Future Generation Computer Systems 176 (2026) 108184

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

A portable framework with generalized runtime features for task graph execution and concurrent multi-application deployment on heterogeneous systems

Serhan Gener [a,*], Sahil Hassan [a], H. Umut Suluhan [a], Liangliang Chang [b], Chaitali Chakrabarti [b], Tsung-Wei Huang [c], Umit Ogras [c], Ali Akoglu [a,*]

[a] University of Arizona, 1230 E Speedway Blvd, Tucson, 85719, Arizona, USA
[b] Arizona State University, 650 E Tyler Mall, Tempe, 85281, Arizona, USA
[c] University of Wisconsin at Madison, 500 Lincoln Dr, Madison, 53706, Wisconsin, USA

ARTICLE INFO

Keywords:
Auto parallelization
Dynamic scheduling
Heterogeneous runtime

ABSTRACT

Heterogeneous computing platforms are widely adopted to meet the diverse performance demands of modern applications, but they present a key challenge of balancing programmability with performance portability. This work introduces a unified and portable framework that addresses this trade-off by integrating the CEDR runtime system with the Taskflow task-graph programming model. By combining CEDR's dynamic scheduling with Taskflow's parallelization capabilities, our system simplifies application development while enabling efficient execution across CPUs, GPUs, and FPGAs. We demonstrate its portability on a broad range of heterogeneous SoC and HPC-scale systems. The framework supports concurrent execution of multiple applications through a centralized coordination layer, overcoming limitations of isolated execution contexts and achieving up to 1.23× speedup on highly heterogeneous platforms. Our solution further generalizes runtime features such as streaming input handling and cached scheduling across applications, yielding up to 6.08× improvements in execution time and a 29.60× reduction in scheduling overhead. Experimental evaluations demonstrate consistent improvements in execution time, validating the effectiveness of this integrated and extensible design.

1. Introduction

Modern computing platforms are increasingly built around heterogeneous architectures, combining CPUs with specialized accelerators such as GPUs, FPGAs, and domain-specific processors. This architectural trend spans from large-scale High-Performance Computing (HPC) systems to compact System-on-Chip (SoC) platforms, offering an opportunity to match diverse application demands to the most suitable compute resources. However, the benefits achieved by heterogeneity are limited due to issues with the higher complexity of application development and runtime management, particularly in managing dynamic workloads that compete for shared heterogeneous resources.

To address these challenges, recent efforts have introduced runtime systems that abstract low-level hardware details and dynamically assign workloads to appropriate processing elements (PEs) based on system state and resource availability [1–3]. One such system, the Compiler-Integrated Extensible DSSoC Runtime (CEDR) [3,4], provides a hardware-agnostic, API-based programming model that enables dynamic scheduling across CPUs, FPGAs, and GPUs. Fig. 1 illustrates the transformations in a sample application ① with Fast Fourier Transform (FFT) and Complex Vector Multiplication (Mult) operations. In Fig. 1 ②, nodes labeled FFT correspond to CEDR API calls, while nodes labeled Mult represent non-API computations. While the Mult computation can be expressed as an API and executed on accelerators such as FPGAs or GPUs, in this discussion, it is treated as a non-API operation to simplify the illustration of the challenges faced by such applications. CEDR efficiently utilizes PEs for the parallel execution of API-mapped (FFTs) regions of the application across heterogeneous PEs. Since it provides limited visibility into the application's full control flow, it misses optimization opportunities for non-API (Mults) regions, such as user-defined loops.

To improve coverage of such regions and capture additional parallelism, task-graph computing frameworks like Taskflow [5] have emerged as a complementary approach. These frameworks promote fine-grained parallelism by requiring developers to express applications as directed acyclic graphs (DAGs) of dependent tasks with minimal

* Corresponding authors.
E-mail addresses: gener@arizona.edu (S. Gener), sahilhassan@arizona.edu (S. Hassan), suluhan@arizona.edu (H.U. Suluhan), lchang21@asu.edu (L. Chang), chaitali@asu.edu (C. Chakrabarti), tsung-wei.huang@wisc.edu (T.-W. Huang), uogras@wisc.edu (U. Ogras), akoglu@arizona.edu (A. Akoglu).

https://doi.org/10.1016/j.future.2025.108184
Received 30 June 2025; Received in revised form 22 September 2025; Accepted 29 September 2025
Available online 13 October 2025

modifications to existing application code. As shown in Fig. 1 ③, Taskflow exposes additional parallelism opportunities (e.g., Mult) for regions of code where task dependencies expose parallelism opportunities across CPU cores. However, these frameworks typically require static task-to-resource mapping and lack dynamic scheduling mechanisms, which can lead to inefficient resource utilization, especially when multiple workloads with overlapping compute demands run concurrently on the system.

Our prior work [6] introduces a novel integration of Taskflow and CEDR, bridging these frameworks to leverage their respective strengths and enabling applications to benefit from both dynamic scheduling and holistic task graph representations. As illustrated in Fig. 1 ④, the combined system enables a single application to exploit fine-grained parallelism and adapt at runtime to the system's state while remaining portable across heterogeneous SoC platforms. However, real-world systems often execute multiple applications concurrently, each with its own set of tasks and performance requirements. When such applications run independently, traditional runtime approaches instantiate separate execution contexts per application, each operating with only a local view of CPU resources, as shown in Fig. 1 *Single Application View*. This lack of a global view of the system prevents coordinated scheduling, increases contention, and reduces system-wide efficiency. In this work, we extend our initial integration efforts to support coordinated multi-application execution through a shared execution layer as illustrated in Fig. 1 *Multi Application View*, enabling centralized task scheduling across all applications while preserving their individual control flows.

This coordination is achieved without requiring developers to refactor application logic or manually synchronize across workloads. By maintaining a single global visibility into all executing applications, our extended runtime framework enhances overall platform utilization and balances CPU workloads. This paper presents the design, implementation, and evaluation of an enhanced runtime framework that offers a generalizable and portable solution for managing concurrent applications on modern heterogeneous platforms. A preliminary version of this work was presented at the Fourth International Workshop on Extreme Heterogeneity Solutions (ExHET 2025), held in conjunction with PPoPP 2025, where we focused on supporting a single-application execution model. In this extended version of our preliminary work, we enable concurrent execution of multiple applications (beyond the prior single-application model), generalize runtime features such as streaming input handling and cached scheduling to work seamlessly across applications, and enhance portability by validating on diverse platforms, including an HPC-scale system with CPUs, GPUs, and FPGAs. The overall technical contributions of our work are as follows:

- We present a unified runtime-task programming framework that integrates dynamic scheduling with task-graph-based application modeling, enabling hardware-agnostic and platform-portable execution across CPUs, GPUs, and FPGAs without requiring changes to application code.
- We propose a general, reusable methodology for interfacing runtime systems with task-based programming frameworks, facilitating seamless, modular, and scalable integration.
- We introduce advanced runtime features such as streaming input execution and cached scheduling to improve adaptability and performance.
- We enable concurrent and coordinated multi-application execution through a centralized coordination layer that provides a global system view and, in turn, improves CPU core utilization and eliminates inefficiencies associated with isolated execution contexts.
- We design global, cross-application scheduling mechanisms that eliminate the need for manual inter-process coordination, allowing applications to be managed through a unified runtime control flow.
- We demonstrate the effectiveness of the proposed framework across a range of platforms, from embedded SoCs to HPC-scale heterogeneous systems, achieving consistent improvements in workload efficiency, including performance gains of up to 1.23× for signal processing workloads on the most heterogeneous system.

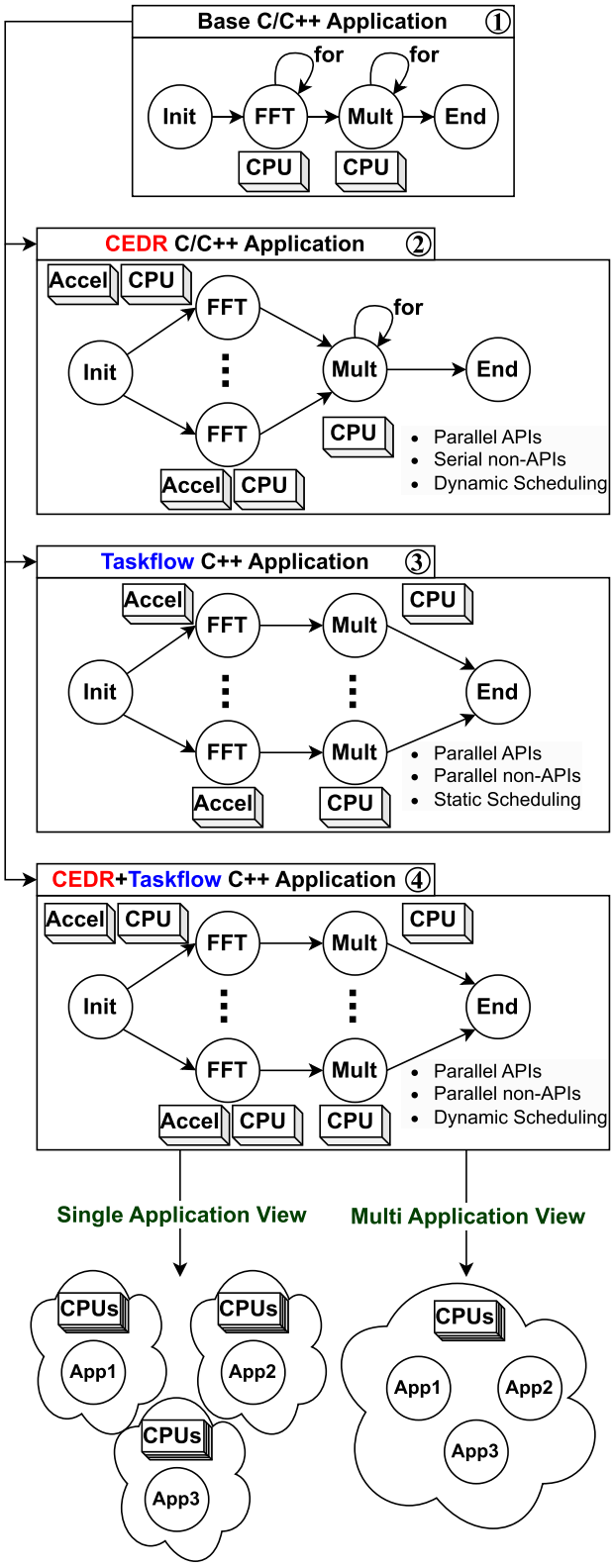


Fig. 1. CEDR and Taskflow integration flow for a base C/C++ application.

The remainder of this paper is organized as follows. Section 2 provides background on runtime systems and programming models, with a focus on their trade-offs. Section 3 presents the integration of the CEDR

Table 1
Overview of key components and APIs.

Position	Name	Type	Description
Application	*CEDR_**	API	CEDR APIs for task execution
	for_each_index	API	Taskflow API for parallel loop
	emplace	API	Taskflow API for standalone single tasks
Taskflow	`tf::Taskflow`	Class	Taskflow semantic for DAG
	`tf::Executor`	Class	Taskflow semantic for executing the DAG
CEDR	*CEDR_DAG_EXTRACT*	New API	CEDR API for extracting the DAG from tf::Taskflow
	CEDR_RUN_DAG	New API	CEDR API for executing the DAG from tf::Taskflow
	`cedr_task_config`	*struct* Variable	Optional CEDR variable storing configuration information for CEDR APIs

runtime and the Taskflow framework, including application preparation steps, runtime coordination mechanisms, and communication protocols. Section 4 describes the experimental setup, and Section 5 evaluates the proposed system across a variety of heterogeneous platforms and workloads, including single- and multi-application scenarios, mixed workloads, and HPC-scale experiments. Section 6 reviews related efforts in heterogeneous task scheduling and runtime design integrations. Finally, Section 7 concludes the paper.

2. Background

A runtime system with a built-in scheduler is crucial for making task-to-PE mapping decisions in real time, especially when workloads arrive dynamically and target a heterogeneous set of PEs. Multiple runtime systems [1–4,7–13] have been developed to support dynamic resource management. These systems aim to maximize performance by adapting to runtime resource availability and workload variability while leveraging heterogeneity at the PE level.

The design and usability of runtime systems strongly influence their adoption. Most systems follow one of two programming models. The first model employs an application programming interface (API)-based approach [3,13], where developers utilize predefined functions to initiate tasks. This model favors ease of use and simplifies application development but restricts control over task dependencies and execution sequencing. The second model organizes applications as directed acyclic graphs (DAGs) of tasks [1,4,12], enabling the runtime system to observe the application's global structure. This visibility allows more fine-grained optimizations and flexible scheduling. However, it also shifts the burden to developers, who must explicitly define task dependencies and execution flows, which complicates support for applications with dynamic control flow, where task sequences are determined at runtime and conditional branches and dependencies are created adaptively rather than statically represented in a traditional application DAG.

Among the DAG-based programming models, Taskflow [5] is a prominent state-of-the-art open-source parallel programming model that tackles the challenges of expressing task-based parallelism by automatically generating the task dependency graph within an application. Unlike traditional task-based frameworks [14–19], Taskflow features a lightweight runtime that efficiently scales parallel execution across many processors. Although Taskflow effectively exposes parallelism, it relies on static task-to-PE mappings defined by the developer. This static mapping limits its ability to adapt to dynamic workload variations or heterogeneous execution environments. For instance, when multiple applications assign the same type of task to a single PE type, such as a hardware accelerator, that resource can become oversubscribed–even if other PEs remain idle and are equally qualified to execute the task. To address this issue, we integrate Taskflow with a runtime system that dynamically schedules tasks to PEs capable of running them (e.g., CPUs and supported accelerators), allowing for more efficient resource utilization and improved performance under dynamic workloads.

Although several runtime systems support either API-based or DAG-based programming models, there are very few systems that are designed around both. In order to combine the ease of programming of API-based models with DAG-based detailed application representation, this work integrates Taskflow with the open-source Compiler-integrated, Extensible DSSoC Runtime (CEDR) [3,4]. CEDR stands out by offering both programming models, initially introducing DAG-based support [4] and later extending its capabilities to include an API-based interface [3]. With its hardware-agnostic API, CEDR enables developers to build, compile, and deploy applications seamlessly on heterogeneous platforms. CEDR runs as a daemon process on Linux-based systems, ensuring portability across a broad range of platforms and reducing the effort needed to migrate between them. Each submitted application is dynamically loaded by the runtime daemon, which spawns a dedicated thread to run the application's main function. This thread-based execution model allows multiple applications to coexist within a single runtime instance. Beyond portability, CEDR supports a broad range of programming environments–including GNURadio [20] and PyTorch [21]–and is capable of targeting multiple hardware architectures, such as x86, RISC-V [22], FPGAs, GPUs, and ARM-based SoCs [4,23]. These features make CEDR a versatile runtime solution for heterogeneous computing. The following section details our integration approach.

3. CEDR-taskflow integration

Both CEDR and Taskflow expose their APIs through header-based interfaces, allowing developers to begin the integration process by including both headers in the application source file. However, the integration involves more than simply invoking existing APIs to ensure seamless application deployment. To enable interaction between the two systems, the application must also transfer the DAG generated by Taskflow to CEDR for runtime execution. Table 1 summarizes the key components and APIs-both reused and newly introduced in Taskflow and CEDR that support this integration. Using a subset of these components, Listing 3 illustrates how to combine CEDR and Taskflow within a C/C++ application. This integration flow not only unifies the capabilities of both frameworks but also showcases the performance and programmability benefits of their combined use.

3.1. Application preparation

We begin with a base application that does not use either CEDR or Taskflow APIs. Listing 1 shows a simple C/C++ program that executes 512 128-point FFTs-referred to as the *FFT for loop* (lines 9–18) or API region-followed by 512 128-point vector multiplications, for the purposes of this example, referred to as the *Multiplication for loop* (lines 19–27) or non-API region. To add CEDR support to this application source code, we highlight the changes made to the base C/C++ code with red in Listing 2, which involves including the `libcedr.h` header (line 1) and replacing the original *FFT* function with the hardware-agnostic CEDR API call *CEDR_FFT* on line 14.

Listing 1
Base C/C++ Code

```
...
int start=0,end=512,size=128;
bool forward=true;
complex input=allocate(512);
complex output=allocate(512);

// FFT for loop



for(int i=start;i<end;i++){
  FFT(input[i],
    output[i],
    size,
    forward);
}
// Multiplication for loop



for(int i=start;i<end;i++){
  for(int j=0;j<size;j++){
    output[i][j] = output[i][j] *
          2;
  }
}



...
deallocate(input);
deallocate(output);
```

Listing 2
C/C++ Code Using CEDR

```
#include <libcedr.h>

...
int start=0,end=512,size=128;
bool forward=true;
complex input=allocate(512);
complex output=allocate(512);

// FFT for loop



for(int i=start;i<end;i++){
  CEDR_FFT(input[i],
    output[i],
    size,
    forward);
}
// Multiplication for loop



for(int i=start;i<end;i++){
  for(int j=0;j<size;j++){
    output[i][j] = output[i][j] *
          2;
  }
}



...
deallocate(input);
deallocate(output);
```

Listing 3
C++ Code Using CEDR and Taskflow

```
#include <libcedr.h>
#include <taskflow.hpp>
...
int start=0,end=512,size=128;
bool forward=true;
complex input=allocate(512);
complex output=allocate(512);
tf::Taskflow taskflow;
// FFT for loop
task0=taskflow.for_each_index
  (ref(start), ref(end), 1,
  [input, &output, size,
  forward](int i){
  CEDR_FFT(input[i],
    output[i],
    size,
    forward);
});
// Multiplication for loop
task1=taskflow.for_each_index
  (ref(start), ref(end), 1,
  [&output, size]
  (int i){
  for(int j=0;j<size;j++){
    output[i][j] = output[i][j] *
          2;
  }
});
task0.precede(task1);
tf::Executor executor;
executor.run(taskflow).wait();
...
deallocate(input);
deallocate(output);
```

Next, we incorporate Taskflow constructs into the application, as shown in Listing 3. We include the `taskflow.hpp` header on line 2 and replace the original *for* loop of API-region with Taskflow's *for_each_index* construct on line 10, which parallelizes the loop using specified `start`, `end`, and increment values (set to 1 in this case). The loop index `i`, defined on line 13, is used inside a lambda function that captures necessary variables. We capture any variables that require updates (e.g., `output`) by reference, as shown in line 12. With these modifications, the *FFT for loop* (lines 9-18) now includes both CEDR and Taskflow API support. The *Multiplication for loop* (lines 19-27 in Listing 1) remains unchanged with CEDR-specific modifications in this example for clarity. In practice, Mult can also be exposed as a CEDR API and executed on CPU, GPU, or FPGA resources, but here we illustrate it as CPU-only to keep the example focused and easy to follow. With the lack of API support, it remains unchanged with CEDR-specific modifications as shown in Listing 2, but is parallelized when we integrate Taskflow constructs. We apply the same *for_each_index* modification to the loop on line 23 in Listing 3, allowing Taskflow to handle the parallel execution of this non-API region that CEDR lacks support for.

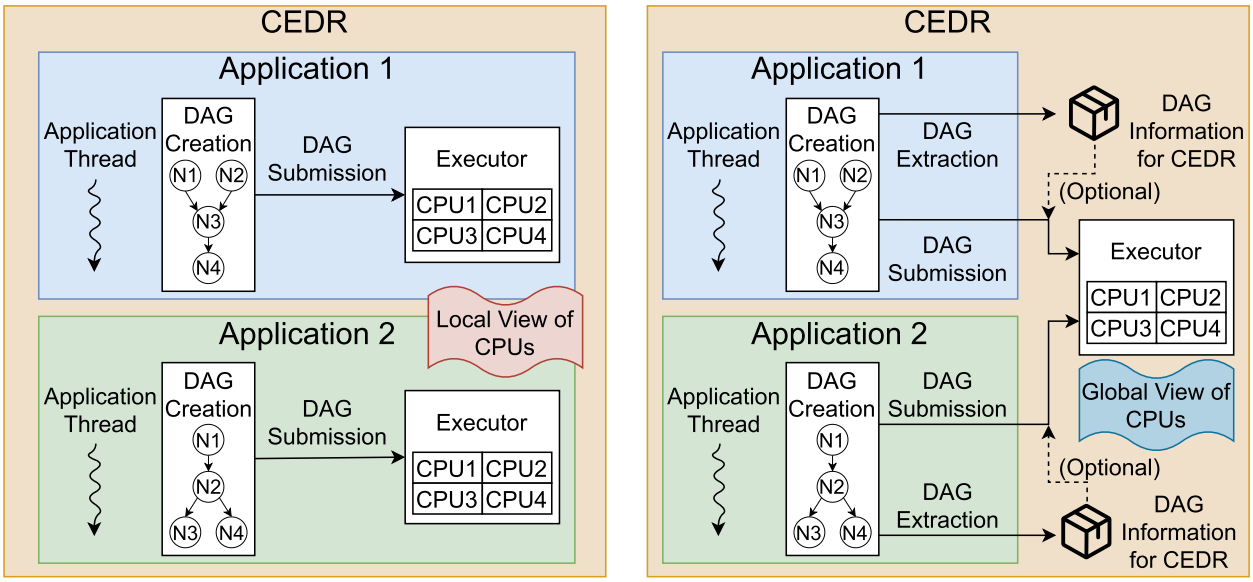
For applications containing tasks outside of a *for* loop, the Taskflow API *emplace* can be used instead of *for_each_index*. After creating tasks, we initialize a `tf::Executor` instance, which manages the execution of the task flow graph, represented by the `tf::Taskflow` class. We construct the task flow graph by defining tasks (e.g., `task0` and `task1` on lines 10 and 20 of Listing 3) and connecting them using predecessor-successor relationships (line 27 in Listing 3). Once the graph is ready, we submit it to the `tf::Executor` instance (line 30 in Listing 3), which starts execution from the head node. Although Taskflow offers many advanced features, the features utilized in the presented examples involve the core APIs that are essential for our integration effort with CEDR.

In this combined programming model shown in Listing 3, CEDR's APIs are placed within Taskflow tasks. When the `tf::Executor` begins executing the task flow graph, any task node that invokes a CEDR API triggers a call to CEDR, which then schedules and executes the task on a suitable PE based on its scheduling policy. This dynamic runtime mechanism eliminates the need to assign tasks to specific PEs statically during graph construction. During application execution, Taskflow's `tf::Executor` handles thread-to-CPU-core assignments for parallel execution of tasks. At the same time, CEDR manages the dispatch and execution of tasks across CPUs, FPGA-based accelerators, and GPUs when the task involves a CEDR API call. With this integration, applications continue to benefit from CEDR's capability of efficiently utilizing heterogeneous platforms, while leveraging Taskflow's DAG representation and capability to optimize/parallelize task thread execution onto CPU cores.

While Listings 1–3 illustrate the porting process for a C/C++ baseline, the same flow applies to applications originally written with frameworks such as CUDA, HIP, or OpenCL. In those cases, only the underlying kernel calls (e.g., FFT) change, while the integration with CEDR and Taskflow follows the same steps.

3.2. CEDR and taskflow communication

With the CEDR and Taskflow APIs now integrated into the application, the next step is to enable CEDR to acquire the DAG constructed by Taskflow. To support this, we introduce the *CEDR_DAG_EXTRACT*



(a) Each application maintains its own view of available CPUs, leading to isolated scheduling decisions.

(b) A shared global view of CPUs is maintained by CEDR, allowing coordinated scheduling across applications.

Fig. 2. Comparison of CEDR runtime execution with local and global CPU views for multi-application execution. (a) Each application maintains its own view of available CPUs, leading to isolated scheduling decisions. (b) A shared global view of CPUs is maintained by CEDR, allowing coordinated scheduling across applications.

API within CEDR, which allows Taskflow to transmit its generated DAG to the runtime. This API has two arguments: (1) the graph produced by Taskflow (`tf::Taskflow`) and (2) a configuration map that provides additional metadata about each graph task. This optional configuration, `cedr_task_config`, allows users to embed pre-scheduling task-to-PE assignment decisions for specific APIs during *CEDR_DAG_EXTRACT* call if desired. CEDR uses this information to identify and execute task nodes that involve CEDR API calls. Once invoked, *CEDR_DAG_EXTRACT* translates the Taskflow structure into CEDR's native DAG format [4] and performs an initial scheduling step for any task nodes involving CEDR APIs. If only a single application is running, users can apply this scheduling result directly by passing the `cedr_task_config` argument to CEDR API calls. Otherwise, CEDR continues with its default runtime scheduling strategy, where each API call is scheduled at the time the executing thread issues it.

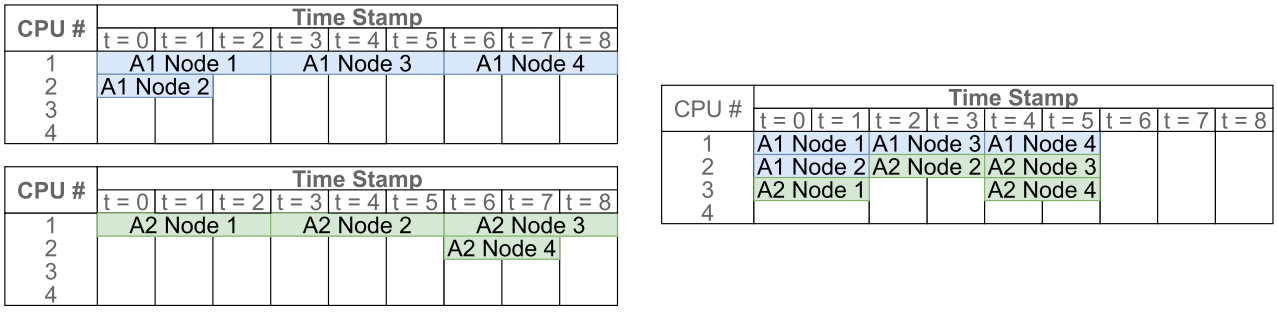
After extracting the DAG using *CEDR_DAG_EXTRACT*, the application can execute it using `tf::Executor`, following the standard Taskflow execution model. In our prior work [6], we introduced *CEDR_RUN_DAG*, an API that transfers the creation of `tf::Executor` within the CEDR runtime, which also enables repeated execution of a single application's DAG directly within CEDR. Specifically, modifying the application in Listing 3 to utilize this API would involve replacing lines 29 and 30 with *CEDR_RUN_DAG(taskflow, repeat_count)*, with a `repeat_count` of one. This design eliminated redundant graph initialization and supported efficient stream-based processing. However, the prior *CEDR_RUN_DAG* model was still limited to managing execution of `a single application`.

When multiple applications call *CEDR_RUN_DAG* function, each one creates its own `tf::Executor` instance as illustrated in Fig. 2a. Similar to Taskflow [5], frameworks such as Intel TBB [14] and OpenMP [24] also adopt this model. Each runtime initializes a private thread pool, performs work stealing only within that pool, and assumes exclusive ownership of the available CPU cores. While this design is effective for single-application execution, the concurrent execution of multiple applications fragments CPU visibility, oversubscribes cores, and results in independent scheduling decisions that are unaware of the global system state. For example, Taskflow executors cannot coordinate across processes. TBB employs per-process task arenas but still lacks inter-application resource sharing. OpenMP constructs pools based on user-specified environment variables or defaults, without global coordination. Consequently, all of these frameworks operate with only a local view of the CPU pool, which restricts system-wide load balancing and increases context switching overhead, leading to oversubscription, underutilization, and degraded overall performance.

To address these challenges, we designed new runtime mechanisms in CEDR that: (1) extend Taskflow's work-stealing queues to allow cross-application task stealing, and (2) enforce global CPU scheduling policies that ensure fairness and prevent starvation across applications. Unlike Linux daemons or process-level services, which rely on coarse-grained context switching or inter-process communication (IPC) mechanisms, our solution coordinates tasks at fine granularity within the runtime itself. This design provides a globally visible execution context where multiple DAGs can coexist and be dynamically balanced, significantly reducing oversubscription and improving overall throughput.

Implementing this model requires decoupling DAG creation from DAG execution. Applications generate DAGs in their own threads, but rather than binding them to local executors (Fig. 2a), they submit them through the CEDR APIs (*CEDR_RUN_DAG*) to a centralized `tf::Executor` (Fig. 2b), allowing the runtime to view all application DAGs simultaneously and make coordinated scheduling decisions across CPUs. This API-level modification ensures applications can remain agnostic to whether DAGs are executed locally or centrally.

In order to manage the execution of multiple applications, we extend our prior work by enabling `coordinated multi-application execution` under a shared runtime-managed `tf::Executor`. From the user perspective, applications invoke *CEDR_RUN_DAG* in the same manner as before, but the runtime automatically binds them to the global executor rather than creating per-application executors. Unlike the previous version of *CEDR_RUN_DAG* API, instead of instantiating separate executors for each application, CEDR now orchestrates the execution of multiple task graphs from distinct applications using a single globally visible `tf::Executor` as illustrated in Fig. 2b. This change enables Taskflow to monitor system-wide CPU load, perform cross-application task stealing, minimize context switching for CPUs, and dynamically balance execution across all active workloads.



(a) Applications 1 and 2 are scheduled independently, causing oversubscription and longer per-application task executions across CPUs.

(b) Applications 1 and 2 are scheduled with awareness of shared resources, resulting in more balanced and efficient execution across CPUs.

Fig. 3. Execution timelines of two applications under local and global CPU views. (a) Applications 1 and 2 are scheduled independently, causing oversubscription and longer per-application task executions across CPUs. (b) Applications 1 and 2 are scheduled with awareness of shared resources, resulting in more balanced and efficient execution across CPUs.

In summary, in addition to supporting streaming-style execution using repeated execution capability, the globally visible `tf::Executor` model enables cross-application scheduling policies and runtime mechanisms that unify execution under a shared coordination layer. In existing systems, coordinating multiple task graphs across concurrently running applications typically requires developers to construct IPC mechanisms, MPI [25] structures, or rely on external libraries such as *Boost.Process* [26] to ensure that a common execution context manages all tasks. These workarounds are both tedious for the application developer and error-prone during execution. By centrally managing multiple graphs under CEDR's runtime, our approach preserves Taskflow's fine-grained CPU scheduling and extends it across application boundaries, enabling globally informed task execution coordination the need for explicit IPC between applications or manual application-level refactoring. This coordination remains fully portable, requiring no hardware-specific assumptions, and can be seamlessly deployed across heterogeneous SoC platforms.

As an illustrative example of how the applications in Fig. 2 execute on a system with four CPU cores, we present a Gantt chart of DAG node executions in Fig. 3. Each node is modeled to complete in two time steps when it has exclusive access to a core. When multiple nodes contend for the same CPU, additional overhead from context switching and resource contention extends their completion time to three steps. This simplified model captures the imbalance that arises when executors operate without global coordination.

The executor assigns nodes sequentially to the lowest-numbered available CPU, mirroring a greedy assignment policy behavior when each executor operates independently with a local CPU view. In Fig. 3a, where each application maintains its own executor, this local view results in imbalanced scheduling, where two nodes are mapped to CPU1, one to CPU2, while CPUs 3 and 4 remain idle. At timestamp 3, for example, Application 1's Node 3 and Application 2's Node 2 are both placed on CPU1, extending their execution due to contention. A similar situation arises in the final region (timestamps 6-8), which mirrors the imbalance observed in the initial region (timestamps 0-2). Multiple nodes accumulate on CPU1, while a single node is on CPU2, leaving other CPUs underutilized and prolonging execution.

In contrast, Fig. 3b shows the case with a single executor and a global CPU view. Here, nodes are distributed across idle CPUs with awareness of cross-application workloads. By eliminating redundant executor instances and coordinating at the runtime level, the global executor prevents oversubscription on a subset of CPUs and achieves higher system-wide utilization. As a result, execution finishes by timestamp 5 instead of 8, with fewer idle cores throughout execution. This example illustrates the key research challenge addressed in this section, where, without global coordination, multiple executors fragment the CPU pool and cause avoidable contention. A unified executor provides globally informed node assignment that improves fairness, utilization, and overall completion time.

3.3. Broader applicability

Although this work focuses on a specific integration of CEDR and Taskflow, the proposed methodology has broader relevance for runtime system design in heterogeneous computing environments. The core ideas are not restricted to these two frameworks–they can be extended to other runtime systems and task-based programming models as long as developers understand the necessary integration steps and architectural constraints. For example, the approach for communicating taskflow graphs or DAGs between frameworks can be adapted to suit different runtime systems. Likewise, an API-centric runtime like IRIS [1] could be combined with a DAG-based library to gain greater flexibility and performance tuning for complex workloads. While our implementation centers on CEDR and Taskflow, the integration strategy outlined here provides a general foundation for enhancing resource coordination and promoting interoperability across various runtime systems.

4. Experimental setup

To comprehensively study the performance trends of our proposed runtime, we consider four different platforms. We utilize the Xilinx Zynq Ultrascale+ development board (ZCU102) [27] to prototype and evaluate heterogeneous architectures. The ZCU102 features a Cortex-A53 with four ARM-based hard CPU cores and five custom hardware accelerators implemented on its programmable logic. These include (1) two FFT accelerators capable of processing up to 2048-point FFTs, (2) two GEMM accelerators, and (3) one ZIP accelerator for point-wise vector operations. The FFT accelerator is generated using Xilinx FFT IP, while the GEMM and ZIP accelerators are developed using High-Level Synthesis (HLS). The CPUs run at 1.2 GHz, and all accelerators operate at 300 MHz. To validate the portability of our approach, we also conducted experiments on the NVIDIA Jetson Xavier AGX platform (Jetson) [28], which features a 512-core Volta GPU clocked at 1.3 GHz and supports FFT, GEMM, ZIP, and CONV_2D (Convolution 2D) computations, along with an eight-core ARM-based CPU running at 2.3 GHz. We utilize a 24-core system 12th Gen Intel Core i9-12900 [29] (I9CPU) with a max frequency of 5 GHz to demonstrate how the multi-application approach scales with a high number of CPU cores. To evaluate performance in an HPC-like environment, we also test on the most heterogeneous system available to us, which includes HPC-scale GPUs and FPGAs. This system (I7CGF) consists of a 12-core Intel Core i7-5820K processor [30] operating at 3.3 GHz, an NVIDIA Tesla K40c GPU [31] running at 745 MHz,

and an Alveo U280 FPGA [32] with a kernel clock rate of 500 MHz. The CPU and GPU support the FFT, GEMM, and ZIP APIs, while the Alveo FPGA supports only the ZIP as an accelerator. In all platforms, for any accelerator (GPU or FPGA), data transfers between CPU and accelerators are managed through CEDR APIs, which encapsulate both computation and the necessary data movement. For example, when a task is mapped to the GPU, the API internally issues the corresponding *cudaMalloc*, *cudaMemcpy*, and *cudaFree* operations. Accordingly, the execution times reported in the Gantt charts include both data allocation, transfer, and computation.

In the Taskflow-integrated CEDR context, each application thread uses Taskflow constructs to create its DAG. In the multi-executor configuration, each thread creates and manages its own `tf::Executor`. In the single-executor configuration, application threads only generate their DAGs, which are then submitted to CEDR for centralized scheduling through a shared `tf::Executor`.

We evaluate the integration of CEDR and Taskflow using benchmark applications drawn from the signal processing domain, including Radar Correlator (RC), Temporal Mitigation (TM), WiFi-TX, Pulse Doppler (PD), and Synthetic Aperture Radar (SAR). The benchmark applications used in this study, including Taskflow-only, CEDR-only, and CEDR-Taskflow implementations, are publicly available.[1] These applications introduce a variety of workload patterns ideal for testing runtime behavior. RC processes incoming radar pulses to estimate range, applying three 256-point FFTs at a rate of 1000 samples per second. Its main control path involves two FFT stages, followed by a spectral correlation and a final inverse FFT (IFFT). TM targets interference suppression by removing weak radar echoes masked by stronger communication signals, leveraging GEMM and ZIP operations. WiFi-TX implements a WiFi transmission pipeline, where each of the 10 packets passes through a 128-point IFFT. PD performs three sequential phases of 256-, 128-, and 128-way parallel 128-point FFTs to compute object range and velocity from Doppler frequency shifts. SAR focuses on reconstructing 3D terrain imagery, executing 1537 FFTs and 768 ZIP operations, with high degrees of parallelism distributed across two main computational phases.

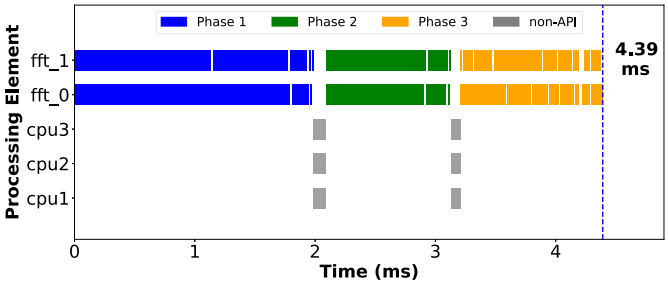
To demonstrate the generality of the framework beyond signal processing applications, we also developed an image-processing workload in the form of a lane-detection application. This pipeline consists of grayscale conversion followed by two Gaussian blur stages (each implemented using the CONV_2D APIs). Sobel filters then process the blurred image in the x- and y-directions (again using CONV_2D), and their outputs are merged. A region-of-interest (ROI) stage and a Hough transform complete the pipeline, yielding a typical lane-detection workflow. In total, this application uses multiple chained and parallel CONV_2D APIs while combining them with non-API tasks such as ROI extraction, thereby offering a different domain and execution structure compared to the signal processing workloads.

5. Experimental evaluations

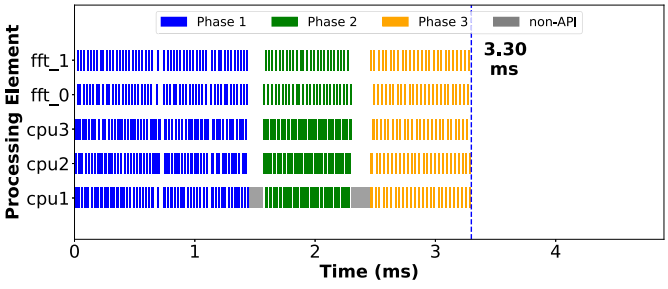
In this section, we first evaluate single-application execution to validate the correctness and efficiency of the integration. We then extend our experiments to multi-application scenarios to demonstrate scalability and the benefits of coordinated application executions.

5.1. Performance analysis

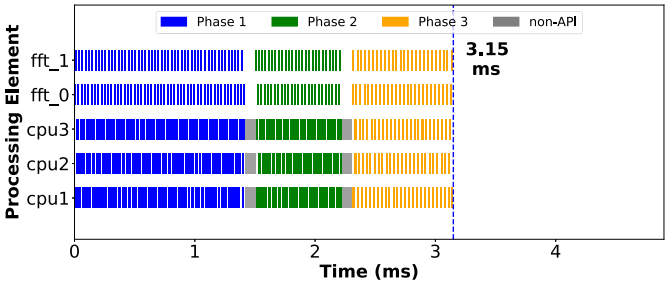
Fig. 4 presents the makespan of a single PD application instance executed on a system configured with 3 CPU cores and 2 FFT accelerators deployed on the ZCU102 platform. Three execution scenarios are evaluated: (a) Taskflow-only, (b) CEDR-only, and (c) the integrated CEDR and Taskflow setup. The execution time in each Gantt chart, marked by a vertical dashed blue line, indicates the makespan from the first API call to the last API call, excluding memory initialization, cleanup, allocation, or deallocation.



(a) Taskflow application



(b) CEDR application



(c) CEDR-Taskflow application

Fig. 4. Single instance of PD running on ZCU102 with (a) Taskflow-only implementation, (b) CEDR-only implementation, and (c) both CEDR and Taskflow-based implementation. (a) Taskflow application, (b) CEDR application, (c) CEDR-Taskflow application.

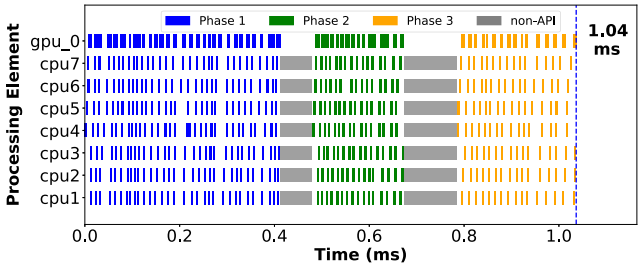


Fig. 5. Single instance of PD running on Jetson with integrated CEDR-Taskflow implementation.

In the Taskflow-only setup (Fig. 4a), all FFT API calls are statically mapped to the FFT accelerators, and non-API regions are parallelized across the available CPU cores. Although CPU cores are capable of executing FFT tasks, they remain idle due to the static assignment of tasks to PEs, resulting in contention on the FFT accelerators and leading to suboptimal performance.

With CEDR alone (Fig. 4b), using a simple Round Robin (RR) scheduler, APIs are dynamically scheduled across all available PEs, including

[1] https://github.com/UA-RCL/CEDR_applications/tree/taskflow

Table 2
Execution time comparison when applications are deployed as a single instance through CEDR only, Taskflow only, and CEDR-Taskflow Integrated setup on ZCU102.

App Name	Taskflow only (ns)	CEDR only (ns)	CEDR and Taskflow (ns)
RC	120,972	120,612	120,162
TM	2,597,770	2,575,658	1,762,166
WiFi-TX	714,621	712,721	651,685
PD	5,144,564	3,868,427	3,790,988
SAR	37,351,722	38,111,968	28,980,316

CPU cores and FFT accelerators, which alleviates accelerator contention and improves execution time. However, non-API regions are executed serially on a single CPU core due to CEDR's lack of support for parallelization in these regions. Despite this limitation, dynamic scheduling reduces the makespan from 4.39 ms to 3.30 ms.

The integrated setup (Fig. 4c) combines CEDR's dynamic API scheduling with Taskflow's parallelization of non-API regions. As a result, both types of tasks are efficiently distributed across the heterogeneous resources, further reducing the makespan to 3.15 ms.

Table 2 reports the execution time for a single instance of each benchmark application on the ZCU102 platform across three configurations: Taskflow-only, CEDR-only, and the integrated CEDR-Taskflow implementation using Earliest Finish Time (EFT) as CEDR's scheduling policy. These measurements reflect the full execution time, including memory allocation, initialization, and deallocation.

The integrated implementation consistently outperforms the standalone versions across all benchmarks. For example, the TM application achieves a speedup of 1.47× over Taskflow-only and 1.46× over CEDR-only. The RC application, which offers limited opportunities for parallelization in both API and non-API regions, shows only minimal gains, suggesting that the integration provides little benefit for such workloads. In the API-heavy PD application, the combined approach yields a 1.36× improvement over Taskflow-only and 1.02× over CEDR-only. For WiFi-TX, the integrated version delivers a modest 1.09× improvement over both standalone implementations. Finally, the SAR application, which contains substantial parallelism in both API and non-API regions, benefits significantly from the integration, with speedups of 1.28× over Taskflow-only and 1.31× over CEDR-only.

These results indicate that the performance gains from the CEDR-Taskflow integration depend on the workload characteristics, particularly the balance between API and non-API regions. Nevertheless, the integrated runtime consistently improves execution efficiency, with improvements of up to 1.47×.

5.2. Portability

The CEDR-Taskflow integration demonstrates portability across SoC platforms. Fig. 5 presents the execution of the PD application on the Jetson platform *without any modifications to the application code* used for generating the Gantt chart in Fig. 4c. This seamless portability is enabled by Taskflow's adaptive parallelism, which automatically utilizes the available CPU cores on the Jetson platform. At the same time, CEDR continues to provide dynamic resource management for hardware-agnostic API calls, demonstrating robustness across various hardware configurations. Given the Jetson's higher core count compared to the ZCU102, non-API regions of the PD application scale automatically across the additional CPU cores. Simultaneously, FFT tasks are dynamically distributed to available CPU cores whenever the GPU is occupied. As a result, the total execution time on the Jetson platform is reduced to 1.04 ms, compared to 3.15 ms on the ZCU102. This seamless portability highlights the flexibility and efficiency of the integrated runtime approach, enabling optimized execution across heterogeneous platforms without requiring additional developer effort.

Table 3
Time spent on scheduling for each application when repeated 1000 times on the ZCU102 platform.

App Name	API Count	Stream (μs)	Stream + Cached (μs)	Improvement
RC	3	2376	283	8.37x
TM	5	3759	643	5.84x
WiFi-TX	10	7662	723	10.59x
PD	512	291,790	10,769	27.09x
SAR	2305	1,405,034	47,475	29.60x

5.3. Features enabled by CEDR-taskflow integration

5.3.1. Streaming input processing

Fig. 6a illustrates the execution of 10 SAR instances run sequentially using the CEDR-Taskflow integrated setup. Each color-coded region represents one SAR instance executed on a system with 3 CPU cores, 2 FFT accelerators, and 1 ZIP accelerator emulated on the ZCU102 board. The white spaces between SAR instances indicate time spent on repeated memory allocation, deallocation, and initialization. By leveraging the repeated execution functionality provided by Taskflow-integrated CEDR, which allows multiple runs of an application with a single graph initialization, these gaps are significantly reduced, as shown in Fig. 6b. This improvement enables the application to process consecutive input streams without reinitializing context, thereby reducing the total makespan from 2,334.54 ms to 383.42 ms.

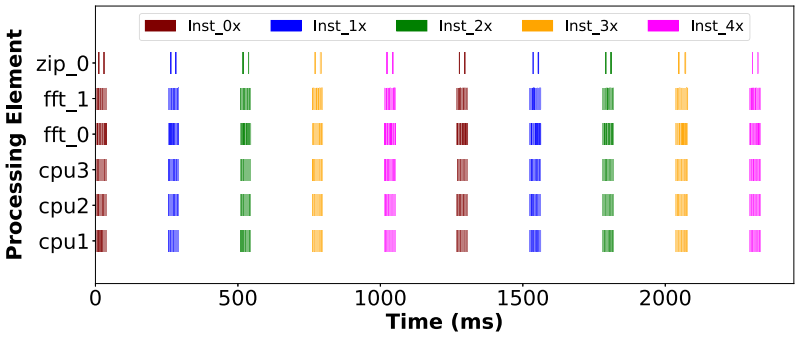
While it is well understood that the repeated execution of the same task graph reduces the overhead of frequent memory allocation, deallocation, and initialization, we explicitly quantify this effect on our heterogeneous SoC platform here. We compare complete application execution against isolated graph executions, and this empirical evidence establishes a baseline that motivates subsequent design decisions.

5.3.2. Cached scheduling

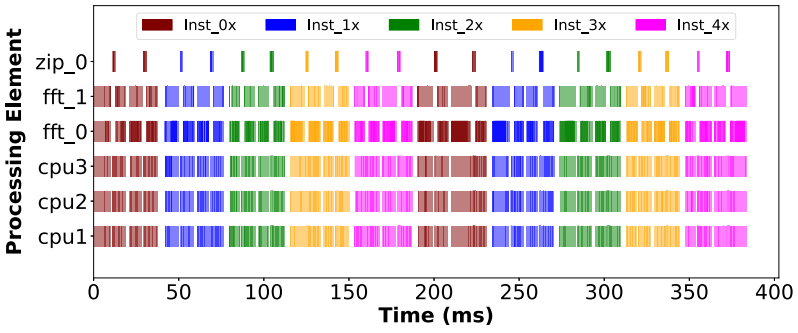
The streaming-enabled execution model also enables the reuse of scheduling decisions across multiple instances of an application by caching the task-to-PE mapping from a prior execution. To evaluate the impact of caching, we measure the scheduling overhead for each benchmark application under streaming execution, both with and without schedule caching. Each application is executed 1000 times using the EFT scheduler, and the results are presented in Table 3. The *API Count* column lists the number of API calls in each application. The *Stream* column reports the total scheduling time in μs without caching, while the *Stream + Cached* column shows the time required to extract the DAG and apply cached scheduling decisions. The results demonstrate a significant reduction in scheduling overhead when caching is used. As shown in Table 3, applications with higher API counts benefit more from caching, indicating that this approach is particularly suitable for API-intensive applications when processing streaming input.

5.4. Multi-application execution experiments

We present the benefits of the proposed single-executor approach, which maintains a global view of the system, compared to the default multi-executor setup when executing multiple applications concurrently. Fig. 7 presents the average makespan results of an experiment that utilizes the PD application with a repetition count of 1000 to simulate a streaming input scenario under concurrent workloads. On the x-axis, each point corresponds to a separate experiment and represents the total number of concurrently running PD instances, all launched simultaneously at the start of the experiment. We use the RR scheduler for task placement, as its deterministic behavior helps isolate the effects we aim to measure, and we do not use cached scheduling for any of the experiments in this section. The number of application instances is increased up to the first integer divisible by five, which exceeds the platform's total CPU core count by at least ten. In each experiment, for both



(a) SAR processing 10 inputs as application instances



(b) SAR processing 10 inputs in a streaming manner

Fig. 6. Ten SAR instances running back to back on ZCU102 by (a) direct injections from CEDR, (b) repeating the task flow graph. Labels Nx show instances N and N+5. (a) SAR processing 10 inputs as application instances, (b) SAR processing 10 inputs in a streaming manner.

single and multi-executor configurations, the ZCU102 utilizes 3 CPUs, 2 FFTs, and 1 ZIP, while the Jetson utilizes 7 CPUs and 1 GPU as PEs. Min-max ranges are shown in the figure to capture the variability among concurrently running instances within each experiment.

Fig. 7(a) and (b) show the scaling behavior of concurrently running applications on ZCU102 and Jetson platforms. As the number of concurrent PD instances increases, both platforms exhibit a generally linear growth in average makespan, with the single-executor configuration consistently outperforming the multi-executor baseline. The single-executor setup results in up to 1.20× and 1.47× speedups on ZCU102 and Jetson, respectively, compared to the multi-executor setup.

On Jetson, under the multi-executor setup, we observe a slight saturation in the 3-8 instance range (Fig. 7b), where the increase in average makespan slows as the number of instances approaches the number of available CPU cores (8 for Jetson). Once this threshold is crossed, the linear scaling trend continues. On ZCU102, which has only 4 CPU cores, this saturation pattern is not noticeable due to the platform's limited core count. To examine this trend in more detail and investigate whether a setup with more CPU cores available than needed causes a similar trend in the single-executor setup, we extend the experiment to the I9CPU platform with 24 CPU Cores, which is homogeneous.

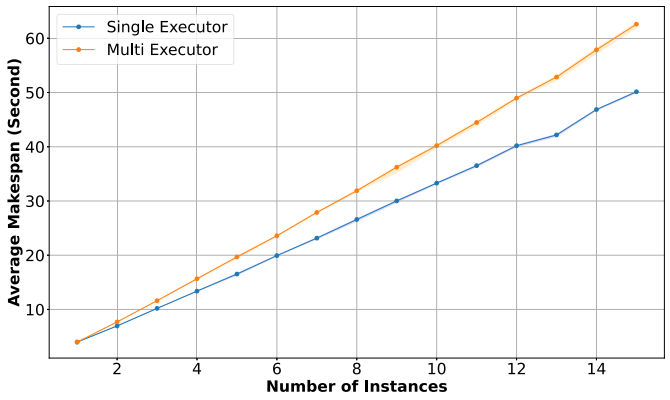
Fig. 7c presents the results for the I9CPU platform. The multi-executor setup again shows a clear saturation trend, starting around 13 concurrent PD instances and continuing until it reaches the number of available cores, after which the trend returns to a linear increase. Additionally, we observe that the slope of the makespan curve is steeper before this region begins and becomes more gradual afterward. In contrast, the single-executor configuration maintains a consistent linear trend throughout, highlighting its scalability. On this platform, the single-executor approach achieves a speedup of up to 2.92× before, 2.71× during, and 1.84× after the saturation region.

Mix of Applications: Next, we conduct an experiment with multiple distinct applications executing concurrently. Table 4 presents the results using the same three platforms, comparing the single-executor and multi-executor configurations. The experiment follows the same setup as Section 5.4, including the platform-specific configurations for ZCU102 and Jetson. Each PD and SAR application instance is set up with a repeat count of 1000 to simulate processing streaming inputs. In the *Single Application Multiple Instance* setup, we report the average execution time across five application instances, each processing 1000 input streams. In the *Mix of Applications* setup, the PD columns present the average execution time across the five PD instances, the SAR column shows the average time for a single SAR execution, and the Total column reports the overall makespan–from the start time of the earliest application to the end time of the last completed one. The bottom row for each platform indicates the speedup achieved by the single-executor configuration over the multi-executor baseline.

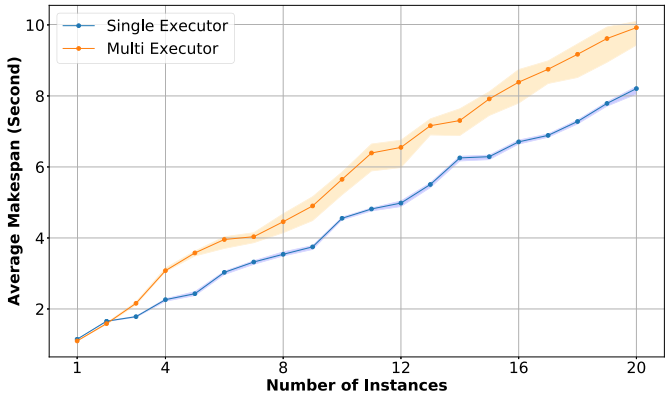
In this experiment, we use five applications for the *Single Application Multiple Instance* setup and a total of six applications for the *Mix of Applications* setup. While this configuration showcases the system's ability to handle diverse concurrent workloads, it also provides representative cases that correspond to different saturation regions observed in prior experiments–post-saturation on ZCU102, near-saturation on Jetson, and pre-saturation on I9CPU.

On the ZCU102 platform, we observe improvements of 1.20× for PD and 1.14× for SAR in the *Single Application Multiple Instance* configuration. In contrast, the *Mix of Applications* workload causes modest improvements, with application-specific and total speedups ranging from 1.06× to 1.07×, primarily due to the platform's limited number of CPU cores. On Jetson, which offers more CPU cores, we observe better scaling, 1.31× for PD and 1.67× for SAR in the *Single Application Multiple Instance* setup. For the mixed workload, the total speedup reaches approximately 1.21×, highlighting the increased flexibility in CPU core management enabled by a higher core count when using the single-executor setup.

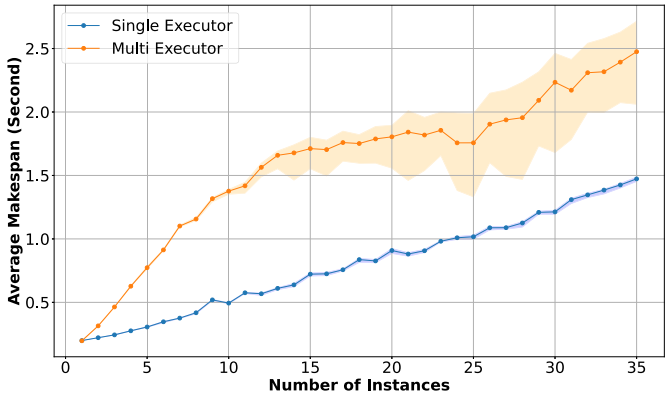
On the I9CPU platform, which offers much more CPU resources, the single-executor setup achieves improvements of 2.21× and 1.82× for PD and SAR, respectively, in the *Single Application Multiple Instance* case. In



(a) ZCU102 - 4 CPU Cores



(b) Jetson - 8 CPU Cores



(c) I9CPU - 24 CPU Cores

Fig. 7. Number of application instances versus average makespan for single-executor (proposed) and multi-executor implementations of the PD application on different platforms. (a) ZCU102 - 4 CPU Cores, (b) Jetson - 8 CPU Cores, (c) I9CPU - 24 CPU Cores.

the *Mix of Applications* scenario, speedups range from 1.59× to 2.19×, demonstrating the system's ability to make more informed decisions when application execution distribution is globally managed across a large number of cores.

In all platforms, we observe a consistent slowdown in PD's average execution time when moving from the *Single Application Multiple Instance* to the *Mix of Applications* setup, despite both scenarios running five PD instances. This behavior is expected as SAR competes for shared resources in the mixed configuration, reducing the availability of resources for PD compared to the single-application scenario.

Looking more closely at the PD results, the values in the *Single Application Multiple Instance* setup directly match the results for five instances presented in Fig. 7. In the *Mix of Applications* setup, the average PD execution time closely aligns with the six-instance region, as all PD applications typically finish before the SAR instance completes. Since SAR is also running together with the 5 PDs, a total of six applications are running on the system during the execution of PDs. The slight variations occur due to the different characteristics of SAR in terms of the resources it uses.

Overall, across all platforms and configurations, the single-executor system consistently outperforms the multi-executor baseline, demonstrating improved parallelization and resource awareness when execution is managed globally with a complete system view.

5.5. HPC scale experiments

To evaluate the system's performance in an HPC-scale heterogeneous environment, we perform an experiment on the I7CGF platform. This system, which represents the most heterogeneous configuration available to us, includes CPUs and a GPU, both of which support FFT and ZIP operations, as well as an Alveo FPGA that supports ZIP operations only. The workload comprises four PD applications and one SAR application, each configured to process 100 streaming inputs using cached scheduling and the EFT policy for scheduling, just as in Section 5.3.2. Compared to earlier experiments, the number of streaming inputs is reduced from 1000 to 100, to make the Gantt charts more readable, as shown in Fig. 8. The SAR application and the first PD instance are submitted simultaneously at the beginning of the experiment, with the remaining PD applications introduced at 500 ms intervals. We select 500 ms as the interval based on the observation that, when running SAR and PD concurrently, each PD instance completes in approximately 350-450 ms. This timing ensures that one PD remains active during the SAR execution window. Sending four PD instances is sufficient to cover the span of the SAR's execution in the fastest system configuration. This experiment combines all key features–streaming inputs, cached scheduling, dynamic load balancing, and heterogeneous execution–to demonstrate that the system operates efficiently while seamlessly leveraging all introduced capabilities under complex workloads on a system with a high degree of heterogeneity.

Fig. 8(a) and (b) show the Gantt charts of this experiment under multi-executor and single-executor configurations, respectively. Overall, the single-executor setup achieves a 1.23× speedup, demonstrating improved coordination between applications and CPU core utilization.

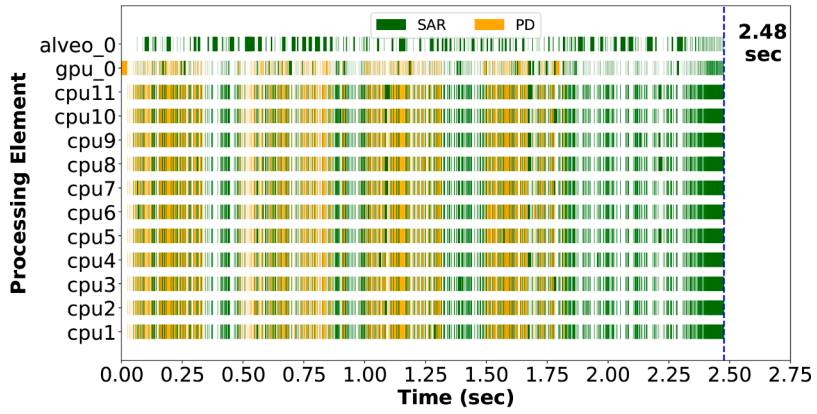
Examining specific regions, particularly between 0.25- 0.50 sec, 1.75-2.00 sec, and on the `alveo_0` PE, we observe that the multi-executor configuration shows noticeable idle periods and large execution gaps, indicating poor CPU workload balancing and underutilization of available accelerators. In contrast, the single-executor setup eliminates most of these idle gaps, reflecting more efficient handling of non-API regions and improved overlap across applications.

Similarly, around the 1.00-1.25 sec interval, the multi-executor setup is dominated by PD tasks, resulting in the SAR application being delayed and trailing behind all PD executions. As a result, SAR completes roughly 0.5 sec after the last PD finishes. In comparison, the single-executor setup achieves a tighter blend of SAR and PD APIs, enabling both applications to complete almost simultaneously. While this synchronized completion is a result of our specific experimental setup (utilizing four PDs with 500 ms intervals), it highlights another outcome. If another PD application were submitted at 2.00 sec, the single-executor configuration would already have completed SAR and could dedicate all resources to the new PD. In contrast, under the multi-executor setup, SAR would still be running, and the new PD workload would further delay SAR completion due to resource contention.

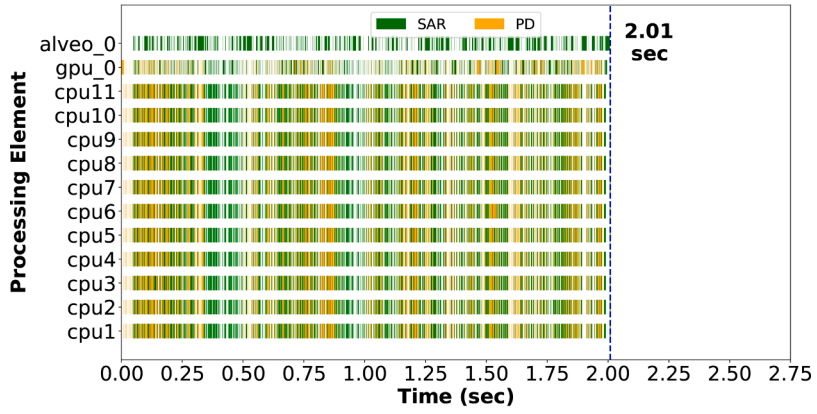
These results highlight the runtime's ability to coordinate application workloads, minimize idle time, and achieve improved overall execution when utilizing a centralized execution model.

Table 4
Single and multiple application results on ZCU102, Jetson, and I9CPU.

Platform	Executor View	Single Application Multiple Instance		Mix of Applications		
		5 PD (ms)	5 SAR (ms)	5 PD and 1 SAR (ms)		
				PD	SAR	Total
ZCU102	Multi	19,747.39	194,709.74	24,375.81	59,035.02	59,563.37
	Single	16,444.32	169,394.47	22,688.54	55,358.64	55,595.45
	Improvement	1.20×	1.14×	1.07×	1.06×	1.07×
Jetson	Multi	3,404.96	31,955.72	4,228.41	8,972.76	9,113.61
	Single	2,597.72	19,133.92	3,360.01	7,435.44	7,529.02
	Improvement	1.31×	1.67×	1.25×	1.20×	1.21×
I9CPU	Multi	770.83	4,697.44	895.95	1,487.10	1,527.65
	Single	348.15	2,571.19	408.10	918.74	955.95
	Improvement	2.21×	1.82×	2.19×	1.61×	1.59×



(a) Multi Executor



(b) Single Executor

Fig. 8. Mix of applications with different injection rates running on HPC scale platform I7CGF for (a) multi executor and (b) single executor (proposed) setups. (a) Multi executor, (b) Single executor.

5.6. Application domain extension

To demonstrate the generality of our approach beyond the signal processing domain, we repeated the multi-executor versus single-executor experiment (shown earlier in Fig. 7b) using the lane detection (LD) application introduced in Section 4. In this experiment, each instance processes 50 streaming inputs while utilizing cached scheduling and the EFT scheduler. Fig. 9 reports the makespan as the number of concurrently executing LD applications increases. Similar to the PD workload, we observe that the multi-executor model results in fragmented resource usage and increasing makespan as the number of applications grows. In contrast, the single-executor model achieves coordinated scheduling and improved load balancing, reducing overall makespan by up to 4.45 s. These results confirm that the benefits of centralized coordination extend across domains and demonstrate the portability of our integration to workloads with different computational patterns.

5.7. Case study

As a final experiment, we compare our single- and multi-executor implementations with another state-of-the-art runtime system. For this study, we selected IRIS [1] and adapted one of its example applications. Specifically, we modified the SAXPY kernel provided by IRIS to increase complexity by switching to floating-point vector-vector multiplication (ZIP-MULT) and introducing task dependencies. In the modified version, each application instance computes two independent ZIP-MULTs,

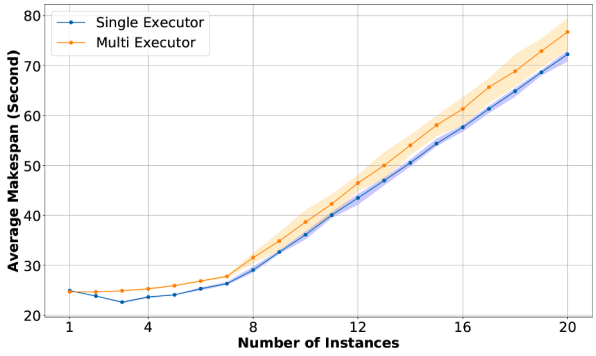


Fig. 9. Number of application instances versus average makespan for single-executor (proposed) and multi-executor implementations on Jetson platform using LD application.

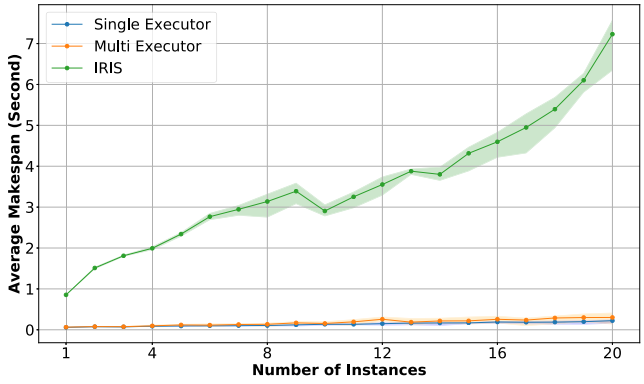


Fig. 10. Scaling trends for multi-instance execution of the ZIP-MULT workload on the Jetson platform. Results show IRIS alongside the single- and multi-executor models of the proposed CEDR-Taskflow integration.

whose results are consumed by a third ZIP-MULT. This setup retains simplicity while incorporating both parallel execution and inter-task dependencies. In our experiments, we use a vector size of 1,024, and each application repeats the ZIP-MULT triple 1000 times on the Jetson platform.

Results are shown in Fig. 10. For the CEDR-Taskflow integrated approach, the single- and multi-executor setups achieve similar scaling, with up to 1.72× improvement in the single-executor case. For IRIS, we observe a reduction in the growth rate between 6 and 9 instances, similar to the single- and multi-executor results in Section 5.4. Beyond 10 instances, the average execution time scales approximately linearly with the number of concurrent applications. We also note that IRIS and CEDR follow different execution models. IRIS treats all CPU cores as a single device and executes parallel regions internally using OpenMP, while CEDR assigns each API execution to a single designated core. Because of these differences, we do not make direct runtime-to-runtime performance claims; instead, we emphasize that the trend lines demonstrate how both runtimes scale with multi-application workloads.

6. Related work

The evolving landscape of heterogeneous computing, from embedded SoCs to large-scale HPC environments, demands runtime frameworks that balance programmability, performance, and efficient resource utilization. Recent works explore programming model design [33], runtime interoperability [34,35], and dynamic scheduling across heterogeneous devices [35]. The approach presented in this work offers a unified and portable solution for task-based execution and coordinated runtime management across diverse heterogeneous architectures.

In [33], the authors focus on achieving performance portability across different hardware platforms. They provide a generic and portable C++ interface for data-parallel computation on accelerators. Built on the HPX [36] asynchronous runtime and incorporating a SYCL-based [37] backend, Copik and Kaiser [33] enables single-source programming for OpenCL-compatible devices. Like our framework, it abstracts hardware-specific details and streamlines portable development, notably by being orthogonal to vendor-specific extensions and not requiring additional compiler markups. However, in [33], the authors primarily target kernel-based execution in HPC environments, whereas our work exposes and manages workloads through a holistic application view that includes both API and non-API regions of the code, enabling the parallelization of segments that were previously executed serially.

In [34], the authors explore asynchronous execution models by integrating multiple programming layers. They demonstrate the interoperability of HPX [38], an asynchronous many-task runtime system, and Kokkos [39], a programming model for portable compute kernels, via SYCL [37]. This integration enables asynchronous execution of GPU kernels without blocking CPU threads by treating SYCL events as HPX tasks, thus integrating them into the HPX task graph. Their approach, particularly using an event polling scheme within the HPX scheduler, achieves improved throughput in Octo-Tiger [40], a real-world astrophysics application. In contrast, our work integrates Taskflow and CEDR at the application level, where Taskflow provides a holistic task graph view, including CPU-only computations (non-API regions), and CEDR dynamically schedules tasks across heterogeneous PEs. Additionally, our system supports multi-application execution under centralized runtime coordination, a capability not addressed by the single-application context of [34].

The challenge of coordinating distinct programming paradigms has been addressed in [35], where the authors introduce an interoperability mechanism between OmpSs-2 [41] and OpenACC [42], both directive-based solutions. Their approach establishes a dominant-subordinate relationship, where OmpSs-2 (task-parallel) orchestrates application control while delegating computation-heavy regions to OpenACC (data-parallel). This relationship is achieved by extending the OmpSs-2 compiler and runtime to support OpenACC as well. While both approaches support hybrid programming, our integration blends Taskflow's DAG-based task parallelism with CEDR's dynamic API scheduling. Our framework further supports centralized execution coordination across multiple active applications and heterogeneous PEs.

In contrast to systems that focus on portable kernel abstractions or single-application pipelines, our work offers a unified framework designed for broader applicability and multi-application coordination. It fully exposes parallelization opportunities by converting entire applications into task graphs, regardless of whether the code resides within APIs or non-API regions. APIs are then dynamically scheduled across available resources, including CPU, FPGA, and GPU, based on the system state, while applications' overall CPU workloads are managed under centralized coordination. The runtime includes support for stream input processing and cached scheduling decisions, all of which are generalized to operate across concurrent application workloads. This design is evaluated across a range of platforms, from SoCs to HPC-grade resources, and demonstrates improved resource utilization and reduced execution time. By providing a centralized execution layer and a global view of system resources and task dependencies, our framework bridges the gap between productive application development and efficient heterogeneous execution.

7. Conclusion

This paper introduces a unified runtime-task programming framework that bridges CEDR and Taskflow to address the growing complexity of executing applications on heterogeneous systems. Our design supports both task-graph-based parallelization and dynamic scheduling, enabling high resource utilization without requiring hardware-

specific knowledge or extensive code modifications. Through this integration, we further extend the system to manage multiple applications concurrently under a globally coordinated runtime framework, resolving inefficiencies introduced by fragmented execution contexts. Experimental results across a range of platforms–from embedded SoCs to HPC-class systems–demonstrate consistent improvements in performance, scalability, and scheduling efficiency. The system also supports advanced features such as cached scheduling and streaming input execution across concurrent workloads, showcasing its robustness in dynamic execution environments.

CRediT authorship contribution statement

Serhan Gener: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization; **Sahil Hassan:** Writing – review & editing, Writing – original draft, Visualization, Validation, Formal analysis, Conceptualization; **H. Umut Suluhan:** Writing – review & editing, Writing – original draft, Validation, Formal analysis, Conceptualization, Visualization; **Liangliang Chang:** Writing – review & editing, Writing – original draft, Validation, Formal analysis, Conceptualization, Visualization; **Chaitali Chakrabarti:** Writing – review & editing, Writing – original draft, Visualization, Formal analysis, Conceptualization; **Tsung-Wei Huang:** Writing – review & editing, Writing – original draft, Visualization, Formal analysis, Conceptualization; **Umit Ogras:** Writing – review & editing, Writing – original draft, Visualization, Formal analysis, Conceptualization; **Ali Akoglu:** Writing – review & editing, Writing – original draft, Visualization, Supervision, Resources, Project administration, Methodology, Funding acquisition, Formal analysis, Conceptualization.

Data availability

Data will be made available on request.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

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 conclusions 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 AFRL and DARPA or the U.S. Government.

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

Dr. Akoglu and Dr. Ogras have disclosed an outside interest in DASH Tech IC to the University of Arizona and University of Wisconsin, respectively. Conflicts of interest resulting from this interest are being managed by the respective universities in accordance with their policies.

References

[1] J. Kim, S. Lee, B. Johnston, J.S. Vetter, IRIS: a performance-portable framework for cross-platform heterogeneous computing, IEEE Trans. Parallel Distrib. Syst. 35 (10) (2024) 1796–1809. https://doi.org/10.1109/TPDS.2024.3429010

[2] R. Nozal, J.L. Bosque, R. Beivide, EngineCL: usability and performance in heterogeneous computing, Future Gener. Comput. Syst. 107 (2020) 522–537. https://doi.org/10.1016/j.future.2020.02.016

[3] J. Mack, S. Gener, S. Hassan, H.U. Suluhan, A. Akoglu, CEDR-API: productive, performant programming of domain-specific embedded systems, in: 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2023, pp. 16–25. https://doi.org/10.1109/IPDPSW59300.2023.00016

[4] J. Mack, S. Hassan, N. Kumbhare, M. Castro Gonzalez, A. Akoglu, CEDR: a compiler-integrated, extensible DSSoC runtime, ACM Trans. Embed. Comput. Syst. 22 (2) (2023). https://doi.org/10.1145/3529257

[5] T.-W. Huang, D.-L. Lin, C.-X. Lin, Y. Lin, Taskflow: a lightweight parallel and heterogeneous task graph computing system, IEEE Trans. Parallel Distrib. Syst. 33 (6) (2022) 1303–1320. https://doi.org/10.1109/TPDS.2021.3104255

[6] S. Gener, S. Hassan, L. Chang, C. Chakrabarti, T.-W. Huang, U. Ogras, A. Akoglu, A unified portable and programmable framework for task-based execution and dynamic resource management on heterogeneous systems, in: Proceedings of the 2025 4th International Workshop on Extreme Heterogeneity Solutions, ExHET '25, Association for Computing Machinery, New York, NY, USA, 2025, p. 1-9. https://doi.org/10.1145/3720555.3721988

[7] J. Auerbach, D.F. Bacon, I. Burcea, P. Cheng, S.J. Fink, R. Rabbah, S. Shukla, A compiler and runtime for heterogeneous computing, in: DAC Design Automation Conference 2012, 2012, pp. 271–276. https://doi.org/10.1145/2228360.2228411

[8] C. Bolchini, S. Cherubin, G.C. Durelli, S. Libutti, A. Miele, M.D. Santambrogio, A runtime controller for openCL applications on heterogeneous system architectures, SIGBED Rev. 15 (1) (2018) 29-35. https://doi.org/10.1145/3199610.3199614

[9] K. Moazzemi, B. Maity, S. Yi, A.M. Rahmani, N. Dutt, HESSLE-FREE: heterogeneous systems leveraging fuzzy control for runtime resource management, ACM Trans. Embed. Comput. Syst. 18 (5s) (2019). https://doi.org/10.1145/3358203

[10] X. Tan, J. Bosch, C. Álvarez, D. Jiménez-González, E. Ayguadé, M. Valero, A hardware runtime for task-based programming models, IEEE Trans. Parallel Distrib. Syst. 30 (9) (2019) 1932–1946. https://doi.org/10.1109/TPDS.2019.2907493

[11] J. Boutellier, J. Wu, H. Huttunen, S.S. Bhattacharyya, PRUNE: dynamic and decidable dataflow for signal processing on heterogeneous platforms, IEEE Trans. Signal Process. 66 (3) (2018) 654–665. https://doi.org/10.1109/TSP.2017.2773424

[12] G. Christodoulis, F. Broquedis, O. Muller, M. Selva, F. Desprez, An FPGA target for the StarPU heterogeneous runtime system, in: 2018 13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2018, pp. 1–8. https://doi.org/10.1109/ReCoSoC.2018.8449373

[13] C. Hsieh, A.A. Sani, 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), 2019, pp. 136–141. https://doi.org/10.1109/VLSI-SoC.2019.8920374

[14] Intel oneTBB, 2021. Accessed: June 23, 2025, https://github.com/oneapi-src/oneTBB.

[15] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Torquati, Fastflow: high-level and efficient streaming on multicore, Programming multi-core and many-core computing systems (2017) 261–280. https://doi.org/10.1002/9781119332015.ch13

[16] H. Carter Edwards, C.R. Trott, D. Sunderland, Kokkos: enabling manycore performance portability through polymorphic memory access patterns, J. Parallel Distrib. Comput. 74 (12) (2014) 3202–3216. https://doi.org/10.1016/j.jpdc.2014.07.003

[17] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, D. Fey, HPX: a task based programming model in a global address space, in: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 1–11. https://doi.org/10.1145/2676870.2676883

[18] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, J.J. Dongarra, PaRSEC: exploiting heterogeneity to enhance scalability, Comput. Sci. Eng. 15 (6) (2013) 36–45. https://doi.org/10.1109/MCSE.2013.98

[19] M. Bauer, S. Treichler, E. Slaughter, A. Aiken, Legion: expressing locality and independence with logical regions, in: SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012, pp. 1–11. https://doi.org/10.1109/SC.2012.71

[20] J. Mack, S. Gener, A. Akoglu, J. Holtom, A. Chiriyath, C. Chakrabarti, D. Bliss, A. Krishnakumar, A. Goksoy, U. Ogras, GNU Radio and CEDR: runtime scheduling to heterogeneous accelerators, in: Proceedings of the GNU Radio Conference, 7, 2022, pp. 1–12.

[21] H.U. Suluhan, S. Gener, A. Fusco, H.F. Ugurdag, A. Akoglu, PyTorch and CEDR: enabling deployment of machine learning models on heterogeneous computing systems, in: 2023 20th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA), 2023, pp. 1–8. https://doi.org/10.1109/AICCSA59173.2023.10479315

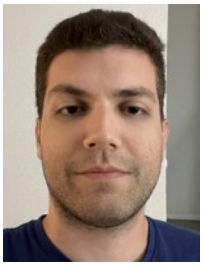
[22] H.U. Suluhan, S. Gener, A. Fusco, J. Mack, I. Dagli, M. Belviranli, C. Edemen, A. Akoglu, A runtime manager integrated emulation environment for heterogeneous SoC design with RISC-V cores, in: 2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2024, pp. 23–30. https://doi.org/10.1109/IPDPSW63119.2024.00013

[23] J. Mack, N. Kumbhare, A. NK, U.Y. Ogras, 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. https://doi.org/10.1109/IPDPSW50202.2020.00016

[24] L. Dagum, R. Menon, OpenMP: an industry standard API for shared-memory programming, J. Comput. Sci. Eng. 5 (1) (1998) 46–55. https://doi.org/10.1109/99.660313

[25] D.W. Walker, J.J. Dongarra, MPI: a standard message passing interface, Supercomputer 12 (1996) 56–68.

[26] B. Schäling, The Boost C++ Libraries, Boris Schäling, 2011.
[27] Xilinx ZCU102 evaluation board. Accessed: June 23, 2025, https://docs.amd.com/v/u/en-US/ug1182-zcu102-eval-bd.
[28] NVIDIA jetson AGX Xavier evaluation board. Accessed: June 23, 2025, https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/.
[29] Intel Core i9-12900 processor 30M cache, up to 5.10 GHz, . Accessed: June 23, 2025, https://www.intel.com/content/www/us/en/products/sku/134597/intel-core-i912900-processor-30m-cache-up-to-5-10-ghz/specifications.html.
[30] Intel Core i7-5820K processor 15M cache, up to 3.60 GHz, . Accessed: June 23, 2025, https://www.intel.com/content/www/us/en/products/sku/82932/intel-core-i75820k-processor-15m-cache-up-to-3-60-ghz/specifications.html.
[31] NVIDIA Tesla K40 specification, . Accessed: June 23, 2025, https://www.nvidia.com/content/PDF/kepler/nvidia-tesla-k40.pdf.
[32] AMD Alveo U280 data center accelerator card, . Accessed: June 23, 2025, https://docs.amd.com/r/en-US/ds963-u280.
[33] M. Copik, H. Kaiser, Using SYCL as an implementation framework for HPX.compute, in: Proceedings of the 5th International Workshop on OpenCL, IWOCL '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 1–7. https://doi.org/10.1145/3078155.3078187
[34] G. Daiß, P. Diehl, H. Kaiser, D. Pflüger, Stellar mergers with HPX-Kokkos and SYCL: methods of using an asynchronous many-task runtime system with SYCL, in: Proceedings of the 2023 International Workshop on OpenCL, IWOCL '23, Association for Computing Machinery, New York, NY, USA, 2023, pp. 1–12. https://doi.org/10.1145/3585341.3585354
[35] O. Korakitis, S.G. De Gonzalo, N. Guidotti, J.a.P. Barreto, J.C. Monteiro, A.J. Peña, Towards OmpSs-2 and openACC interoperation, in: Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 433-434. https://doi.org/10.1145/3503221.3508401
[36] T. Heller, H. Kaiser, P. Diehl, D. Fey, M.A. Schweitzer, Closing the performance gap with modern C++, in: High Performance Computing: ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P 3MA, VHPC, WOPSSS, Frankfurt, Germany, June 19–23, 2016, Revised Selected Papers 31, Springer, 2016, pp. 18–31.
[37] The Khronos SYCL Working Group, SYCL 2020 Specification (Revision 10), Technical report, Khronos Group, 2021. Accessed: June 23, 2025, https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf.
[38] H. Kaiser, P. Diehl, A.S. Lemoine, B.A. Lelbach, P. Amini, A. Berge, J. Biddiscombe, S.R. Brandt, N. Gupta, T. Heller, et al., HPX-the C++ standard library for parallelism and concurrency, J. Open Source Softw. 5 (53) (2020) 2352.
[39] C.R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D.S. Hollman, D. Ibanez, et al., Kokkos 3: programming model extensions for the exascale era, IEEE Trans. Parallel Distrib. Syst. 33 (4) (2021) 805–817.
[40] D.C. Marcello, S. Shiber, O. De Marco, J. Frank, G.C. Clayton, P.M. Motl, P. Diehl, H. Kaiser, Octo-Tiger: a new, 3D hydrodynamic code for stellar mergers that uses HPX parallelization, Mon. Not. R. Astron. Soc. 504 (4) (2021) 5345–5382.
[41] Barcelona supercomputing center, OmpSs-2 specification, . Accessed: June 23, 2025, https://pm.bsc.es/ftp/ompss-2/doc/spec/Accessed: 2021-03-30.
[42] S. Wienke, P. Springer, C. Terboven, D. an Mey, OpenACC-first experiences with real-world applications, in: Euro-Par 2012 Parallel Processing: 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings 18, Springer, 2012, pp. 859–870.



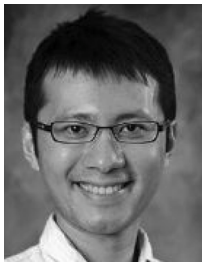
Serhan Gener is a PhD candidate in the Electrical and Computer Engineering department at the University of Arizona. He received his BS and MS degrees in Computer Engineering from Yeditepe University, Istanbul, Turkey, in 2015 and 2017, respectively. His research interests include heterogeneous computing, resource management, reconfigurable computing, embedded systems, image processing, and software security.

Sahil Hassan is a Postdoctoral Research Associate in the Department of Electrical and Computer Engineering at University of Arizona. He received his PhD in 2024 from the same institution. His research focuses on heterogeneous and brain-inspired computing systems.

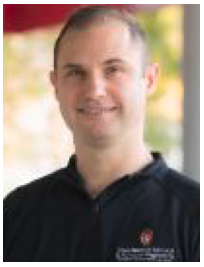
H. Umut Suluhan is a PhD student in the Department of Electrical and Computer Engineering at The University of Arizona. He received his BS degree in Computer Science from Ozyegin University in 2023. His research interests are broadly in high-performance computing and reconfigurable architectures with a focus on runtime software design and resource management algorithm development for heterogeneous computing systems.

LiangLiang Chang received his BS degree from Jilin University, Changchun, China, and his MS degree from Tsinghua University, Beijing, China. He is currently pursuing a PhD degree at Arizona State University, Tempe, AZ, USA. His research interests include profile-guided compiler optimizations and code generation for heterogeneous architectures.

Chaitali Chakrabarti is a Professor with the School of Electrical Computer and Energy Engineering, Arizona State University (ASU), Tempe, and a Fellow of the IEEE. Her research interests are in the areas of low power embedded systems design, distributed machine learning and VLSI architectures and algorithms for signal processing and communications.



Tsung-Wei Huang is an Assistant Professor in the ECE Department at the University of Wisconsin at Madison. He earned his PhD in ECE from University of Illinois at Urbana-Champaign (2017) and BS/MS in CS from Taiwan's NCKU (2011). His research focuses on software systems for performance-critical applications, including CAD, machine learning, and quantum computing. Dr. Huang has received several awards, including the ACM SIGDA Outstanding PhD Dissertation Award, NSF CAREER Award, Humboldt Research Fellowship, ACM SIGDA Outstanding New Faculty Award, the ICCAD 10-Year Most Influential Paper Award, and DAC Under 40 Innovator Award.



Umit Ogras is the Gene Amdahl Professor in the Dept. of Electrical and Computer Engineering at the University of Wisconsin-Madison. He worked at the Arizona State University as a faculty member between 2013-2020 and at Intel as a research scientist between 2008-2013 before receiving his PhD degree in Computer Engineering from Carnegie Mellon University in 2007. His research interests include chiplet-based platforms, edge AI, domain-specific systems, and low-power VLSI.

Ali Akoglu Ali Akoglu received his PhD in Computer Science from Arizona State University in 2005. He is a Professor in the Department of Electrical and Computer Engineering and a member of the BIO5 Institute at the University of Arizona. His research focuses on domain-specific computing systems, with emphasis on resource management for heterogeneous platforms and the design of reconfigurable and neuromorphic architectures. Dr. Akoglu serves as the site director of the NSF IUCRC on Cloud and Autonomic Computing.