GNURadio and CEDR: Runtime Scheduling to Heterogeneous Accelerators

Joshua Mack JMACK2545@ARIZONA.EDU
Serhan Gener GENER@ARIZONA.EDU
Ali Akoglu AKOGLU@ARIZONA.EDU
University of Arizona

Jacob Holtom JHOLTOM@ASU.EDU
Alex Chiriyath ACHIRIYA@ASU.EDU
Chaitali Chakrabarti CHAITALI@ASU.EDU
Daniel Bliss DWBLISS@ASU.EDU
Center for Wireless Information Systems and Computational Architectures (WISCA), Arizona State University, Tempe, AZ, 85281, USA

Anish Krishnakumar ANISH.N.KRISHNAKUMAR@WISC.EDU
Alper Goksoy AGOKSOY@WISC.EDU
Umit Ogras UOGRAS@WISC.EDU
University of Wisconsin-Madison

Abstract

Accelerators in GNURadio have been previously limited to requiring the selection of an accelerator framework/block at design time. In this paper, we present a preliminary investigation into supporting two new capabilities with GNURadio: first, we illustrate the ability to execute GNURadio blocks across a variety of heterogeneous accelerators (FPGA and GPU). Second, we demonstrate that we are able to dynamically schedule these blocks across our pool of accelerators using easily-customizable scheduling policies. We do this via a combination of out-of-tree modules and by embedding GNU Radio itself as an application in a heterogeneous runtime called CEDR (Mack et al., 2022). The CEDR ecosystem provides a productive environment for researching the combined challenges of application design, systems software, and hardware prototyping for heterogeneous systems, and namely, it provides a flexible intelligent scheduling (IS) interface by which the user can easily adjust or prioritize how tasks are dispatched at runtime to the various accelerators present on a given system. The IS integrates a variety of runtime schedulers that optimize for metrics such as application performance and minimum scheduling algorithm overheads. The IS in tandem with CEDR improves the performance of applications through dynamic scheduling by efficiently utilizing the resources at runtime.

We demonstrate GNURadio running on heterogeneous hardware using CEDR across both Xilinx Zynq Ultrascale+ ZCU102 and Nvidia Jetson AGX Xavier systems. We choose three applications that can leverage FFT acceleration: we implement a radar correlator in GNURadio, deploy in CEDR and execute it along with non-GNURadio-based WiFi-TX and Synthetic Aperture Radar applications. We find that, when GNURadio shares the system with other unrelated applications, our integrated IS dispatches FFT tasks to the accelerator on both the FPGA and GPU platforms along with the CPU cores fairly without compromising target throughput for each application. In summary, we show that running the GNURadio runtime and applications inside/with CEDR enables better scheduling options and easier accelerator access by eliminating the need for users to partition their workloads at design time. We believe this is a stepping stone in broadening GNURadio's support for heterogeneous execution and enabling it to hook more flexibly into a variety of scheduling heuristics.

Proceedings of the 1st GNU Radio Conference, Copyright 2016 by the author(s).

1. Introduction

Heterogeneous computing systems, while offering a large potential for performance gains relative to homogeneous counterparts, traditionally pair efficiency gains with reductions in ease of use and programmer productivity. Platforms such as Domain-specific System on Chip (DSSoC) devices have been proposed as one such solution for addressing this divergence with the hope that the focus on a smaller domain of applications will enable more productive software and programming abstractions. However, despite the advantages gained through reducing the problem size, there is a need for an intelligent runtime system and programming framework to enable effective utilization of DSSoC platforms and take full advantage of their underlying hardware without requiring users to become hardware experts in the process. We envision that the DSSoC system should also enable a productive programming and deployment experience in such a way that multiple users can coexist and share the hardware as a service by supporting execution of any combination of dynamically arriving applications. In traditional heterogeneous programming paradigms, massive amounts of effort are put into offline performance analysis by domain experts to determine the portions of an application that must be accelerated, the type of accelerators needed, and effective implementation strategy for the target hardware configuration. Low-performance serial implementations are then replaced with their optimized heterogeneous implementations, and a static binary that represents a single, expertly-tuned instance of that application is produced. Such static and offline resource allocation decisions result in a greedily optimized implementation that assumes it does not share the heterogeneous accelerators with any other applications. However, this assumption has the potential to lead to drastic mismanagement or under-utilization of the target hardware in a highly heterogeneous computing environment. Towards this end, as part of the Domain-Focused Advanced Software-Reconfigurable Heterogeneous System on Chip (DASH-SoC) team, we are building a framework to develop flexible, high-performance, low-power, domain-specific SoCs, while assuring non-expert programmability. We are developing an example SoC for software-defined RF systems: radios; radars; spectral awareness; positioning, navigation, and timing; and RF convergence. DASH allows RF system designers to escape the traditional power & development-cost limits to innovation (Bliss, 2020; Chiriyath et al., 2021).

One of the key outcomes of this project is the design and development of a novel open-source ecosystem that we refer to as CEDR: a Compiler-integrated, Extensible, DSSoC Runtime (Mack et al., 2022). This ecosystem integrates compile-time application analysis, a Linux-based runtime system, and an intelligent scheduling framework to holistically target the aforementioned requirements and capabilities. In this study we present integration of CEDR with the GNRURadio runtime and demonstrate our ability to deploy GNURadio workflows on both FPGA and GPU based COTS SoC platforms without requiring users to have prior knowledge on FPGA or GPU based design experience. We show that the CEDR and GNURadio integration allows developers to remain in their familiar programming environment, provides users with a hardware-agnostic application development experience, empowers them to automatically and dynamically deploy their applications on heterogeneous compute systems composed of a pool of general purpose processors and accelerators. In the following sections we start with an overview of the CEDR ecosystem followed by a discussion on our approach to CEDR and GNURadio integration. We use Radar Correlator applications as a case study and present its implementation in GNURadio, compilation through our integrated flow and finally deployment on Nvidia Jetson AGX Xavier platform. We finally emulate an SoC composed of a pool of FFT accelerators and ARM CPU using Xilinx ZCU102 Zynq platform. On this emulation platform we demonstrate dynamic task to processing element mapping decision capability where we execute the GNURadio based Radar Correlator as a continuous process and two signal processing applications, Synthetic Aperture Radar (SAR) and WiFiTX, arrive dynamically.

1.1. Contributions

- GNURadio C++ and Python (via cython3 embedding) Flowgraphs run in CEDR enabling runtime accelerator selection in GNURadio
- GR OOT Module (`gr-cedr`) collection of flexible runtime adaptive accelerator blocks (Found at https://github.com/wisca/gr-cedr)
- Intelligent scheduler available from GNURadio applications

1.2. Future Work and Research Directions

- Verify External Hardware RF Source/Sink blocks running in CEDR
- Enable CustomBuffers in `gr-cedr blocks`
- Build a GNURadio 4.0 scheduler utilizing the DASH Intelligent Scheduler

2. Background

2.1. CEDR

As illustrated in Figure 1, CEDR is composed of two components: a compilation workflow and a runtime workflow.

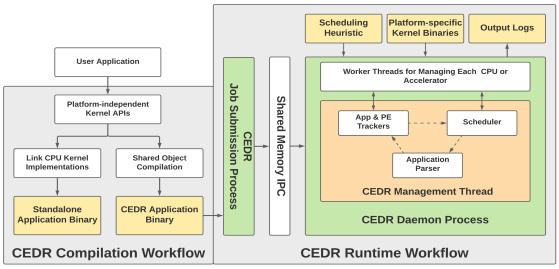


Figure 1. High level architectural overview of CEDR ecosystem.

The compilation workflow is used to convert C/C++ applications into CEDR-compatible binaries, and the runtime workflow is leveraged to then parse, schedule, dispatch, and execute those applications across a heterogeneous pool of resources on a given compute platform. CEDR provides these capabilities while remaining independent of any scheduling heuristic or hardware platform, ensuring that it can be ported across any number of execution environments without requiring major development effort to port any given scheduler to a new SoC platform or vice versa.

In this ecosystem we designed and developed front-end compilation flow to refactor applications into a sequence of hardware agnostic function calls and generate an application representation in the form of a flexible binary structure. This representation allows the run time system to invoke each function call on its supported processing elements (PEs) based on the task to PE mapping decisions made by the integrated intelligent scheduler (IS) at runtime. Programmer is given specification for API calls for key kernels that have support for acceleration. Our front-end compilation flow refactors applications into a sequence of hardware agnostic function calls and generates an application representation in the form of a flexible binary structure. We designed and implemented DASH-APIs and validated their support by both our compile time and run time system. Here when users make an API call, they do not need to specify where this task will be executed. The IS framework makes these scheduling decisions dynamically in the order of nanoseconds. The goal of the IS framework is to remove the burden of the user to study the intricate details of the hardware architecture and processing elements. It automatically chooses the best PE to execute the ready tasks as a function of the computational kernels that make up the tasks and the PE states. For example, if there is a hardware accelerator tailored for a specific task, IS checks its current workload and schedules the task to the PE that maximizes the design objectives, such as performance. IS achieves this objective by using a suite of machine learning (ML)-based and traditional algorithms. Our novel ML-based schedulers formulate the scheduling as a classification problem and imitate complex optimization algorithms with orders of magnitude faster runtime. Similarly, our suite of schedulers, such as real-time heterogeneous earliest finish time (HEFT-RT) (Fusco et al., 2022), provide a wide range options to the target users.

CEDR provides runtime capabilities while remaining independent of any one scheduling heuristic or hardware platform, ensuring that it can be ported across any number of execution environments without requiring major development effort to port any given scheduler to a new SoC platform or vice versa. We have built a runtime manager that operates in the Linux user space, operates as a background Daemon Process, and the user submits jobs for execution via inter-process communication (IPC) using the Job Submission Process and allows end users to interact with the heterogeneous architecture by submitting their compiled applications as if they are interacting with an HPC system.

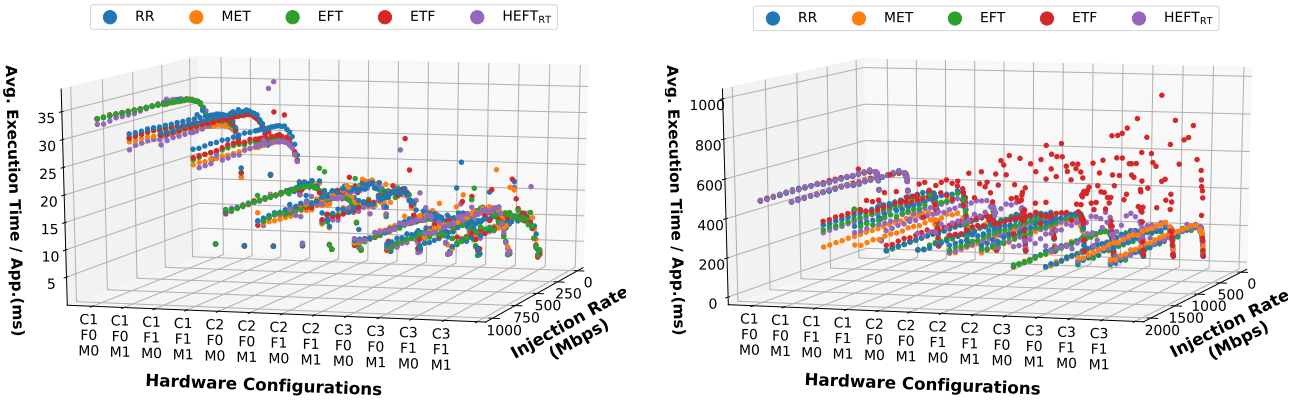
The daemon process consists of two key components: the Worker Threads and the CEDR Management Thread. For each resource in the system – whether it is a CPU core or an accelerator – we spawn one worker thread that is tasked with receiving, executing, and reporting back on work assigned to that particular resource. As an example, suppose we are running on a system with one CPU core (CPU 1) and one FFT accelerator (FFT 1). In this case, the worker thread for CPU 1 is, itself, assigned via its processor affinity to run on CPU 1, and as it receives tasks that are scheduled to CPU 1, it executes them and reports back to the management thread on their completion. Meanwhile, the worker thread for FFT 1 is tasked with running on one of the CPU cores and facilitating data transfers to and from the underlying accelerator. One of the advantages of this architecture is that we can easily scale to systems with any number of resources simply by changing the number of worker threads spawned to manage them. These worker threads are managed using the widely utilized POSIX thread library by the main CEDR management thread, and when coupled with the fact that all of the components shown here operate in Linux userspace, we can see that CEDR is trivially portable across a wide range of Linux-based SoC platforms.

The CEDR management thread, shown with orange filled box in Figure 1, operates in a continuous loop of application parsing, application PE tracking, and task scheduling on a system where user application binaries arrive dynamically. The application parser forms the entry-point by which applications are received by the runtime. Each CEDR application is submitted in the form of a flexible binary format (also known as a "Fat Binary") that contains the various invocations needed by each node for each heterogeneous PE. As applications are received, the application parser reads the provided binary objects and initializes CEDR's internal application representation. These parsed applications are themselves cached and stored as "application prototypes" such that, if they are to be submitted

again in the future, the runtime doesn't need to re-parse, instead just instantiating another copy. As parsing completes, applications are handed off to the application PE tracker, which begins by pushing the head nodes from each application – the nodes with empty predecessors lists – into the runtime's ready queue for scheduling and dispatch. From there, tasks are allocated by the user's specified scheduling heuristic to run on particular PEs by passing them to their corresponding worker threads and choosing the appropriate function that was previously parsed from their platforms list. As tasks are received and executed by the various worker threads, they signal their completion back to the application PE tracker, which responds by checking the dependency resolution of their successor nodes and pushing them into the ready queue as necessary, after which the process repeats. If an application runs out of successors to enqueue, it is marked as completed by the runtime, timing logs are generated representing its execution, and the memory associated with the application instance is released. These timing logs capture all of the relevant scheduling and timing information about when each task in a given application ran, on which PE it ran, and so on. This cycle of application parsing, application dispatch, and log generation repeats indefinitely until an IPC command is received that signals for the runtime to terminate.

The runtime manager relies on the integrated IS for task to PE mapping decisions, manages execution of incoming applications to completion, monitors the state of execution on each PE, and collects performance counters at task level. CEDR allows users to specify a workload composed of any number and combination of distinct applications and evaluate performance of pre-silicon hardware configurations composed of mixtures of ARM CPU cores and accelerators (e.g., FFT, matrix multiplication). Then, the runtime process consists of two key components: the Worker Threads and the CEDR Management Thread. For each resource in the system – whether it is a CPU core or an accelerator – we spawn one worker thread that is tasked with receiving, executing, and reporting back on work assigned to that particular resource. One of the advantages of this architecture is that we can easily scale to systems with any number of resources simply by changing the number of worker threads spawned to manage them. These worker threads are managed using the widely utilized POSIX thread library by the main CEDR management thread, and when coupled with the fact that all of the components shown here operate in Linux userspace, we can see that CEDR is trivially portable across a wide range of Linux-based SoC platforms. In the literature, various frameworks have been proposed that enable exploration of certain aspects of this design space – such as SoC and application design without a focus on scheduling (Chen et al., 2016; Mantovani et al., 2020; Nazarian & Bogdan, 2020; Shao et al., 2016) or standalone application programming interfaces that are independent of hardware (Huang et al., 2019; Sujeeth et al., 2014) – to the best of our knowledge, no frameworks thus far (Auerbach et al., 2012; Bolchini et al., 2018; Boutellier et al., 2018; Christodoulis et al., 2018; Hsieh et al., 2019; Moazzemi et al., 2019; Tan et al., 2019) have been presented that bring together application development, resource management, and accelerator design capabilities into a single unified compilation and runtime toolchain that targets DSSoC hardware. We believe that the CEDR ecosystem, with its integrated compile-time and runtime workflows, empowers researchers to conduct design space explorations, and consequently, it will help the research community move towards establishing a more general understanding of DSSoCs and their broader role in an era of increasingly heterogeneous computing systems. We demonstrated capabilities of the integrated software ecosystem on FPGA based emulation platforms (Xilinx Zynq UltraScale MPSoC-ZCU102, Xilinx Virtex Ultrascale+ -VCU128) through successful execution of a wide variety of workloads composed of randomly arriving mixture of five applications (Pulse doppler, Temporal mitigation, Radar correlator, SAR RDA, and WiFi TX) with ability to dispatch tasks to available pool of compute resources based on heuristic schedulers (Heterogeneous Earliest Finish Time, Earliest Finish Time, Earliest Task First, Minimum Execution Time, Round Robin) along with Imitation Learning (Regression Tree and Deep Neural Network) based schedulers.

Figure 2 shows an example experiment where we executed mixture of low latency applications (Radar Correlator, Temporal Mitigation in Figure 2(a)) and high latency applications (WiFiTX, Pulse Doppler Figure 2(b)) over 12 different SoC configurations, 29 different job injection and five different scheduling heuristics. This experiment covers total of 3480 configurations, and involves scheduling over 10 million tasks. The entire experiment took less than 3 hours on the ZCU102 platform. This is by orders of magnitude faster than cycle accurate and discrete event based simulators. The DASH software ecosystem is portable across a wide number of Linux-based systems, ensuring that effort to migrate across systems is minimal for all developers involved. We verified portability of our ecosystem across other platforms such as x86, Odroid-XU3, and Nvidia Xavier GPU through dynamically arriving workload scenarios. Because CEDR is portable, we can implement an approach that meets the program goals with regards to effective utilization of COTS hardware while also providing a path forward for fully custom hardware designs.



(a) (b)

Figure 2. Average execution time per application for 12 resource pool configurations, 5 schedulers and 29 injection rates, using (a) *low* latency, (b) *high* latency workloads.

2.2. CEDR Programming Model

Developers in general are incredibly familiar with the contracts and expected behavior that come with APIs. As such, by developing support for APIs, our aim is to increase productivity among current DASH software developers by providing a familiar workflow and the basic idea is to refactor a program into two threads. The first thread would contain the execution for all the extraneous non-kernel code that an application needs to run correctly (such as one-time file I/O, printing, or memory allocation and deallocation), and then the second thread contains all the code that is determined to be "kernels". The program executes by handing off back-and-forth between these two threads by "resolving barriers" in each thread (that are placed to ensure we preserve the program's execution). In this workflow, from the non-kernel code's perspective, each kernel is represented as a self-contained collection of code, and if we simply were to insert the intelligent scheduler into that gap between the non-kernel and kernel messaging, it would be conceivably quite simple to interchangeably swap those kernel invocations to run on different PEs. As we can see, this methodology relies on having a well-defined function-like interface sitting between the kernel and non-kernel code. And for that, we decided representing our DASH kernels as API calls as illustrated in Figure 3.

Taking an FFT execution as the motivating example, when we originally started the project, we first answered the question of "what does a DASH program look like" with any arbitrary collection of code or library calls that implement an FFT. While this is still the eventual goal, as a starting point, because it consisted of arbitrary code, it was fairly difficult for machines to deeply understand and reason about. Likewise, because of that, it was difficult for downstream compiler tooling to effectively map that code to heterogeneous PEs. As a next step, labels were introduced into the DASH corpus, and these have proven to be quite useful for enabling machine classification of code as we've seen in previous ontology results. However, for current developers and compilers, they were still a bit difficult to leverage correctly. For developers looking to just test the software toolchain, it was difficult to know the exact boundaries of where your label should be placed in the code for the compilation tooling to effectively recognize it, and for the corresponding compilers, this ambiguity around kernel entrances/exits made it difficult to reason about which variables corresponded to which inputs/outputs of the kernel, and that made it difficult to swap those kernels out for implementations on different hardware units.

2.3. CEDR and API Integration

From the user's perspective, they only need to write code using a library of APIs in place of their kernels similar to how previous toolchains required hand-labeling the existing kernels in their code. The code written with their APIs is then passed through an LLVM opt pass that swaps out those API calls for calls to invoke the intelligent scheduler, and a barrier is inserted into the code that halts execution while the IS schedules and dispatches that kernel so that we can ensure we preserve correctness – although the "removal" of these barriers based on dynamic analysis of execution dependencies is something we're actively investigating. An example of this with regards to FFT is provided in Figure 4

In this case, the user simply needs to write their code using the DASH_FFT API, and then during compilation, that DASH_FFT call is replaced (at the LLVM level) with code that looks like the tool-generated code on the right. In

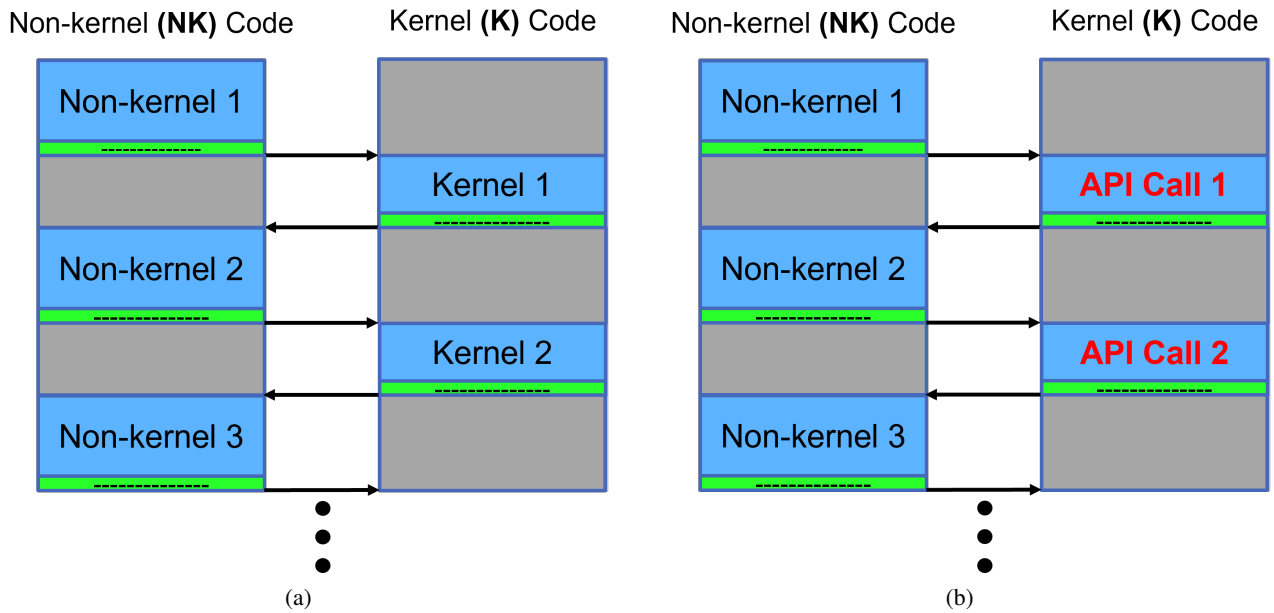


Figure 3. (a) Kernel, Non-Kernel based application representation (b) API-based representation

the replaced code, a pthread_barrier is initialized, the "enqueue_kernel" function is called to push a node into the intelligent scheduler's todo queue along with all required arguments, the IS picks the best invocation of that kernel (CPU/accelerator) based on the system state, and eventually it signals completion by resolving the barrier. In the tool generated code, FFT API call is replaced with the enqueue kernel function with barriers placed before and after. This is our way of setting up hooks for the runtime system so that FFT is dispatched to a PE after all its dependencies are resolved. We then compile this refactored application code to generate a flexible binary structure such that each task in the application can be invoked on any of the processing elements that support that task. For example the binary for the FFT task has three versions for possible execution on ARM core, DAP or a dedicated accelerator. This binary representation gives flexibility to the run time system on choosing where to invoke each function call among the supported processing elements. We don't want application developer to make greedy choices and implement an application with static schedules favoring only an accelerator for certain tasks because this system will be used by multiple applications concurrently. More importantly we don't want users to worry about how to do workload partitioning and programming on a complex heterogenous architecture.

For choosing our initial set of APIs, we started by identifying the most common labeled kernels among our DASH corpus, we came up with set of FFT and GEMM API calls. With these APIs developed, we took the original unmodified source code for five of our most frequently used applications. We measured developer productivity in a scenario where developers, without prior exposure to our software tooling, ported and validated SAR, WiFiTX, Pulse Doppler, Radar Correlator and Temporal Mitigation aapplications, and the average time spent per developer was approximately 3 hours including the time to port their code to leverage the DASH-APIs as well as compile, execute, and validate its functionality. In summary, the DASH ecosystem comes with a hardware-software co-design environment that provides a productive abstraction layer over which software can be programmed in a hardware-independent manner and then dynamically mapped and executed over a variety of heterogeneous computation units (Mack et al., 2022).

2.4. Intelligent Scheduler

Heterogeneous SoCs provide substantial improvements in performance and energy by integrating a variety of hardware accelerators. A myriad of PEs provides an abundance of scheduling decision choices for streaming applications. Streaming scenarios present significant challenges to scheduling since the incoming tasks and applications observe different system states, i.e. busy states of the PEs and partially filled wait and execution queues. For example, an incoming FFT task can potentially be executed in the general-purpose cores, GPU and dedicated hardware accelerators. In this case, hardware accelerators are natural choices since they optimize for specific operations and provide superior performance and energy efficiency. However, having a task to wait for highly busy accelerator units

```c
#include "dash.h"
int main(){
  double *input = (double*) malloc…
  double *output = (double*) malloc…
  DASH_FFT(input, output, size, forwardTrans);
}
```

(a)

```c
#include "dash.h"
int main(){
  double *input = (double*) malloc…
  double *output = (double*) malloc…
  //---------------- Replaced portion ---------------
  pthread_barrier_t kernel_1_barrier;
  pthread_barrier_init(&kernel_1_barrier, nullptr, 2);
  enqueue_kernel("FFT", &input, &output, &size,
                        &forwardTrans, &kernel_1_barrier);
  pthread_barrier_wait(&kernel_1_barrier);
  // -------------------------------------------------
}
```

(b)

Figure 4. API based application development approach (a) FFT API call in the user application (b) Refactored user application with enqueue kernel and thread barriers

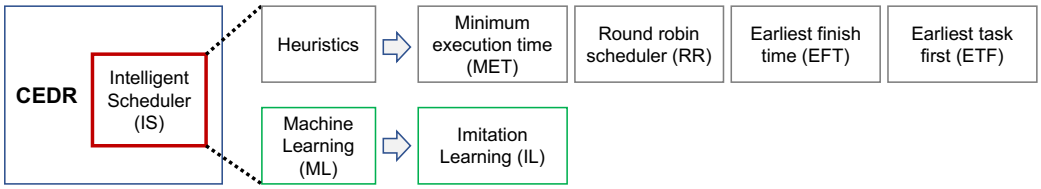


Figure 5. The intelligent scheduler (IS) framework integrates into CEDR and comprises a variety of heuristic and machine learning scheduling algorithms.

can override the benefits they provide. So, schedulers must carefully consider the system state and optimally choose a PE for a task at runtime to exploit the potential of heterogeneous platforms.

Specialized processors such as GPUs and hardware accelerators execute tasks significantly faster than general-purpose cores, even in the order of nanoseconds. So, scheduling decisions must incur ultra-low overheads to facilitate the fast execution of PEs in DSSoCs. Scheduling algorithms are implemented as optimization based approaches, heuristic methods and machine learning methods. Approaches that rely on optimization techniques formulate the problem at hand using mathematical expressions, corresponding constraints and objective functions. Solving the optimization problem can be severely prohibitive in terms of runtimes, and hence cannot be used for dynamic scheduling in DSSoCs. Heuristics are commonly used in practice as means to trade off the scheduling complexity and overheads with the quality of the decisions. They are also tailored to particular objectives such as performance, power, and energy consumption. To this end, we introduce the intelligent scheduler (IS) (shown in Figure 5) which provides a flexible interface to seamlessly assign tasks to PEs and prioritize them for dispatch to execution. The IS integrates a suite of schedulers, and provides the scheduling algorithm choice to the user by integrating into CEDR.

Heuristic schedulers are tailored to a particular set of scenarios and fail to generalize well, and also incur signficant runtime overheads. Machine learning techniques provide the adaptability and ability to learn in several environments. In particular, reinforcement learning (RL) has been used for scheduling in data clusters. But, it suffers from convergence issues for large problem sizes such as DSSoCs. RL also demands substantial computational resources and time to effectively explore the solution space and learn the optimal decisions. All prior approaches suffer from runtime overheads, sub-optimality, computational time and resource overheads. To address these challenges, we use an imitation learning (IL)-based scheduling approach that generates an Oracle using any complex scheduler offline and approximates it using light-weight decision tree classifiers at runtime. Generating the Oracle offline allows us to deploy optimal/near-optimal schedulers offline without being limited by runtime overheads. The use of supervised machine learning techniques such as decision trees approximates the complex decisions using light-weight classifiers that can be easily deployed at runtime. Therefore, IL-based scheduling approach addresses the challenges of scheduling decision quality, runtime overheads and generalization to multiple use cases.

Figure 6 provides an overview of the proposed IL-based scheduling framework. As a reminder, the heterogeneous SoC under consideration is organized into processing clus-

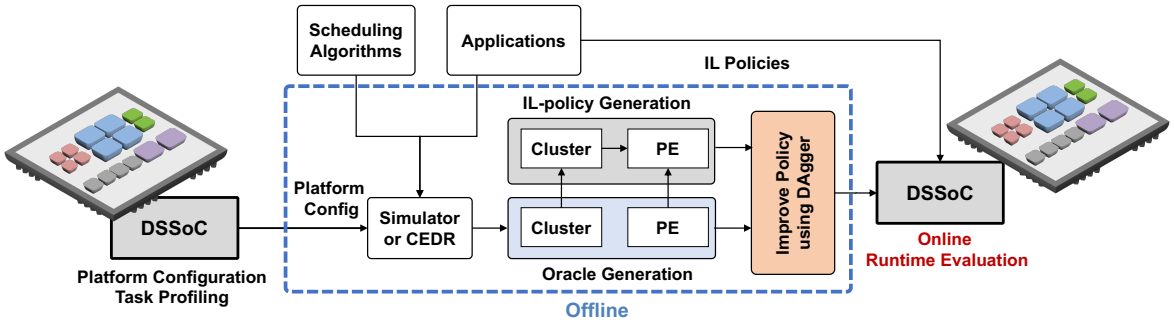


Figure 6. An overview of the IL-based Scheduling Framework.

ters, each of which contain multiple identical cores. In the figure, the cores of the same type are shaded by the same color. Real-world applications are profiled and characterized on commercially available hardware platforms to obtain task latency and power consumption estimates. A simulator or CEDR use the task profiling information, application directed flow graphs (DFGs) and scheduling algorithms to generate the Oracle. In this framework, we use a hierarchical approach to breakdown the complex scheduling problems into smaller sub-problems. So, we generate the cluster-level Oracle and PE-level Oracles to reduce the complexity. Then, we train IL policies using any supervised machine learning techniques, such as logistic regression, decision tree classifier and multilayer perceptrons. The first-level of IL policy predicts the cluster to which a task should be scheduled to. Then, the second level of IL policies predict the exact PE within that chosen cluster for task execution. We then train IL policies, improve them using data aggregation (DAgger) and utilize them for runtime evaluation.

2.5. Interaction with the GNURadio Scheduler

The current GNURadio scheduling system operates with a a thread per block model and execution and and data flow is driven by backpressure. This appears similar to a completely fair scheduler to the IS. The IS inside of the CEDR runtime can then perform predictions and accelerator selection based on what operations are currently being requested in each thread as driven by the backpressure.

GNURadio 4.0 is in the process of revising this model and enabling alternative sceduling algorithms. We intend to take a future research direction and develop a GR 4.0 scheduler implementation that leverages the IS and the `gr-cedr` OOT.

3. GNURadio Interface to CEDR

As discussed previously, CEDR binaries are compiled applications bundled up as shared libraries with a `main` entrypoint. GNURadio flowgraphs can be formed into this structure simply through appropriate compilation and linking. C++ only flowgraphs make this very easy through CMake and a few extra lines to configure the appropriate build options.

Python flowgraphs can be built into a binary to be submitted to CEDR by leveraging Cython3 to embed an interpreter into a shared binary (`cython3 --embed`). This adds an additional step and time between building the flowgraph and execution time, but as CEDR improves and GNURadio evolves, we expect to be able to elimniate this additional complexity.

Once the flowgraph has been compiled and linked with the adapted process, the GNURadio flowgraph is then launchable within the CEDR runtime.

The `gr-cedr` Out-of-Tree (OOT) Module provides API calls and blocks for each of CEDR's supported runtime flexible accelerators.

3.1. CEDR/libdash API

CEDR is accompanied by a library called `libdash` that implements a number of traditional signal processing operations and includes accelerated implemntations for multiple platforms. These platforms include general purpose CPU, GPU, FPGA, and custom silicon accelerator blocks, such as the DSSoC-DASH Systolic Array Processor and other silicon accelerators.

3.2. Blocks

Currently, there is only one fully implemented block within the `gr-cedr` OOT. This blocks wraps arbitrary dimension FFT operations with CPU, GPU, FPGA, and systolic array implementations, and is able to dynamically switch between them at runtime.

CEDR and `libdash` already have a number of implemented operations available to accelerate and we intend to quickly implement more blocks leveraging those APIs.

3.3. Applications

We developed a simple range correlator in GNURadio as shown in Figure 7 to demonstrate broadened support for DASH Runtime integration into other runtime frameworks where the GNURadio user can instantly leverage DASH runtime and its scheduler, implement application flowgraphs, and compile for heterogeneous SoC execution.

Accelerators in GNU Radio have been previously limited to requiring the selection of an accelerator tor framework/block at design time. In this study, we present a preliminary investigation into supporting two new capabilities with GNU Radio: first, we illustrate the ability to execute GNU Radio blocks across a variety of heterogeneous accelerators. Second, we demonstrate that we are able to dynamically schedule these blocks across our pool of accelerators using easily-customizable scheduling policies. This work shows the broadened support for DASH runtime integration into other runtime frameworks where GNURadio developer is able to seamlessly execute applications compiled for the target off the shelf SoC platform and benefit from the accelerator support through CEDR based execution. We do this via a combination of out-of-tree modules and by embedding GNU Radio itself as an application in a heterogeneous runtime called CEDR.

Figure 8 shows the execution of native GNURadio application (Radar Correlator) with other DASH applications (WiFiTX and SAR) on an SoC composed of 2 FFT accelerators and 3 ARM cores on ZCU102 MPSoC where applications arrive dynamically, integrated scheduler handles task to PE mapping decisions and CEDR runtime workflow manages execution of all applications in coordination with the GNURadio runtime. We find that, when GNU Radio shares the system with other unrelated applications, our integrated IS dispatches FFT tasks to the accelerator along with the CPU cores fairly without compromising target throughput for each application. In summary, we show that running the GNU Radio runtime and applications inside/with CEDR enables better scheduling options and easier accelerator access by eliminating the need for users to partition their workloads at design time. We believe this is a stepping stone in broadening GNU Radio's support for heterogeneous execution and enabling it to hook more flexibly into a variety of scheduling heuristics. Taken together, the DASH software ecosystem is a capable environment for enabling seamless deployment of user applications and exploring the boundaries of productive application development, resource management development, and hardware configuration analysis for heterogeneous architectures.

4. Intelligent Scheduler

This section presents the demonstration of the intelligent IL-based scheduler (described in Section 2.4) for a workload that comprises a mix of GNU Radio and other DASH applications on two different platforms, Xilinx Zynq UltraScale+ ZCU102 FPGA and Nvidia Jetson AGX Xavier. Specifically, the pulse Doppler application constitutes the GNU Radio application. WiFiTX and SAR constitute the other DASH applications chosen for this demonstration. As discussed in Section 2.4, we first instrument CEDR to run a complex scheduler and generate the Oracle. We choose the EFT heuristic for these experiments. During this step, CEDR populates the list of features and labels for each in task the workload. This information is then used to train a DT classifier in Python using the scikit-learn library. The trained model is then plugged back into CEDR to perform runtime scheduling for the workload with streaming application arrivals. The rest of the section focuses on specific details and results on the two evaluation platforms.

4.1. Xilinx Zynq UltraScale+ ZCU102

To demonstrate the IL-based scheduling approach on Zynq ZCU102 FPGA, we choose an SoC configuration with 3 CPU cores and 2 FFT hardware accelerators. The EFT scheduler uses only one of the CPUs and both FFT hardware accelerators for scheduling at runtime. As a consequence, the DT-based IL policy also performs likewise, as shown in Figure 9(a). We can observe that the hardware accelerators are the obvious PE choice for the FFT dominated workload. The scheduling policy only reverts to the CPU core when both hardware accelerators are busy executing other tasks and have a significant waiting time. Therefore, the IL scheduling policy effectively utilizes the PEs at runtime.

4.2. Nvidia Jetson AGX Xavier

On the Nvidia Jetson AGX Xavier platform, we choose a resource configuration of 2 CPU cores and the GPU to execute the workload. The GPU offers substantially lower execution times than the CPU cores and hence, a majority of the workload is executed on the GPU. The scheduling policy prefers to execute the tasks on the CPU cores only when the GPU is significantly blocked for a long period of time, as shown in Figure 9(b).

4.3. Summary

We successfully demonstrate that the IL-based scheduling polices are applicable and portable across different platforms. Furthermore, the framework seamlessly integrated into CEDR, thereby providing an easy plug-and-play environment for user evaluation.

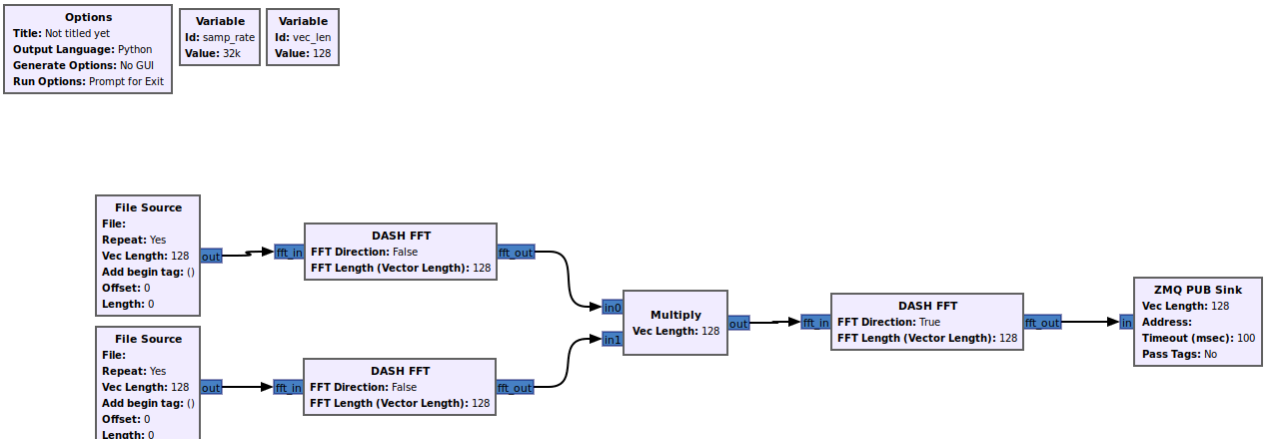


Figure 7. GNURadio Flow graph used for the radar correlator example application with run-time accelerator selection

5. Conclusion

In this paper, we demonstrate the ability to execute GNURadio flowgraphs across a variety of heterogeneous accelerators (FPGA and GPU) by replacing common computational blocks with those from the `gr-cedr` OOT module. We also demonstrate the dynamic scheduling of these blocks across our pool of accelerators using easily-customizable scheduling policies at runtime.

References

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

Bliss, Daniel W. Development of next gen processors for rf systems. In *DARPA Electronics Resurgence Initiative Summit*, 2020.

Bolchini, Cristiana, Cherubin, Stefano, Durelli, Gianluca C., Libutti, Simone, Miele, Antonio, and Santambrogio, Marco D. A runtime controller for opencl applications on heterogeneous system architectures. *SIGBED Rev.*, 15(1):29–35, March 2018. doi: 10.1145/3199610.3199614. URL https://doi.org/10.1145/3199610.3199614.

Boutellier, Jani, Wu, Jiahao, Huttunen, Heikki, and Bhattacharyya, Shuvra S. Prune: Dynamic and decidable dataflow for signal processing on heterogeneous platforms. *IEEE Transactions on Signal Processing*, 66(3):654–665, 2018. doi: 10.1109/TSP.2017.2773424.

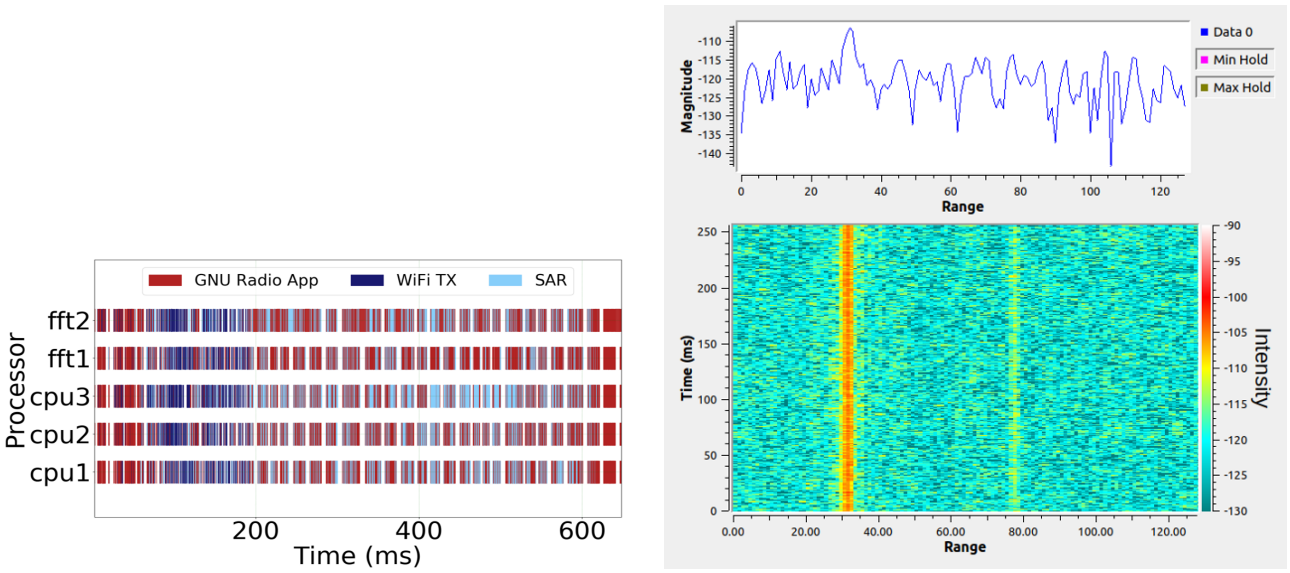
Chen, Yu-Ting, Cong, Jason, Fang, Zhenman, Xiao, Bingjun, and Zhou, Peipei. ARAPrototyper: Enabling Rapid Prototyping and Evaluation for Accelerator-Rich Architectures. *arXiv:1610.09761 [cs]*, October 2016. URL http://arxiv.org/abs/1610.09761. arXiv: 1610.09761.

Chiriyath, Alex R., Herschfelt, Andrew, Srinivas, Sharanya, and Bliss, Daniel W. Technological advances to facilitate spectral convergence. In *2021 55th Asilomar Conference on Signals, Systems, and Computers*, pp. 623–628, 2021. doi: 10.1109/IEEECONF53345.2021.9723312.

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

Fusco, Alexander, Hassan, Sahil, Mack, Joshua, and Akoglu, Ali. 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)*, October 2022. doi: https://arxiv.org/abs/2207.11360.

Hsieh, Chenying, Sani, Ardalan Amiri, and Dutt, Nikil. 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)*, pp. 136–141, October 2019. doi: 10.1109/VLSI-SoC.2019.8920374. ISSN: 2324-8432.



(a) (b)

Figure 8. (a) GNU radio application (Radar Correlator) executing with dynamically arriving WiFiTx and SAR applications through CEDR on a SoC with 3 ARM cores and 2 FFT accelerators emulated on ZCU120 FPGA platform. (b) Waterfall image collected at runtime from CEDR based execution is identical to the output of the baseline implementation.

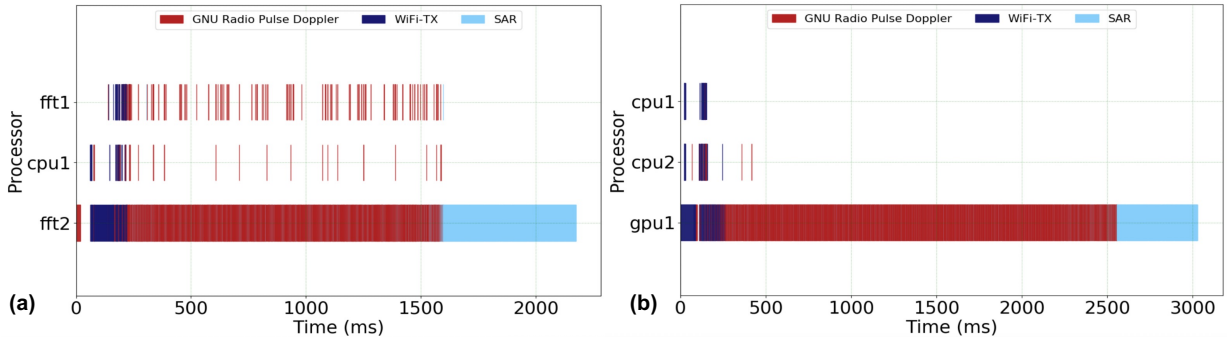


Figure 9. Gantt charts of the workload execution on (a) Xilinx Zynq ZCU102 FPGA and (b) Jetson AGX Xavier for a workload comprising the GNU Radio Pulse Doppler, WiFiTX and SAR applications.

Huang, Tsung-Wei, Lin, Chun-Xun, Guo, Guannan, and Wong, Martin. Cpp-Taskflow: Fast Task-Based Parallel Programming Using Modern C++. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 974–983, May 2019. doi: 10.1109/IPDPS.2019.00105. ISSN: 1530-2075.

Mack, Joshua, Hassan, Sahil, Kumbhare, Nirmal, Gonzalez, Miguel Castro, and Akoglu, Ali. Cedr - a compiler-integrated, extensible dssoc runtime. *ACM Trans. Embed. Comput. Syst.*, mar 2022. ISSN 1539-9087. doi: 10.1145/3529257. URL https://doi.org/10.1145/3529257. Just Accepted.

Mantovani, Paolo, Giri, Davide, Di Guglielmo, Giuseppe, Piccolboni, Luca, Zuckerman, Joseph, Cota, Emilio G., Petracca, Michele, Pilato, Christian, and Carloni, Luca P. Agile soc development with open esp : Invited paper. pp. 1–9, Nov 2020. ISSN 1558-2434.

Moazzemi, Kasra, Maity, Biswadip, Yi, Saehanseul, Rahmani, Amir M., and Dutt, Nikil. HESSLE-FREE: Heterogeneous systems leveraging fuzzy control for runtime resource management. *ACM Transactions on Embedded Computing Systems*, 18(5s):74:1–74:19, October 2019. ISSN 1539-9087. doi: 10.1145/3358203. URL http://doi.org/10.1145/3358203.

Nazarian, Shahin and Bogdan, Paul. S4oC: A Self-Optimizing, Self-Adapting Secure System-on-Chip Design Framework to Tackle Unknown Threats — A Network Theoretic, Learning Approach. In *2020 IEEE International Symposium on Circuits and Sys-*

tems (ISCAS), pp. 1–8, October 2020. doi: 10.1109/ISCAS45731.2020.9180687. ISSN: 2158-1525.

Shao, Yakun Sophia, Xi, Sam Likun, Srinivasan, Vijayalakshmi, Wei, Gu-Yeon, and Brooks, David. Co-designing accelerators and SoC interfaces using gem5-Aladdin. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, October 2016. doi: 10.1109/MICRO.2016.7783751.

Sujeeth, Arvind K., Brown, Kevin J., Lee, Hyoukjoong, Rompf, Tiark, Chafi, Hassan, Odersky, Martin, and Olukotun, Kunle. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Transactions on Embedded Computing Systems*, 13(4s):134:1–134:25, April 2014. ISSN 1539-9087. doi: 10.1145/2584665. URL http://doi.org/10.1145/2584665.

Tan, Xubin, Bosch, Jaume, Álvarez, Carlos, Jiménez-González, Daniel, Ayguadé, Eduard, and Valero, Mateo. A hardware runtime for task-based programming models. *IEEE Transactions on Parallel and Distributed Systems*, 30(9):1932–1946, 2019. doi: 10.1109/TPDS.2019.2907493.