Hardware Acceleration with Zero-Copy Memory Management for Heterogeneous Computing

Oren Bell
Comp. Sci. and Eng. Dept.
Washington University in St Louis
St Louis, MO, USA
oren.bell@wustl.edu

Chris Gill
Comp. Sci. and Eng. Dept.
Washington University in St Louis
St Louis, MO, USA
cdgill@wustl.edu

Xuan Zhang
Electrical and Systems Engineering Dept.
Washington University in St Louis
St Louis, MO, USA
xuan.zhang@wustl.edu

***Abstract*—The ROS2 software framework is increasingly prevalent in component-based applications for robots and other autonomous systems. Recently added ROS2 features to support zero-copy semantics may significantly reduce latency and latency variation when passing data from one component to another.**

Additionally, there is a growing trend of developing autonomous robotic systems on heterogeneous computing platforms to exploit hardware acceleration. However, support for portable and reusable zero-copy semantics on heterogeneous compute systems is limited. Such systems thus must either use low-level techniques to manage memory operations directly, which may be tedious and error-prone, or they may not adequately address substantial memory overheads that can arise from repeatedly copying messages and data into and out of device memory associated with GPUs and FPGAs.

Towards addressing that limitation of the current state of the art, this paper introduces *Hazcat*, a new zero-copy framework that automatically performs device memory operations when needed, and avoids copying and other costly operations otherwise. Hazcat is integrated specifically with ROS2 but is also designed for portability to other component-based software frameworks.

***Index Terms*—ROS2, zero-copy, GPU, FPGA, multi-core, hardware acceleration, heterogeneous computing, robotics**

I. Introduction

Real-time embedded systems support mission-critical and safety-critical applications ranging from automated and connected driving [41] to real-time hybrid simulation in earthquake engineering experiments [9]. In recent years, the combination of component-based modularity and customizable thread and memory management in ROS2 has made it especially attractive for developing real-time embedded robotics applications [5], including the TurtleBot 4 open source robotics platform for education and research [2], the Hamster robust micro Autonomous Unmanned Ground Vehicle [1], and the xArm series of robotic manipulators [3].

Such systems often have timing constraints that require careful management of hardware and software overheads and other sources of timing variation as well as scheduling to ensure real-time tasks meet their deadlines. Even atop traditional multi-core platforms, memory operations may introduce significant overheads, e.g., when the number of cores used exceeds a single chip socket so that the cost of memory operations between threads on cores in different chip sockets limits how many cores can be exploited at fine-grained timescales [9], or when frequent communication among components causes the aggregate cost of memory operations to approach other less frequent but more expensive overheads (e.g., thread context switches). Thus, reducing the frequency of memory operations or making them more efficient [13] is an important issue for these systems.

For heterogeneous computing platforms that support hardware acceleration of computations (e.g., Nvidia Jetson and Xilinx Kria) latency-aware memory management is even more salient. For example, it has been shown [31] [4] that GPUs can be a cause of non-deterministic behavior in real-time systems. As one particularly egregious example, memory operations on Nvidia GPUs, particularly *cudaFree*, may cause implicit device wide synchronization operations [43]. With or without hardware acceleration, a multi-component application can reduce its memory bandwidth by leveraging shared memory structures and passing references to data instead of copying the data itself [19], whenever possible. Such "zero-copy" approaches are intuitive in principle, but must enforce ownership of data to avoid race conditions, which may raise further engineering challenges in practice.

Though the ROS2 application programming interface (API) now supports zero-copy semantics atop multi-cores, support for portable and reusable zero-copy semantics on heterogeneous computing platforms that autonomous systems may exploit for hardware acceleration is limited: currently they must either use low-level techniques to manage memory operations directly, which may be tedious and error-prone, or they may encounter potentially substantial memory overheads that may arise from repeatedly copying messages and data into, within, and out of different device memories associated with multi-core processors, GPUs, and FPGAs.

Towards addressing that limitation of the current state of the art, we introduce Hazcat, a new zero-copy framework that automatically performs device memory operations when needed, and avoids copying and other costly operations otherwise. Hazcat is integrated specifically with ROS2 but is also designed for portability to other component-based software frameworks. Hazcat assumes full responsibility for all memory operations between components, and eliminates memory copies when consecutive components are on the same computational device (i.e., within the same device memory).

The contributions of this paper include: (1) a system model for zero-copy semantics between and within device memory for multi-cores, GPUs, and FPGAs in section III; (2) requirements for extending Hazcat support to new hardware in section IV; and (3) empirical evaluations of the Hazcat framework in comparison to other approaches in section V.

II. Background and Related Work

Memory allocation on uniprocessors and multi-core processors is fairly mature [21] [24] [12] [39] [26] [40] [34] [28] [18], but memory allocation on heterogeneous computing systems is newer and less studied [14] [32]. A recent example by Nvidia recycles memory with the RAPIDS memory manager [13], thus avoiding costly and non-deterministic memory deallocation: as Section I mentioned, calls to *cudaFree* may cause implicit device-wide synchronization on a GPU.

A survey on memory management of heterogeneous computing systems was published by Hazarika et al [14]. There has been extensive research into using hardware acceleration in robotics applications [25] [30] [37] [33] [7] [22] [35] [23] [27] [36] [29]. However, most of this research assumes developers have detailed knowledge of the use of FPGAs/GPUs and tends to target a particular product rather than discussing hardware acceleration more broadly.

ROS2: The ROS2 software framework is often considered a de facto standard for robotics. Subtasks within a robotics application are separated into components called *nodes*, each communicating with others via a publish/subscribe communication model: when a node sends data it *publishes* a *message* on a *topic*. Interested nodes *subscribe* to that topic to receive a copy of the message. Recent research on ROS2 [5] has focused on processing chains. When one node publishes, this will naturally trigger the execution of its subscribers, which may then publish and trigger execution of their subscribers, and so on. This results in a natural chain of execution, which can be modelled as a tree. The responsiveness of an application can be quantified by looking at the end-to-end latency of these processing chains. A response-time analysis for ROS2 was developed in [5] and later modified in [38], both of which include inter-component interference in their analysis.

The lowest layer of the ROS2 stack, called the ROS Middleware Abstraction Interface (RMW), serves as a wrapper for a networking framework, such as FastDDS [8] or CycloneDDS [10], which directly facilitates this communication. Recently developments in ROS2 have added API calls to the RMW specification that enable zero-copy through the use of borrow/return semantics. Before a publisher wishes to send data, it must *borrow* a message which it can then populate before publishing. Subscribers receive an immutable pointer to the same data, and when all subscribers have returned their pointers, the memory for the message is freed or recycled, depending on the implementation.

In contrast to its predecessor (ROS) ROS2 is designed to cater to real-time concerns. ROS2 has recently added 5 new calls to its RMW specification to support zero-copy semantics: *rmw_borrow_loaned_message*, *rmw_return_loaned_message*, *rmw_publish_loaned_message*, *rmw_take_loaned_message*, and *rmw_release_loaned_message*. Although these new additions can help to reduce nondeterminism caused by excessive memory operations, we posit that it doesn't go far enough. Many robotics applications leverage hardware acceleration for performance reasons, but ROS2 zero-copy features are as yet unaware of device memory boundaries, so components leveraging hardware acceleration must assume responsibility for copying data into, within, and out of device memory which may lose the benefits of zero copy, and may introduce more nondeterminism.

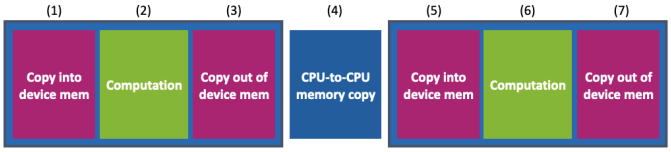


Fig. 1: Sequence with two hardware accelerated components

Hardware Accelerated Workloads: For components sending data to each other, we want to minimize the latency that is due to 3 types of operations, depicted in Figure 1: intra-component memory copies (magenta), compute operations (green), and inter-component memory copies (blue). With hardware acceleration, whether using GPUs or FPGAs, there are 3 steps: copy to device memory, run the acceleration kernel, and copy out of device memory [42] [6]. These 3 operations happen within a component. We aim to remove intra-component memory copies entirely by having middleware assume responsibility for device memory operations. Inter-component memory copies also may be eliminated when conditions are right for zero-copy to occur, that is, when two consecutive nodes operate within the same memory domain.

Zero Copy in ROS2: A popular implementation of zero-copy semantics for ROS2 is in the Iceoryx project [11], which provides a third-party daemon that manages a shared memory pool. Publishers and subscribers wishing to make use of zero copy are required to use borrow and return semantics to access memory chunks [20] as follows: (1) publishers and subscribers register with the Iceoryx daemon, specifying their topics of interest; (2) a publisher makes a borrow request and the daemon provides the address of the next available chunk large enough to fit the message; (3) the publisher populates the chunk with data and (4) publishes the chunk, relinquishing ownership; (5) Iceoryx notifies all interested subscribers of available data after which (repeatedly): (6) a subscriber awakes and borrows the chunk, receiving a pointer to newly available data (7) processes the data and (8) returns their reference to the chunk; (9) the Iceoryx daemon marks a chunk as available for reuse once all subscribers have processed it.

Multi-core performance numbers for Iceoryx show minimal overhead [19], but Iceoryx does not support zero-copy semantics for other devices' memory: a developer wishing to leverage hardware acceleration must manually copy messages into and out of device memory. Additionally, the memory pools are pre-allocated by a third-party daemon, which the

end-developer must configure to optimize memory usage. Under the default settings, memory chunks are likely to be oversized, and may not be plentiful enough to accommodate the message backlog of a worst-case scenario. Iceoryx also only supports zero-copy of fixed-size plain-old-data types. Any messages which require dynamic allocation of memory may be partially allocated on the heap, potentially resulting in invalid pointers and segmentation faults when sharing messages between processes. This is a limitation our implementation also shares, a fix for which would likely require updates to the ROS2 API. Another zero-copy mechanism is available higher in the ROS2 software stack [16], but it only supports intra-process zero-copy semantics. In the interest of generality, we do not assume two components share an address space so we do not consider that solution further in this paper.

III. System Model

We use the term *memory domain* to refer to any area of memory that is tied to a particular set of processors, and is guaranteed to be readable from processors in that set: host memory is a separate domain from GPU memory, both are separate domains from FPGA memory, and data in one domain must be copied to be accessible in another domain.

We specify special *heterogeneity aware* (HA) allocators that manage memory for a particular memory domain, according to a domain-agnostic interface. They have functionality to access each other's memory, and perform copy operations as needed when memory is not in an appropriate domain. We also specify a shared message queue, analogous to a ROS2 topic, which stores references to messages allocated by an HA allocator. Publishers put messages into the queue for later consumption by subscribers. When a subscriber expresses a preference to receive a message in a different memory domain, a copy is made on the fly in the correct memory domain and a token to the duplicate copy is also stored in the message queue. This is the worst-case scenario: when data moves across memory domains, a copy operation must occur. Each topic gets a unique message queue. Unlike Iceoryx, our system handles zero-copy in a completely decentralized manner, and developers are allowed to apply custom allocation strategies instead of being forced to use Iceoryx's static ring buffer.

Hazcat is not truly zero-copy in all circumstances. If an application is only partially hardware accelerated, memory copies will be required between portions operating in host memory and portions operating in device memory. However, if consecutive nodes operate in the same memory domain, we offer the benefits of zero-copy regardless of what that memory domain is, be it host, GPU, or (in the future) FPGA memory.

A. Heterogeneity Aware Allocator

Custom memory allocators are already used in real-time systems, e.g. to allocate memory statically, or to reuse previously allocated objects to avoid overheads of an operating system's default dynamic memory management mechanisms. Sophisticated features such as allocator-aware containers and polymorphic allocators [17] have been developed to allow flexible use of allocators, through which containers allocated with different strategies can be completely interoperable without subverting type-safety. A minimal allocator offers 2 functions: *allocate* and *deallocate*. Some allocators also provide object construction, but we omit this in our design: we assume that our allocators, while potentially managing peripheral device memory, will run on a CPU and should not attempt to dereference the device memory they allocate.

Managing multiple memory domains within our framework incurs stricter design requirements beyond supporting allocate and deallocate functions. We first detail a new extended interface, describe garbage collection features required whenever sharing memory, present the structure of the allocator and its memory pool, and show a way to ensure fidelity when reconstructing the allocator in other processes.

1) Allocator Interface: We propose a new allocator interface, shown in Listing 1, that adds functionality to convert arbitrary memory allocated by a separate allocator into a domain readable to the current allocator. This conversion may operate as a simple passthrough function if the two allocators operate in the same memory domain (this is our zero-copy condition), or it may perform the operations necessary to copy data from a separate domain.

Listing 1: Allocator interface for heterogeneous computing

```
class Alloc {
  static int domain;
  static void * create_alloc(...);

  static void * remap(Alloc * alloc);
  void unmap();

  int allocate(int size);
  void share(int offset);
  void deallocate(int offset);

  void * copy_from(const void* ptr, void* cpu_ptr, int size
     );
  void * copy_to(void* ptr, const void* cpu_ptr, int size);
  void * copy(const Alloc* a, void* dst, const void* src,
     int size);

  int shmem_id;
  int device_type;
  int device_number;
  int strategy;
};
```

Multiple allocators can be written for the same domain. We assume that any allocation can be accessed by any component in the same domain, provided any requisite memory mapping operations have been performed. The added functionality is six-fold: *remap* maps an allocator previously created in a different process into this process address space; *unmap* removes the allocator from this process address space; *share* increases a reference counter for a specific allocation; *copy_from* copies from this allocator to a pointer in CPU memory; *copy_to* copies to this allocator from CPU memory; and *copy* copies from an arbitrary allocator into this one, possibly going through host memory as an intermediary. Our allocators return an integer offset rather than a pointer: allocators are mapped at different places in each process's address space, so allocations

are measured relative to the beginning of the allocator and absolute pointers are recalculated in each process.

2) Deallocation and Reference Counting: Deallocation of shared memory may need to be performed by any thread: while a message is created by a publisher, one of its subscribers will likely be responsible for deallocation. So, for time-sensitive threads, deallocation must be a time-bounded operation, which limits the types of allocation strategies we can use. For now, we only provide a static ring buffer and intend to provide additional options in the future, but stipulate that all allocation strategies must offer O(1) deallocation. We assume that every memory domain is copyable into host memory and vice-versa. However, not every memory domain may be copyable into each other, so each allocator must specify *copy_from*, to copy memory from itself into host memory, and *copy_to*, to copy memory to itself from host memory. The *copy* function is often a wrapper for those two steps in sequence: copying from the source domain to host memory, and then from host memory to the target domain. However, depending on available hardware, developers may code special conditions that bypass host memory, e.g., for GPU-GPU copies.

An HA allocator must implement a reference counting strategy, so the *deallocate* function will not free an allocation until it is called more times than there have been calls to *share* the same allocation. Note this creates a race condition that is resolved by the message queue as discussed in Section III-C. When implementing new allocators for device memory, the allocator methods cannot access the memory they are allocating, so strategies like boundary tags [21] are not applicable.

Allocators provide additional information: *device_type* identifies the hardware device the allocator is managing memory for; for multiple instances of that *device_type*, *device_number* can distinguish them; and *strategy* identifies the allocator approach, such as ring buffers, TLSF [24], or best-fit [21]. Two allocators' *device_type* and *device_number* must be the same to take advantage of zero copy, with *device_type* and *strategy* used to perform function lookups when an allocator is mapped into a new process, based on its ID according to *System V shared memory* (a POSIX standard that creates unique system-wide IDs for different shared memory segments).

3) Allocator Structure: Each allocator comprises 3 contiguous memory mappings: a local portion, a shared portion visible across processes, and a pool of mapped device memory. The local portion stores function pointers as a way to emulate the convenience of object-oriented polymorphism, which we use so our message queue does not need to concern itself with type information. True polymorphism is not possible for objects in shared memory, due to uncertainty of the structure of some data types, which combined with inherent type-erasure that occurs during inter-process communication would prevent us from knowing the structure of an allocator created in another process. The goal is for different allocator implementations to have an identical structure in their first few bytes, so we use plain-old-data structures for our allocator design to guarantee this.

Since function pointers are not valid across process boundaries, the allocator portion that holds them is not a shared mapping. This leads to a design in which our allocator object straddles different memory mappings, the bulk of which is visible across processes, with the local partition only visible to the current process. Due to granularity requirements, a minimum of 4kB is allotted to the local partition, but only 56 bytes are needed for the 7 non-static function pointers. The remainder is unstructured space that may be used as appropriate for the convenience of the allocator implementation. The start of the allocator's shared portion contains required type information that is relevant when remapping it into a new process, using the new allocator interface described above in Listing 1. The rest of the shared mapping varies by implementation. After the shared mapping is the actual memory pool, often mapped device memory, which again varies by implementation. For dynamic allocators, it's useful to start with an arbitrarily large virtual memory pool unbacked by physical memory. This gives the allocator room to grow, while still guaranteeing it can be reconstructed in a different process.

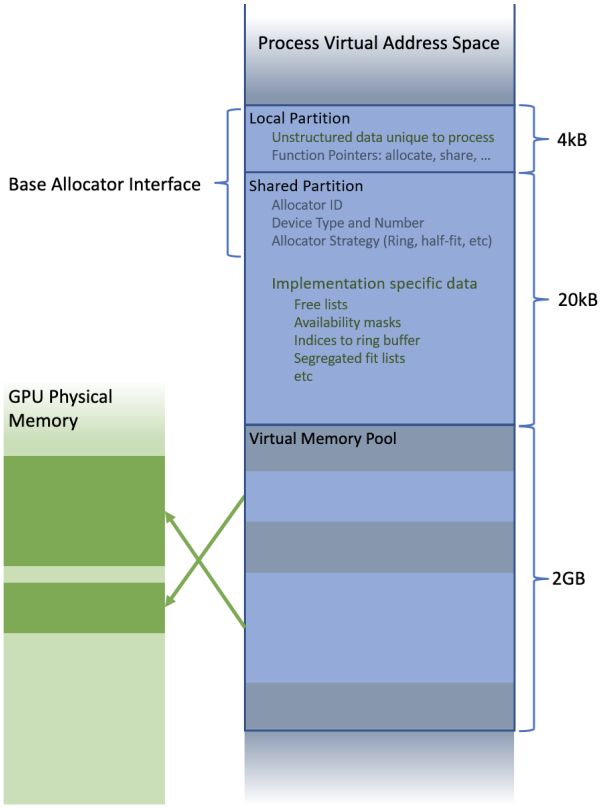


Fig. 2: Structure of allocator and partitions (not to scale)

4) Reconstructability of Allocators: Reconstructibility is essential: since allocators are not guaranteed to start at the same virtual memory address in different processes, memory allocations are expressed in terms of offsets from the start of their allocator and the relative structure of the allocator and its memory pool must be identical between processes. This is complicated by alignment restrictions on shared memory for

different devices and systems. The minimum size of the local partition is a page (typically 4kB). The granularity of shared memory varies (4kB on x64 systems, but 16kB on ARM), as does the required granularity of device memory.

The allocator is only reconstructible in a location that satisfies all these granularity requirements. The shared mapping must end at an address that's a multiple of the shared granularity. Similarly, the device mapping must start at an address that's a multiple of the device granularity. Since the shared mapping and the device mapping are contiguous, then the boundary between them must be at an address that is a multiple of the least-common-multiple of both their granularities: $lcm(m, n)$. We make an initial reservation of unmapped virtual memory at approximate location $\hat{x}$ of size $a+b+c+lcm(m, n)$. That is, the collective sizes of the local, shared, and device mappings, plus a buffer zone to align it correctly. A valid position satisfying all granularity constraints is guaranteed to exist in any memory range of this size.

If the boundary of shared and device memory partitions must be at a multiple of $\text{lcm}(m, n)$, then some modular arithmetic reveals that our reservation x must start at $\hat{x} + \text{lcm}(m, n) - ((x + a + b) \mod \text{lcm}(m, n))$. Any virtual memory before this point or past the tail can be released. Virtual memory within this range will be remapped to local, shared, and device memory in contiguous blocks guaranteed to begin at an address that satisfies their granularity requirements.

B. Message Queue

In a publish/subscribe application, topics can be modelled as a queue of messages. This is because execution of subscription processing may be delayed and a backlog of work may accumulate. In soft real-time systems, the queue can have a maximum length and stale messages may be dropped.

In our framework, these message queues exist as named shared memory files, with the name taken from the topic name. In ROS2, topics are named with strings. So a ROS2 topic named */perception/rear_camera* would be saved as a shared file */dev/shm/ros2_hazcat.perception.rear_camera*. When using Hazcat outside of ROS2, these message queues can be named arbitrarily. The structure and example contents of a message queue are shown in Table I. We describe the formatting of this design, and the methods for interacting with it, including registration and the *publish* / *take* calls.

1) Design of a Message Queue: Each message is stored in the queue in three parts: an allocator ID, the memory offset, and the message length. The allocator ID is used to look up an allocator, which may or may not be already mapped into the current process. These mapping operations typically occur once near the beginning of a program's run. After an allocator is mapped into the process's address space, the memory offset is added to the allocator's starting address to get the absolute address of the message. Each message may have multiple copies, one for each memory domain. The entire message queue is then structured as a collection of arrays: one per memory domain, plus an extra array for metadata.

The message length is a latent design decision for now. As mentioned in Section II, we only support fixed-size plain-old-data datatypes for messages, so specifying message length is redundant, as it could be inferred from the topic. In the future, however, we plan to implement support for zero-copy of dynamically sized messages. Each entry in the metadata array contains 96 bits, as follows. 32 bits serve as a subscriber counter to indicate if the message is still in use. 32 more bits serve as an availability map for up to the 32 supported memory domains, where a 0 indicates the message hasn't been copied to this domain, and a 1 indicates it is ready for zero-copy reading. The last 32 bits serve as locks for each memory domain to prevent redundant copy operations from colliding.

This allows up to 4 billion publishers and subscribers per topic, and 32 supported memory domains. The zero-th memory domain will always be host memory, but the rest are assigned in order of registration and have no relation to the *device_type* or *device_number* mentioned above in Section III-A.

In addition to the message queue itself, there is header data (not illustrated) to track the size of the message queue, the number of registered memory domains, and a global iterator pointing to the most recent message index. Subscribers also store their own iterator to the most recent message they haven't yet read. The sub counter tracks how many registered subscribers haven't read the affiliated message yet. Whenever a subscriber takes a message (covered in Section III-B4 on the *take* command), the sub count is decremented.

2) Publisher and Subscriber Registration: Before interacting with the message queue, a publisher or subscriber must register with Hazcat. A publisher or subscriber can only be affiliated with a single topic which they request during registration. The associated message queue will be mapped into the current process (or created from scratch). Pursuant to the subscriber's requested backlog, the message queue may be resized. Other processes will be informed of this when they attempt to fetch data and note the message queue's self-reported size does not match the size of their own mapping, at which point they remap the message queue with its larger capacity.[1]. We assume that all publishers and subscribers are registered during an application's initialization phase, so these resizing operations will not be a part of steady state operation.

When the application terminates, publishers and subscribers are also required to unregister, which is largely just decrementing an entity count. The last process to unregister the last of the publishers and subscribers for a message queue will also destroy the message queue.

When resizing message queues for topics, we consider the design philosophy behind ROS2 when developing Quality of Service (QoS) policies [15]. The general principle is that publishers offer a quality of service, and subscribers request a quality of service. Since we are not concerned with packet loss over a network, the ability of publishers to retain a message

[1]In practice, this resizing would seldom occur, due to the shared memory object being page aligned. A 4kB page is enough to hold tokens for well over 60 messages across 4 different memory domains. Few applications, would require more than this.

Metadata			Mem Domain 1			Mem Domain 2		
Sub Count	Avail Bits	Locks	Alloc ID	Offset	Size	Alloc ID	Offset	Size
0	0	0	∼	∼	∼	∼	∼	∼
1	b11	b10	0x0000 0014	0x0000 D0F8	0x0001 4100	0x0000 001A	0x0000 04D0	0x0001 4100
3	b10	b00	0x0000 0015	0x0021 0A00	0x0000 0B00	∼	∼	∼
3	b10	b11	0x0000 0014	0x0001 21F8	0x0001 4100	∼	∼	∼
...			...			...		

TABLE I: Example Message Queue. Each column is 32 bits. Each row represents a message.

history for resending is unnecessary. Thus, only the requests of the subscribers are relevant and the message queue should be the largest depth of all interested subscribers. Since shared memory objects must be page aligned, the minimum footprint is still large enough to accommodate over 60 backlogged messages across 4 memory domains. 4 arrays of 96 bit entries, plus another metadata array of 96 bit entries, makes 60 bytes per entry. A 4kB page then equates to 68 entries.

3) Publish command: The *publish* command is called by a publisher with a message to publish. First, it atomically fetches and increments the index of the next available row. It then secures a lock on the entire row, which ensures that any subsequent attempt to modify that row will block until the current call finishes.

If there are any remaining messages in that row (due to wrap around of the message queue) they are deallocated. This dropping of messages will only occur if there are best-effort components in the system, which are assumed to be tardy and unaffected by the missed messages. Any hard real-time tasks have to consider this deallocation as a source of interference when computing their response time. In a well-designed application with only hard real-time components, the message queue would never overflow, as an accumulated backlog exceeding the quality-of-service setting is by definition a system failure. The alloc id, offset, and size fields illustrated in Table I are updated accordingly and then the write lock is removed. Finally, a signal is sent on the associated FIFO to inform other processes that a message is available.

4) Take command: The *take* command is called by a subscriber after being notified that a message was available (a convenience, not a required step). First, the row for the oldest unread message is found. If the subscriber is up-to-date and no new messages are available, the call returns NULL. After inspecting the availability bitmask for the row, if the message is available in the subscriber's preferred domain, it will *share* the message and then prepare to return it. If the message isn't available in the message's preferred domain, an available copy is identified and the entry for that is fetched. Then the subscriber's allocator allocates a new message and performs a copy operation as shown in Listing 2.

This new message copy is also stored in the row with the original, the availability mask is updated accordingly, and the allocator ID and message offset are prepared to be returned. Whether or not a copy occured, one last check is made. If this subscriber is the last to access this message, the message queue will release its references to all copies of this message across all domains, potentially deallocating some, if no other subscribers hold a reference. Messages with a non-zero reference count remain available to access for any subscriber that is already running. When they finish, they also decrement their reference count and the last to finish deallocates the copy in their particular domain. This does mean that subscribers frequently use messages that are no longer tracked by the message queue. We do this because tracking message ownership in the message queue requires locking a message reference as read-only until all subscriptions return. These locks would provide an opportunity for best-effort subscriptions to indefinitely block a real-time publisher trying to submit a message, which is unacceptable, so we require our HA allocators to implement reference counting themselves.

Listing 2: Copy a message into the preferred domain when zero-copy conditions are not met

```
alloc = lookup(sub.alloc_id)
src_alloc = lookup(entry.alloc_id)
msg = src_alloc + entry.offset
len = entry.len

here = alloc.allocate(len)

if (CPU == src_alloc.domain) {
  alloc.copy_to(here, msg, len)
} else if (CPU == alloc.domain) {
  src_alloc.copy_from(msg, here, len)
} else {
  alloc.copy(here, src_alloc, msg, len)
}
```

C. Message Lifecycle

As a recap to our system model, we cover the steps to create, use, and dispose of a message. We conclude with comments on thread safety and steps to prevent race conditions of shared memory. The steps to interact with Hazcat are 4-part: allocate, publish, take, and deallocate. The *allocate* method is called on an allocator to create memory for the message, as detailed in Section III-A1. During a component's computation, the message is populated. After the first component is finished, it calls *publish*, which places the allocator ID and message offset in the relevant message queue, as described in III-B3. When a second subscribed component has been informed of a message, it calls *take*. This modifies the message queue as described in Section III-B4 and also affects the messages directly: it calls *share* on the message copy in the component's memory domain, which increments the reference counter stored in the allocator. If this subscriber is the last to read the message,

it will also call *deallocate* on all the message copies, which decrements the reference counter. This is akin to clearing the message queue's references to the message copies. The message won't be garbage collected until this subscribed component calls *deallocate* one more time to clear it's own reference to the message, which is the final step.

In Section III-A2, we mention that the *share* and *deallocate* commands create a natural race condition. This is resolved by the fact that *share* is only ever called from within the *take* command, while the message is owned by the message queue. Calling *deallocate* from a separate thread concurrently with the *take* command is always guaranteed to do nothing, as the message queue itself retains a reference to the message. The only circumstance where calling *deallocate* will truly deallocate a message is when the message is no longer tracked by the message queue, so there's no possibility of *share* being called on it simultaneously.

IV. Design and Implementation

Any component-based framework using a CORBA or pub/-sub communication model can be modified to use Hazcat as its communication layer. Calls can be made to hazcat to register publishers and subscribers, create messages, publish to a topic, take from a topic, and finally deallocate messages. All that is needed is a translation layer to incorporate this into a desired framework.

For sake of demonstration, we created an RMW to interface with ROS2 systems and provide these zero-copy benefits to ROS2 applications. We also provide 2 allocators, one each for CPU and GPU, both implementing a static ring buffer. Additional allocators can be added readily in the future, such as one for the real-time dynamic memory strategy TLSF [24].

These allocators can be specified by the end developer when instantiating publishers or subscribers. These are done through the use of subscription_options and publisher_options, which each contain a field called rmw_implementation_payload. This mechanism can be used to pass options specific to an RMW implementation. In this case, a Hazcat allocator. If this field is ignored, then a default CPU-based allocator is assumed.

When extending the Hazcat platform to new products and hardware, we make certain stipulations for features the drivers must provide: an IPC mechanism to share device memory allocations with unrelated processes; support for unified-virtual-addressing, including virtual memory reservations; and mapping of physical memory to an explicit virtual address, subject to page granularity constraints.

First, the hardware driver must implement some interprocess communication (IPC) mechanism for its device memory. Any device memory allocated must be able to be accessed by any unrelated process using a globally unique token. Depending on how this mechanism is implemented, or whether it's implemented at all, may naturally limit a product's performance and its viability to be used with Hazcat. As an illustrative example, CUDA now provides shareable handles for IPC communication with their new driver API calls. As of CUDA 10.2, a developer can use *cuMemExportToShareableHandle* and *cuMemImportFromShareableHandle* to create a handle allegedly shareable between processes. However, these calls only work for related processes. The shareable handles are process-specific file descriptors, and any attempt to translate these descriptors to another process, via the *procfs* and the new *pidfd_getfd* syscall, will inevitably fail. Why this happens not unexplained in the current CUDA documentation. The handle is intended to be used before a *fork()* operation, and is not conducive to IPC between arbitrary processes.

Additionally, the drivers for the hardware device in question must support unified-virtual-addressing. Remapping allocators requires concatenating shared memory with device memory in a way that they appear as a single object. While their absolute location may vary, their relative positioning must be intact between processes. Thus, once a shared memory mapping is created for the allocator, its device memory pool must be placed in a particular location in a process's address space. If this ability is not present, we can't make guarantees about the positional relationship between an allocator and its memory pool. These guarantees are essential to correctly locate messages using nothing more than an offset.

Lastly, to avoid race conditions when claiming address space, the device needs a call to reserve virtual memory. Then the allocator and device memory can be mapped in without another thread claiming adjacent virtual memory.

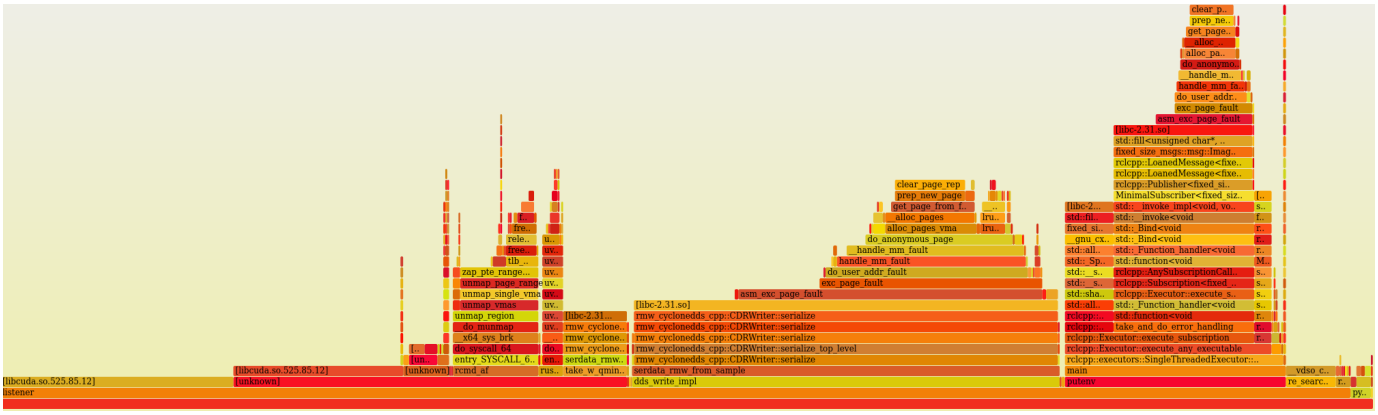
CUDA's driver APIs support all the necessary virtual memory features mentioned above, but lack an adequete IPC mechanism. The traditional CUDA API does not support address reservations and explicit memory mapping, but does have IPC functionality that meets our constraints. However, the two APIs provided are not compatible. We can conclude that based on the aggregate functionality present in both APIs, Nvidia hardware has the capability to work with Hazcat, but as it stands, Hazcat cannot support CUDA on it for independent-process workloads. The experiments below in Section V therefore use components in a single process or in related processes.

V. Empirical Evaluation

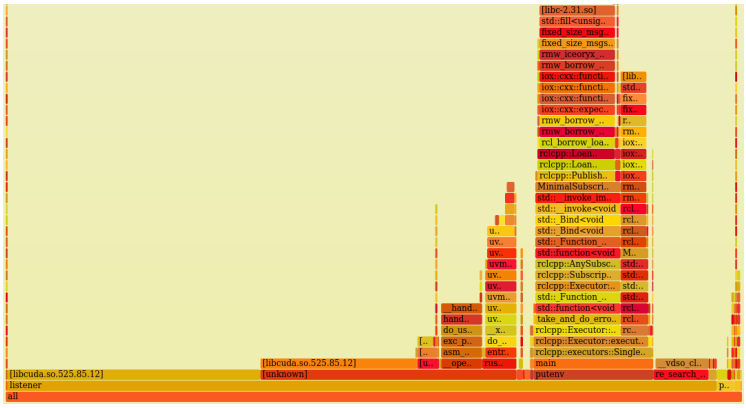
A. Two Component Experiments

As a demonstrative example, we evaluate a sample application with two nodes. Each performs a simple bilateral filter on a random 4k image, to serve as an arbitrary parallelizable computation. These experiments were performed on a desktop system with an AMD Ryzen7 3700X CPU and an Nvidia RTX 2070 Super. We measure the end-to-end latency and perform stacktrace sampling on this application while it runs on three different middlewares: CycloneDDS, Iceoryx, and our Hazcat.

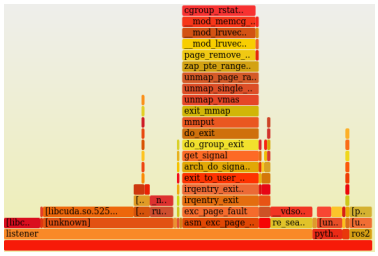
Due to the limitations of CUDA IPC mentioned in Section IV, these nodes where run in the separate threads in the same process, instead of in different processes, though we expect that the performance differences with that other case would be negligible. We measure the steady state operation of the system, so the first 1 or 2 samples were dismissed as outliers if their runtime latencies differed significantly from the subsequent samples. Such outliers were seen in the Iceoryx



(a) CycloneDDS Flame Graph
(end-to-end latency: 50.8ms)



(b) Iceoryx Flame Graph
(end-to-end latency: 27.2ms)



(c) Hazcat Flame Graph
(end-to-end latency: 13.7ms)

Fig. 3: Stacktrace Sampling

uniprocessor example, as well as in all of the GPU examples (presumably due to kernel loading).

Figure 1 illustrates the different operations performed by this two-component sample application. Each operation type is detailed in Section II, and mean timing information for each variation is illustrated in Figure 4. Variation on the end-to-end latency for all three middlewares is displayed in the violin graphs in Figure 5. These figures were created by manually placing userspace probes in the application to measure the entry and exit points for GPU memory operations, GPU kernels, and the callback functions for each component.

Stacktrace sampling can be found in Figures 3a, 3b, and 3c. The width of these flame graphs has been normalized according to their respective end-to-end latencies, so the time spent in each function can be better visualized.

Our tracing tools are unfortunately incapable of extracting stack traces from within CUDA code, so any usage, be it from kernel execution or memory operations, are combined into ”libcuda”. We do observe that CycloneDDS and Iceoryx spend roughly the same amount of time in the CUDA library, independant of their other sources of overhead. Hazcat, however, spends much less time running CUDA code. Since the three

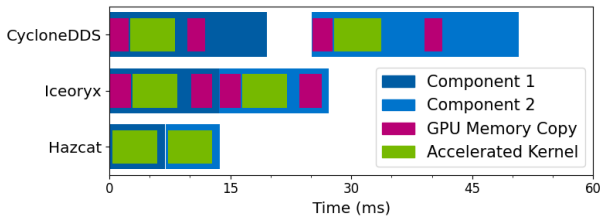


Fig. 4: Two Component Experiment with Hardware Acceleration

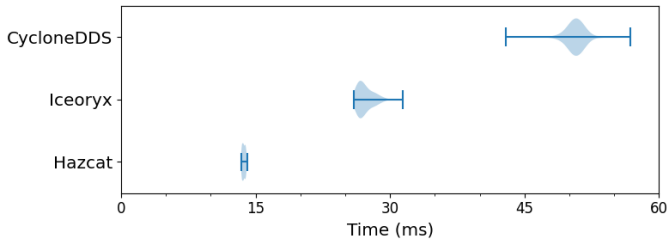


Fig. 5: End-to-end Latency Distributions for Two Component Experiment

are running the same kernel, we can assume time spent on GPU computation remains constant, and the variation observed is a result of reduced GPU memory operations.

We see this confirmed in Figure 4. Iceoryx sees reduced latency compared to CycloneDDS, despite running the same code. This is because of CycloneDDS's worse overhead and required memory copies between components (as dissected in 3a). Hazcat runs slightly modified code, where the memory copies in userspace have been removed. Almost all of the end-to-end latency under Hazcat is from GPU computation.

The Iceoryx and Hazcat variations both implement some type of zero copy and as a result have reduced inter-component (and even intra-component) latency – the latter is due to performing a *publish* call on the underlying middleware while the node is running. This call performs a memory copy in CycloneDDS but has minimal overhead in Iceoryx and Hazcat.

CycloneDDS and Iceoryx incur the overhead of additional GPU memory operations when using hardware acceleration. This is the primary performance benefit of Hazcat over Iceoryx: eliminating unnecessary GPU memory operations.

Figure 5 shows density plots of the end-to-end latency of each experiment variation. The distributions on the hardware accelerated workloads show that both Hazcat and Iceoryx have tight lower bounds, highlighting that, with them, the best case is the typical case. CycloneDDS, with its higher overhead and excess inter-component memory copies, sees more variation.

The most interesting finding is that Hazcat also has a tight upper bound. Iceoryx's use of GPU memory copies creates a source of latency jitter. Hazcat, on the other hand, has no such issue, making it behave much more deterministically compared to the other two frameworks.

B. Synthetic ROS Graphs

To demonstrate Hazcat's performance in more complex workloads, we synthetically generated 98 ROS graphs with randomized message sizes for topics and randomized CPU or GPU computation for nodes. Graphs ranged from 2 to 14 nodes, 1 to 17 edges, and had critical paths ranging from 2 nodes to 5.

Each graph was ran under Hazcat, Iceoryx, and CycloneDDS, and their end-to-end latencies were measured. The first three samples were discarded from each run, to eliminate outliers caused by transient effects during startup or shutdown.

Different graphs will naturally have vastly varying end-to-end latencies. We observed mean latencies ranging from 15 to 206ms. Since plotting all 98 graphs would be infeasible, instead measurements were normalized relative to the mean end-to-end latency of the same graph ran on Iceoryx. That is, if 3 runs of the same graph on Iceoryx take 10ms, 12ms, and 17ms, each run will then be plotted in Figure 6 as -2Δms, -1Δms, and 4Δms. If a run of the same graph on Hazcat takes 8ms, it will be plotted as -5Δms.

Hazcat had a better mean performance than CycloneDDS for every graph. It performed comparably ($\pm 10\%$) with Iceoryx for 60% of the graphs, and out-performed Iceoryx for 9% of the graphs.

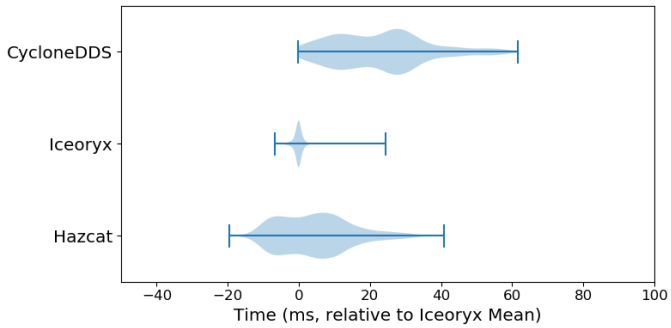


Fig. 6: Relative performance with randomized ROS graphs

When examining individual graphs, we notice that Hazcat performs comparably to Iceoryx in graphs dominated by CPU work, as well as graphs with wide work but a short span. Such graphs comprised the majority of our test set. The largest performance benefits come when multiple successive GPU nodes are chained together, and can benefit from the device-memory zero copy.

There was a greater deal of temporal variance in Hazcat, which we ascribe to its experimental implementation.

VI. Conclusions and Future Work

The primary contribution of Hazcat's heterogeneity aware zero-copy features for heterogeneous computing environments is that memory copies are only performed when data in one memory domain is needed in a different domain. In all other cases, memory is recycled, drastically reducing the worst case end-to-end latency of an application as well as removing sources of nondeterminism.

In our synthetic benchmarks with randomized ROS graphs, we saw an average case speedup of 1-73% over the unoptimized CycloneDDS middleware, depending on the graph in question, with 27% being typical. When compared to Iceoryx, we observed anywhere between nearly a 1.8x slowdown to a 2x speedup, again, depending heavily on the graph in question.

The largest performance benefits are mostly seen when consecutive components in the application all operate in the same domain. Even in the worst case, for an application using n domains, we never saw more than n-1 copy operations per message.

We observe that in addition to performance gains in hardware acceleration workloads, our framework has a decentralized design, which additionally reduces the requisite developer knowledge, as there is no 3rd party daemon to launch and no ring buffers to configure. Built-in defaults and runtime initialization allow it to perform well even when limited information is provided about the application.

In the future, we aim to support a wider range of hardware. This paper only discusses CUDA based usage of Nvidia hardware, but FPGAs are an important target for future research as well. Given the aforementioned lapses in the CUDA drivers' support of IPC, we plan to investigate other approaches to support GPUs [6] [31].

Hazcat also lays the groundwork for future exploration of custom allocation strategies. Though it currently only supports static ring buffers, we plan to develop other allocators in the future, including ones for dynamic allocation of uniprocessor, multi-core, and device memory.

References

[1] Hamster (ros2 robots) https://robots.ros.org/hamster/.

[2] Turtlebot4 (ros2 robots) https://robots.ros.org/turtlebot4/.

[3] xarm (ros2 robots) https://robots.ros.org/xarm/.

[4] Tanya Amert, Nathan Otterness, Ming Yang, James H Anderson, and F Donelson Smith. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115. IEEE, 2017.

[5] Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn Brandenburg. Response-time analysis of ros 2 processing chains under reservation-based scheduling. In *31st Euromicro Conference on Real-Time Systems*, pages 1–23. Schloss Dagstuhl, 2019.

[6] Roberto Cavicchioli, Nicola Capodieci, Marco Solieri, and Marko Bertogna. Novel methodologies for predictable cpu-to-gpu command offloading. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[7] Marc Eisoldt, Steffen Hinderink, Marco Tassemeier, Marcel Flottmann, Juri Vana, Thomas Wiemann, Julian Gaal, Marc Rothmann, and Mario Porrmann. Reconfros: Running ros on reconfigurable socs. In *Proceedings of the 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings*, pages 16–21. 2021.

[8] eProsima. Fastdds. https://github.com/eProsima/Fast-DDS, 2016.

[9] David Ferry, Gregory Bunting, Amin Megareh, Shirley Dyke, Arun Prakash, Kunal Agrawal, Chris Gill, and Chenyang Lu. Real-time system support for hybrid structural simulation. In *International Conference on Embedded Software (EMSOFT)*. ACM, 2014.

[10] Eclipse Foundation. Cyclonedds. https://github.com/eclipse-cyclonedds/cyclonedds, 2019.

[11] Eclipse Foundation. Iceoryx. https://github.com/eclipse-iceoryx/iceoryx, 2019.

[12] David Gay and Alex Aiken. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 313–323, 1998.

[13] Mike Harris. Fast, flexible allocation for nvidia cuda with rapids memory manager, Dec 2020.

[14] Anakhi Hazarika, Soumyajit Poddar, and Hafizur Rahaman. Survey on memory management techniques in heterogeneous computing systems. *IET Computers & Digital Techniques*, 14(2):47–60, 2020.

[15] Michel Hidalgo, Shane Loretz, Chris Lalancette, Michael Jeromino, M. Mei, Marya Belanger, and Christophe Bedard. About quality of service settings, Jul 2022.

[16] Michel Hidalgo, Ivan Paunovic, Chris Lalancette, and Esther Weon. Setting up efficient intra-process communication, Jul 2022.

[17] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Third edition, September 2011.

[18] Mark S Johnstone and Paul R Wilson. The memory fragmentation problem: Solved? *ACM Sigplan Notices*, 34(3):26–36, 1998.

[19] Karsten Knese and Michael Pöhnl. A true zero-copy rmw implementation for ros2. 2019.

[20] Karsten Knese, William Woodall, and Michael Carroll. Zero copy via loaned messages, Apr 2020.

[21] Donald E Knuth. The art of computer programming, vol 1: Fundamental. *Algorithms. Reading, MA: Addison-Wesley*, 1968.

[22] Daniel Pinheiro Leal, Midori Sugaya, Hideharu Amano, and Takeshi Ohkawa. Automated integration of high-level synthesis fpga modules with ros2 systems. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 292–293. IEEE, 2020.

[23] Daniel Pinheiro Leal, Midori Sugaya, Hideharu Amano, and Takeshi Ohkawa. Fpga acceleration of ros2-based reinforcement learning agents. In *2020 Eighth International Symposium on Computing and Networking Workshops (CANDARW)*, pages 106–112. IEEE, 2020.

[24] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and Jorge Real. Tlsf: A new dynamic memory allocator for real-time systems. In *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.*, pages 79–88. IEEE, 2004.

[25] Víctor Mayoral-Vilches, Sabrina M Neuman, Brian Plancher, and Vijay Janapa Reddi. Robotcore: An open architecture for hardware acceleration in ros 2. *arXiv preprint arXiv:2205.03929*, 2022.

[26] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 66–77, 1995.

[27] Yasuhiro Nitta, Sou Tamura, and Hideki Takase. A study on introducing fpga to ros based autonomous driving system. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 421–424. IEEE, 2018.

[28] Takeshi Ogasawara. An algorithm with constant execution time for dynamic storage allocation. In *Proceedings Second International Workshop on Real-Time Computing Systems and Applications*, pages 21–25. IEEE, 1995.

[29] Takeshi Ohkawa, Yuhei Sugata, Harumi Watanabe, Nobuhiko Ogura, Kanemitsu Ootsu, and Takashi Yokota. High level synthesis of ros protocol interpretation and communication circuit for fpga. In *2019 IEEE/ACM 2nd International Workshop on Robotics Software Engineering (RoSE)*, pages 33–36. IEEE, 2019.

[30] Takeshi Ohkawa, Kazushi Yamashina, Takuya Matsumoto, Kanemitsu Ootsu, and Takashi Yokota. Architecture exploration of intelligent robot system using ros-compliant fpga component. In *2016 International Symposium on Rapid System Prototyping (RSP)*, pages 1–7. IEEE, 2016.

[31] Nathan Otterness and James H Anderson. Amd gpus as an alternative to nvidia for supporting real-time workloads. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[32] Cenk Özer. A dynamic memory manager for fpga applications. Master's thesis, Middle East Technical University, 2014.

[33] Ariel Podlubne and Diana Göhringer. Fpga-ros: Methodology to augment the robot operating system with fpga designs. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–5. IEEE, 2019.

[34] Paul W Purdom, Stephen M Stigler, and Tat-Ong Cheam. Statistical investigation of three storage allocation algorithms. *BIT Numerical Mathematics*, 11(2):187–195, 1971.

[35] J Peña Queralta, Fu Yuhong, Lassi Salomaa, Li Qingqing, Tuan Nguyen Gia, Zhuo Zou, Hannu Tenhunen, and Tomi Westerlund. Fpga-based architecture for a low-cost 3d lidar design and implementation from multiple rotating 2d lidars with ros. In *2019 IEEE SENSORS*, pages 1–4. IEEE, 2019.

[36] Yuhei Sugata, Takeshi Ohkawa, Kanemitsu Ootsu, and Takashi Yokota. Acceleration of publish/subscribe messaging in ros-compliant fpga component. In *Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, pages 1–6, 2017.

[37] Yuhei Suzuki, Takuya Azumi, Shinpei Kato, and Nobuhiko Nishio. Real-time ros extension on transparent cpu/gpu coordination mechanism. In *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pages 184–192. IEEE, 2018.

[38] Yue Tang, Zhiwei Feng, Nan Guan, Xu Jiang, Mingsong Lv, Qingxu Deng, and Wang Yi. Response time analysis and priority assignment of processing chains on ros2 executors. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 231–243. IEEE, 2020.

[39] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and computation*, 132(2):109–176, 1997.

[40] Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management*, pages 1–116. Springer, 1995.

[41] Falk Wurst, Dakshina Dasari, Arne Hamann, Dirk Ziegenbein, Ignacio Sanudo, Nicola Capodieci, Paolo Burgio, and Marko Bertogna. System performance modelling of heterogeneous hw platforms: An automated driving case study. In *22nd Euromicro Conference on Digital System Design (DSD)*. Euromicro, 2019.

[42] Yecheng Xiang and Hyoseung Kim. Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 392–405. IEEE, 2019.

[43] Ming Yang. Avoiding pitfalls when using nvidia gpus for real-time tasks in autonomous systems. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems*, 2018.