DOI: http://dx.doi.org/10.1145/3563946

[3] Kasra Moazzemi, Biswadip Maity, Saehanseul Yi, Amir M. Rahmani, and Nikil Dutt. 2019. HESSLE-FREE: Heterogeneous Systems Leveraging Fuzzy Control for Runtime Resource Management. ACM Transactions on Embedded Computing Systems 18, 5s (Oct. 2019), 74:1–74:19. DOI: http://dx.doi.org/10.1145/3358203

[4] Cristiana Bolchini, Stefano Cherubin, Gianluca C. Durelli, Simone Libutti, Antonio Miele, and Marco D. Santambrogio. 2018. A Runtime Controller for OpenCL Applications on Heterogeneous System Architectures. SIGBED Rev. 15, 1 (March 2018), 29–35. DOI: http://dx.doi.org/10.1145/3199610.3199614

[5] Georgios Christodoulis, François Broquedis, Olivier Muller, Manuel Selva, and Frédéric Desprez. 2018. An FPGA target for the StarPU heterogeneous runtime system. In 2018 13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC). 1–8. DOI: http://dx.doi.org/10.1109/ReCoSoC.2018.8449373

[6] Xubin Tan, Jaume Bosch, Carlos Álvarez, Daniel Jiménez-González, Eduard Ayguadé, and Mateo Valero. 2019. A Hardware Runtime for Task-Based Programming Models. IEEE Transactions on Parallel and Distributed Systems 30, 9 (2019), 1932–1946. DOI: http://dx.doi.org/10.1109/TPDS.2019.2907493

[7] Jani Boutellier, Jiahao Wu, Heikki Huttunen, and Shuvra S. Bhattacharyya. 2018. PRUNE: Dynamic and Decidable Dataflow for Signal Processing on Heterogeneous Platforms. IEEE Transactions on Signal Processing 66, 3 (2018), 654–665. DOI: http://dx.doi.org/10.1109/TSP.2017.2773424

[8] Joshua Auerbach, David F. Bacon, Ioana Burcea, Perry Cheng, Stephen J. Fink, Rodric Rabbah, and Sunil Shukla. 2012. A compiler and runtime for heterogeneous computing. In DAC Design Automation Conference 2012. 271–276. DOI: http://dx.doi.org/10.1145/2228360.2228411 ISSN: 0738-100X.

[9] Chenying Hsieh, Ardalan Amiri Sani, and Nikil Dutt. 2019. 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). 136–141. DOI: http://dx.doi.org/10.1109/VLSI-SoC.2019.8920374 ISSN: 2324-8432.

[10] J. Mack, N. Kumbhare, A. NK, U. Y. Ogras, and A. Akoglu. 2020. User-Space Emulation Framework for Domain-Specific SoC Design. In 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). 44–53. DOI: http://dx.doi.org/10.1109/IPDPSW50202.2020.00016

[11] Joshua Mack, Sahil Hassan, Nirmal Kumbhare, Miguel Castro Gonzalez, and Ali Akoglu. 2023. CEDR: A Compiler-Integrated, Extensible DSSoC Runtime. ACM Trans. Embed. Comput. Syst. 22, 2, Article 36 (1 2023), 34 pages. DOI: http://dx.doi.org/10.1145/3529257

[12] Joshua Mack, Serhan Gener, Sahil Hassan, H. Umut Suluhan, and Ali Akoglu. 2023. CEDR-API: Productive, Performant Programming of Domain-Specific Embedded Systems. In 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). 16–25. DOI: http://dx.doi.org/10.1109/IPDPSW59300.2023.00016

[13] Umit Ogras, Joshua Mack, and Ali Akoglu. CEDR: A Novel Runtime Environment for Accelerator-Rich Heterogeneous Architectures. ([n. d.]). https://esweek.org/education-class/ec6 2023 Embedded Systems Week (ESWEEK), https://esweek.org/education-class/ec6.

[14] Joshua Mack, Serhan Gener, Ali Akoglu, Jacob Holtom, Alex Chiriyath, Chaitali Chakrabarti, Daniel Bliss, Anish Krishnakumar, Alper Goksoy, and Umit Ogras. 2022. GNU Radio and CEDR: Runtime Scheduling to Heterogeneous Accelerators. In Proceedings of the GNU Radio Conference, Vol. 7.

[15] L. Chang, J. Mack, B. Willis, X. Chen, J. Brunhaver, A. Akoglu, and C. Chakrabarti. 2022. Profile-Guided Parallel Task Extraction and Execution for Domain Specific Heterogeneous SoC. In 2022 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom). IEEE Computer Society, Los Alamitos, CA, USA, 913–920. DOI: http://dx.doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom57177.2022.00121

[16] Joshua Mack, Samet E. Arda, Umit Y. Ogras, and Ali Akoglu. 2022. Performant, Multi-Objective Scheduling of Highly Interleaved Task Graphs on Heterogeneous System on Chip Devices. IEEE Transactions on Parallel and Distributed Systems 33, 9 (2022), 2148–2162. DOI: http://dx.doi.org/10.1109/TPDS.2021.3135876

[17] Alexander Fusco, Sahil Hassan, Joshua Mack, and Ali Akoglu. 2022. A Hardware-based HEFT Scheduler Implementation for Dynamic Workloads on Heterogeneous SoCs. In 2022 IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC). 1–6. DOI: http://dx.doi.org/10.1109/VLSI-SoC54400.2022.9939623

[18] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. 2014. A Fully Pipelined and Dynamically Composable Architecture of CGRA. In 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines. 9–16. DOI: http://dx.doi.org/10.1109/FCCM.2014.12

[19] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A. Patterson, and Krste Asanović. 2017. BOOM v2: an open-source out-of-order RISC-V core. Technical Report UCB/EECS-2017-157. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.html

[20] Anish Krishnakumar, Hanguang Yu, Tutu Ajayi, A. Alper Goksoy, Vishrut Pandey, Joshua Mack, Sahil Hassan, Kuan-Yu Chen, Chaitali Chakrabarti, Daniel W. Bliss, Ali Akoglu, Hun-Seok Kim, Ronald G. Dreslinski, David Blaauw, and Umit Y. Ogras. 2024. FALCON: An FPGA Emulation Platform for Domain-Specific SoCs (DSSoCs). IEEE Design & Test 41, 1 (2024), 70–80. DOI: http://dx.doi.org/10.1109/MDAT.2023.3291331

[21] Timothy Roscoe. 2021. It's Time for Operating Systems to Rediscover Hardware. USENIX Association.
[22] Jeffrey D. Ullman. 1975. NP-complete scheduling problems. Journal of Computer and System sciences 10, 3 (1975), 384–393.
[23] A. Alper Goksoy, Sahil Hassan, Anish Krishnakumar, Radu Marculescu, Ali Akoglu, and Umit Y. Ogras. 2023. Theoretical Validation and Hardware Implementation of Dynamic Adaptive Scheduling for Heterogeneous Systems on Chip. Journal of Low Power Electronics and Applications 13, 4 (2023). DOI: http://dx.doi.org/10.3390/jlpea13040056
[24] A. Alper Goksoy, Anish Krishnakumar, Md Sahil Hassan, Allen J. Farcas, Ali Akoglu, Radu Marculescu, and Umit Y. Ogras. 2022. DAS: Dynamic Adaptive Scheduling for Energy-Efficient Heterogeneous SoCs. IEEE Embedded Systems Letters 14, 1 (2022), 51–54. DOI: http://dx.doi.org/10.1109/LES.2021.3110426
[25] H. Topcuoglu and S. Hariri. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. 13, 3 (2002), 260–274. DOI: http://dx.doi.org/10.1109/71.993206 00000.
[26] L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira. 2010. DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish Time Algorithm. In 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing. 27–34. DOI: http://dx.doi.org/10.1109/PDP.2010.56
[27] H. Arabnejad and J. G. Barbosa. 2014. List Scheduling Algorithm for Heterogeneous Systems by an Optimistic Cost Table. IEEE Transactions on Parallel and Distributed Systems 25, 3 (2014), 682–694. DOI: http://dx.doi.org/10.1109/TPDS.2013.57
[28] Naqin Zhou, Deyu Qi, Xinyang Wang, Zhishuo Zheng, and Weiwei Lin. 2017. A list scheduling algorithm for heterogeneous systems based on a critical node cost table and pessimistic cost table. Concurrency and Computation: Practice and Experience 29, 5 (2017), e3944.
[29] G. Xie, G. Zeng, X. Xiao, R. Li, and K. Li. 2017. Energy-Efficient Scheduling Algorithms for Real-Time Parallel Applications on Heterogeneous Distributed Embedded Systems. IEEE Transactions on Parallel and Distributed Systems 28, 12 (12 2017), 3426–3442. DOI: http://dx.doi.org/10.1109/TPDS.2017.2730876
[30] M. F. Akbar, E. U. Munir, M. M. Rafique, Z. Malik, S. U. Khan, and L. T. Yang. 2016. List-Based Task Scheduling for Cloud Computing. In 2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). 652–659. DOI: http://dx.doi.org/10.1109/iThings-GreenCom-CPSCom-SmartData.2016.143
[31] Anish Krishnakumar, Samet E. Arda, A. Alper Goksoy, Sumit K. Mandal, Umit Y. Ogras, Anderson L. Sartor, and Radu Marculescu. 2020. Runtime Task Scheduling Using Imitation Learning for Heterogeneous Many-Core Systems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39, 11 (2020), 4064–4077. DOI: http://dx.doi.org/10.1109/TCAD.2020.3012861
[32] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource Management with Deep Reinforcement Learning. In Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16). Association for Computing Machinery, New York, NY, USA, 50–56. DOI: http://dx.doi.org/10.1145/3005745.3005750
[33] TraceAtlas. https://github.com/ruhrie/TraceAtlas/. ([n. d.]). accessed date: Jan. 20, 2020.
[34] Vivek Yadav. Small U-Net for vehicle detection, github.com/vxy10/p5_VehicleDetection_Unet. ([n. d.]). Retrieved June 5, 2023 from github.com/vxy10/p5_VehicleDetection_Unet Accessed: 2023-06-05.
[35] Imagenette Dataset, github.com/fastai/imagenette. ([n. d.]). Retrieved June 5, 2023 from github.com/fastai/imagenette Accessed: 2023-06-05.
[36] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In 2009 IEEE conference on computer vision and pattern recognition. Ieee, 248–255.
[37] P. Warden. 2018. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. ArXiv e-prints (April 2018). arXiv:cs.CL/1804.03209 https://arxiv.org/abs/1804.03209
[38] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html
[39] CEDR Tutorial Set https://github.com/UA-RCL/CEDR/blob/tutorial/CEDR_tutorial.md . ([n. d.]). Retrieved January 31, 2024 from https://github.com/UA-RCL/CEDR/blob/tutorial/CEDR_tutorial.md
[40] Joshua Mack, Sahil Hassan, and Ali Akoglu. CEDR: A Holistic Software and Hardware Design Environment for FPGA-Integrated Heterogeneous Systems. ([n. d.]). https://www.isfpga.org/workshops-tutorials/#t8 Tutorial: 2024 International Symposium on Field Programmable Gate Arrays, March 3-5, 2024, Monterey, CA, https://www.isfpga.org/workshops-tutorials/#t8.
[41] Bryan Donyanavard, Tiago Mück, Santanu Sarma, and Nikil Dutt. 2016. SPARTA: Runtime task allocation for energy efficient heterogeneous manycores. In 2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS). 1–10.
[42] Bryan Donyanavard, Tiago Mück, Amir M. Rahmani, Nikil Dutt, Armin Sadighi, Florian Maurer, and Andreas Herkersdorf. 2019. SOSA: Self-Optimizing Learning with Self-Adaptive Control for Hierarchical System-on-Chip Management. In Proceedings of the 52nd Annual

IEEE/ACM International Symposium on Microarchitecture (MICRO '52). Association for Computing Machinery, New York, NY, USA, 685–698. DOI: http://dx.doi.org/10.1145/3352460.3358312

[43] Biswadip Maity, Bryan Donyanavard, Anmol Surhonne, Amir Rahmani, Andreas Herkersdorf, and Nikil Dutt. 2021. SEAMS: Self-Optimizing Runtime Manager for Approximate Memory Hierarchies. ACM Transactions on Embedded Computing Systems 20, 5 (July 2021), 48:1–48:26. DOI: http://dx.doi.org/10/gm3hnz

[44] André Luís del Mestre Martins, Alzemiro Henrique Lucas da Silva, Amir M. Rahmani, Nikil Dutt, and Fernando Gehm Moraes. 2019. Hierarchical adaptive Multi-objective resource management for many-core systems. Journal of Systems Architecture 97 (2019), 416–427. DOI: http://dx.doi.org/10.1016/j.sysarc.2019.01.006

[45] Chenying Hsieh, Ardalan Amiri Sani, and Nikil Dutt. 2019. 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). 136–141. DOI: http://dx.doi.org/10.1109/VLSI-SoC.2019.8920374 ISSN: 2324-8432.

[46] Carsten Heinz, Jaco Hofmann, Jens Korinth, Lukas Sommer, Lukas Weber, and Andreas Koch. 2021. The TaPaSCo Open-Source Toolflow. J. Signal Process. Syst. 93, 5 (May 2021), 545–563. DOI: http://dx.doi.org/10.1007/s11265-021-01640-8

[47] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience 23, 2 (2011), 187–198.

[48] Cristiana Bolchini, Stefano Cherubin, Gianluca C. Durelli, Simone Libutti, Antonio Miele, and Marco D. Santambrogio. 2018. A Runtime Controller for OpenCL Applications on Heterogeneous System Architectures. SIGBED Rev. 15, 1 (March 2018), 29–35. DOI: http://dx.doi.org/10.1145/3199610.3199614

[49] Jungwon Kim, Seyong Lee, Beau Johnston, and Jeffrey S. Vetter. 2021. IRIS: A Portable Runtime System Exploiting Multiple Heterogeneous Programming Systems. In 2021 IEEE High Performance Extreme Computing Conference (HPEC). 1–8. DOI: http://dx.doi.org/10.1109/HPEC49654.2021.9622873

[50] Giorgos Vasiliadis, Rafail Tsirbas, and Sotiris Ioannidis. 2022. 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). 55–64. DOI: http://dx.doi.org/10.1109/IPDPSW55747.2022.00017

[51] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Advances in Neural Information Processing Systems, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[52] GNU Radio Website, https://www.gnuradio.org. ([n. d.]). Retrieved January 31, 2024 from https://www.gnuradio.org

[53] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. IEEE Transactions on Parallel and Distributed Systems 33, 6 (2022), 1303–1320. DOI: http://dx.doi.org/10.1109/TPDS.2021.3104255

[54] Richard Uhrie, Chaitali Chakrabarti, and John Brunhaver. 2020. Automated Parallel Kernel Extraction from Dynamic Application Traces. (2020). arXiv:2001.09995

[55] Hoda Naghibijouybari, Esmaeil Mohammadian Koruyeh, and Nael Abu-Ghazaleh. 2022. Microarchitectural Attacks in Heterogeneous Systems: A Survey. *Comput. Surveys* 55, 7 (2022), 1–40.

[56] Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. 2019. Anomaly Detection and Classification using Distributed Tracing and Deep Learning. In 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). 241–250. DOI: http://dx.doi.org/10.1109/CCGRID.2019.00038

[57] ARM AMBA. AXI4-stream protocol specification. Volume IHI 51A ([n. d.]).

[58] udmabuf. Udmabuf, A Userspace mappable DMA Buffer, https://github.com/ikwzm/udmabuf. ([n. d.]). Retrieved October 13, 2021 from https://github.com/ikwzm/udmabuf

[59] H. Umut Suluhan, Serhan Gener, Fusco Alexander, Joshua Mack, Ismet Dagli, Mehmet Belviranli, Cagatay Edemen, and Ali Akoglu. 2024. A Runtime Manager Integrated Emulation Environment for Heterogeneous SoC Design with RISC-V Cores. In 2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). 1–8.

Received 2 April 2024; revised 10 July 2024; accepted 24 July 2024