Mozart: Taming Taxes and Composing Accelerators with Shared-Memory

Vignesh Suresh
University of Illinois Urbana-Champaign
Urbana, Illinois, USA
vv15@illinois.edu

Bakshree Mishra
University of Illinois Urbana-Champaign
Urbana, Illinois, USA
bmishra3@illinois.edu

Ying Jing
University of Illinois Urbana-Champaign
Urbana, Illinois, USA
yingj4@illinois.edu

Zeran Zhu
University of Illinois Urbana-Champaign
Urbana, Illinois, USA
zzhu35@illinois.edu

Naiyin Jin
University of Illinois Urbana-Champaign
Urbana, Illinois, USA
naiyinj2@illinois.edu

Charles Block
University of Illinois Urbana-Champaign
Urbana, Illinois, USA
coblock2@illinois.edu

Paolo Mantovani
Columbia University
New York, New York, USA
paolo@cs.columbia.edu

Davide Giri
Columbia University
New York, New York, USA
davide_giri@cs.columbia.edu

Joseph Zuckerman
Columbia University
New York, New York, USA
jzuck@cs.columbia.edu

Luca Carloni
Columbia University
New York, New York, USA
luca@cs.columbia.edu

Sarita Adve
University of Illinois Urbana-Champaign
Urbana, Illinois, USA
sadve@illinois.edu

Abstract

Resource-constrained system-on-chips (SoCs) are increasingly heterogeneous with specialized accelerators for various tasks. Acceleration taxes due to control and data movement, however, diminish end-to-end speedups from hardware acceleration. Meanwhile, emerging workloads are increasingly task-diverse with several, potentially shared, fine-grained acceleration candidates. This motivates a paradigm of parallel and disaggregated acceleration. Compared to a monolithic accelerator, disaggregation provides higher flexibility, reuse, and utilization, but at the cost of higher control and data acceleration taxes.

We propose a novel SoC architecture, *Mozart*, that enables efficient accelerator disaggregation by leveraging shared-memory to tame control and data acceleration taxes. To address the control tax, Mozart includes a lightweight, modular, and general accelerator synchronization interface (ASI). ASI eliminates the typical CPU-centric accelerator control in favor of a decentralized, uniform synchronization interface through shared-memory. This enables accelerators to directly and transparently synchronize with each other (or CPUs) using the same shared-memory interface as CPUs. To address the data tax, Mozart leverages the Spandex-FCS heterogeneous coherence protocol, which supports decentralized data movement and per-word coherence specialization. We demonstrate the first RTL implementation of Spandex-FCS and the first evaluation of its benefits for a heterogeneous SoC with fixed-function accelerators, running real-world applications with Linux.

Mozart simultaneously enables, for the first time, (1) *finer-grained acceleration* than previously possible, (2) *programmable and transparent composition* of fine-grained, disaggregated accelerators, (3) efficient *accelerator pipelining* through shared-memory and decentralization, and (4) *a performance-competitive disaggregated alternative* to specialized monolithic accelerators. We demonstrate these capabilities of Mozart with a comprehensive one-of-a-kind evaluation of more than 70 hardware configurations prototyped on an FPGA employing various accelerators, running real-world applications on Linux, and a scalability analysis with up to 15 accelerators. We also present an analytical performance model to understand and explore system design choices and to validate the results.

CCS Concepts

• **Computer systems organization → Parallel architectures**.

Keywords

Heterogeneous Systems, Shared-Memory, Cache Coherence, Disaggregated Acceleration, Accelerator Synchronization

ACM Reference Format:
Vignesh Suresh, Bakshree Mishra, Ying Jing, Zeran Zhu, Naiyin Jin, Charles Block, Paolo Mantovani, Davide Giri, Joseph Zuckerman, Luca Carloni, and Sarita Adve. 2024. Mozart: Taming Taxes and Composing Accelerators with Shared-Memory. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '24), October 14–16, 2024, Long Beach, CA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3656019.3676896

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

PACT '24, October 14–16, 2024, Long Beach, CA, USA

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0631-8/24/10
https://doi.org/10.1145/3656019.3676896

1 Introduction

With the end of Dennard Scaling, hardware specialization has emerged as a key technique to improve performance. It is common to evaluate the benefit of a standalone hardware accelerator

Table 1: Opportunities for disaggregated fine-grained accelerators in different domains.

Extended Reality [38]	Reuse of Cholesky, Jacobian, GEMM, Gauss-Newton within Visual Intertial Odometry (VIO); Reuse of Cholesky, Jacobian, Gauss-Newton between VIO and Scene Reconstruction; FMADDs and Trascendentals across Audio and Hologram; FFT within Audio.
Earable Computing [13]	Reuse of FFT, GEMM and convolution across multiple applications (e.g., FFT in TinySR, Ambisonic, and HRTF).
Hyperscale Computing [30]	Reuse of common microservices and components of the "datacenter tax" [29, 41, 74] across datacenter workloads. Recent work [30] introduces the concept of a "sea of accelerators" for big data workloads that leverage accelerator reuse and chaining (although no chained implementations are proposed).
Database Management Services (DBMS) [86]	SQL queries used in DBMS can be decomposed into fine-grained producer-consumer tasks (Col Select, Col Filter, Joiner, Sorter, etc.) and reused across different query types.
Genomics [33]	Genomic data processing tasks can be represented as Extended SQL queries, reusing DBMS primitives.
Machine Learning [93]	Deep learning workloads contain matrix multiply (MM) layers of different sizes and can be reused across different layers and networks.
Wearables [79]	Finger-gesture recognition, CNN-based image recognition and transportation context-detection, etc., comprise producer-consumer chains of fine-grained tasks like FFT, convolution, dynamic time warp (DTW), and AES encryption.
Brain-Computer Interfaces (BCI) [42, 73]	Frequently used BCI pipelines like Seizure Detection can reuse common primitives such as FFT, discrete wavelet transform (DWT) and support vector machine (SVM).
Medical Imaging [23]	Denoise, Deblur, Registration, Segmentation, etc. reuse FInv, FSqrt, FDiv, and Polynomial-16 (Poly16).
Cross-domain [43, 83]	Several kernels like FFT, logistic regression (LR) and MPEG-decode are used across different domains like "memory-enhance" and "video-sync".

and then use simple models like Amdahl's Law to project end-to-end application performance benefits from acceleration. However, when integrating such accelerators into a system-on-chip (SoC), the end-to-end application performance benefits can diminish due to system costs that are often not captured in standalone evaluations. These costs include accelerator invocations, data movement to and from the accelerator, pre- and post-processing of data for accelerators, etc.

We refer to the above costs as the *acceleration tax*: the additional cost paid by an application to offload a task to an accelerator. Prior work has introduced taxes that impact end-to-end performance in other domains like datacenter tax [41] and AI tax [64]. Our work introduces acceleration taxes in a similar vein to highlight the impact of these costs on end-to-end hardware accelerated performance.

Along with the rise in hardware specialization, emerging workloads are also showing high task diversity, covering multiple domains [30, 38, 42, 43, 83]. These workloads do not exhibit a single dominant task that can benefit from acceleration; instead, end-to-end performance benefit requires accelerating several, often fine-grained, tasks. For instance, Gonzalez et al. [30] advocate for a sea of (fine-grained) accelerators for key datacenter and system tax operations. Huzaifa et al. [38] show that a baseline extended reality (XR) system consists of 27 high-level tasks, each of which has several finer-grained kernels common across the high-level tasks that could be specialized. Karageorgos et al. [42] propose a brain-computer interface (BCI) system that includes processing units for small kernels frequently used in BCI pipelines. Recent works [43, 83] also highlight the importance of reusing multiple accelerators across application domains.

Table 1 summarizes past work from diverse application domains that can exploit multiple fine-grained accelerators for their constituent tasks, each of which can be composed on-demand, reused and shared among various tasks. This leads to a paradigm of parallel and fine-grained disaggregated accelerators [23, 30, 43, 44, 54, 83], analogous to the ideas of disaggregated compute, memory, storage, and networking successfully deployed in other contexts [2, 24, 31, 49, 56, 68, 71]. Compared to using overly specialized monolithic accelerators, disaggregation affords increased flexibility, reuse, and utilization of the fine-grained accelerators. Unfortunately, while monolithic accelerators achieve efficiency through custom hardware interfaces and data paths to tightly integrate the constituent fine-grained kernels [10, 51, 61], disaggregation can compromise efficiency due to higher acceleration taxes.

This work seeks to leverage shared-memory to reduce the impact of two acceleration taxes for an SoC: control taxes due to bulky accelerator invocations, and data taxes due to additional data movement.

Taming the Control Tax. We define the control tax as the overhead to invoke the accelerator and ascertain its completion. Traditionally, accelerators are invoked by host CPUs through bulky operating system (OS) calls such as `ioctl` [69]. The overhead of such an invocation increases the overall accelerator execution time, limiting the speedup from hardware acceleration (as also observed by [7, 9, 48, 53]).

To reduce the control tax, we propose a lightweight *Accelerator Synchronization Interface (ASI)* to synchronize with the CPU or with other accelerators through shared-memory. We envision ASI as a modular, general synchronization interface for accelerators that eliminates the CPU-centric model of accelerator control, and promotes accelerators as first-class system compute units. With ASI, all units (accelerators or CPUs) have a uniform synchronization interface mediated by shared-memory. Thus, we construct a view of *transparent acceleration* where compute units are agnostic to whether they synchronize with a CPU, a specialized monolithic accelerator, or a sea of disaggregated accelerators (that synchronize with each other).

To enable this vision, we design ASI as a modular shim that can be seamlessly integrated with an accelerator, irrespective of its level of programmability or specialization. We implement a lightweight finite-state machine (with negligible area overhead) that performs shared-memory reads and writes for synchronization, with the option of leveraging benefits from the provided cache coherence protocols.

Taming the Data Tax. We define the data tax as the overhead to move data to and from accelerators (relative to without accelerator offload). Traditionally, data movement to and from accelerators was achieved through explicit data copying between different address spaces through DRAM. Recently there has been a significant momentum towards systems where CPUs and accelerators are part of a common cache-coherent shared-memory address space, fostered by various industry standards [5, 19, 28, 70, 75, 76]. Such a solution eliminates the need for expensive data copies and improves programmability and efficiency through implicit data orchestration.

Despite the benefits of coherent shared-memory, a coherence protocol such as MESI can incur significant overhead when used in heterogeneous systems [3, 4, 72]. To reduce this overhead, we

leverage the Spandex protocol [4] with fine-grained coherence specialization (FCS) [3]. Spandex-FCS is a state-of-the-art modular and extensible protocol that allows per-access specialization of coherence to meet the diverse needs of a heterogeneous system. Prior to this work, Spandex-FCS was evaluated only for systems with GPUs (and CPUs) and only in architectural simulation. This work, for the first time, provides an RTL-level implementation of Spandex-FCS for a heterogeneous SoC with several fixed-function accelerators and RISC-V CPUs — enabling validation of its claimed flexibility, modularity, and performance. We specifically use Spandex-FCS to enable efficient data forwarding directly to a consumer's cache, virtually hiding all read latency (which is on the critical path).

Mozart. We propose a novel SoC architecture called *Mozart* that combines the benefits of ASI and Spandex-FCS to tame control and data taxes. Mozart leverages the benefits of shared-memory to provide a programmable, transparent, and performant interface for accelerator synchronization and communication. As a result, Mozart simultaneously enables, for the first time, (1) *finer-grained acceleration* than previously possible, (2) *programmable and transparent composition* of fine-grained, disaggregated accelerators, (3) efficient *accelerator pipelining* through shared-memory and decentralization, and (4) *a performance-competitive disaggregated alternative* to specialized monolithic accelerators.

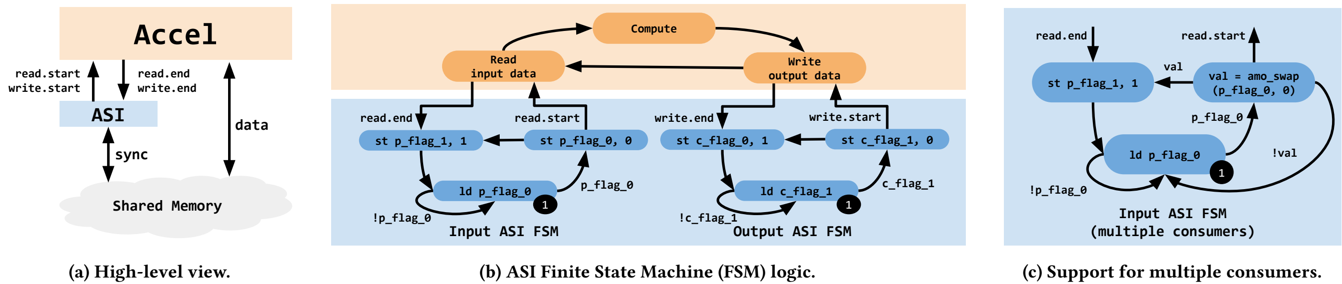
Evaluation. We leverage the ESP heterogeneous SoC platform [55] which provides multiple accelerator designs and RISC-V CPU integration in a modular tiled architecture, with the option to include private (L2) caches within the tile and a last-level cache (LLC) shared by all tiles. ESP also supports MESI-based cache coherence and Coherent DMA – an implementation in ESP optimized for accelerator communication. ESP has been used in prior work for FPGA prototyping [26, 94] and tape-outs [17, 18, 40]. We prototype multiple configurations of Mozart (with a RISC-V core and multiple accelerators integrated over a mesh-based network-on-chip (NoC)) capable of running Linux and evaluated on an FPGA.

We consider over 70 distinct hardware configurations in our work – 9 different accelerator combinations (including 3 fine-grained single accelerator systems, 3 fine-grained multi-accelerator systems, 2 coarse-grained monolithic accelerator systems and a configurable synthetic accelerator system with up to 15 accelerators), 3 accelerator invocation strategies (baremetal, OS, and ASI), and 3 coherence protocols (MESI, Coherent DMA, and Spandex-FCS). For the single-accelerator systems and a synthetic multi-accelerator system, we perform parameter sweeps to explore the design space of workloads at different acceleration granularities. For the multi-accelerator systems, we evaluate accelerator chaining and pipelining where applicable, and explore the design space of varying chain and pipeline lengths (1 to 15) with the synthetic multi-accelerator system. Finally, we study the performance gap between monolithic and disaggregated accelerators with 4 multi-accelerator systems.

The accelerators we study include FFT, FIR, Viterbi, Sort, GeMM, and a synthetic accelerator for design space exploration. The applications we map to the multi-accelerator systems include *audio*, 3D spatial audio decoder [38]; *Mini-ERA*, a workload from the autonomous driving domain [39]; *FCNN*, a fully connected neural network inspired by the architecture of Instant-NGP [58]; and a synthetic workload for further design space exploration. Together, our workloads represent a large diversity, including acceleration tasks ranging from very fine to very coarse granularities.

Contributions. Our major contributions and findings are:

- *Impact of Acceleration Taxes:* We highlight the impact of control and data acceleration taxes on end-to-end performance in shared-memory SoCs. Specifically, we show that these taxes significantly degrade and often eliminate the potential performance benefits from fine-grained acceleration.
- *Taming the Control Tax:* We propose ASI, a lightweight, modular and general interface that replaces expensive CPU-centric accelerator control with efficient shared-memory synchronization between accelerators (and CPUs). This constructs a view of transparent acceleration where compute units are agnostic to the type of entity they synchronize with. ASI supports these capabilities with a low area overhead of just 1%.
- *Taming the Data Tax:* We show that supplementing ASI with Spandex-FCS to reduce the data tax is effective at further improving the performance of fine-grained acceleration. This work is the first RTL-level implementation and evaluation of Spandex-FCS (or Spandex). We evaluate an FPGA prototype with RISC-V CPUs and multiple fixed-function accelerators in an SoC capable of running multi-core Linux OS. Our implementation has a modest area overhead over MESI and confirms the performance benefits of Spandex-FCS in a complex heterogeneous system, previously shown only in simulation for a CPU-GPU system without an OS. Further, qualitatively, we also found that the claimed extensibility of Spandex did indeed allow us to bring up the system in a modular way with incrementally increasing specializations.
- *Mozart, a Novel SoC Architecture:* Combining our acceleration tax mitigation techniques, we propose *Mozart* that enables performance benefits for finer-grained acceleration than previously seen. Mozart provides new opportunities for accelerator-level parallelism by chaining and pipelining fine-grained accelerators through shared-memory, with no additional hardware modifications.
- *Disaggregated, Fine-grained Acceleration:* Mozart, for the first time, demonstrates programmable and transparent composition (through shared-memory) of fine-grained, disaggregated accelerators as a performance-competitive alternative to specialized monolithic accelerators. Furthermore, the capabilities of Mozart are comprehensively evaluated in complex settings: with multiple fixed-function disaggregated accelerators, real-world applications running with an OS, and state-of-the-art coherence protocols, all realized on an FPGA with RTL implementations.
- *Signi cant Results:* Our evaluation provides several surprising new quantitative results. For half of the more than 70 configurations with single and multiple disaggregated accelerators (Figures 4 and 5), the baseline system with OS invocation of accelerators and MESI coherence provides virtually no benefit or a degradation relative to pure software. Mozart enables speedup of 1.8X to 49X in these cases (relative to pure software). For audio, disaggregated accelerators with the baseline system see a 1.8X slowdown relative to a monolithic accelerator; Mozart improves the performance of both and reduces the slowdown to a minor 1.08X (Figure 7). Our scalability study shows Mozart's performance benefits scale well with increasing accelerator-level parallelism. For 15 disaggregated accelerators (the maximum number of coherent accelerators supported by ESP), the slowdown compared to a monolithic accelerator is just 1.1X.



(a) High-level view. (b) ASI Finite State Machine (FSM) logic. (c) Support for multiple consumers.

Figure 1: ASI overview. (a) A high-level view of the interface between the accelerator, ASI, and shared-memory. (b) Input and output ASI FSM logic for a single-producer single-consumer case, with initial state marked ①. Here, p_flag_0/1 are used to synchronize with the (upstream) producer. c_flag_0/1 are used to synchronize with the (downstream) consumer. (c) Modified input ASI FSM to support multiple consumers, where a read-modify-write (amo_swap [65]) is used to update a common flag.

• *Analytical Performance Model:* In addition to a comprehensive evaluation on an FPGA with a variety of accelerators and workloads, we also provide an analytic model for additional insight into our empirical results.

Overall, our work reveals a wide new space of designs for acceleration with little area cost and familiar shared-memory programmability, aligned with the needs of emerging applications with many diverse tasks. Our work has significant implications for heterogeneous SoC design, including implications for the system architecture, design-space exploration tools, schedulers, and compilers.

2 Taming the Control Tax

2.1 Accelerator Synchronization Interface (ASI)

2.1.1 Concept. Traditionally, offloading computation to an accelerator requires the CPU to invoke the accelerator, typically through heavyweight OS calls treating the accelerator as a device. This centralizes and elevates the role of the CPU, treating it effectively as a master controlling the accelerators.

ASI departs from the above attributes in two ways. First, we promote decentralized control, where accelerators are first-class entities empowered to synchronize with and trigger computation directly on other accelerators, without the need for CPU mediation. Second, similar to synchronization among CPU cores, we enable accelerators to synchronize with each other and with the CPU through shared-memory. This provides a lightweight and uniform synchronization mechanism where a synchronizing entity is agnostic to whether the other participating entity is an accelerator or a core.

Figure 1a shows the high-level view of ASI. It is a generalized modular shim interface that can be integrated with an accelerator without modifications to the accelerator design, and can be adapted to the synchronization needs of any application. While programmable coarse-grained accelerators such as GPUs are already becoming increasingly like CPUs, our work targets and elevates finer-grained, fixed-function and limited-programmable accelerators to be first-class compute units. ASI enables these accelerators to circumvent CPU centralization, and independently and efficiently synchronize with other compute units through a common shared-memory abstraction. Finally, performing software-based shared-memory synchronization within a fixed-function accelerator (e.g., as in HSA [28]) would require full instruction pipelines and incur relatively large area and power overheads; therefore, ASI supports these synchronizations directly in hardware with minimal design complexity.

2.1.2 Design. The applications studied in this work mostly exhibit producer–consumer interactions (as is often the case for heterogeneous SoCs). Specifically, these are single producer, single consumer interactions, where an accelerator (repeatedly) consumes (loads) data stored by a prior producer, computes, and then produces (stores) data for a subsequent consumer. We therefore focus our proposed design towards supporting these interactions efficiently.

Figure 1b illustrates our lightweight ASI design. It consists of two finite state machines (FSMs), one for each producer-consumer interaction described above, labeled as *input* and *output* ASI FSMs. Each FSM relies on two shared-memory flags (flag_0/1)[1]. The input ASI FSM of an accelerator uses p_flag_0 to determine whether its producer has new data to send, and p_flag_1 to inform the producer that it is ready to receive new data. Similarly, the output ASI FSM of the accelerator synchronizes with its consumer using c_flag_0 and c_flag_1. We follow the RISC-V weak memory ordering (RVWMO) model [65], and treat the loads and stores of the flags as acquires and releases[2]. To enforce acquire ordering, all subsequent accesses in the accelerator are implicitly stalled by ASI until the required load value 1 for the acquire is returned. To enforce release ordering, we ensure all prior accesses have been performed in shared-memory before the release is attempted.

ASI can be extended to support other scenarios; e.g., multiple producer-consumer. Figure 1c illustrates a modified input ASI FSM to support multiple consumers, where only one consumer at a time consumes an input from the producer. We adapt the two-flag implementation by replacing the store to flag_0 with a read-modify-write to resolve concurrent updates to the same flag from multiple consumers. This is only one possible design. ASI can also be implemented with primitives better suited for other use cases like queues or barriers.

2.1.3 Implementation. We leverage the ESP platform [17, 55] to implement the ASI design for a heterogeneous SoC. (Our design is

[1] While our design uses two flags, alternate designs with a single flag for single producer, single consumer interactions are also possible.
[2] Technically, only the synchronization stores that set the value 1 have release semantics, but we conservatively ignore this distinction.

not limited to the ESP platform.) Accelerators in ESP follow a load-store architecture with three steps: (i) inputs are read from memory into the accelerator's local scratchpad using a DMA interface, (ii) the accelerator computes using the inputs and produces outputs stored into its scratchpad, and (iii) the outputs in the scratchpad are written to memory using the DMA interface. The use of the DMA interface for memory accesses is independent of the cache hierarchy and coherence protocol implemented in the system; e.g., even accelerators with MESI-coherent private L2 caches use the DMA interface for single loads and stores.

To implement ASI in ESP, we similarly use the accelerator DMA interfaces (originally for bulk reads and writes) to implement the fine-grained acquire and release synchronization accesses, avoiding the design complexity of dedicating a specialized load-store unit for the ASI. Thus, the accelerator's load DMA engine reads the appropriate flag in shared-memory, which is tested by the ASI FSM. The FSM proceeds to the next phase if the flag is set to 1; if not set, the ASI continues testing. Similarly, the ASI FSM uses the store DMA engine to set flags in shared-memory.

```
init:
  // mapping flags for accel
  map(accel.p_flag_0 -> flag0_0)
  map(accel.p_flag_1 -> flag0_1)
  map(accel.c_flag_0 -> flag1_0)
  map(accel.c_flag_1 -> flag1_1)
  // mapping flags for CPU
  map(cpu.p_flag_0 -> flag1_0)
  map(cpu.p_flag_1 -> flag1_1)
  map(cpu.c_flag_0 -> flag0_0)
  map(cpu.c_flag_1 -> flag0_1)
```

(a) ASI initialization.

```
wait_for_consumer:
  while (*cpu.c_flag_1 != 1) {;}
  *cpu.c_flag_1 = 0
  // produce data for consumer
  *cpu.c_flag_0 = 1

wait_for_producer:
  while (*cpu.p_flag_0 != 1) {;}
  *cpu.p_flag_0 = 0
  // consume data from producer
  *cpu.p_flag_1 = 1
```

(b) Producer-consumer example.

Figure 2: ASI software interface for a single CPU and single accelerator system. (a) CPU pseudocode for mapping memory addresses of shared-memory flags for the CPU and accelerator (with ASI registers). (b) CPU pseudocode for synchronizing with the accelerator.

2.1.4 Software Interface. We reserve configuration registers in the accelerator to configure the virtual addresses of the shared-memory flags. ESP offers support for address translation with the DMA engine, and we rely on the same support for ASI. Figure 2a illustrates the initialization for a system with a single CPU and accelerator. During application startup, the CPU configures the mapping of shared-memory flags for the CPU and accelerator, where the latter is done by configuring ASI registers with the addresses of the appropriate flags. To create a producer-consumer chain, `p_flag_0/1` for the consumer and `c_flag_0/1` for the producer are mapped to the same shared-memory flag. This is an infrequent configuration required only at application startup or upon reconfiguration of the producer-consumer chain, as opposed to CPU intervention through OS calls for every accelerator invocation. Our implementation leverages the ESP API and device drivers for configuration; however, subsequent CPU-initiated accelerator invocations replace the ESP API with inexpensive shared-memory synchronization. The latter can be placed directly in the user application or underlying scheduler using standard libraries (e.g., atomic_flag in C++, Boost [14, 62]). Figure 2b illustrates the pseudocode to manipulate these shared-memory flags to produce data for the consumer or consume the data from the producer. Although Figure 2b demonstrates a single accelerator system, the software interface is agnostic to whether the synchronizing entity is a CPU, a monolithic accelerator, or a sea of disaggregated accelerators.

2.2 Capabilities

ASI enables chaining and software-level pipelining of disaggregated accelerators through shared-memory, without any CPU or OS intervention. ASI also enables tiling large datasets across chained and pipelined accelerators, as well as supporting streaming use cases.

Accelerator chaining. If the workload exhibits producer-consumer relationships among a sequence of accelerators, ASI enables composing these accelerators into a producer-consumer chain using shared-memory synchronization. No additional hardware or mediation from the CPU or OS is required (except when the CPU itself is a producer or consumer in the chain), reducing the time for accelerator invocation and synchronization while also freeing up the CPU for other tasks. From a program's perspective, the chain of disaggregated accelerators transparently emulates a monolithic accelerator.

Software pipelining across disaggregated accelerators. Chained accelerators can be further optimized by pipelining them entirely in software, requiring no hardware changes in ASI. The first accelerator (or CPU) in the pipeline can produce data for the next one, even while subsequent accelerators in the chain are yet to complete. The shared-memory flags in ASI allow a producer to indicate data being ready and the consumer to indicate readiness to accept data, independent of each other and the rest of the pipeline. As a result, the accelerators in a chain can be pipelined — improving the end-to-end application performance by improving the individual accelerator utilization (and any inefficiencies resulting from pipeline imbalances).

Tiling and streaming. For large datasets, ASI enables producer-consumer pairs to compute on and communicate data in smaller tiles and inexpensively synchronize with each other for every tile. This can improve data locality and avoid performance degradation due to cache thrashing. ASI also supports streaming use cases, where producer-consumer pairs synchronize with each other for every new stream of data.

3 Taming the Data Tax

3.1 Background

Spandex-FCS provides state-of-the-art heterogeneous coherence specialization with demonstrated low design complexity and high performance. The original Spandex coherence interface [4] provides coherence specialization at a device granularity. The follow-on Spandex-FCS (Spandex with fine-grained coherence specialization) [3] extends this flexibility to a per-request (or per-word) granularity. For example, a CPU application may use MESI (write ownership and writer-initiated invalidation) for some accesses to ensure high data locality while using GPU coherence (with write-through and self-invalidation) for other accesses. Such fine-grained specialization can avoid unnecessary data movement on the NoC. The FCS work introduces additional request types to the Spandex protocol, including write-through forwarding (`ReqWTFwd`), which relies on the read-with-ownership (`ReqO+data`) request. The `ReqO+data` request can be used to perform loads while allocating the data in owned state in the private cache of the requestor. The `ReqWTFwd`

request can be used to perform write-through stores. If the data is owned by another private cache upon the ReqWTFwd arriving at the LLC, the update is forwarded to the current owner without changing the coherence state at the LLC or the current owner. If the data is not owned in another private cache, this request is treated like a typical write-through request (ReqWT).

3.2 Producer-consumer Data Forwarding

The producer-consumer forwarding capability in Spandex-FCS lends itself well to our design goal of decentralized data movement to reduce the data tax. A MESI-style protocol is inefficient at supporting producer-consumer data movement as a result of needing to (1) invalidate shared consumer lines during producer stores, and (2) revoke ownership of producer lines during consumer loads, all centralized via the directory. With Spandex-FCS, all loads from consumers use the ReqO+data request to allocate the data in owned state in the consumer's cache, and all store accesses from producers use the ReqWTFwd request to forward the update directly to the owner (here, the consumer) cache. The consumer benefits from ReqO+data because it eliminates read misses which are typically on the critical path. The producer benefits from ReqWTFwd because it avoids the overheads of transient states associated with stores using a MESI-style protocol. Further, the producer is expected to exhibit less reuse of the written data than the consumer; therefore, not allocating the data in the producer's cache avoids the cost of future write-backs. Overall, the producer forwarding data directly into the consumer's cache models a dedicated peer-to-peer data transfer interface in hardware, while providing the programmability of coherent shared-memory.

The above capability of Spandex-FCS also benefits the ASI implementation by enabling forwarded updates for shared-memory synchronization flags as follows. First, each device uses ReqO+data to *own* the two synchronization flags that it tests — for the flags presented in Figure 1b, the device *owns* c_flag_1 and p_flag_0. Second, all synchronization flag updates are forwarded directly to the owner without unnecessary coherence overheads.

3.3 Implementation

Our Spandex-FCS cache implementation [92] with ESP comprises designs for two separate controllers, one for the L2 cache for a compute element (accelerator or CVA6 RISC-V core [27, 95]) and one for the shared LLC. This subsection highlights important aspects of the implementation that are necessary to support features of Spandex-FCS in ESP. While the current Spandex-FCS implementation is for a private cache for an accelerator, we plan to explore Spandex-FCS for an accelerator's private scratchpad as directly addressable yet globally visible coherent memory, as in [45].

Fine-grained Coherence Specialization. One key feature of Spandex-FCS is the ability to specialize coherence at the granularity of individual requests. We augment the CVA6 RISC-V CPU and accelerator DMA to support this feature in ESP. For CVA6, we introduce custom load and store ISA extensions that contain dedicated fields to specify Spandex-FCS request types for loads and stores. These fields are included in the AXI [5] transaction to the L2 cache as part of the *AxUSER* bit field. For accelerators, we add Spandex-FCS options to the DMA control interface driven by software configurable registers in ESP. Request types can be inferred using static analysis during compilation or an offline tracing tool from [3]. In this work, we determine request types manually and focus on the types described in Section 3.1. Fine-grained specialization of request types are for performance and not correctness; unspecified requests will default to MESI. Thus, programs can be incrementally converted for performance and leverage the programmability advantages of shared-memory.

Write-Coalescing Buffer. Spandex-FCS offers support for word-granularity requests. To improve spatial locality in the line (if present) and reduce the number of word-granularity requests sent on the network, we supplement the L2 cache controller with a write-coalescing buffer that coalesces multiple word-granularity writes. The write-buffer entries are evicted in a FIFO fashion, or completely drained at a release synchronization. This enables the Spandex-FCS cache to benefit from the spatial locality in the line, while performing writes and tracking ownership at word granularity. This implementation shares commonalities with several coalescing write buffer proposals such as QuickRelease [34] and VIPS [66].

RISC-V Weak Memory Ordering (RVWMO). Our caches implement the full RVWMO specification [65]. We support atomic memory operations (AMO), load-reserve/store-conditional (LR-SC), and all combinations of RISC-V fences. We enforce ordering using a self-invalidation of valid data on an acquire (aq bit set or read in the successor set of fence) and a write-buffer flush on a release (rl bit set or write in the predecessor set of fence).

Lazy Self-Invalidation. In addition to writer-invalidated reads, Spandex also supports self-invalidated reads by allocating words in valid state using the ReqV request type. The directory does not track which cache has a valid copy of the word since the caching device is responsible for self-invalidating all valid words in its cache.[3] While this reduces the invalidation traffic from the directory, a naive self-invalidation requires checking for cache lines in valid state at every acquire synchronization. To address this, we use a lazy scheme with versioning in the cache state [92].

Spandex directory. We implement the Spandex directory with the LLC for holding the cache line states. The directory is similar to the one for conventional protocols, but it does not need to track which caches have valid state copies since they are self-invalidated. Tracking ownership for a line also requires tracking which word is owned in the line. To mitigate overhead, we reuse the data array of the word to store the owner ID [4, 20], and track the ownership status of each word in a separate set of entries in the directory. These tradeoffs are analyzed in [20]. It is possible to support an LLC non-inclusive of owned data by implementing a state-only LLC using mechanisms similar to those for a conventional MESI protocol.

4 Methodology

4.1 Evaluation Hardware

4.1.1 Hardware setup and flow. We implement Spandex-FCS caches in SystemVerilog RTL. All evaluated accelerators are implemented

[3] DeNovo[20] uses regions to invalidate only the potentially stale data based on software information, but we do not implement region optimizations here.

in SystemC, and ASI is implemented as a modular shim to be easily integrated with the accelerators. We generate synthesizable RTL for the accelerators using Cadence Stratus HLS [15]. We use ESP's SoC generator templates [55] to create our SoC design, synthesize using Xilinx Vivado [87] and evaluate on a VCU118 FPGA board [88].

4.1.2 Coherence Protocols. We evaluate three protocols: MESI coherence, Coherent DMA and Spandex-FCS. Coherent DMA is an optimized implementation of MESI supported by ESP that bypasses the L2 cache of the accelerators, and provides best performance for finer-grained data sizes in ESP [94]. It leverages a dedicated DMA interface in the LLC controller that can service bulk transfers without needing to be broken into consecutive line-granularity requests. In contrast, MESI and Spandex-FCS do not bypass the accelerator L2 cache. The L2 cache does not service bulk transfers, and any bulk request from the accelerator is broken down as consecutive line-granularity requests to the L2 cache. As a result, Coherent DMA provides a higher throughput DMA interface that can effectively pipeline reads and writes from an accelerator, especially in cases where they hit in the LLC (i.e., no L2 cache in the system has a local copy). The increased throughput is further amplified by the unusual fact that the LLC clock frequency in ESP is twice of the L2, due to other implementation reasons. On the other hand, the DMA interface does not service multiple bulk transfers concurrently. This behaviour of Coherent DMA is purely a limitation of the ESP system and other implementations that service bulk transfers are possible. A similar DMA interface can also potentially be integrated with Spandex-FCS.

4.1.3 Evaluated Designs. All our designs are modular SoCs with a mesh-based NoC connecting multiple tiles, including CPU, multiple accelerators, LLC and IO. We use a 64-bit in-order RISC-V CVA6[27] CPU, 32KiB 512-set 4-way L2 caches, and a 128KiB 1024-set 8-way LLC. The CPU and L2 caches are clocked at 78MHz, while the LLC is clocked at 156MHz (as required by ESP). We use the following system configurations for our evaluations: **(1) Baseline system** uses OS-based invocation for accelerators and MESI for cache coherence. **(2) M+** uses ASI for accelerator invocation and MESI for cache coherence. **(3) D+** uses ASI for accelerator invocation and Coherent DMA for cache coherence. **(4) Mozart** uses ASI for accelerator invocation and Spandex-FCS for cache coherence.

4.2 Benchmarks

We use multiple benchmark categories to cover a large design space of heterogeneous workloads. We start with benchmarks that have one dominant kernel offloaded to a single accelerator. We then consider multi-accelerator workloads and evaluate complex real-world multi-kernel applications that invoke multiple accelerators.[4] Finally, for a broad design space exploration sweep for added insight, we study a multi-accelerator workload using synthetic accelerators.

4.2.1 Single Accelerator Benchmarks. We use three single accelerators for standard kernels, FFT, Sort, and GeMM. We use (open source) RTL from ESP for the accelerators, and augment them with ASI, illustrated in Figure 1b, to enable shared-memory synchronizations. We run these kernels with different input sizes (128 to 32K elements for FFT, 32 to 1K elements for Sort, matrices of size 8x8 to 64x64 for GeMM) to study impact of control and data taxes. For simplicity, we use the term GeMM to refer to both GeMV and GeMM in this work, and report results for GeMV over 1xM vector and M*N matrix as 1*M*N, and GeMM over MxN and NxM matrices as M*N.

4.2.2 3D Spatial Audio Decoder. The 3D spatial audio decoder (audio) is based on libspatialaudio [47] and a component of the ILLIXR system [38] that plays back the encoded audio sources based on the pose of the user in an XR system. Audio is a complex real-world application that runs end-to-end for playable audio output, features a rich diversity of tasks, and is implemented with 6.5K lines of C++ source code. A detailed data flow graph of the computation required for a single audio block is shown in Figure 3, along with the size of data transferred along each edge. We identify the four dominant tasks in the application: psycho-acoustic filter, rotate order, zoomer, and the binauralizer filter. The psycho-acoustic and binauralizer filters contain 16 and 32 independent instances of a chain of fine-grained tasks, respectively. The chain consists of an FFT, FIR and IFFT, which we refer to as the FFT-FIR-IFFT chain. We accelerate the FFT-FIR-IFFT chain in hardware and execute the overlap task on the CPU (we do not design an accelerator for overlap as it not a generic kernel like FFT or FIR, with differences even between the two instances in the audio application). As the FFT accelerators operate on fixed-point data, the CPU is responsible for converting input data for the FFT-FIR-IFFT chain from floating-point to fixed-point and vice-versa.

To evaluate audio in hardware, we design three configurations of a heterogeneous SoC to accelerate the FFT-FIR-IFFT chain. First, we design a monolithic accelerator that accelerates the FFT-FIR-IFFT tasks as sequential stages within this single monolithic accelerator. Next, we design a composable disaggregated system by reusing two FFT accelerators from the single accelerator benchmark to accelerate FFT and IFFT, and develop a custom accelerator for FIR. Using these fine-grained accelerators, we offload the entire FFT-FIR-IFFT chain as a disaggregated producer-consumer chain. Finally, for an in-depth analysis of all scenarios, we design a second monolithic accelerator that internally pipelines FFT-FIR-IFFT computations across each other, at a hardware level. To achieve this aggressive hardware-level pipelining, we include additional private scratchpads within the monolithic accelerator to allow double-buffering, which in turn enables overlapping data movement and compute. However, other simpler but potentially less performant strategies are also possible.

4.2.3 Mini-ERA. Mini-ERA [39] is a vehicular swarm perception workload that consists of multiple tasks with 5K lines of code, including a radar component to find the nearest obstacle in the vehicle's lane and Viterbi decoder to decode messages [40]. We leverage the Viterbi accelerator in ESP's accelerator suite, augment it with ASI, and reuse the FFT accelerator from the audio benchmark. We note that accelerator disaggregation enables accelerators (here, FFT) to be directly reused across workloads. In the radar component, the radar data is used to perform an FFT on the FFT accelerator, which the CPU uses to calculate distance. In Viterbi,

[4]We considered the Yin-Yang benchmarks [43] and the cross-domain benchmarks by Wang et al. [83] for our evaluation, but not all accelerators and key software functions were open-sourced, making it hard to test our ideas.

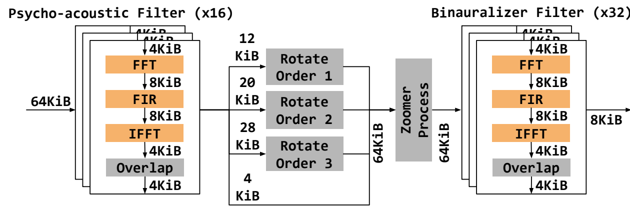


Figure 3: Block diagram of audio from ILLIXR [38]. The blocks in orange are accelerated in hardware.

encoded messages are first decoded by the Viterbi accelerator before a de-scramble is performed by the CPU. To mimic the actual hardware on an autonomous vehicle, we create synthetic sensor accelerators for both components that provide the input (pre-loaded from traces) at runtime. Using ASI, we can chain the radar sensor, CPU, and the FFT accelerator to form a three-device chain in the radar component; and the Viterbi sensor, Viterbi accelerator, and CPU to form another chain in the Viterbi component. Mini-ERA has limited opportunities for accelerator pipelining (only one inter-accelerator producer-consumer relationship), so we do not evaluate its pipelining.

4.2.4 Fully Connected Neural Network (FCNN)

Fully Connected Neural Networks are widely used in applications like image and video processing [80], recommender systems [89], and recent neural graphics [50, 57, 58, 84]. Instant-NGP [58], for instance, proposes a model for neural radiance fields (NeRF) consisting of a 1-layer multi-layer perceptron (MLP) for density and a 2-layer MLP for color, where each layer consists of 64 neurons. We design a 3-layer FCNN benchmark inspired by the Instant-NGP model, where each FCNN layer consists of 64 neurons. We instantiate three GeMM accelerators, capable of computing ReLU, and map individual layers of the FCNN to each accelerator in a producer-consumer chain. During initialization, the CPU writes the read-only weights for the layers once. During the benchmark execution, the CPU writes a 1x64 vector input for the disaggregated FCNN accelerator system, and reads back 1x64 vector output.

4.2.5 Synthetic Benchmark

The synthetic benchmark enables wide design space exploration. Based on the micro-benchmark described for Gables [36, 52], the benchmark loads a word from an input array, performs several multiply-and-accumulate (MAC) operations, and stores the word to an output array. As in the original micro-benchmark, the synthetic benchmark provides tuning knobs for the input/output data size as well as the operational intensity (Op Int, number of operations performed per byte loaded from shared-memory). These knobs are used to configure the work done (number of operations performed). We evaluate the synthetic benchmark with two cases of total work – Synth-large with work of 1500 and Synth-small with work of 150 – over stream data of size 4KiB. To evaluate the benchmark in hardware, we design a synthetic accelerator that includes a load kernel (to read from the input array), a store kernel (to write to an output array), and one or more compute kernels. We use the synthetic benchmark to evaluate strong scaling of up to 15 disaggregated synthetic accelerators in chained and pipelined configurations, and use the Op Int knob to partition the work evenly among the accelerators.

4.3 Power Consumption

We qualitatively discuss Mozart's power consumption here. Our optimizations primarily improve execution time by reducing the work done and/or network traffic; however, disaggregated systems introduce additional network traffic. We do not quantitatively evaluate power consumption because the measured power from an FPGA is not representative of an ASIC technology, and power-aware RTL simulations are infeasible for our benchmarks with an OS.

ASI: ASI eliminates the need for CPU orchestration of accelerators and thus, reduces the work done by the CPU for OS invocation. On the other hand, it expends additional power to spin on flags in shared-memory. However, spinning largely happens in the private cache and our implementation is amenable to traditional back-off solutions to reduce the impact.

Spandex-FCS: By enabling read hits and reducing coherence traffic, Spandex-FCS reduces traffic in system compared to MESI and thus lowers dynamic power consumption from the NoC and caches.

Disaggregated Accelerators: Relative to a monolithic accelerator, a disaggregated system incurs additional network traffic to communicate data between accelerators, potentially increasing power consumption. Spandex-FCS can reduce this compared to MESI with producer-consumer forwarding.

Static Power: Our optimizations incur small area overheads and thus do not expect a significant increase in static power. Instead, disaggregated accelerators may potentially incur lower static power than monolithic if managed appropriately [23].

5 Results

This section presents the quantitative results from our comprehensive evaluation. Appendix A provides a analytical performance model for additional insights to accompany our empirical results.

5.1 ASIC Area Analysis

Table 2 shows the area overhead of our ASI and Spandex implementations. We obtain ASIC area results using Genus [16] and use the 12nm process technology. For ASI, we compare the area of the audio accelerators with and without ASI. For Spandex-FCS, we compare its area against the MESI implementation in ESP for the L2 and LLC for sizes in Section 4.1.3. We also show the area overhead from adding a write-buffer (WB) in the Spandex-FCS L2.

Low area overhead for ASI and Spandex-FCS: ASI only incurs a 1% area overhead in the FFT and the FIR accelerators. For Spandex-FCS L2, compared to the baseline L2 in ESP, the area overheads are 7% and 5% with and without a WB. Spandex-FCS LLC is 1% smaller than the baseline LLC in ESP since we do not implement the coherent DMA handler for Spandex-FCS[5].

5.2 Taming Control Tax for a Single Accelerator

Figure 4a shows the benefit of ASI in taming the control tax for single-accelerator benchmarks, with execution time normalized to the software-only case. We evaluate the single-accelerator benchmarks in 3 systems: (1) baseline system, (2) baseline system with no OS (i.e., benchmark is run bare metal), (3) M+.

[5]Only results for D+ rely on the DMA handler, while M+ does not. The design of the handler is deeply integrated with the LLC in ESP, therefore we were unable to measure the area without it.

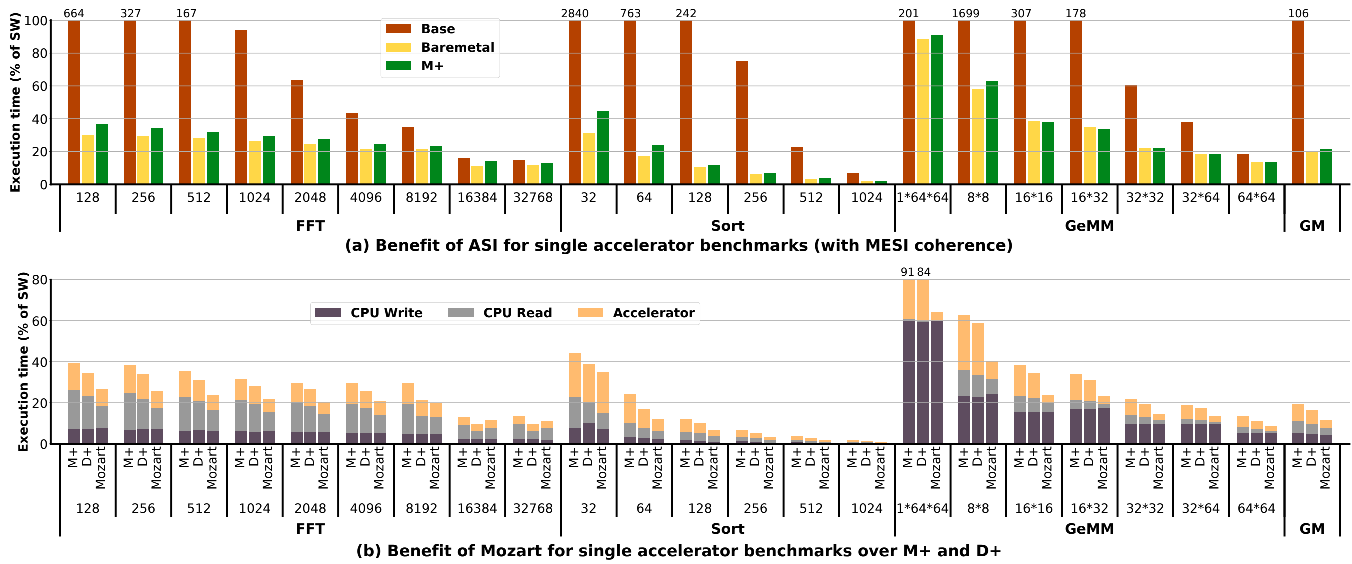


Figure 4: Benefit of ASI and Spandex-FCS for single accelerator benchmarks with different input sizes and geometric mean (GM). The bars show execution time normalized to pure software.

Table 2: Area of the baseline designs (accelerators and caches in ESP) and overheads of ASI and Spandex-FCS, respectively.

Accelerator	Area (` < 2)	ASI Overhead
FFT	30881	1%
FIR	80081	1%

Cache	Area (` < 2)	Spandex-FCS Overhead
L2	61082	5% (7% w/ write buffer)
LLC	260259	-1%

Baseline System can lead to slowdown, while M+ significantly reduces the control tax, with comparable performance to bare metal: Across all standalone accelerator benchmarks, the baseline performs worst, and in several fine-grained cases, it results in a slowdown over software (e.g., 28.4X slowdown for Sort 32). As the input sizes and the amount of work done in the accelerator increase, the impact of the OS-invocation, which is a constant overhead independent of the input size, is less severe and we see increasing speedups over software. ASI, by bypassing the bulky OS invocation, brings down this large constant overhead to just the time taken to perform a shared-memory synchronization. As a result, M+ offers an average speedup of 5.31X over baseline, and comparable performance to the bare metal configuration (which has the lowest control tax).

5.3 Taming Data Tax for a Single Accelerator

Figure 4b shows the benefit of Spandex-FCS in taming the data tax for single accelerator benchmarks, with execution time presented as a percentage of the software-only case. We evaluate the benchmarks in 3 systems: (1) M+, (2) D+, (3) Mozart.

Mozart provides the lowest data tax for virtually all configurations: Across virtually all single-accelerator benchmarks and input sizes, Mozart provides the best performance, with average speedups of 1.7X and 1.4X over M+ and D+, respectively. The improvements come from the direct producer-consumer data forwarding enabled by Spandex-FCS. The large performance improvements (of up to 2.2X relative to M+) are seen when the data sizes are smaller than the L2 cache size of the accelerator. With increasing input size, Mozart continues to outperform M+ with a similar speedup. However, we observe an inflection point in the larger FFT sizes when the input exceeds the L2 cache size; e.g., speedup over M+ drops from 1.5X to 1.1X when changing input size from 8192 to 16384. At this point, the consumer no longer hits in the L2 cache for all read data, resulting in cache misses to the LLC. However, Mozart still outperforms M+ because the consumer benefits from read hits at the LLC, whereas modified data needs to be revoked from the producer's cache for M+.

Mozart outperforms D+ for most cases despite the implementation difference, especially for finer-grained input sizes that fit within the L2 cache. At coarser-grained sizes, D+ performs better, providing a speedup of 1.2X over Mozart for FFT 32768. Here, the data from the CPU's L2 cache spills over to the LLC. The accelerator, having bypassed its own L2 cache, benefits from data locality in the LLC for high-throughput DMA reads and writes. While this work aims to primarily enable finer-grained acceleration, we note that Spandex-FCS can be extended to support coarser-granularity transfers like D+ for similar benefits. We plan to explore this direction in future work.

5.4 Accelerator to Accelerator Chaining

Figures 5 and 6 show accelerator chaining benefits for four benchmarks: three real-world use cases – audio, Mini-ERA and FCNN – and a synthetic benchmark for a broad sweep of chain sizes and two cases of total work (as described in Section 4.2.5). We evaluate these benchmarks in 4 systems: (1) baseline system, (2) M+, (3) D+, (4) Mozart. The accelerator "chain" for a baseline system is orchestrated entirely by the CPU, where the CPU invokes each accelerator with OS-based invocations sequentially, after the previous accelerator in the computation chain finishes its computation.

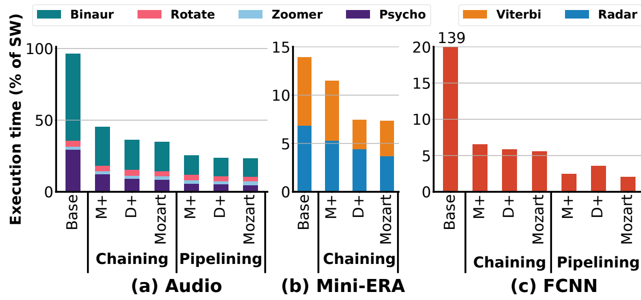


Figure 5: Execution time for three real-world applications with accelerator chaining and pipelining. Execution time is normalized to the software-only version.

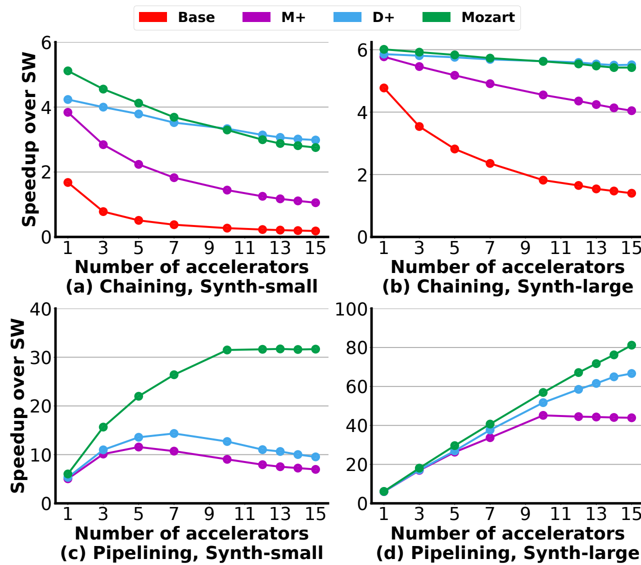


Figure 6: Speedup over software for the synthetic benchmark for accelerator chaining and pipelining. Two benchmarks – Synth-large and Synth-small – are evaluated.

Baseline system can lead to slowdown over software and Mozart enables speedup: We observe that chaining accelerators with baseline does not always provide benefits over software, and can even result in a slowdown. Specifically, the baseline provides a speedup of 7.2X over software for Mini-ERA, but provides virtually no speedup for audio, and results in a slowdown of 1.4X over software for FCNN. M+ improves upon baseline and enables speedups over software for the three cases by 1.2X to 21.3X. Mozart further improves upon M+ by 1.4X on average, and provides better or comparable performance to D+ for all cases.

Mozart broadens the design space that can benefit from accelerator chaining: Figures 6a and 6b show that with the baseline system, accelerator chains can provide speedups over software until length 15 with Synth-large, or cause a slowdown at length 3 with Synth-small. The speedup over software monotonically decreases with increase in chain length due to the additional control and data taxes incurred by each accelerator. We observe that Mozart significantly improves upon the baseline and enables all accelerator chain lengths studied to provide speedups over software. The speedup curve for accelerator chains with Mozart is much higher and has a lower slope than both baseline and M+. Mozart is comparable with D+ for accelerator chaining, with the best configuration depending on the accelerator chain length.

5.5 Accelerator to Accelerator Pipelining

Figures 5 and 6 show the benefits of software pipelining of disaggregated accelerators, enabled by ASI (as described in Section 2.2), for the same benchmarks as in Section 5.4. We evaluate these benchmarks for 3 systems: (1) M+, (2) D+, and (3) Mozart. Accelerator pipelining improves performance over chaining across all configurations, and Mozart results in the best performance for all cases.

Pipelining with ASI significantly improves performance for all benchmarks: As discussed in Section 2.2, the accelerator pipeline is orchestrated entirely in software, with no hardware changes to the ASI necessary. Further, it improves accelerator utilization, and any inefficiencies are a result of imbalances in the pipeline. For the two real-world applications (that were amenable to pipelining), pipelining provides an average speedup of 1.8X over chaining.

Mozart gives the best performance across both real-world benchmarks: On average, Mozart achieves speedups of 1.2X and 1.3X over M+ and D+ with pipelining. The GeMMs performed in the FCNN are particularly fine-grained and data load time dominates the accelerator execution time. As a result, for FCNN, Mozart provides a speedup of 1.2X over M+ and 1.75X over D+ with pipelining. D+ performs the worst for FCNN with pipelining due to the bottlenecked DMA interface, discussed in Section 4.1.2. Here, the read-only weights (requiring a majority of the read accesses) are concurrently read from the LLC by all three accelerators in D+, while M+ and Mozart benefit from read hits in the private caches.

Performance of Mozart scales best with increasing accelerator-level parallelism: Figures 6c and 6d show that Mozart provides the highest speedup for all pipeline lengths (i.e., number of accelerators) from 1 to 15. For Synth-small, Mozart shows increasing speedups until pipeline length of 10, after which performance saturates. This is because the CPU read and write time (CPU time) increasingly becomes the dominant stage, and eventually limits the maximum speedup achievable from adding more accelerators to the system. In contrast, M+ and D+ reach their maximum speedups much earlier at pipeline lengths of 5 and 7 respectively, after which their speedups degrade. This is because of the longer CPU (read and write) time in the case of M+ and D+, thus becoming the dominant stage at smaller pipeline lengths. Furthermore, the LLC is a point of centralization for all reads and writes for M+ and D+. As the pipeline length increases, the LLC is unable to efficiently service multiple concurrent requests from all the accelerators, leading to degrading speedups for both systems. In case of Synth-large, Mozart is again the best performing system, achieving linear speedup over software until pipeline length 15. M+, on the other hand, reaches its maximum speedup at pipeline length 10. In summary, Mozart gives the best scalability and performance for both synthetic workloads across the pipeline lengths studied.

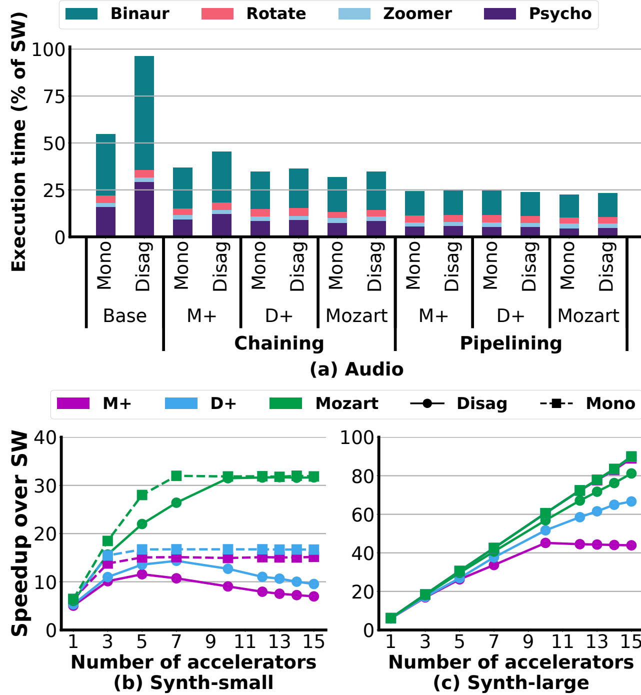


Figure 7: Execution time for (a) audio and for speedup over software for (b,c) synthetic benchmark for monolithic (Mono) and disaggregated (Disag) accelerators. Results are presented for the chained and pipelined audio benchmark, and the pipelined synthetic benchmark. Execution time of audio is normalized to the software-only version. Two synthetic benchmarks – Synth-large and Synth-small – are evaluated.

5.6 Monolithic vs. Disaggregated Performance

Figure 7 shows the performance gap between monolithic (dashed lines) and disaggregated (solid lines) acceleration for audio (Figure 7a) and synthetic benchmarks (Figures 7b and 7c) for different systems.

Baseline shows a large gap between monolithic and disaggregated, Mozart techniques significantly improve upon the baseline for both systems. In audio(Figure 7a), we observe a performance gap of 1.8X between the monolithic accelerator and the chained disaggregated accelerators with the baseline system (between Mono and Disag in Base). This performance gap exists because the monolithic system incurs control and data taxes only once in a chain, as opposed to multiple times for the disaggregated accelerators. Mozart improves the performance for both and reduces the performance gap to 1.1X (between Mono and Disag in Mozart Chaining).

Pipelining narrows performance gap and Mozart gives the best benefits. In audio, applying pipelining improves both monolithic and disaggregated systems, with Mozart providing the best performance. Further, in audio, pipelining virtually closes the gap for all coherence protocols. This is because the CPU time (including writing data for FFT, reading output from IFFT and overlap) is the dominant pipeline stage for audio and determines the maximum speedup possible from pipelining. Thus, by reducing CPU read time, Mozart shows a speedup of 1.1X over M+. Figures 7b and 7c show that the pipelined monolithic and disaggregated accelerators provide increasing speedup over software up to a certain number of pipeline stages until the performance is bound by CPU time. For Synth-small, monolithic and disaggregated accelerators for all three protocols become bound by CPU time at different number of pipeline stages. Mozart provides the best performance for both the monolithic and disaggregated systems at all pipeline lengths. On the other hand, M+ and D+ observe an increase in performance gap with increase in pipeline length beyond 7 for Synth-small.

Disaggregated accelerators are a performant alternative, with no additional design cost for accelerator pipelining: As discussed before in Sections 2.2 and 4.2.2, introducing pipelining with disaggregated accelerators requires no hardware change and is orchestrated entirely in software, whereas there is an added design complexity cost to designing monolithic accelerators. For audio, the monolithic accelerator is designed with aggressive internal pipelining that is specialized for the workload (here, the FFT-FIR-IFFT chain), and requires additional scratchpads to orchestrate the pipeline efficiently. Whether to use a monolithic accelerator or to reuse existing IPs for disaggregated accelerators of various constituent tasks is an important design decision when building an SoC. Mozart opens up the design space of possible acceleration candidates by promoting disaggregated systems that would have otherwise sacrificed performance for lower design complexity.

6 Related Work

There is decades of work on coherence, synchronization, and heterogeneous SoCs. We aim to move beyond one-off ideas to holistically realize and evaluate a full heterogeneous SoC (1) with multiple fixed-function accelerators capable of synchronizing with each other over shared-memory, (2) running real-world applications with an OS and demonstrating software programmable chains and pipelines of accelerators, (3) with a state-of-the-art heterogeneous coherence protocol that enables per-word coherence specialization, and (4) demonstrating programmable, transparent composition of disaggregated accelerators as a performant alternative to monolithic acceleration.

6.1 Accelerator Disaggregation and Composability

Cong et al. [23] propose using accelerator building blocks (ABBs) to emulate a coarse-grained loosely-coupled accelerator. Stitch [79] proposes composing tightly coupled accelerators and proposes network and compiler support to achieve this. Baskaran et al. [11, 12] explore similar motivations for decentralizing control and data movement. The Accelerator Store [54] has a shared SRAM for accelerator data communication. Yin-Yang [43] also advocates the use of disaggregated accelerators from different domains. However, all of the above work rely on architecture simulations, do not consider bulky control taxes, do not consider overheads of coherent shared memory or propose invasive changes to the CPU pipeline. VIP [59] proposes IP chains and frame bursts to reduce overheads due to CPU orchestration or memory stalls. Mozart is complementary to the software stack of VIP.

Cohort [85] studies transparent acceleration using shared-memory synchronizations with an FPGA evaluation. However, the work does

not evaluate accelerator chaining or pipelining, and does not consider how to bridge the performance gap between disaggregated and monolithic acceleration. ASI is lightweight with minimal area overhead, while Cohort adds significant area to enable shared-memory synchronization. Gonzalez et al. [30] present an analytical model to study the benefits of a sea of glue accelerators in hyperscale workloads and also advocate chained fine-grained accelerators. However, they do not provide an actual performant design. AuRORA [44] proposes an interface for integrating virtualized accelerators for multi-tenant execution; however, their control interface for accelerators does not leverage shared-memory and is CPU-centralized. Wang et al. [83] propose an accelerator for data restructuring (DRX) to enable accelerator-accelerator chaining, without indirection via the host CPU. The idea of DRX is complementary to our work; e.g., ASI could potentially reduce the control tax for invoking the DRX accelerator. While their system focuses solely on optimizing inter-chip PCIe traffic, Mozart leverages Spandex-FCS to optimize on-chip coherence traffic. Yuan et al. [91] present a holistic approach to accelerator integration and motivate the need to tame datacenter taxes. However, their approach for accelerator interfacing through x86-specific instructions restricts its general applicability and requires invasive changes to the CPU, whereas ASI relies on a more general and ubiquitous shared-memory abstraction. While data movement in their work is through shared virtual memory, they rely on a non-coherent PCIe interconnect, whereas our work leverages the benefit of a coherent interface with Spandex-FCS.

6.2 Kernel Bypass and Control Tax

Lustig et al. [53] and Blockmaestro [1] identify both the overheads of invocation in the context of GPUs and propose schemes that reduce the control tax by invoking GPUs ahead of time. NUCD [9] proposes an ISA extension for kernel-bypass, but requires CPU modifications and does not support communication between accelerators. The M^3 series of works [6–8] promote accelerators as first-class citizens at the OS level with support for virtualization. Several other studies [35, 48, 85, 90] also explore kernel bypass through shared-memory to eliminate control taxes. However, these works primarily target programmable and coarse-grained accelerators, do not consider a cache coherent heterogeneous SoC, and do not propose solutions to optimize data movement in such SoCs. Commercial products typically adopt various approaches for invocation that (1) are not uniform or interoperable, (2) are designed with programmable accelerators in mind, and/or (3) rely on underlying mechanisms that are typically wrappers of OS calls. To remedy this, HSA [28] was proposed as a standard interface to enable heterogeneous devices to communicate over shared-memory with task descriptors enqueued in a circular buffer. However, ASI (1) is generalized to support arbitrary synchronization primitives, (2) is lightweight in implementation with small area overhead, and (3) supports accelerators with limited programmability, whereas HSA is mostly implemented with programmable accelerators [81].

6.3 Heterogeneous Coherence and Data Tax

There is a rich literature on coherence for heterogeneous systems [5, 19, 20, 22, 46, 60, 63, 66, 67, 69, 70, 75–78, 82, 94]. Spandex was proposed as a flexible and extensible coherence interface that supports a variety of access patterns and coherence requirements at a per-device granularity [4]. Spandex-FCS extended Spandex to provide coherence specialization at a per-access granularity, also demonstrating extensibility [3]. We are not aware of any other heterogeneous coherence protocol with such properties, and therefore chose Spandex-FCS as state-of-the-art. The Spandex papers provide a detailed comparison with other protocols. For example, Fusion [46] is a hierarchical protocol that supports forwarding within an accelerator tile. HCC enables efficient work stealing [82] (its primitive operations are supported by Spandex and the specific protocol is complementary to our work). However, most such work is evaluated in simulation with small benchmarks and no OS. Cohmeleon [94] develops an algorithm to dynamically choose a coherence protocol at runtime for each accelerator and is complementary to our work. For instance, the algorithm can be used to choose the Spandex-FCS request type at runtime. Further, we evaluate against MESI and coherent DMA, which perform best for small workload sizes in their work. gem5-Aladdin [69] studies trade-offs between a DMA and cache-based approach in simulation, again complementing our work. None of the above works consider the control tax, accelerator chaining or pipelining, or disaggregated vs. monolithic accelerators. RELIEF [32] proposes a scheduler for leveraging data forwarding techniques proposed in prior work [46, 59] and colocation of producer-consumer tasks to reduce data tax. This is complementary to our work as Spandex also offers similar forwarding primitives. Rui et al. [37] optimize data streaming for on-chip accelerators in a system with a snooping protocol. Our work does not require a snooping protocol, and supports streaming accesses efficiently thanks to specialized request types in Spandex-FCS. Streaming accelerators [10, 21, 25, 51, 61] reduce the data tax with custom data paths and hardwired interfaces that restrict accelerator reuse and have limited programmability. Our work offers shared-memory programmability features, flexibility to specialize coherence for every request, and enables disaggregated accelerator sharing.

7 Conclusions and Future Work

Mozart is a novel architecture that enables finer-grained acceleration than previously possible; programmable, transparent composition of fine-grained disaggregated accelerators; efficient accelerator pipelining via shared-memory and decentralization; and fine-grained, disaggregated accelerators as a performance-competitive alternative to specialized monolithic accelerators. Mozart leverages a general, lightweight, and modular ASI to enable uniform shared-memory synchronization across all compute units, supported by a state-of-the-art heterogeneous coherence protocol, Spandex-FCS, that enables decentralized data movement and per-word coherence specialization. ASI is open source and available as a part of ESP, and can be easily integrated with any of the accelerators at less than 1% area overhead. Equally significant, this is the first open sourced RTL implementation of Spandex-FCS that offers per-word heterogeneous coherence specialization, and with only a modest cache area increase. Our results show that Mozart can tame control and data acceleration taxes for fine-grained accelerators and accelerate real-world applications with software programmable chains and

pipelines of disaggregated accelerators, and that these performance benefits scale to complex SoCs of up to 15 accelerators.

Overall, Mozart reveals a new space of designs for acceleration with little area cost and familiar shared-memory programmability, aligned with the needs of emerging applications with large task diversity. Our future work includes designing SoCs using these ideas for full-scale workloads such as extended reality and data centers, designing schedulers and resource management mechanisms, building automated design space exploration tools, extending these ideas to multi-SoC systems, and incorporating complementary techniques such as near-data acceleration for reduced off-chip data movement.

Acknowledgments

This work was supported in part by the DARPA DSSOC program, the Applications Driving Architectures (ADA) Center, a JUMP Center co-sponsored by SRC and DARPA, the IBM-Illinois Discovery Accelerator Institute (IIDAI), the National Science Foundation under grants 2120464 and 2217144 and the Graduate Research Fellowship Program, and a generous donation from the AMD Xilinx University Program.

A Analytical Performance Model

We develop an analytical model for the various configurations evaluated in this work to further understand the quantitative results.

A.1 Model for Disaggregated Accelerators

We consider a system with one CPU and N disaggregated accelerators that can be chained and pipelined. The system runs in a loop, with the computational chain starting and ending with the CPU.

We define C_i, I_i and D_i as the compute time, the control taxes, and the data taxes incurred by the i^{t} accelerator running in a steady state, where C_0, I_0, and D_0 are incurred by the CPU. The total execution time with accelerator chaining is defined in Equation 1.

$$Chain_{dis} = \sum_{i=0}^{N} (C_i + I_i + D_i) \quad (1)$$

When the CPU and accelerators are pipelined, the system performance is dominated by the CPU or accelerator with the longest execution time. Equation 2 calculates this dominant time which defines the inverse of the pipeline throughput.

$$Pipe_{dis} = \max_{0 \le i \le N} (C_i + I_i + D_i) \quad (2)$$

ASI reduces I_i and Spandex-FCS reduces D_i resulting in more efficient chaining and pipelining. The quantitative results are presented in Sections 5.2 through 5.5.

A.2 Monolithic vs. Disaggregated Accelerators

We compare the performance of monolithic and disaggregated accelerators with both chaining and pipelining. The monolithic accelerator has internal compute stages, where each internal stage is identical to the compute of the corresponding disaggregated accelerator. We define the compute time of each internal stage as C_{M_i}, the total compute time as $C_M = \sum_{i=1}^{N} C_{M_i}$, and as stated above, we assume $C_{M_i} = C_i$. Finally, we define I_M and D_M as control and data taxes paid by the monolithic accelerator.

A.2.1 Chaining.

Equation 3 defines the ratio of execution time of a system with a CPU and (1) a monolithic accelerator over (2) N disaggregated accelerators running in a chain (without pipelining).

$$Gap_{chain} = \frac{Chain_{dis}}{Chain_{mono}} = \frac{\sum_{i=0}^{N} (C_i + I_i + D_i)}{(C_0 + I_0 + D_0) + (C_M + D_M + I_M)} = \frac{C_0 + C_M + (\sum_{i=0}^{N} (I_i + D_i))}{C_0 + C_M + (I_0 + D_0 + I_M + D_M)} \quad (3)$$

Gap_{chain} increases monotonically with the number of accelerators due to additional acceleration taxes. This gap also depends on whether the total compute time ($C_0 + C_M$) is significantly larger than the taxes. Figures 6a and 6b for Synth-small and Synth-large illustrate this well (the x-axis value of 1 accelerator is the same as monolithic). Gap_{chain} increases for both cases with increasing chain lengths; but with significantly larger compute time, the gap is much lower for Synth-large than Synth-small. However, when the total compute time does not dominate the taxes, then the gap between the two cases depends on the cumulative acceleration taxes. We observe that this gap with Mozart is much lower than for other coherence protocols in audio (Figure 7a) and for most chain lengths for Synth-small and Synth-large (Figures 6a and b). Pipelining can narrow this gap much further as discussed next.

A.2.2 Pipelining.

For a monolithic accelerator that can be internally pipelined (as discussed in Section 5.6), Equation 4 calculates the dominant time that defines the inverse of the monolithic pipeline throughput. Here, each compute and data movement stage of the monolithic accelerator is pipelined across each other. Equation 5 defines the ratio of execution time of (1) an internally pipelined monolithic accelerator over (2) N pipelined disaggregated accelerators.

$$Pipe_{mono} = \max_{i}((C_0 + I_0 + D_0), C_{M_i}, (I_M + D_M)) \quad (4)$$

$$Gap_{pipe} = \frac{Pipe_{dis}}{Pipe_{mono}} = \frac{\max_i((C_0 + I_0 + D_0), (C_i + I_i + D_i))}{\max_i((C_0 + I_0 + D_0), C_{M_i}, (I_M + D_M))} \quad (5)$$

We expect Mozart to provide the best performance due to its lower taxes for both monolithic and disaggregated cases. When the CPU time ($C_0 + I_0 + D_0$) dominates, monolithic and disaggregated accelerators offer similar performance (Gap_{pipe} is nearly 1). When the compute time of an accelerated stage dominates, the gap between monolithic and disaggregated is far lower for Mozart (and often closes) with pipelining than with just chaining since the disaggregated taxes are no longer additive.

The presented analytical model can explore tradeoffs for different design decisions. For example, a powerful CPU can potentially reduce OS-based CPU control taxes. However, control taxes due to OS-based invocations would still be greater than ASI's near-baremetal overhead. Further, a powerful CPU reduces C_0 and potentially increases its sensitivity to relatively longer latency accesses in the cache hierarchy, making Spandex-FCS and Mozart more effective.

B Artifact Appendix

B.1 Abstract

This appendix describes how to reproduce the results from Section 5 of the paper. We implement our design on the ESP platform for prototyping heterogeneous SoCs [55]. We integrate the source code for the Accelerator Synchronization Interface (ASI), Spandex-FCS, and all the accelerators that are part of our comprehensive evaluation in the ESP platform. Our source code is available as a separate fork of ESP on GitHub.

As part of our methodology, we first generate a variety of fixed-function accelerators (disaggregated and monolithic) through high-level synthesis (HLS). Our evaluation includes three different coherence protocols – MESI, Coherent DMA, and Spandex-FCS – and two different strategies for accelerator invocation – OS and ASI. We synthesize multiple bitstreams of SoCs – each SoC has RISC-V CPUs and accelerators in a variety of design configurations, connected with a mesh-based network-on-chip (NoC). Once the hardware is generated, we evaluate the performance of our designs on an FPGA, using multiple real-world applications running on top of a Linux operating system. This appendix provides detailed instructions for reproducing the results from our evaluation.

B.2 Artifact check-list

- **Run-time environment:** SoC prototypes on an FPGA running Linux. Host machine for using ESP must either use Docker or support an OS supported by ESP; i.e., CentOS 7, Ubuntu 18.04, or Red Hat Enterprise Linux 7.8.
- **Hardware:** Xilinx VCU118 FPGA boards [88].
- **Benchmarks:** Multiple single-accelerator and multi-accelerator benchmarks including real-world applications and comprehensive scalability studies.
- **Compilation:** Automated cross-compilation for RISC-V processors using ESP's workflow.
- **Metrics:** Execution time in CPU cycles.
- **Output:** Prints on the UART console and CSV files with tabulated results for each benchmark.
- **Experiments:** Single accelerator benchmarks for different invocation and coherence strategies (Figures 4a and 4b), multi-accelerator applications for different invocation and coherence strategies (Figure 5), comparison of a monolithic and disaggregated accelerator system (Figure 7), scalability study with different combinations of a synthetic accelerator in chained and pipelined configurations (Figure 6).
- **How much disk space required (approximately)?:** 64 GB.
- **How much time is needed to prepare workflow (approximately)?:** Setting up ESP takes 5-6 hours assuming all required tools are already installed. High-level synthesis of each accelerator takes 30 minutes on average. The generation of each FPGA bitstream takes up to 3 hours.
- **How much time is needed to complete experiments (approximately)?:** Once bitstreams are ready, entire evaluation takes 2 hours.
- **Publicly available?** Yes.
- **Code licenses (if publicly available)?:** Apache 2.0.
- **Archived?:** Yes: 10.5281/zenodo.13207536.

B.3 Description

B.3.1 Hardware dependencies. We use a Xilinx VCU118 FPGA board for our evaluations. However, our system will also work with any of the other FPGA boards supported by ESP. Deploying the compiled software on the FPGA requires an Ethernet connection to the FPGA. Further, a UART connection can be used to view the output from the FPGA using a terminal console. Instructions on the FPGA setup can be found at https://esp.cs.columbia.edu/docs/singlecore/singlecore-guide/, specifically in the sections "Debug link configuration" and "UART interface".

B.3.2 Software dependencies. The software and tool dependencies of ESP for different operating systems are listed at https://esp.cs.columbia.edu/docs/setup/setup-guide/. Users who prefer to work with a Docker image can find the steps to setup with Docker here: https://esp.cs.columbia.edu/docs/setup/docker/. In terms of CAD tools, evaluating our system requires only Xilinx Vivado 2019.2 and Cadence Stratus HLS 19.22-s100 (other versions of Stratus should work too). Finally, our evaluation requires the RISC-V software toolchain for compiling the benchmarks to run on the ESP platform. Steps to setup the toolchain can be found at: https://esp.cs.columbia.edu/docs/setup/setup-guide/#software-toolchain.

B.3.3 How to access. We integrated our source code into ESP and released it on Zenodo (https://zenodo.org/records/13363750). Once the hardware and software dependencies have been setup and installed, download the archived repository (or clone from the linked GitHub release) and initialize all submodules recursively. The root folder of the repository includes a template `esp_env.sh` script with the required environment variables that need to be modified to the appropriate values in the user's environment.

B.3.4 Important files and directories. The most relevant files and directories are as follows:

- `accelerators/stratus_hls`: Accelerators (both hardware and software benchmarks) used in the experiments, namely:
  - `audio_fft_stratus`
  - `audio_fir_stratus`
  - `audio_ffi_stratus`
  - `audio_dma_stratus`
  - `sort_stratus`
  - `gemm_stratus`
  - `tiled_app_stratus`
  - `viterbi_stratus`
  - `asi_viterbi_stratus`
  - `sensor_dma_stratus`
- `accelerators/stratus_hls/common/inc/core/accelerators/esp_accelerator_asi.hpp`: Modular implementation of the Accelerator Synchronization Interface (ASI), that allows easy integration with any accelerator in `stratus_hls`.
- `rtl/caches/spandex-caches/`: RTL implementation of the Spandex-FCS protocol, including the L2 and LLC controller.
- `rtl/caches/*_wrapper.vhd`: Cache wrapper files to instantiate Spandex-FCS caches as well as synchronization-related support (`aq/rl/fence`).
- `socs/*`: **SoC design folders.** The SoCs used for the experiments are labeled with the prefix `*-xilinx-vcu118-xcvu9p`. Each folder comes with scripts and configuration files for running the experiments relevant to that folder. Below, we list the design folders in this artifact along with a mapping of the results in the paper that they reproduce:

socs/audio-mono-esp-* : Figures 5a and 7.
socs/audio-mono-spx-* : Figures 5a and 7.
socs/fcnn-*-esp-* : Figures 4a (Sort and GeMM), 4b (Sort and GeMM) and 5c.
socs/fcnn-spx-* : Figures 4b (Sort and GeMM) and 5c.
socs/miniera-esp-* : Figures 4a (FFT), 4b (FFT) and 5b.
socs/miniera-spx-* : Figures 4b (FFT) and 5b.
socs/synth-esp-* : Figure 6.
socs/synth-spx-* : Figure 6.

socs/*/soft-build/ariane/sysroot : The Linux le system that is populated after building the toolchain and Linux.
soft/common/drivers/linux/esp/esp.c : Changes to the ESP device driver to enable ASI-based accelerator invocation.
rtl/cores/ariane/ariane/src/decoder.sv : Changes CVA6 decoder stage to generate Spandex-FCS based loads and stores based on custom RISC-V instructions.
soft/ariane : Software folder containing submodules for audio_pipeline and mini-era .

B.4 Experiment work ow

This section presents a step-by-step work ow for our experiments using the ESP platform[6]. As listed in Section B.3.4, we provide multiple design folders for reproducing the various results in the paper. For each step below, we provide automated scripts that can be run from each design folder to evaluate the benchmarks relevant to that folder:

(1) Run HLS. Generate the RTL implementation of all the accelerators listed in Section B.3.4.
Script: source ./gen-hls.sh .
(2) Generate FPGA Bitstream. Con gure the SoC layouts using the socgen.tar binary, modify wrappers and generate the bitstream. Here, socgen.tar is included in each design folder as a reference to recreate the required SoC con guration to run our benchmarks.
Script: source ./gen-vivado.sh .
(3) Build Software. Generate all the software executables required to run that benchmark (scripts in *-spx-* folders generate executables to run on Spandex hardware). We also provide pre-compiled binaries in the test folder for audio and mini-era benchmarks (whose compilation instructions can be found in the respective submodules).
Script: source ./gen-sw.sh
(4) Run baremetal tests on FPGA. Run the baremetal tests that were compiled in Step 3.
Script: source ./run-baremetal-test.sh
(5) Boot Linux on FPGA. Build Linux using the pre-built le system tarball in socs/sysroot.tar , program the bitstream and boot Linux on the FPGA. Once the boot reaches the login prompt, log in with the username root and the password openesp
Script: source ./gen-run-linux.sh
(6) Run Linux tests on FPGA. Navigate to /applications/test and run all the software executables required to run that benchmark.
Script: source ./run-linux-test.sh

[6]The resources available on the ESP website (https://esp.cs.columbia.edu/resources/) include several hands-on tutorial guides, the recordings of conference tutorials, and an overview paper.

B.5 Evaluation and Expected Results

To evaluate the results in this artifact, (1) ensure the hardware and software dependencies are met as described in Section B.3.1 and B.3.2, (2) download the source repository (included in Section B.3.3) with all submodules, (3) source the esp_env.sh script, (4) navigate to each design folder listed in Section B.3.4 and run scripts (1)-(6) from Section B.4.

B.6 User Interface

All benchmarks print out the execution results via UART, which can be viewed through a console such as minicom. An example command to open minicom is shared below:
minicom -D <FPGA UART port> -b 38400 -C <minicom.log>

B.7 Tabulating Results

In order to tabulate the results into a CSV le, we parse a print log from UART (minicom.log in the above example) using gen-results.py included in the root folder of the ESP repository, and generate results.csv in the current directory. We recommend running the parser script after all benchmarks have been evaluated from their respective design folders. Note: Linux tends to sporadically generate unavoidable internal prints (like random: fast init done). We recommend rerunning the benchmark script if this happens for consistent results.
Script: python3 gen-results.py <minicom.log> .

References

[1] AmirAli Abdolrashidi, Hodjat Asghari Esfeden, Ali Jahanshahi, Kaustubh Singh, Nael Abu-Ghazaleh, and Daniel Wong. 2021. Blockmaestro: Enabling programmer-transparent task-based execution in gpu systems. In 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). IEEE, Association for Computing Machinery, New York, NY, USA, 333 346.
[2] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi. 2023. UBFT: Microsecond-Scale BFT Using Disaggregated Memory. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (Vancouver, BC, Canada) (ASPLOS 2023). Association for Computing Machinery, New York, NY, USA, 862 877. https://doi.org/10.1145/3575693.3575732
[3] Johnathan Alsop, Weon Taek Na, Matthew D Sinclair, Samuel Grayson, and Sarita Adve. 2022. A case for ne-grain coherence specialization in heterogeneous systems. ACM Transactions on Architecture and Code Optimization (TACO) 19, 3 (2022), 1 26.
[4] Johnathan Alsop, Matthew Sinclair, and Sarita Adve. 2018. Spandex: A Flexible Interface for E cient Heterogeneous Coherence. In 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA). Association for Computing Machinery, New York, NY, USA, 261 274. https://doi.org/10.1109/ISCA.2018.00031
[5] ARM. 2017. AMBA AXI and ACE protocol speci cation. https://developer.arm.com/Architectures/AMBA.
[6] Nils Asmussen, Sebastian Haas, Carsten Weinhold, Till Miemietz, and Michael Roitzsch. 2022. E cient and Scalable Core Multiplexing with M³v. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (Lausanne, Switzerland) (ASPLOS '22). Association for Computing Machinery, New York, NY, USA, 452 466. https://doi.org/10.1145/3503222.3507741
[7] Nils Asmussen, Michael Roitzsch, and Hermann Härtig. 2019. M³x: Autonomous Accelerators via Context-Enabled Fast-Path Communication. In 2019 USENIX Annual Technical Conference (USENIX ATC 19). USENIX Association, Renton, WA, 617 632.

[8] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. 2016. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (Atlanta, Georgia, USA). Association for Computing Machinery, New York, NY, USA, 189 203. https://doi.org/10.1145/2872362.2872371

[9] Mochamad Asri, Curtis Dunham, Roxana Rusitoru, Andreas Gerstlauer, and Jonathan Beard. 2020. The N2on-Uniform Compute Device (NUCD) Architecture for Lightweight Accelerator O oad. In 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). IEEE, IEEE, New York, NY, USA, 85 96.

[10] Steven Bailey. 2018. Rapid ASIC Design for Digital Signal Processors. Ph. D. Dissertation. UC Berkeley.

[11] Saambhavi Baskaran, Mahmut Taylan Kandemir, and Jack Sampson. 2022. An architecture interface and o oad model for low-overhead, near-data, distributed accelerators. In 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, Association for Computing Machinery, New York, NY, USA, 1160 1177.

[12] Saambhavi Baskaran and Jack Sampson. 2020. Decentralized o oad-based execution on memory-centric compute cores. In The International Symposium on Memory Systems. Association for Computing Machinery, New York, NY, USA, 61 76.

[13] Nathaniel Bleier, Muhammad Husnain Mubarik, Srijan Chakraborty, Shreyas Kishore, and Rakesh Kumar. 2022. Rethinking programmable earable processors. In Proceedings of the 49th Annual International Symposium on Computer Architecture. Association for Computing Machinery, New York, NY, USA, 454 467.

[14] Boost. 2024. Atomic Flags. https://www.boost.org/doc/libs/1_78_0/doc/html/atomic/interface.html

[15] Cadence. 2022. Stratus High-Level Synthesis. https://www.cadence.com/en_US/home/tools/digital-design-and-signo /synthesis/stratus-high-level-synthesis.html, .

[16] Cadence. 2023. Genus Synthesis Solution. https://www.cadence.com/en_US/home/tools/digital-design-and-signo /synthesis/genus-synthesis-solution.html.

[17] Maico Cassel Dos Santos, Tianyu Jia, Martin Cochet, Karthik Swaminathan, Joseph Zuckerman, Paolo Mantovani, Davide Giri, Je Jun Zhang, Erik Jens Loscalzo, Gabriele Tombesi, Kevin Tien, Nandhini Chandramoorthy, John-David Wellman, David Brooks, Gu-Yeon Wei, Kenneth Shepard, Luca Carloni, and Pradip Bose. 2022. A Scalable Methodology for Agile Chip Development with Open-Source Hardware Components. In Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD). IEEE, New York, NY, USA, 85 96.

[18] Maico Cassel Dos Santos, Tianyu Jia, Joseph Zuckerman, Martin Cochet, Davide Giri, Erik Jens Loscalzo, Karthik Swaminathan, Thierry Tambe, Je Jun Zhang, Alper Buyuktosunoglu, Kuan-Lin Chiu, Giuseppe Di Guglielmo, Paolo Mantovani, Luca Piccolboni, Gabriele Tombesi, David Trilla, John-David Wellman, En-Yu Yang, Aporva Amarnath, Ying Jing, Bakshree Mishra, Joshua Park, Vignesh Suresh, Sarita Adve, Pradip Bose, David Brooks, Luca P. Carloni, Kenneth L. Shepard, and Gu-Yeon Wei. 2024. 14.5 A 12nm Linux-SMP-Capable RISC-V SoC with 14 Accelerator Types, Distributed Hardware Power Management and Flexible NoC-Based Data Orchestration. In 2024 IEEE International Solid-State Circuits Conference (ISSCC), Vol. 67. IEEE, New York, NY, USA, 262 264. https://doi.org/10.1109/ISSCC49657.2024.10454572

[19] CCIX. 2019. An Introduction to CCIX - White paper. https://www.ccixconsortium.com/library/white-paper/.

[20] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. 2011. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In 2011 International Conference on Parallel Architectures and Compilation Techniques. Association for Computing Machinery, New York, NY, USA, 155 166. https://doi.org/10.1109/PACT.2011.21

[21] Ziaul Choudhury, Anish Gulati, and Suresh Purini. 2023. FlowPix: Accelerating Image Processing Pipelines on an FPGA Overlay using a Domain Speci c Compiler. ACM Trans. Archit. Code Optim. 20, 4, Article 60 (dec 2023), 25 pages. https://doi.org/10.1145/3629523

[22] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. 2022. Enzian: An Open, General, CPU/FPGA Platform for Systems Software Research. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (Lausanne, Switzerland) (ASPLOS '22). Association for Computing Machinery, New York, NY, USA, 434 451. https://doi.org/10.1145/3503222.3507742

[23] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. 2012. CHARM: A Composable Heterogeneous Accelerator-Rich Microprocessor. In Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (Redondo Beach, California, USA). Association for Computing Machinery, New York, NY, USA, 379 384. https://doi.org/10.1145/2333660.2333747

[24] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. 2023. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (Vancouver, BC, Canada) (ASPLOS 2023). Association for Computing Machinery, New York, NY, USA, 727 741. https://doi.org/10.1145/3582016.3582031

[25] David Durst, Matthew Feldman, Dillon Hu , David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-directed scheduling of streaming accelerators. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (London, UK) (PLDI 2020). Association for Computing Machinery, New York, NY, USA, 408 422. https://doi.org/10.1145/3385412.3385983

[26] Guy Eichler, Luca Piccolboni, Davide Giri, and Luca P. Carloni. 2021. MasterMind: Many-Accelerator SoC Architecture for Real-Time Brain-Computer Interfaces. In 2021 IEEE 39th International Conference on Computer Design (ICCD). IEEE, New York, NY, USA, 101 108. https://doi.org/10.1109/ICCD53106.2021.00027

[27] ETH Zurich Integrated Systems Laboratory. 2020. Ariane Github. Available at https://github.com/lowRISC/ariane (accessed May 10, 2021).

[28] HSA Foundation. 2015. Heterogeneous System Architecture Foundation. http://hsafoundation.com.

[29] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. Association for Computing Machinery, New York, NY, USA, 3 18.

[30] Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanovic, and Parthasarathy Ranganathan. 2023. Pro ling Hyperscale Big Data Processing. In Proceedings of the 50th Annual International Symposium on Computer Architecture. Association for Computing Machinery, New York, NY, USA, 1 16.

[31] Donghyun Gouk, Miryeong Kwon, Hyeonjun Bae, Sangwon Lee, and Myoungsoo Jung. 2023. Memory Pooling With CXL. IEEE Micro 43, 2 (2023), 48 57. https://doi.org/10.1109/MM.2023.3237491

[32] Sudhanshu Gupta and Sandhya Dwarkadas. 2024. RELIEF: Relieving Memory Pressure In SoCs Via Data Movement-Aware Accelerator Scheduling. In 2024 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, IEEE, New York, NY, USA, 110 119.

[33] Tae Jun Ham, David Bruns-Smith, Brendan Sweeney, Yejin Lee, Seong Hoon Seo, U Gyeong Song, Young H Oh, Krste Asanovic, Jae W Lee, and Lisa Wu Wills. 2020. Genesis: A hardware acceleration framework for genomic data analysis. In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, Association for Computing Machinery, New York, NY, USA, 254 267.

[34] Blake A Hechtman, Shuai Che, Derek R Hower, Yingying Tian, Bradford M Beckmann, Mark D Hill, Steven K Reinhardt, and David A Wood. 2014. QuickRelease: A throughput-oriented approach to release consistency on GPUs. In 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA). IEEE, IEEE, New York, NY, USA, 189 200.

[35] Tayler Hicklin Hetherington, Maria Lubeznov, Deval Shah, and Tor M. Aamodt. 2019. Edge: Event-driven gpu execution. In 2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT). IEEE, Association for Computing Machinery, New York, NY, USA, 317 330.

[36] Mark Hill and Vijay Janapa Reddi. 2019. Gables: A roo ine model for mobile socs. In 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, IEEE, New York, NY, USA, 317 330.

[37] Rui Hou, Lixin Zhang, Michael C Huang, Kun Wang, Hubertus Franke, Yi Ge, and Xiaotao Chang. 2011. E cient data streaming with on-chip accelerators: Opportunities and challenges. In 2011 IEEE 17th International Symposium on High Performance Computer Architecture. IEEE, HPCA, New York, NY, USA, 312 320.

[38] Muhammad Huzaifa, Rishi Desai, Samuel Grayson, Xutao Jiang, Ying Jing, Jae Lee, Fang Lu, Yihan Pang, Joseph Ravichandran, Finn Sinclair, Boyuan Tian, Hengzhi Yuan, Je rey Zhang, and Sarita V. Adve. 2021. ILLIXR: Enabling End-to-End Extended Reality Research. In 2021 IEEE International Symposium on Workload Characterization (IISWC). IEEE, New York, NY, USA, 24 38. https://doi.org/10.1109/IISWC53511.2021.00014

[39] IBM. 2021. Mini-ERA. https://github.com/IBM/mini-era.

[40] Tianyu Jia, Paolo Mantovani, Maico Cassel Dos Santos, Davide Giri, Joseph Zuckerman, Erik Jens Loscalzo, Martin Cochet, Karthik Swaminathan, Gabriele Tombesi, Je Jun Zhang, Nandhini Chandramoorthy, John-David Wellman, Kevin Tien, Luca Carloni, Kenneth Shepard, David Brooks, Gu-Yeon Wei, and Pradip Bose. 2022. A 12nm Agile-Designed SoC for Swarm-Based Perception with Heterogeneous IP Blocks, a Recon gurable Memory Hierarchy, and an

800MHz Multi-Plane NoC. In ESSCIRC 2022- IEEE 48th European Solid State Circuits Conference (ESSCIRC). IEEE, New York, NY, USA, 269 272. https://doi.org/10.1109/ESSCIRC55480.2022.9911456

[41] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Pro ling a warehouse-scale computer. In Proceedings of the 42nd Annual International Symposium on Computer Architecture. Association for Computing Machinery, New York, NY, USA, 85 96.

[42] Ioannis Karageorgos, Karthik Sriram, Ján Veselý, Michael Wu, Marc Powell, David Borton, Rajit Manohar, and Abhishek Bhattacharjee. 2020. Hardware-software co-design for brain-computer interfaces. In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, Association for Computing Machinery, New York, NY, USA, 391 404.

[43] Joon Kyung Kim, Byung Hoon Ahn, Sean Kinzer, Soroush Ghodrati, Rohan Mahapatra, Brahmendra Yatham, Shu-Ting Wang, Dohee Kim, Parisa Sarikhani, Babak Mahmoudi, Divya Mahajan, Jongse Park, and Hadi Esmaeilzadeh. 2022. Yin-Yang: Programming Abstractions for Cross-Domain Multi-Acceleration. IEEE Micro 42, 5 (2022), 89 98. https://doi.org/10.1109/MM.2022.3189416

[44] Seah Kim, Jerry Zhao, Krste Asanović, Borivoje Nikolić, and Yakun Sophia Shao. 2023. AuRORA: Virtualized Accelerator Orchestration for Multi-Tenant Workloads. In 2023 56rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, Association for Computing Machinery, New York, NY, USA, 85 96.

[45] Rakesh Komuravelli, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V. Adve, and Vikram S. Adve. 2015. Stash: Have your scratchpad and cache it too. In 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA). Association for Computing Machinery, New York, NY, USA, 707 719. https://doi.org/10.1145/2749469.2750374

[46] Snehasish Kumar, Arrvindh Shriraman, and Naveen Vedula. 2015. Fusion: Design Tradeo s in Coherent Cache Hierarchies for Accelerators. SIGARCH Comput. Archit. News 43, 3S (jun 2015), 733 745. https://doi.org/10.1145/2872887.2750421

[47] Video Labs. 2021. Ambisonic encoding / decoding and binauralization library. https://github.com/videolabs/libspatialaudio.

[48] Michael LeBeane, Brandon Potter, Abhisek Pan, Alexandru Dutu, Vinay Agarwala, Wonchan Lee, Deepak Majeti, Bibek Ghimire, Eric Van Tassell, Samuel Wasmundt, Brad Benton, Mauricio Breternitz, Michael L. Chu, Mithuna Thottethodi, Lizy K. John, and Steven K. Reinhardt. 2016. Extended task queuing: Active messages for heterogeneous systems. In SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, Association for Computing Machinery, New York, NY, USA, 85 96.

[49] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (Vancouver, BC, Canada) (ASPLOS 2023). Association for Computing Machinery, New York, NY, USA, 574 587. https://doi.org/10.1145/3575693.3578835

[50] Sixu Li, Chaojian Li, Wenbo Zhu, Boyang Yu, Yang Zhao, Cheng Wan, Haoran You, Huihong Shi, and Yingyan Lin. 2023. Instant-3D: Instant Neural Radiance Field Training Towards On-Device AR/VR 3D Reconstruction. In Proceedings of the 50th Annual International Symposium on Computer Architecture. Association for Computing Machinery, New York, NY, USA, 1 13.

[51] Qiaoyi Liu, Je Setter, Dillon Hu , Maxwell Strange, Kathleen Feng, Mark Horowitz, Priyanka Raina, and Fredrik Kjolstad. 2023. Uni ed Bu er: Compiling Image Processing and Machine Learning Applications to Push-Memory Accelerators. ACM Transactions on Architecture and Code Optimization 20, 2 (2023), 1 26.

[52] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J Ligocki, Matthew J Cordery, Nicholas J Wright, Mary W Hall, and Leonid Oliker. 2015. Roo ine model toolkit: A practical tool for architectural and program analysis. In High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers 5. Springer, Springer, New York, NY, USA, 129 148.

[53] Daniel Lustig and Margaret Martonosi. 2013. Reducing GPU o oad latency via ne-grained CPU-GPU synchronization. In 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA). IEEE, IEEE, New York, NY, USA, 85 96.

[54] Michael J. Lyons, Mark Hempstead, Gu-Yeon Wei, and David Brooks. 2012. The Accelerator Store: A Shared Memory Framework for Accelerator-Based Systems. ACM Trans. Archit. Code Optim. 8, 4, Article 48 (jan 2012), 22 pages. https://doi.org/10.1145/2086696.2086727

[55] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. 2020. Agile SoC development with open ESP. In 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD). IEEE, IEEE, New York, NY, USA, 85 96.

[56] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (Vancouver, BC, Canada) (ASPLOS 2023). Association for Computing Machinery, New York, NY, USA, 742 755. https://doi.org/10.1145/3582016.3582063

[57] Muhammad Husnain Mubarik, Ramakrishna Kanungo, Tobias Zirr, and Rakesh Kumar. 2023. Hardware Acceleration of Neural Graphics. In Proceedings of the 50th Annual International Symposium on Computer Architecture (Orlando, FL, USA) (ISCA '23). Association for Computing Machinery, New York, NY, USA, Article 50, 12 pages. https://doi.org/10.1145/3579371.3589085

[58] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. 2022. Instant Neural Graphics Primitives with a Multiresolution Hash Encoding. ACM Trans. Graph. 41, 4, Article 102 (July 2022), 15 pages. https://doi.org/10.1145/3528223.3530127

[59] Nachiappan Chidambaram Nachiappan, Haibo Zhang, Jihyun Ryoo, Niranjan Soundararajan, Anand Sivasubramaniam, Mahmut T. Kandemir, Ravi Iyer, and Chita R. Das. 2015. VIP: Virtualizing IP chains on handheld platforms. In 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA). Association for Computing Machinery, New York, NY, USA, 655 667. https://doi.org/10.1145/2749469.2750382

[60] Abishek Ramdas, Michael Giardino, Runbin Shi, Adam Turowski, David Cock, Gustavo Alonso, and Timothy Roscoe. 2022. ECI: a Customizable Cache Coherency Stack for Hybrid FPGA-CPU Architectures. https://doi.org/10.48550/ARXIV.2208.07124

[61] Jason Redgrave, Albert Meixner, Nathan Goulding-Hotta, Artem Vasilyev, and Ofer Shacham. 2018. Pixel visual core: Google's fully programmable image vision and AI processor for mobile devices. In Proc. IEEE Hot Chips Symp. (HCS). IEEE, New York, NY, USA, 1 18.

[62] CPP Reference. 2024. std::atomic_ ag. https://en.cppreference.com/w/cpp/atomic/atomic_ ag

[63] Xiaowei Ren, Daniel Lustig, Evgeny Bolotin, Aamer Jaleel, Oreste Villa, and David Nellans. 2020. HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems. In 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, New York, NY, USA, 582 595. https://doi.org/10.1109/HPCA47549.2020.00054

[64] Daniel Richins, Dharmisha Doshi, Matthew Blackmore, Aswathy Thulaseedharan Nair, Neha Pathapati, Ankit Patel, Brainard Daguman, Daniel Dobrijalowski, Ramesh Illikkal, Kevin Long, David Zimmerman, and Vijay Janapa Reddi. 2020. Missing the forest for the trees: End-to-end ai application performance in edge data centers. In 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, IEEE, New York, NY, USA, 515 528.

[65] RISC-V. 2024. RVWMO Memory Consistency Model, Version 2.0. RISC-V. https://ve-embeddev.com/riscv-isa-manual/latest/rvwmo.html

[66] Alberto Ros and Stefanos Kaxiras. 2012. Complexity-e ective multicore coherence. In Proceedings of the 21st international conference on Parallel architectures and compilation techniques. Association for Computing Machinery, New York, NY, USA, 241 252.

[67] Alberto Ros and Stefanos Kaxiras. 2015. Callback: E cient synchronization without invalidation with a directory just for spin-waiting. In 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA). Association for Computing Machinery, New York, NY, USA, 427 438. https://doi.org/10.1145/2749469.2750405

[68] Chaoyi Ruan, Yingqiang Zhang, Chao Bi, Xiaosong Ma, Hao Chen, Feifei Li, Xinjun Yang, Cheng Li, Ashraf Aboulnaga, and Yinlong Xu. 2023. Persistent Memory Disaggregation for Cloud-Native Relational Databases. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (Vancouver, BC, Canada) (ASPLOS 2023). Association for Computing Machinery, New York, NY, USA, 498 512. https://doi.org/10.1145/3582016.3582055

[69] Yakun Sophia Shao, Sam Likun Xi, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David Brooks. 2016. Co-designing accelerators and SoC interfaces using gem5-Aladdin. In 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, Association for Computing Machinery, New York, NY, USA, 85 96.

[70] Debendra Das Sharma. 2022. Compute Express Link: An open industry-standard interconnect enabling heterogeneous data-centric computing. In 2022 IEEE Symposium on High-Performance Interconnects (HOTI). IEEE, New York, NY, USA, 5 12. https://doi.org/10.1109/HOTI55740.2022.00017

[71] Junyi Shu, Ruidong Zhu, Yun Ma, Gang Huang, Hong Mei, Xuanzhe Liu, and Xin Jin. 2023. Disaggregated RAID Storage in Modern Datacenters. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (Vancouver, BC, Canada) (ASPLOS 2023). Association for Computing Machinery, New York, NY, USA, 147 163. https://doi.org/10.1145/3582016.3582027

[72] Inderpreet Singh, Arrvindh Shriraman, Wilson WL Fung, Mike O'Connor, and Tor M Aamodt. 2013. Cache coherence for GPU architectures. In 2013 IEEE 19th

International Symposium on High Performance Computer Architecture (HPCA). IEEE, IEEE, New York, NY, USA, 578 590.

[73] Karthik Sriram, Raghavendra Pradyumna Pothukuchi, Michał Gerasimiuk, Muhammed Ugur, Oliver Ye, Rajit Manohar, Anurag Khandelwal, and Abhishek Bhattacharjee. 2023. SCALO: An Accelerator-Rich Distributed System for Scalable Brain-Computer Interfacing. In Proceedings of the 50th Annual International Symposium on Computer Architecture. Association for Computing Machinery, New York, NY, USA, 1 20.

[74] Akshitha Sriraman and Thomas F. Wenisch. 2018. μSuite: A Benchmark Suite for Microservices. In IEEE International Symposium on Workload Characterization. IEEE, New York, NY, USA, 85 96.

[75] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. 2015. CAPI: A Coherent Accelerator Processor Interface. IBM Journal of Research and Development 59, 1 (2015), 7:1 7:7. https://doi.org/10.1147/JRD.2014.2380198

[76] J. Stuecheli, W. J. Starke, J. D. Irish, L. B. Arimilli, D. Dreps, B. Blaner, C. Wollbrink, and B. Allison. 2018. IBM POWER9 opens up a new era of acceleration enablement: OpenCAPI. IBM Journal of Research and Development 62, 4/5 (2018), 8:1 8:8. https://doi.org/10.1147/JRD.2018.2856978

[77] Hyojin Sung and Sarita V. Adve. 2015. DeNovoSync: Efficient Support for Arbitrary Synchronization without Writer-Initiated Invalidations. In Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (Istanbul, Turkey) (ASPLOS '15). Association for Computing Machinery, New York, NY, USA, 545 559. https://doi.org/10.1145/2694344.2694356

[78] Hyojin Sung, Rakesh Komuravelli, and Sarita V. Adve. 2013. DeNovoND: Efficient Hardware Support for Disciplined Non-Determinism. In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (Houston, Texas, USA) (ASPLOS '13). Association for Computing Machinery, New York, NY, USA, 13 26. https://doi.org/10.1145/2451116.2451119

[79] Cheng Tan, Manupa Karunaratne, Tulika Mitra, and Li-Shiuan Peh. 2018. Stitch: Fusible heterogeneous accelerators enmeshed with many-core architecture for wearables. In 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA). IEEE, Association for Computing Machinery, New York, NY, USA, 575 587.

[80] Zhengzhong Tu, Hossein Talebi, Han Zhang, Feng Yang, Peyman Milanfar, Alan Bovik, and Yinxiao Li. 2022. Maxim: Multi-axis mlp for image processing. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. IEEE, New York, NY, USA, 5769 5780.

[81] Thiruvengadam Vijayaraghavan, Yasuko Eckert, Gabriel H. Loh, Michael J. Schulte, Mike Ignatowski, Bradford M. Beckmann, William C. Brantley, Joseph L. Greathouse, Wei Huang, Arun Karunanithi, Onur Kayiran, Mitesh Meswani, Indrani Paul, Matthew Poremba, Steven Raasch, Steven K. Reinhardt, Greg Sadowski, and Vilas Sridharan. 2017. Design and Analysis of an APU for Exascale Computing. In 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, IEEE, New York, NY, USA, 85 96.

[82] Moyang Wang, Tuan Ta, Lin Cheng, and Christopher Batten. 2020. Efficiently Supporting Dynamic Task Parallelism on Heterogeneous Cache-Coherent Systems. In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). Association for Computing Machinery, New York, NY, USA, 173 186. https://doi.org/10.1109/ISCA45697.2020.00025

[83] Shu-Ting Wang, Hanyang Xu, Amin Mamandipoor, Rohan Mahapatra, Byung Hoon Ahn, Soroush Ghodrati, Krishnan Kailas, Mohammad Alian, and Hadi Esmaeilzadeh. 2024. Data Motion Acceleration: Chaining Cross-Domain Multi Accelerators. In 2024 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, IEEE, New York, NY, USA, 85 96.

[84] Zirui Wang, Shangzhe Wu, Weidi Xie, Min Chen, and Victor Adrian Prisacariu. 2022. NeRF−−: Neural Radiance Fields Without Known Camera Parameters. arXiv:2102.07064 [cs.CV] https://arxiv.org/abs/2102.07064

[85] Tianrui Wei, Nazerke Turtayeva, Marcelo Orenes-Vera, Omkar Lonkar, and Jonathan Balkind. 2023. Cohort: Software-Oriented Acceleration for Heterogeneous SoCs. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (Vancouver, BC, Canada) (ASPLOS 2023). Association for Computing Machinery, New York, NY, USA, 105 117. https://doi.org/10.1145/3582016.3582059

[86] Lisa Wu, Andrea Lottarini, Timothy K Paine, Martha A Kim, and Kenneth A Ross. 2014. Q100: The architecture and design of a database processing unit. ACM SIGARCH Computer Architecture News 42, 1 (2014), 255 268.

[87] Xilinx. 2022. Xilinx Vivado. https://www.xilinx.com/products/design-tools/vivado.html.

[88] Xilinx. 2024. VCU118 Evaluation Board User Guide. https://www.xilinx.com/content/dam/xilinx/support/documentation/boards_and_kits/vcu118/ug1224-vcu118-eval-bd.pdf.

[89] Amir Khani Yengikand, Majid Meghdadi, Sajad Ahmadian, Seyed Mohammad Jafar Jalali, Abbas Khosravi, and Saeid Nahavandi. 2021. Deep representation learning using multilayer perceptron and stacked autoencoder for recommendation systems. In 2021 IEEE international conference on systems, man, and cybernetics (SMC). IEEE, IEEE, New York, NY, USA, 2485 2491.

[90] Yifan Yuan, Jinghan Huang, Yan Sun, Tianchen Wang, Jacob Nelson, Dan R. K. Ports, Yipeng Wang, Ren Wang, Charlie Tai, and Nam Sung Kim. 2023. Rambda: RDMA-driven Acceleration Framework for Memory-intensive μs-scale Datacenter Applications. In 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, New York, NY, USA, 499 515. https://doi.org/10.1109/HPCA56546.2023.10071127

[91] Yifan Yuan, Ren Wang, Narayan Ranganathan, Nikhil Rao, Sanjay Kumar, Philip Lantz, Vivekananthan Sanjeepan, Jorge Cabrera, Atul Kwatra, Rajesh Sankaran, Ipoom Jeong, and Nam Sung Kim. 2024. Intel Accelerators Ecosystem: An SoC-Oriented Perspective : Industry Product. In 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA). IEEE, New York, NY, USA, 848 862. https://doi.org/10.1109/ISCA59077.2024.00066

[92] Zeran Zhu. 2021. Hardware implementation and evaluation of the Spandex cache coherence protocol. Master's thesis. University of Illinois at Urbana-Champaign.

[93] Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Yubo Du, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Deming Chen, Jason Cong, and Peipei Zhou. 2023. CHARM: Composing Heterogeneous AcceleRators for Matrix Multiply on Versal ACAP Architecture. In Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays. Association for Computing Machinery, New York, NY, USA, 153 164.

[94] Joseph Zuckerman, Davide Giri, Jihye Kwon, Paolo Mantovani, and Luca P Carloni. 2021. Cohmeleon: Learning-based orchestration of accelerator coherence in heterogeneous socs. In MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture. Association for Computing Machinery, New York, NY, USA, 350 365.

[95] Joseph Zuckerman, Paolo Mantovani, Davide Giri, and Luca P. Carloni. 2022. Enabling Heterogeneous, Multicore SoC Research with RISC-V and ESP. In Workshop on Computer Architecture Research with RISC-V (CARRV). IEEE, New York, NY, USA.