Thread-Safe Abstraction in KIPR Robotics Controllers

Rocky Ratia

***Abstract* — Heterogeneous environments, such as the Raspberry Pi paired with the STM32 coprocessor, are popular in educational and practical robotics to separate high-level architectures from low-level operations. While higher-level robotics software frequently utilizes multithreading for various actions, standard Serial Peripheral Interface (SPI) drivers are inherently thread-unsafe. This paper presents a compact solution utilizing lock-free shared-memory techniques, allowing concurrent hardware access without blocking on mutexes. This paper also presents preliminary performance measurements of the proposed architecture.**

I. Introduction

Heterogeneous computing is a computing paradigm in which multiple processor architectures cooperate to execute different workloads [1]. In the KIPR Wombat, a Raspberry Pi 3B+ is used alongside an STM32 coprocessor [2]. The protocol used to transfer data between them is called SPI, or Serial Peripheral Interface. Modern robotics typically makes use of multithreading, allowing multiple tasks to execute concurrently. The existing SPI communication layer used by libwallaby and the STM32 firmware is not thread-safe, meaning concurrent accesses from multiple threads can result in undefined behavior.

II. The SPI Protocol and Embedded Constraints

The communication between the Raspberry Pi and the STM32 coprocessor takes place over an SPI interface. Both devices maintain identical copies of a buffer containing all sensor data and motor states. To keep both buffers synchronized, libwallaby exchanges the entire memory map between devices during each hardware update cycle. While this approach minimizes transaction overhead, it introduces thread safety challenges. If multiple threads attempt to read from or write to the local buffer simultaneously, data corruption and undefined behavior can occur.

III. Current Thread-Safety Mechanisms

Existing approaches to thread safety vary considerably. Frameworks such as ROS2 and DDS use callback groups [4]. KIPR's library for interfacing with the hardware, libwallaby, has a basic approach for making the SPI calls safe for multithreading. The approach used by KIPR's library (libwallaby) is wrapping POSIX threads (see Fig. 1).

```
class EXPORT_SYM Thread
{
public:
    Thread();
    virtual ~Thread();

    void start();
    void join();

    virtual void run() = 0;

private:
#ifndef WIN32
    pthread_t m_thread;
#else
    HANDLE m_thread;
#endif
};
```

Fig. 1. The base class for libwallaby's thread implementation.

IV. Threading and Concurrency

C++ has methods to avoid mutexes while remaining thread-safe. C++ atomic objects provide lock-free synchronization primitives [5]. CPU cache architecture also affects multithreaded performance. A CPU cache line is a block of data that is

transferred between the CPU and RAM [6]. The Pi 3B+ features a CPU with multiple cores. When two or more cores attempt to access the same section of the memory, false sharing can occur [7]. This causes many issues, namely the cores will repeatedly invalidate and transfer ownership of the cache line instead of executing in parallel.

V. Implementation and Optimization

The Cortex-A53 in the Raspberry Pi 3B+ uses 64-byte cache lines. The proposed data structure will be aligned to that cache line size. This eliminates the risk of false sharing. The data structure itself will consist of lock-free atomic objects, to avoid the use of mutexes. For this implementation, each motor and servo occupies its own cache line. This will prevent false sharing for unrelated commands between them. The entire structure will use 14 cache lines. 8 will be for the motors and servos, and 6 will be for sensor telemetry and timing. A basic implementation of these data structures is demonstrated in Fig. 3.

```
constexpr size_t CACHE_LINE = 64;

struct alignas(CACHE_LINE) MotorCommand {
    std::atomic<int16_t> speed{0};
    std::atomic<bool> clear_position{false};
    std::atomic<bool> stop{false};
    std::atomic<bool> freeze{false};
};

struct alignas(CACHE_LINE) ServoCommand {
    std::atomic<uint16_t> position{0};
    std::atomic<bool> enable{false};
};

struct alignas(CACHE_LINE) DriverCommands {
    alignas(CACHE_LINE) MotorCommand motor[4];
    alignas(CACHE_LINE) ServoCommand servo[4];
};

struct alignas(CACHE_LINE) MotorTelemetry {
    std::atomic<int32_t> position{0};
};

struct alignas(CACHE_LINE) ImuTelemetry {
    std::atomic<int32_t> x{0};
    std::atomic<int32_t> y{0};
    std::atomic<int32_t> z{0};
};

struct alignas(CACHE_LINE) AnalogTelemetry {
    std::atomic<uint16_t> value[6]{0, 0, 0, 0, 0, 0};
};

struct alignas(CACHE_LINE) DigitalTelemetry {
    std::atomic<bool> port[10]{false, false, false, false, false, false, false, false, false, false};
};

struct alignas(CACHE_LINE) SensorTelemetry {
    alignas(CACHE_LINE) MotorTelemetry motor[4];
    alignas(CACHE_LINE) ImuTelemetry gyro;
    alignas(CACHE_LINE) AnalogTelemetry analog;
    alignas(CACHE_LINE) DigitalTelemetry digital;
};

struct alignas(CACHE_LINE) ImperativeCommand {
    using MilliTime = std::chrono::time_point<std::chrono::steady_clock, std::chrono::milliseconds>;
    static constexpr MilliTime FarFuture = MilliTime::max();
    std::atomic<MilliTime> start_time{FarFuture};
};

struct SharedVehicleChannel {
    alignas(CACHE_LINE) DriverCommands commands;
    alignas(CACHE_LINE) SensorTelemetry telemetry;
    alignas(CACHE_LINE) ImperativeCommand command;
};
```

Fig. 3. The basic implementation of the data structure, aligned to the cache size, utilizing atomics.

A standard thread will be used to update the data structures and make the actual hardware calls. The updater thread is the sole thread permitted to access the SPI hardware. The implementation is encapsulated within a class, ensuring no accidental external write operations occur. Raspberry Pi OS uses the Linux Completely Fair Scheduler, which may preempt application threads. Such preemption can interfere with the updater thread because scheduling delays directly increase hardware communication latency. To prevent this, a single core is reserved, and the highest priority is assigned using SCHED_FIFO to this thread to ensure it is not deprioritized. This optimization is achieved by targeting a specific core and assigning maximum priority to the thread (see Fig. 4).

```
cpu_set_t cpuset;
CPU_ZERO(&cpuset);
CPU_SET(2, &cpuset);
pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpuset);

struct sched_param param;
param.sched_priority = 99;
pthread_setschedparam(pthread_self(), SCHED_FIFO, &param);
```

Fig. 4. CPU Management and priority scheduling for the hardware update thread.

Within the thread's repeat loop, the data structure values are updated, pushing and reading values. Both the compiler and CPU may reorder memory accesses. To prevent this, memory_order_release is used when setting the atomic values, paired with memory_order_acquire on readers. Lastly, to make the functions usable to threads, wrapper calls can be implemented to abstract the actual call.

VI. Performance Evaluation

Performance evaluation of the lock-free shared memory architecture was conducted on the competition legal Wombat controller. The platform utilizes a Raspberry Pi 3B+ running standard Debian

Linux alongside an STM32 coprocessor, and communicates over a DMA-enabled SPI bus. Latency testing was executed by measuring the round-trip time required for a high-priority worker thread to propagate motor objective updates across the cache-aligned atomic structures. Preliminary measurements on the Wombat indicate a mean update latency of approximately 0.5 μs. Due to time constraints, a comprehensive evaluation—including worst-case latency, jitter, scalability under contention, and comparisons across workloads—is reserved for future work.

VII. Conclusion

Shared memory is desirable for applications using multithreading, requiring real-time, low-latency hardware access. For devices that use a hardware-level protocol that is thread-unsafe, there are limited options for operating the hardware from different threads. The current library relies on mutex-protected POSIX threads, which, while safe, is inefficient and not suitable for applications that require quick access. There are many challenges to overcome, from manually managing memory safety and cache lines to ensuring hardware threads are not deprioritized. In all, shared memory is well suited for applications in robotics that use parallel threads but require efficient, rapid hardware calls.

References

[1] Arm, "What is Heterogeneous Compute?," Arm Glossary, 2026. [Online]. Available: https://www.arm.com/glossary/heterogeneous-compute. [Accessed: Jun. 15, 2026].

[2] KIPR, "Docs/Schematics," KIPR-Development-Toolkit, master branch, 2026. [Online]. Available: https://github.com/kipr/KIPR-Development-Toolkit/tree/master/Docs/Schematics. [Accessed: Jun. 15, 2026].

[3] EventHelix, "Direct Memory Access (DMA) and Interrupt Handling," EventHelix, [Online]. Available: https://www.eventhelix.com/fault-handling/dma-interrupt-handling/. [Accessed: Jun. 15, 2026].

[4] Open Robotics, "Using Callback Groups," ROS 2 Documentation: Foxy documentation, 2026. [Online]. Available: https://docs.ros.org/en/foxy/How-To-Guides/Using-callback-groups.html. [Accessed: Jun. 15, 2026].

[5] "std::atomic," cppreference.com, 2026. [Online]. Available: https://cppreference.com/cpp/atomic/atomic. [Accessed: Jun. 15, 2026].

[6] S. Slotin, "Cache Lines," Algorithmica, 2022. [Online]. Available: https://en.algorithmica.org/hpc/cpu-cache/cache-lines/. [Accessed: Jun. 15, 2026].

[7] The Linux Kernel Development Community, "False Sharing," The Linux Kernel documentation, 2026. [Online]. Available: https://docs.kernel.org/kernel-hacking/false-sharing.html. [Accessed: Jun. 15, 2026].