Cohort: Software-Oriented Acceleration for Heterogeneous SoCs

Tianrui Wei  
UC Berkeley, USA  
tianruiwei@berkeley.edu

Nazerke Turtayeva  
UC Santa Barbara, USA  
nurtayeva@ucsb.edu

Marcelo Orenes-Vera  
Princeton University, USA  
movera@princeton.edu

Omkar Lonkar  
UC Santa Barbara, USA  
omkarlonkar@ucsb.edu

Jonathan Balkind  
UC Santa Barbara, USA  
jbalkind@ucsb.edu

ABSTRACT
Philosophically, our approaches to acceleration focus on the extreme. We must optimise accelerators to the maximum, leaving software to fix any hardware-software mismatches. Today’s software abstractions for programming accelerators leak hardware details, requiring changes to data formats and manual memory and coherence management, among other issues. This harms generality and requires deep hardware knowledge to efficiently program accelerators, a state which we consider hardware-oriented.

This paper proposes Software-Oriented Acceleration (SOA), where software uses existing abstractions, like software shared-memory queues, to interact with accelerators. We introduce the Cohort engine which exploits these queues’ standard semantics to efficiently connect producers and consumers in software with accelerators with minimal application changes. Accelerators are even usable in chains which can be runtime reconfigured by software. Cohort significantly reduces the burden to add new accelerators while maintaining system-level guarantees. We implement a Cohort FPGA prototype which supports SOA applications running on multicore Linux. Our evaluation shows speedups for Cohort over traditional approaches ranging from 1.83× to 8.38× over MMIO, and from 1.69× to 11.24× for DMA baselines. Our software-oriented batching optimisations within Cohort also improve performance from 2.32× to 8.10×, demonstrating the power of SOA.

CCS CONCEPTS
• Computer systems organization → Multicore architectures; Reconfigurable computing; Heterogeneous (hybrid) systems.

KEYWORDS
accelerators, heterogeneous systems, programming models, shared memory

ACM Reference Format:

Permission to make digital or hard copies of part or all 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).

ASPL ’23, March 25–29, 2023, Vancouver, BC, Canada  
© 2023 Copyright held by the owner/author(s).  
ACM ISBN 978-1-4503-9918-0/23/03.  
https://doi.org/10.1145/3582016.3582059

1 INTRODUCTION
To meet users’ efficiency and performance demands, systems-on-chip (SoCs) feature many efficient, specialised accelerators alongside general-purpose cores [16, 21, 37, 39, 62, 81]. The architecture of these SoCs suggests the accelerators are subservient to the general-purpose cores, which manage their memory allocation, scheduling, and communication. One could thus view this as a software-oriented SoC rather than an accelerator- or hardware-oriented one. However, we argue that the system is driven by the demands of accelerators, which have special needs and frustratingly bespoke programming paradigms, lacking commonality [4, 23, 31, 46, 52, 61, 73] and requiring more code with each additional accelerator. Further, the hardware accelerators cannot be easily composed together since layers of privileged software must mediate communication. If heterogeneity is to further grow, we must find more efficient means for management of and communication with hardware accelerators.

Due to accelerators’ piecemeal adoption, their software and hardware interfaces are routinely afterthoughts. There is little consideration for hardware or software approaches which are common across accelerators and smooth adoption. An ideal approach would feature:

• A common, flexible hardware interface to efficiently communicate between software and hardware elements.
• A software paradigm using existing abstractions to minimise software modification to add new accelerators.

Figure 1: Proposed Cohort System Design
We introduce an approach, termed Software-Oriented Acceleration, to emphasise that accelerator communication, isolation, and memory management respect existing software paradigms. Our chosen software abstraction is shared-memory queues, with which we reduce hardware and software overhead and ease accelerator adoption. Our system design (Figure 1) has cores and accelerators communicating arbitrarily as peers, forming a Cohort. This enables accelerator composition without software imposition, a pattern which has been underutilised to date.

The Cohort SOA model focuses on accelerators which follow a stream/buffer in and stream/buffer out (SBIO) communication pattern. Many accelerators follow this model, using memory-mapped I/O (MMIO) or direct memory access (DMA) to consume and produce buffers or streams of data [26, 43, 47, 52, 75]. For accelerators with broader communication patterns than SBIO (e.g., GPGPUs or sparse graph and neural network accelerators [7, 39, 59, 62, 72]), Cohort can enable more efficient communication for the use cases which are SBIO.

Cohort provides a common, standard queue interface both to software and to hardware, enabling reuse both of existing multithreaded software and of existing hardware accelerators with little to no modification. To bridge between software-friendly shared-memory queues and hardware-friendly interfaces, we created the Cohort engine. With it, existing software produces data into queues for consumption by hardware or software without special memory allocation routines, cache coherence management, etc. Likewise, accelerators are connected unmodified with existing, high-performance, latency-insensitive [12, 13, 71] hardware interfaces including AXI-Stream [3].

We implement Cohort on an SoC using the native coherence protocol to enable efficient operation. Cohort’s primitives could also be implemented atop emerging cache coherent interconnects beyond the SoC scale, e.g. CXL, CCIX, and CAPI [15, 70, 74]. We have implemented a software-oriented SoC prototype on FPGA with RISC-V cores booting Linux and accelerators connected via Cohort engines (shown in Figure 2). Our evaluation tests lightly modified software running on Linux, communicating between accelerators and software threads with no consideration for the nature of the producer or consumer on the other side of the queues.

Our paper makes the following contributions:

- The software-oriented acceleration approach, using existing communication, isolation, and memory management from software to manage hardware accelerators.
- The identification of queue coherence semantics to enable efficient hardware support for software shared-memory queues.
- Implementation of a Cohort-enabled SoC with Linux support and multiple integrated accelerators.
- SHA benchmark speedup from Cohort’s Engine and API range from 5.44x to 8.38x over MMIO and from 7.27x to 11.24x over coherent DMA baselines.
- Open-source hardware and software of Cohort available at https://github.com/cohort-project

2 MOTIVATION

Having built and programmed many heterogeneous SoCs, we have found issues across the stack. Here we describe how they motivated us to create a more software-oriented SoC design approach. These tie to the two key features mentioned in Section 1: a common, flexible hardware interface, and a software-friendly programming paradigm.

2.1 Software Programmability Issues

Accelerator vendors provide varying software interfaces [29, 30, 36, 45, 49]. Some provide compiler passes or domain specific languages, some call into privileged kernel modules, and others adopt vendor-specific libraries, making for a fragmented software environment. To exploit multiple vendors’ accelerators becomes difficult, requiring a variety of bespoke expertise. This results in fragile software, as vendors’ requirements conflict, if composition is at all possible.

We have also been challenged by memory allocation and management for accelerators. It is routine for new accelerators to require special memory allocations or the addition of manual cache flushing [52, 59, 74]. These requirements come for a variety of reasons, including accelerators’ use of IOMMUs, differing coherence models, and special addressing and alignment constraints. As a result, the task of complexity management falls to software.

In existing SoCs which use MMIO to configure and interact with accelerators and I/O devices, MMIO loads and stores have special semantics which hurt performance. MMIO operations have side effects, meaning that they cannot be performed speculatively and that interrupts generally should not be taken while they are outstanding in the memory system. Many cores, particularly slim in-order ones, will simply stall on these until a load or store returns from the accelerator or device. This behaviour is often relied on for synchronisation, with the return guaranteeing that an operation has completed. In terms of programmability, performing these MMIO operations from user mode is fraught with problems to ensure that operations are not optimised out or reordered by the compiler. We argue that MMIO is undesirable for performance and programmability and believe this is backed up by other recent approaches in, for example, high-performance networking, where shared-memory queues have replaced such operations [57, 80].
2.2 Hardware Interfacing Issues
It is now essential to optimise the data-movement interface, both between cores and accelerators and among accelerators. Coarse-grained accelerators in SoCs today are connected in various ways [14, 39, 59]. Some act as MMIO devices, with private registers or memories that are filled prior to invocation, and read out upon completion. Others use DMA engines, programmed prior to invocation, to fetch data from and store data to memory.

Accelerators also vary in their coherence models, often operating partially- or non-coherently, which complicates software. While fully-coherent accelerators are convenient, they have costs from participating in cache-coherence. Balancing these models is sufficiently complex that research has even been done on machine learning techniques to choose between them [83].

In desktop- and server-class settings with PCIe, protocols like CCIX, CXL, and CAPI [15, 70, 74] enable accelerator cache coherence. However, building on PCIe brings overhead and latency, meaning applications with strict latency requirements usually perform poorly. Neugebauer et al. [58] characterised end host networking and found that for a 128B payload, 90% of the observed 1000ns round-trip latency comes from PCIe itself. In addition, coherence systems like CCIX can offload coherence management to accelerators, meaning coherent access to accelerator-managed memory regions is PCIe topology dependent, which further complicates the system [15]. Many mobile-class SoCs limit use of PCIe due to its power and area, relying instead on native (shared) memory for many of their accelerators and peripherals [54, 55, 60, 65, 66].

3 COHORT APPROACH
3.1 Software-Oriented Acceleration Philosophy
At a high level, SOA reuses existing software mechanisms for communication and application design. Software is routinely optimised in unforeseen ways to drive performance and efficiency. Using programming languages’ and libraries’ existing abstractions, we enable new avenues for optimisations across many hardware accelerators. We need not adopt special I/O operations, memory and cache management, or complex accelerator-specific drivers, unlike existing approaches [52, 59].

SOA is user-mode-oriented to give applications and language runtimes greater flexibility, isolation, and security. Mechanisms which provide kernel interposition are preferred as they enable enforcing system-level policy, profiling, etc. However, encouraging user-level operation should not fundamentally require context switches and kernel interaction for operation, which hurt performance.

3.2 Queue Coherence
In Cohort, we adopt shared-memory queues as the lingua franca of our heterogeneous SoC, as shown in Figure 1. In high-performance parallel software, shared-memory queues enable the decoupling of threads to gain parallelism [53, 76]. We work to provide a common producer-consumer communication abstraction in order to improve programmability and composition, and to enable new performance tuning opportunities. Cohort users replace existing queue-decoupled software threads with accelerators while maintaining their existing code and its chains of queues connecting producers and consumers.

Many parallelisation techniques rely on the queue-based producer-consumer design pattern for communication and decoupling [33, 37, 38, 51, 61, 69, 78]. Producers and consumers maintain a common queue for communication of data and synchronisation. When available, the producer hands data to the consumer by pushing it into the queue. When a consumer is able to process more data, it pops it from the queue. The producer-consumer design pattern is used in both shared-memory multithreaded applications and for inter-process communication by sharing a small memory region between the processes for the queue. Figure 3 shows a simple example code snippet following the generic queue API shown in Table 1.

Much effort has gone into developing high-performance lock-free queue libraries. These libraries provide performance by minimising coherence effects and taking advantage of the memory-level parallelism of modern cores. With their widespread use, these queues’ semantics are well understood and ready to be exploited in Cohort. We focus here on lock-free single producer-single consumer (SPSC) queues, which are supported in widely-adopted software libraries [11] and further boost performance.

We introduce the term Queue Coherence to refer to the semantic behaviour of SPSC queues. In existing libraries, there is an agreed meaning to enqueue/push and dequeue/pop with respect to both cache coherence and memory consistency. When the producer completes an enqueue and the consumer observes a change in the queue’s write pointer, there is a guarantee that the consumer will also observe the updated data items in the queue. Cohort exploits queue coherence by implementing queue operations in an engine connected to the cache coherence system. This enables software to operate as designed while feeding accelerators with the highest performance and efficiency that general-purpose cores offer.

---

Table 1: Cohort API Listing

<table>
<thead>
<tr>
<th>Existing generic SPSC queue API calls</th>
<th>Additional Cohort-specific API calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>int fifo_init(int element_size, int queue_length);</td>
<td>int cohort_unregister(int acc_id, fifo_t *acc_in, fifo_t *acc_out);</td>
</tr>
<tr>
<td>void push(int element, fifo_t *q);</td>
<td>int cohort_register(int acc_id, fifo_t *acc_in, fifo_t *acc_out);</td>
</tr>
<tr>
<td>int pop(fifo_t *q);</td>
<td></td>
</tr>
<tr>
<td>int fifo_deinit(fifo_t *q);</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: Generic producer-consumer code snippet

```c
1  
2  
3  pid_t pid = fork();
4  if (pid == 0) { // producer
5     printf("element: %d
\n", element);
6  }
7  
8  else { // consumer
9     printf("element: %d\n", element);
10    }
```
Figure 4: Producer-consumer code snippet for Cohort with a single accelerator

```c
fifo_t compute_fifo = fifo_init(...);
fifo_t result_fifo =fifo_init(...);
cohort_register(acc, compute_fifo, result_fifo);
push(acc_in_elem, compute_fifo);
int acc_out_elem = pop(result_fifo);
printf("element is %d\n", acc_out_elem);
```

Figure 5: Producer-consumer code snippet for Cohort accelerator chaining

```c
fifo_t encrypt_fifo = fifo_init(...);
fifo_t hash_fifo = fifo_init(...);
fifo_t result_fifo = fifo_init(...);
cohort_register(encrypt_acc, encrypt_fifo, hash_fifo);
cohort_register(hash_acc, hash_fifo, result_fifo);
push(data, encrypt_fifo);
int chain_result = pop(result_fifo);
```

### 3.3 Software-Hardware Composition

Cohort enables replacement of a software thread with a Cohort engine to enable transparent acceleration, as Figure 4 shows. This “accelerator thread” communicates with our software threads in the same way that another software thread would. Instead of leaning on complex accelerator management code and driver support, offloading computation to a Cohort-enabled accelerator is as transparent as pushing data into the software queue connected to the accelerator’s input. To receive results back, the software thread simply pops data from another software queue connected to the accelerator’s output.

In addition, the interoperability brings the benefit of transparent accelerator chaining, enabling chaining of a series of computations through several accelerators, whilst being transparent to software. If we have a hashing and an encryption accelerator, then to perform an encryption followed by a hash, we only need the code snippet in Figure 5.

Our approach, introducing Cohort, provides this exact ease of programming. With a minor addition in the form of the queue registration routine (which does not modify the underlying memory allocation and could be incorporated into the queue library), software queues are connected to Cohort units on-chip, ready for production and consumption by accelerators. This enables runtime reconfiguration of the hardware, with accelerator chains created dynamically by software. The programmer can maintain their high-level producer-consumer abstraction while incrementally moving functionality to hardware accelerators.

### 4 COHORT IMPLEMENTATION

This section explains the implementation of Cohort and its integration into the SoC as well as the library and operating system support we developed to enable booting SMP Linux on the Cohort SoC on FPGA.

4.1 Programming Model

Cohort is designed for use from user mode with minimal operating system involvement. There is a single Cohort driver to support all Cohort-enabled accelerators, regardless of the variety of accelerators available. This is in contrast to many existing accelerator drivers which grow the size and complexity of our trusted software over time.

To support maximally transparent use of SPSC queues, the Cohort engine features an MMU compatible with the cores’ ISA’s MMU. In a modern SoC, such an MMU is low cost and its benefits are significant, as we show in Section 6. Queues are virtually contiguous and are accessed sequentially, which significantly improves TLB hit rates. Beyond this, Cohort benefits from the very same optimisations as software programmers adopt for multicore processors. E.g. If the programmer adopts huge pages in their software, the Cohort MMU will take advantage of them and see performance improvements just as cores would.

4.1.1 Queue Descriptors. A common goal for high-performance software is to choose a queue structure that minimises coherence traffic. Software queues are thus organised in a variety of ways that Cohort must support. To register a queue with Cohort, its structure must be described to properly configure the Cohort engine. For Cohort to achieve broad compatibility, desirable features of SPSC queues include configurable element size, configurable queue size, and use of read and write indices versus pointers.

To support a range of queue formats, we develop a queue descriptor struct which the queue library developer uses to describe their queue. The descriptor also contains (virtually addressed) pointers to the queue elements in question, such as the read or write index. At present, we support the following attributes for the queue.

- `write_pointer/index`
- `read_pointer/index`
- `fifo_base_address`
- `fifo_element_size`
- `fifo_length`

4.1.2 User-Mode API. Figure 7 shows a code snippet demonstrating the Cohort user-mode programming interface. The programmer starts by allocating two cohort-enabled queues with queue descriptors in their `fifo_t` structures from their queue library. The programmer then registers the two queues connected to a
Thanks to queue coherence, the Cohort engine bridges from software-the Cohort engine and otherwise the accelerator avoids polling. endpoints provided by the Cohort engine, for consumption of input. Instead, the accelerator connects to two latency-insensitive interfaces, including standard valid/ready FIFOs. For SBIO accelerators, Cohort does not require the accelerator to have any knowledge of cache coherence system, how to optimize it for their specific needs, and uses it to hide coherence system details from the accelerators. For SBIO accelerators, Cohort does not require the accelerator to have any knowledge of cache coherence. Instead, the accelerator connects to two latency-insensitive endpoints provided by the Cohort engine, for consumption of input and production of results. When data is available, it is provided by the Cohort engine and otherwise the accelerator avoids polling.

As shown in Figure 6, Cohort is separated into several functional units, each handling a specific task. Our prototype Cohort engine is integrated into the P-Mesh cache coherence system of Open-Piton [8, 10], which is an open-source tile-based SoC framework. The components of P-Mesh that Cohort directly interacts with are shown in orange at the top of the figure. Cohort itself is rendered in blue in the center of the figure, while the connected accelerator is shown at the bottom in green. Cohort is integrated on its own tile within the SoC: a zoomed out view of a complete SoC is shown in Figure 2.

The Cohort Engine has a few main components: uncached configuration registers, the Memory Transaction Engine (MTE) which contains the MMU, a Consumer endpoint, and a Producer endpoint. CPU cores may configure Cohort through its uncached configuration registers, which are the only MMIO component of Cohort. The producer and consumer endpoints are responsible for generating queue operations. When the endpoints need to communicate with memory, the MTE translates from higher level memory operations and invalidations to physically addressed coherence operations. The endpoints effectively translate the software queue structure into a data stream for the connected accelerator to handle, with a hardware-friendly interface.

4.2 Cohort Engine

Cohort-enhanced accelerator by calling the cohort_register system call with the ID of the accelerator to be used plus its input and output queues. From that point, the standard push and pop functions from their queue library are used to feed data to and from their accelerator. At the end of the application, they then call cohort_unregister to end their use of the queues with the accelerator by again providing its ID and input and output queues.

From the programmer’s perspective, the programming experience is extremely close to programming for an everyday multi-threaded environment. Unlike existing approaches which either expose complexity to the programmer or try to hide it in allocation routines and syscalls, Cohort-enabled library code continues nearly unmodified. The queue library developer simply describes their queue via a Cohort queue descriptor to make the queue usable by the programmer, with no need for specialised memory allocation or deallocation routines. The only addition are the cohort_register and cohort_unregister syscalls provided by the Cohort kernel driver. A generic representation of the existing queue APIs on which Cohort can be built is shown in Table 1 alongside the two new API calls required for Cohort.

Queue Library Support. Thanks to Cohort’s flexible queue descriptors, it is reusable in a variety of queue settings. As noted in Section 3.2, SPSC queues are commonly adopted in software libraries. We designed Cohort’s hardware based on established queue implementations in software and thus have seen remarkably straightforward development and integration. While our evaluation uses a hand-rolled C implementation, we have also demonstrated Cohort’s cohesive integration with a high-level software library by implementing support in the C++ Boost Lockfree library to communicate with Cohort-enabled accelerators [11].

4.2.2 Producer Endpoint. The producer endpoint performs the reciprocal operations of the consumer endpoint. When the accelerator produces data (its valid signal goes high) and the queue is not full, the accelerator will take the data, store it into the queue, and then update the write pointer. Both operations are performed with appropriate consistency guarantees and the updates are communicated to the consumer via the coherence system.

4.2.3 Semantics and Coherence. As discussed in section 3.2, SPSC queues signal elements being pushed or popped by changing the write or read pointer, respectively. When a Cohort or software endpoint alters a pointer in the queue structure via the MTE, it means that elements are being pushed to or popped from the queue, following software semantics. In the SoC’s microarchitecture, we assume that the coherence system will cause the line to be invalidated in other caches, which acts as a signal to Cohort endpoints to look for an update.

We exploit this behaviour in two ways for Cohort to transparently observe the changes made by software. The Reader Coherency Manager (RCM) monitors an address passed from the uncached registers for incoming invalidations. Whenever an invalidation matches the address the RCM monitors, it enters the backoff state. The length of the backoff is controlled by the backoff unit. After the
backoff ends, the RCM issues a coherent read to bring the most up
to date copy of the data. Consistency is guaranteed by the writer
ordering the data write before the pointer write using appropri-
ate fences or store semantics, which is the standard behaviour in
lock-free SPSC queue libraries. Just as the software library does,
the Write Coherency Manager (WCM) carefully orders its write
operations to ensure that a reader will see the update to the write
pointer before seeing the data update.

4.2.4 Cohort MMU. For our implementation of Cohort in Open-
Piton+Ariane, we include a RISC-V Sv39-compliant MMU, based
on the MMU in the Ariane core itself. This MMU enables Cohort’s
various components to make use of virtual addresses, significantly
simplifying the programming of Cohort-enhanced accelerators. As
a result of the ISA-native MMU, queues are allocatable with malloc
and the Cohort endpoints seamlessly translate VAs to PAs.

The Cohort MMU features a TLB and page table walker to max-
imise its independence from the cores in the SoC. The operating
system manages the Cohort MMU coherently with the others in
the SoC, as we describe in Section 4.4. Each MMU has the page
base pointer pointing to the root address of the page table of the
corresponding process. Upon a TLB miss, the page table walker
traverses the page table and refills the entry transparently. When
there is a page fault due to a miss in the TLB and an unsuccessful
page table walk, the Cohort MMU raises an interrupt to a core to
resolve the page fault. Once resolved, the core writes to one of two
MMU registers: the first simply resolves the fault and requires the
page table walker to complete its own page table walk, while the
second enables the core to write the page table entry directly into
the TLB. Besides these registers, the Cohort MMU’s TLB is flushed
via a write to another register, in order to maintain TLB coherence.

4.3 Accelerator Interface

Figure 6 shows the accelerator connecting to the consumer and pro-
ducer endpoints. Our prototype supports both simple valid-ready
handshakes [12, 13, 71] and AXI-Stream [3] as latency insensitive
interfaces to and from the accelerator. As the consumer endpoint
retrieves data from the queue feeding into the accelerator, it sets
its valid signal to the accelerator and waits for the accelerator to
be ready to consume the data. Similarly, the accelerator sets its
valid signal high to produce data to the producer endpoint, which
makes the corresponding updates to the queue feeding out of the
accelerator. Many SBIO accelerators are usable unmodified with the
Cohort accelerator interface, while others could have their LSUs
straightforwardly replaced with the Cohort engine.

Accelerators need not produce data with the same interface
width as it consumes data. Cohort produces and consumes data
to/from the accelerator using blocks of parameterised size, with
appropriate ratchet logic to resize data to the accelerator’s required
sizes. As an optimisation, the producer and consumer endpoints
reduce coherence traffic commensurate with the accelerator’s data
block input or output size, updating the read or write pointers by
the data block size.

Upon queue registration to initiate execution, the user is also able
to provide a data buffer akin to a control and status register (CSR)
bank to configure the accelerator. The user simply points Cohort to
the virtually contiguous data block, which the programmer formats
as an accelerator-specific struct. The Cohort-enhanced accelerator
directly receives this configuration data at registration time before
data is passed. For accelerators with CSRs configured by AXI-Lite,
Cohort also supports writing the data over AXI-Lite.

We have so far connected four accelerators into our Cohort SoC.
Section 5.2 describes the integration of three of these: SHA-256,
AES, and an H264 encoder; we have also connected a short-time
Fourier transform accelerator which we do not describe here. Each
accelerator uses valid-ready handshakes for input and output data.
We have demonstrated AXI-Stream functionality using an AXI-
Stream FIFO as a “null” accelerator, which is easily replaceable with
a more complete accelerator.

4.4 OS Support

Cohort provides a kernel driver to establish a complete and safe
data flow and to abstract hardware details from the user. It shrinks
the user space API to Queue Coherent semantics only and passes the
low-level details to kernel via two simple system calls. Unlike
many embedded accelerator environments, user space may not
touch Cohort’s configuration registers and need not be aware of
physical addressing. This, in turn, justifies the usage of Cohort
further, as functionality is solely, but safely implemented via the
Cohort driver, without the need of wrappers. This also simplifies the
usage of multiple Cohort engines using established OS abstractions.

Our driver supports the following features:

- **MMU notifier for TLB flushes.** To maintain TLB coherence, Cohort’s ISA-native MMU is also flushed alongside cores’ TLBs. To enable this, MMU notifiers in Linux, normally used by platforms with IOMMUs, unified virtual memory, or hypervisors. The Cohort kernel driver registers its TLB flush function with an MMU notifier for processes at queue registration.

- **Page fault resolution via interrupt.** As noted in Section 4.2.4, when the Cohort MMU sees a page fault, it triggers an inter-
rupt and invokes a handler registered by the Cohort kernel
driver.

- **Registering queues with Cohort.** When a user mode applica-
tion calls cohort_register, the driver virtually maps the Cohort engine’s configuration register bank and registers the MMU notifier and page fault handler. Then, the driver writes the queue descriptor information to Cohort. After this simple syscall, the requested Cohort engine is ready to use.

- **Unregistering queues from Cohort.** Calling cohort_unregister deallocates and unmaps the prior re-

resources.

The driver is first probed at kernel boot time to request interrupts and MMU notifier resources. When applications launch, they call cohort_register to register queues. The applications then push and pop data to and from Cohort via the registered queues. User space applications invoke cohort_unregister at exit to clear the allocated memory and MMU notifier service.

4.5 Cohort Capabilities and Extensions

**Inter-thread and Inter-process Communication.** Cohort is able to accelerate inter process communication with the same queue abstraction, just as two software threads can communicate via a
shared-memory queue. Such communication is commonly done by allocating the queue once and sharing its memory across two processes. This enables Cohort to receive input data from one process and produce output to another using the same push and pop methods as within a single process.

**Accelerator Chaining.** As more workloads necessitate hardware acceleration, accelerator chaining is a perfect way to accelerate complex computations. Cohort supports chaining together multiple accelerators using SPSC queues, bringing acceleration to more diverse workloads with minimal overhead and great flexibility. As an SoC designer moves to add new accelerators to their system, they can take existing decoupled software threads and directly replace them with Cohort-enhanced accelerators. The queues that are already used to communicate with the software thread are maintained, but registered with the relevant Cohort engine instead. This accelerator chaining capability also enables runtime reconfiguration of accelerator chains based on applications’ needs.

**Multi Producer/Consumer.** Cohort sticks with the SPSC model, essentially a restricted dataflow where queues are not split or merged. Enabling queues supporting multiple producers or multiple consumers would provide value for a broader set of multithreaded use cases and for multiple accelerators to process data to/from a single queue. Generally these queues require atomic memory operations to guarantee correct operation, for which there is somewhat less standardisation of queue organisation. As a result, we leave support for these queues and design of their queue descriptors to future work.

5 EVALUATION METHODOLOGY

This section details our methodology, accelerator configurations, and benchmarks. We integrate Cohort into OpenPiton [8–10] and boot Linux (v5.6-rc4) on a Xilinx Alveo U200 FPGA running at 100 MHz. We use a four tile design with two 64-bit 6-stage Ariane RISC-V RV64GC cores and two accelerators (shown in Figure 2). We use the default OpenPiton configuration of 8KiB L1D, 16KiB L1I, 8KiB L1.5, and 64KiB 4-way L2 caches. The Cohort TLB has 16 entries and the producer and consumer endpoint accelerator interfaces are 64-bit wide. We use a minimal user-mode driver for our experiments.

5.1 Baselines

For our baselines, we repurposed a MAPLE decoupling unit [61] to connect with accelerators. In its original setting, MAPLE is an out-of-core, highly memory parallel load-store unit. It is designed to enable efficient data movement in manycores with slim in-order cores. Its particular focus is decoupled access-execute multithreading and prefetching models for applications with many indirect memory accesses, like graph processing. We modified MAPLE to instead host accelerators and provide MMIO-based and coherent DMA-based invocation (two common approaches). Performance counter data comes from each Cohort Engine, Ariane core or MAPLE unit.

**MMIO Baseline.** For some workloads, an MMIO-based invocation can make sense, as queue contents are not saved in cache and are hence disconnected from the coherence system. However, as noted in Section 3.2, MMIO often requires non-speculative round-trips from core to accelerator for each data word, a fact that was highlighted in the original presentation of MAPLE [61]. We use MAPLE’s MMIO queue interface to provide data to and collect data from the accelerator with no coherence effects.

**Coherent DMA Baseline.** DMA enables bulk data movement between memory and the device without host interference. The DMA must be enabled and programmed by a core, and accelerators must still wait for sufficient data to operate. We use MAPLE’s coherent LLC data prefetching feature to provide data to the accelerator and use the P-Mesh TRI [9] to coherently store results. Note that MAPLE’s coherent DMA is more software efficient than typical DMA approaches as it uses a RISC-V MMU rather than requiring an IOMMU.

5.2 Accelerators

We adopt SHA-256, AES-128, and H264 accelerators for their streaming yet computationally-intensive nature. Their continuous data flow makes good use of the Cohort interface, while their computational complexity justifies acceleration. Figure 2 shows our Cohort SoC with two accelerators connected: SHA-256 and AES.

**SHA-256.** The first accelerator is an open-source SHA-256 cryptographic core [24], which is usable from Linux. Our accelerator accepts input in 512 bit data blocks and hashes them to a 256 bit irreversible hash digest [27]. The accelerator accepts incoming 64 bit data blocks from the consumer endpoint and uses a ratchet to build a 512 bit input block. The hash digest is fed back using a ratchet to the producer unit in four chunks of 64 bits.

**AES-128.** Advanced Encryption Standard (AES) is a symmetric encryption algorithm [28]. In our prototype, we connected an open-source accelerator for AES encryption [2] which generates a ciphertext in blocks of 128 bits with a key of the same width. With Cohort we added a ratchet to consume 128 bits of data from 64 bit chunks from the consumer endpoint and the reverse for sending encrypted ciphertext to the producer endpoint. The encryption key is passed via a coherent CSR struct as described in Section 4.3.

**H264 Encoder.** H264 is a video compression standard widely used in the video industry. It provides good video quality at significantly reduced bitrates. We integrated an H264 encoder from Zexia [79] (using Context-adaptive variable-length coding or CAVLC) with the Cohort engine and confirmed its correct operation. This example also illustrates how Cohort can handle variable input size. The existing instance of the accelerator that we adopted accepts the number of frames at the start of its input to enable variable input length. The integration code includes a ratchet to prepare each frame for H264 in a similar manner to AES and SHA.

5.3 Benchmarks

To characterise the Cohort Engine, we run benchmark applications and test the entire system. Benchmark parameters are illustrated in Table 2.

Here batch size refers to an optimisation that updates the read and write pointers in batches instead of incrementally. This helps to reduce the coherency traffic in the system and improve performance.
Table 2: Benchmark Tuning Parameters.

<table>
<thead>
<tr>
<th>Accelerators of Interest</th>
<th>AES, SHA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication Modes</td>
<td>Cohort, MMIO, DMA</td>
</tr>
<tr>
<td>Min/Max Queue Size</td>
<td>64/8192 elements</td>
</tr>
<tr>
<td>Min/Max Batching Factor</td>
<td>2/64 elements</td>
</tr>
<tr>
<td>Baseline DMA Granularity</td>
<td>256 Bytes</td>
</tr>
</tbody>
</table>

This is an optimisation that would be applied to multithreaded software but is also exploitable by Cohort, demonstrating Software-Oriented Acceleration in action. Below, we briefly describe our benchmark applications.

**Benchmark Implementation in Cohort.** Benchmarks in Cohort initialise the SPSC queues then push and pop the data in sequence. To hash 1 block of text we push 64 bits of data 8 times and fetch the corresponding hash with 4 pops. For AES, there are 2 pushes and 2 pops. As mentioned earlier, we encapsulate these movements into batches and run applications until queue size is reached.

**Benchmark Implementation in Baselines.** Benchmarks for our baselines are logically similar to Cohort’s, but more complicated in their implementation. With MMIO, the core cannot achieve memory-level parallelism and so must receive the accelerator’s output word by word before passing the next input word. This affects performance versus Cohort which can interleave input/output and expose MLP within batches. With Coherent DMA, special API functions (also containing MMIO writes) are called for each data block to be copied to/from the modified MAPLE unit, which matches common DMA programming mechanisms.

## 6 RESULTS

The goal of our evaluation is to demonstrate Cohort in a realistic heterogeneous SoC with multicore Linux support while providing a combination of programmability and performance improvements over the state-of-the-art (SOTA).

### 6.1 Latency Analysis

Figure 8 illustrates the latency to queue size relationship for our SHA benchmark running with different batch sizes and communication APIs. "Cohort batch=N" indicates the Cohort application optimisation of batching its write/read pointer updates to occur only after N elements have been copied into the queue. The same goes for AES in Figure 9.

Across the board, Cohort outperforms the MMIO and Coherent DMA baselines for the SHA application. Over all queue sizes tested, Cohort performs the best with larger batches and the batching optimisation works robustly. Cohort starts at a batch size of 8 elements to reflect one SHA input of 512 bits. Table 3 shows the speedup for Cohort AES and SHA with 64 element batches over the baselines.

<table>
<thead>
<tr>
<th>Queue size</th>
<th>4</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES Speedup</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vs MMIO</td>
<td>2.32</td>
<td>2.07</td>
<td>1.74</td>
<td>1.75</td>
<td>1.48</td>
<td>2.03</td>
<td>2.30</td>
<td>1.86</td>
</tr>
<tr>
<td>Vs DMA</td>
<td>2.32</td>
<td>2.07</td>
<td>1.74</td>
<td>1.75</td>
<td>1.48</td>
<td>2.03</td>
<td>2.30</td>
<td>1.86</td>
</tr>
</tbody>
</table>

For AES, Cohort improves over the baselines as batch size increases. Batch sizes larger than 16 elements always perform equal or better than both MMIO and DMA baselines. Table 3 shows a range in speedup from Cohort on AES versus MMIO and DMA ranging from 1.83× to 2.07× for MMIO and from 1.69× to 2.03× for DMA. For AES, batching increases performance by 5.30× to 8.10×.

### 6.2 IPC Analysis

Beyond seeing an overall improvement in latency, we argued that Cohort would make more efficient use of the core as it provides data to and reads data from accelerators. Figure 10 and Figure 11
We perform FPGA implementation with Xilinx Vivado 2022.1 and report post-synthesis resource utilisation in Table 4. The empty Cohort engine comprises around 10% of the LUTs and 20% of the registers of a Cohort tile, or less than 4% of the LUTs and 10% of the registers of an Ariane tile. A tile with an empty Cohort Engine is about 39% and 46.6% of the Ariane tile by LUTs and registers (both tiles feature OpenPiton’s NoC routers and L1.5 and L2 caches). Despite this small area, Cohort packs a punch in terms of the functionality and features it provides.

The Cohort engine has roughly 68% the LUTs and 45% the registers of the AES accelerator while consuming no BRAM, compared to the large memory needed for AES which would consume considerable SRAM in an ASIC process (note that the AES BRAM is larger than that of an Ariane tile with its previously stated cache configuration totalling around 100KiB). The Cohort engine consumes roughly 27% more LUTs and 57% more registers than the relatively small SHA accelerator. Compared to the H264 encoder, the empty Cohort engine consumes around 37% of the LUTs and 71% of the registers. H264 consumes 4 BRAM slices (tens of kilobytes of memory) and unlike Cohort or the other accelerators, it also consumes 6 DSP slices. Full tiles including Cohort plus AES, SHA, or H264 are significantly smaller than the size of an Ariane tile.

The MMU itself is area efficient, consuming 1081 LUTs, 1206 registers, and no BRAM. Of that, the TLB makes up 911 LUTs and 1029 registers, while the page table walker makes up 168 LUTs and 113 registers. This small MMU leads to savings in memory (no need for separate I/O page tables), runtime (less page table management in software), and software/OS porting. While our demonstrated accelerators are relatively small, the same MMU and Cohort engine are usable with larger accelerators.

## 7 RELATED WORK

As Moore’s Law has slowed and Dennard Scaling has ended, system designs are increasingly exploiting heterogeneous parallelism and domain-specific accelerators to scale performance at acceptable power [23, 64]. Using accelerators often relies on the programmer configuring them manually [32, 63] or through a domain specific language, for the class of applications that are being executed [44, 82].

Decoupling of data-access and compute was originally proposed by J. Smith [69] to mitigate latency of data-supply to processor cores, by having a core accessing the data and feeding it to another core, which handles the computation. After that, several hardware implementations have been proposed, where data communication
occurs through architectural queues [33, 51, 78]. These works aim to hide memory latency as a simpler alternative to superscalar processors. DeSC [37, 38] repurposed decoupling to supply computation-exclusive units like accelerators. DeSC’s programming model offers a producer-consumer relationship between heterogeneous processing units. However, DeSC uses architectural queues that are exposed to software through new ISA instructions, which requires software changes. Furthermore, there is a strict one-to-one relation between supply and compute elements, determined at tapeout time.

Data streaming proposals also have this restriction. MAD [41] and HWPEs [19, 22] have a data-access engine optimised for dataflow computation, which is integrated with cores or accelerators to perform memory-access portion of programs.

Cohort offers similar producer-consumer relationships without restricting them in hardware, and making them transparent to the ISA. In Cohort, accelerators can be configured at runtime to be consuming from or producing to any core or accelerator on-chip. This creates a more rich space of possible heterogeneous communication patterns.

Although MAPLE [61] offers a flexible decoupling mechanism where cores communicate via a network-connected memory-fetching engine, it still requires software changes with explicit produce and consume operations. Moreover, MAPLE’s communication scheme resorts to side-effectful, non-idempotent memory-mapped I/O instructions, which sets a significant minimal latency threshold between devices. Cohort communicates using a simple software queue, which enables greater speculation and parallelism of produce and consume operations.

Loosely coupled and tightly coupled accelerators. Accelerators come in different flavors and sizes, but we classify them regarding their hardware integration into tightly-coupled and loosely-coupled [20]. Often, the classification is in terms of where the accelerator sits in the architecture, namely, close to the CPU or not.

In the tightly-coupled category, we can find accelerators as a functional (computing) unit physically connected to the CPU pipeline (e.g. FPU, or inline accelerators [73]), and accelerators that share the private cache with the CPU (e.g. RocketChip [4], Hwacha [48], and Gemmini [31]). In the loosely-coupled category, we find accelerators placed on the main chip interconnect [14, 39, 59], or placed on a separate chip which communicates to the chip interconnect through an I/O interface (e.g. PCIe).

Comparison with interconnects. CXL and UCIe are emerging interconnect specifications that tackle host-to-host and chiplet-to-chiplet connections respectively. Although CXL enables inter-host coherent communication at a low latency and high throughput, it has a large area and power overhead associated with underlying PCIe protocol stack. By contrast, our implementation of Cohort tackles efficient coherent communication within a single SoC. Cohort would also be implementable on top of the CXL coherence system with the correct semantics. UCIe takes a layered approach to accommodate challenges in die-to-die communication for emerging chiplets. Currently the protocol layer supports four standards: PCIe 6.0, CXL3.0, CXL2.0 and raw data streaming. We envision Cohort would be implemented on top of emerging coherence protocols in new UCIe protocol layers, or atop CXL as previously described. However, the control and physical layers of UCIe are out of scope for this paper.

Area-wise, in a state of the art design such as Intel Alder Lake fabricated with 7nm process, an 8 lane PCIe 5 PHY takes 2.44mm² [50]. By contrast, a high performance CPU core on the same chip with L1 cache but without L2 cache or power gating logic takes around 5.37mm², only a little over twice the PCIe PHY block size. Note that this does not take into account the area of the PCIe controller which would also be required.

System-level considerations. The advantage of loosely-coupled accelerators is that they do not modify the CPU or the ISA. They can be integrated in a relatively plug-and-play manner, through the interconnect, overcoming the verification overhead of tightly-coupled integration. This is aligned with trends in fast chip prototyping [1, 52, 77], where an SoC is made of reusable, third-party IP blocks. However, tightly-coupled accelerators have the advantage of having the CPU dealing with VM translation. Loosely-coupled, driver-triggered accelerators like those of ESP [52] have the strategy to pre-fill a specialised TLB and pin memory so that no page-faults can occur, while NVDLA [59] uses a reserved, non-cacheable memory region, that the OS cannot use, and only the accelerator and its device driver have access to it. Cohort enjoys the advantages of loosely-coupled accelerators while supporting VM in a more flexible manner by incorporating a fully-capable MMU that handles page faults and TLB shootdowns.

Programmability. While tightly-coupled accelerators are triggered by ISA instructions, and thus the result goes to the register file [4, 73], loosely-coupled ones often rely on interrupts or spin-polling for the CPU to know that they finished. Several works have enhanced the I/O software stacks, by reducing interrupt overheads [42, 67], or combining them with spin-polling as a hybrid notification mechanism [25]. Recently, HyperPlane [57] proposed a hardware mechanism to accelerate these notifications. However, these techniques focus on I/O accelerators, while Cohort leverages the coherence protocol to get notified that another processing element has produced or consumed data, thus also supporting on-chip accelerators. Cong et al. [17, 18] proposed an allocation protocol to avoid OS overhead in scheduling tasks to on-chip accelerators.

### Table 4: FPGA Synthesis results for resource utilisation: Cohort with or compared to Ariane, MAPLE, and accelerators

<table>
<thead>
<tr>
<th>Resource Type</th>
<th>Ariane Tile</th>
<th>Empty Cohort Tile</th>
<th>Empty Cohort Engine</th>
<th>Cohort + AES</th>
<th>Cohort + SHA</th>
<th>MAPLE + AES + SHA</th>
<th>AES Only</th>
<th>SHA Only</th>
<th>H264 Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUTs</td>
<td>67083</td>
<td>26390</td>
<td>2594</td>
<td>6679</td>
<td>4524</td>
<td>21066</td>
<td>3837</td>
<td>2041</td>
<td>6851</td>
</tr>
<tr>
<td>Registers</td>
<td>39879</td>
<td>18591</td>
<td>3799</td>
<td>12176</td>
<td>6064</td>
<td>28276</td>
<td>8531</td>
<td>2420</td>
<td>5341</td>
</tr>
<tr>
<td>BRAM</td>
<td>41.5</td>
<td>9.5</td>
<td>0</td>
<td>47.5</td>
<td>0</td>
<td>47.5</td>
<td>47.5</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

ASPLOS ’23, March 25–29, 2023, Vancouver, BC, Canada

T. Wei, N. Turtayeva, M. Orenes-Vera, O. Lonkar, J. Balkind
However, this still requires large workloads to be efficient, while Cohort is also suited for finer-grain tasks.

M3 [6], M3X [5], Rackscale Microkernel [40], Solros [56] bake hardware semantics into OS mechanisms for tight integration. All target simple, efficient, and scalable heterogeneous communication, but implement brand new (micro)kernels (and new DTU hardware [6, 40]). Cohort exploits existing OS and multi-threaded application best practices, via well-known queue semantics and enables accelerators and cores to share queues in coherent memory for communication. We improve performance over the baselines with low cost and a more intuitive API. Similarly to our MMIO baseline, DTU queues are either kept in (limited-size) scratchpads and modified by MMIO or, similarly to our DMA baseline, are kept in DRAM with pointers updated by MMIO and NoC messages. Both DTU mechanisms require software changes, unlike Cohort.

**Acceleration standards** A number of accelerator standardisation efforts are taking place, particularly to standardise on programming models for GPGPUs and similar programmable accelerators. Vulkan [35] and SPIR-V [34], with features like pipelined barriers, expose deeper hardware details in a traditional, hardware-oriented fashion, implying a deep asymmetry between the accelerators and cores. Cohort on the other hand effectively embeds common software semantics into the accelerator integration environment (note: not the accelerator itself). Cohort-enabled accelerators can be decoupled from memory operations and act as peers to cores in a software-oriented manner. By leveraging software semantic awareness to provide a high performance API that is implementation agnostic, we see Cohort as additional, rather than a competitor to these solutions. For accelerators designed or programmed with Vulkan/SPR-V, any communication of a SBIO nature (whether through explicit SPSC queues or other data movement behaviours) could target a specialised Cohort engine instead of the usual LSU for a potential uplift in performance.

**Queue Libraries and Language Support** Beyond adoption for Boost, we are investigating using Cohort with Unix pipes. Of further interest for high-performance, asynchronous acceleration use cases is the Linux io_uring subsystem [68]. The io_uring API has enabled a variety of new high-performance I/O use cases in Linux, particularly for networking and file I/O. Integrating Cohort with io_uring would enable a rich runtime for managing accelerators. More intriguingly, Cohort accelerators could also use io_uring to request services from the kernel via their native queue interfaces.

With simple runtime support (comparable to the library support we added for Boost), languages could automatically retarget their queues to make use of Cohort. We leave automatic identification of queues to future work.

**8 CONCLUSION**

In this paper we argue for the importance of software-oriented acceleration, bridging the gap between hardware-centric SoCs and software programmability. We propose Software-Oriented Acceleration (SOA), an elegant and pragmatic solution for orchestrating the ever-increasing heterogeneity of modern SoCs. To hack the idea up, we identify Queue Coherence semantics for communication between accelerators and present the Cohort engine and API support to enhance common producer-consumer software queues in a full-system heterogeneous environment. This concept makes SoCs more easily programmable while remaining performant. Cohort’s implementation spans the application to architecture stack and shows powerful performance improvements ranging from 1.83× to 8.38× over MMIO, and from 1.69× to 11.24× for DMA baselines. The hardware and software of Cohort, plus a subset of the accelerators, are available open-source at https://github.com/cohort-project.

**ACKNOWLEDGMENTS**

This work was partially funded by a grant from Western Digital Technologies, Inc. We thank Daniel Jiménez Mature, Katie Lim, Guillem López-Paradis, Brian Li, Arjun Vinod, and Guy Wilks for providing code and other useful assistance needed to realise Cohort.

**REFERENCES**


Jeffrey Stuecheli, Bart Blaner, CR Johns, and MS Siegel. 2015. CAPI: A coherent


Received 2022-10-20; accepted 2023-01-19