# Bringing Concurrency to Embedded Graphical Applications with Async Rust

## PARS Workshop 2025

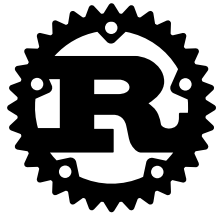
Paul Kailer and Frédéric Wagner
DATAMOVE Inria Grenoble / CAPP ITEC KIT
17/09/2025

# Rust

*a new programming language for developing reliable and efficient systems* [1]

- low-level: compiles to Assembly, performance on-par with C [2]
- memory-safe: guaranteed by the compiler
- high-level: abstractions and genericity included
- async concurrency built-in

All very useful properties in an embedded context



# Making an Operating System obsolete

Central OS responsibilities include

- Memory protection
- Scheduling

# Making an Operating System obsolete

Central OS responsibilities include

- Memory protection – Guaranteed by the compiler ✓
- Scheduling – Async support in the language ✓

# Making an Operating System obsolete

Central OS responsibilities include

- Memory protection – Guaranteed by the compiler ✓
- Scheduling – Async support in the language ✓

With Rust, these can be achieved with bare-metal code. That leads to

- No runtime memory management overhead
- Significantly reduced context switching overhead

# Making an Operating System obsolete

Central OS responsibilities include

- Memory protection – Guaranteed by the compiler ✓
- Scheduling – Async support in the language ✓

With Rust, these can be achieved with bare-metal code. That leads to

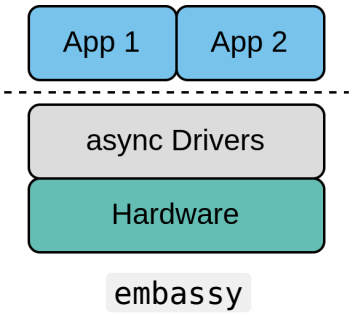
- No runtime memory management overhead
- Significantly reduced context switching overhead

→ **Can Async Rust eliminate an operating system dependency?**

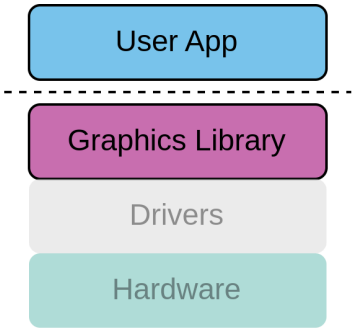
# Research Objective

Existing projects already provide significant features:

- The `embassy` project [3] provides drivers, an async *runtime*, mutex, etc.
- `embedded-graphics` [4] is an easy-to-use graphics library
-




`embassy`

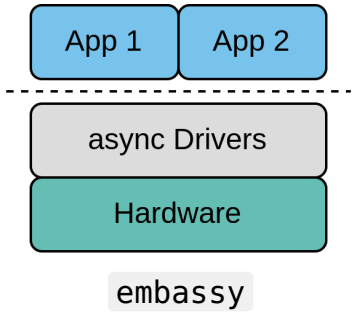


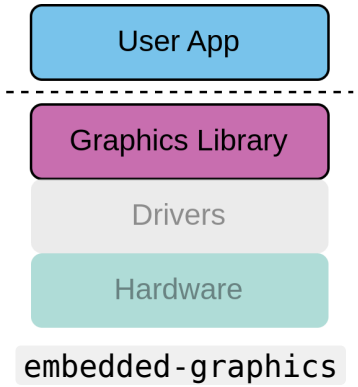

`embedded-graphics`

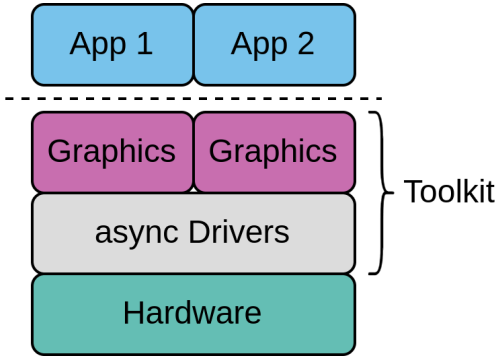
# Research Objective

Existing projects already provide significant features:

- The `embassy` project [3] provides drivers, an async *runtime*, mutex, etc.
- `embedded-graphics` [4] is an easy-to-use graphics library
- In this paper: **asydis** – an **asy**nc shared **dis**play toolkit



# Related Work

- Full-fledged embedded graphics libraries exist, e.g. Bangle.js [5] or LVGL [6]
  - either non-optimal performance (JavaScript) or
  - limited/no async support (C)
- Embedded (RT)OSes provide classical thread concurrency
  - higher overhead for memory management and context switching
  - increased complexity for the programmer

None of the above provide easy, safe and concurrent embedded development.

GRENOBLE INP UGA

1. Context
2. **Async Rust**
3. System Design
4. Evaluation

# Async Concurrency in Rust

- Rust has built-in support for cooperative concurrency with *Futures*
- *Future* – the result of an operation that will be available in the future
- Marking a function `async` transforms its return type into a Future automatically
- To execute a future the `await` keyword is used

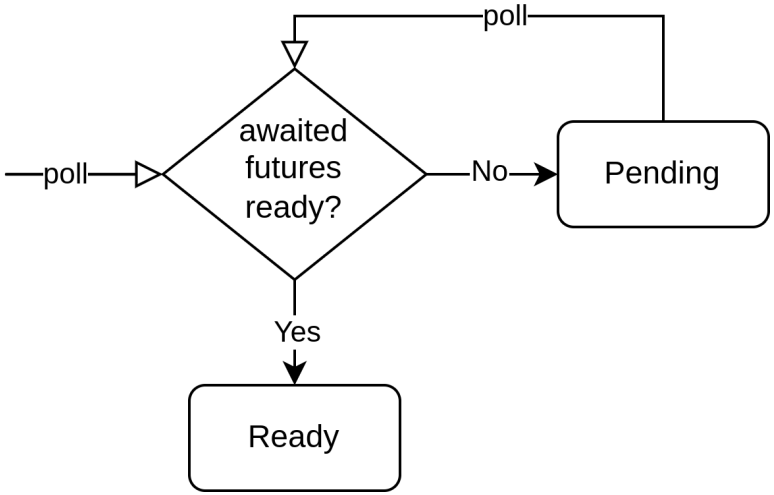
```rust
async fn flush_buffer(buffer: &[u8], interface: I2c) -> bool {
    return interface.send_data(buffer).await;
}
```

Rust

Flushing a buffer asynchronously

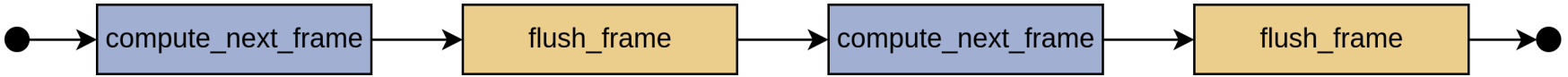
# Future State Machine

The compiler generates a state machine for every `async` function:

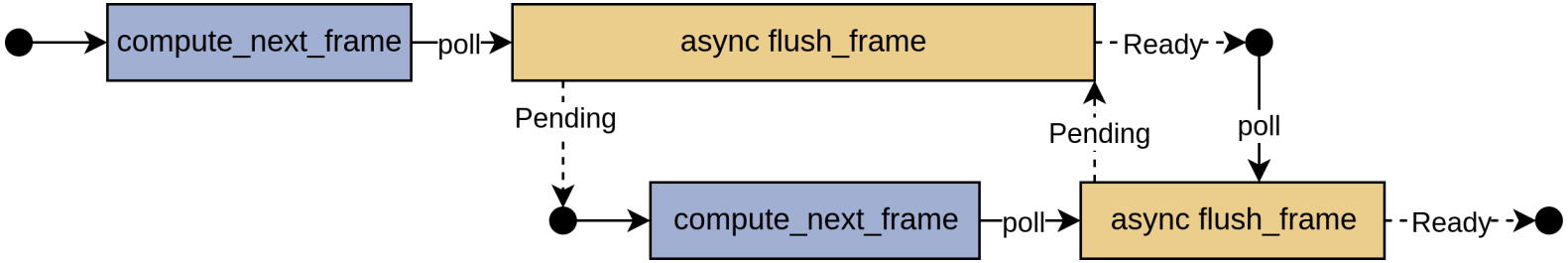



When a future is pending, futures from other tasks can be executed.

# Overlap from Asynchronous Execution



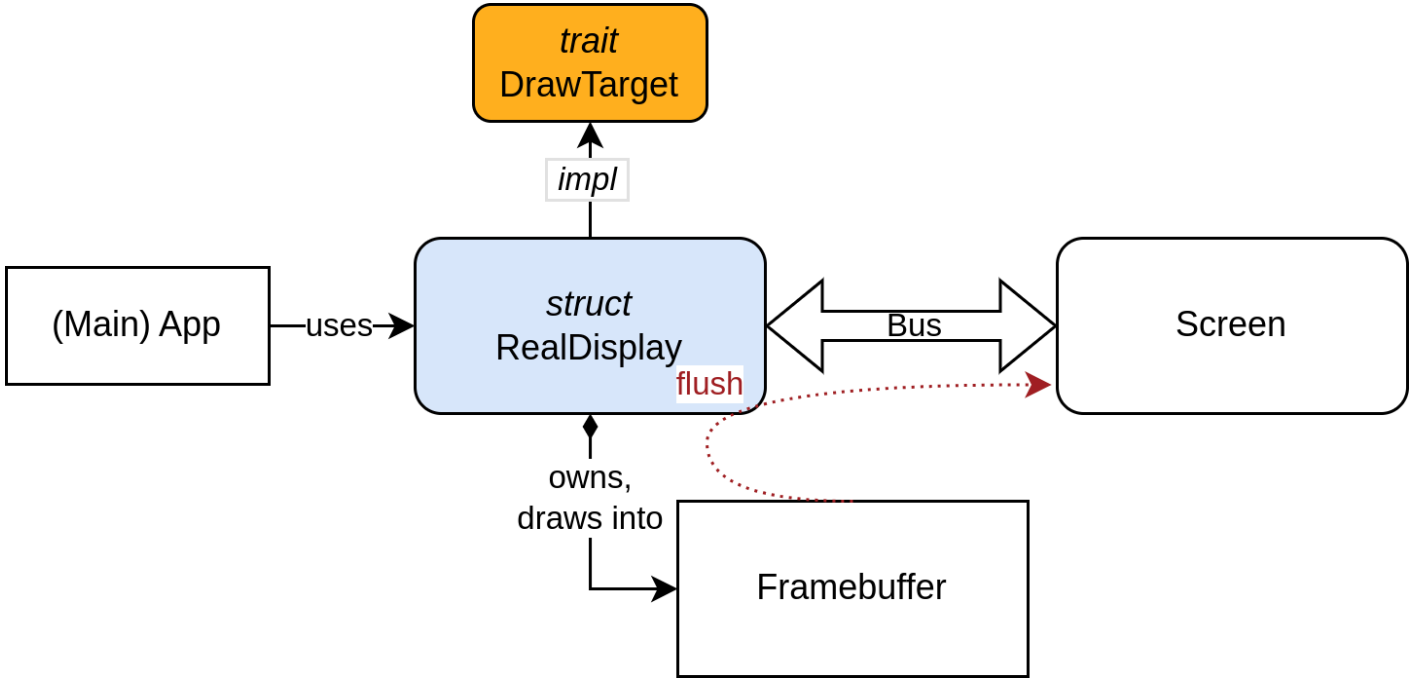

(a) Synchronous Flushing.




(b) Asynchronous Flushing, distributed over 2 tasks.

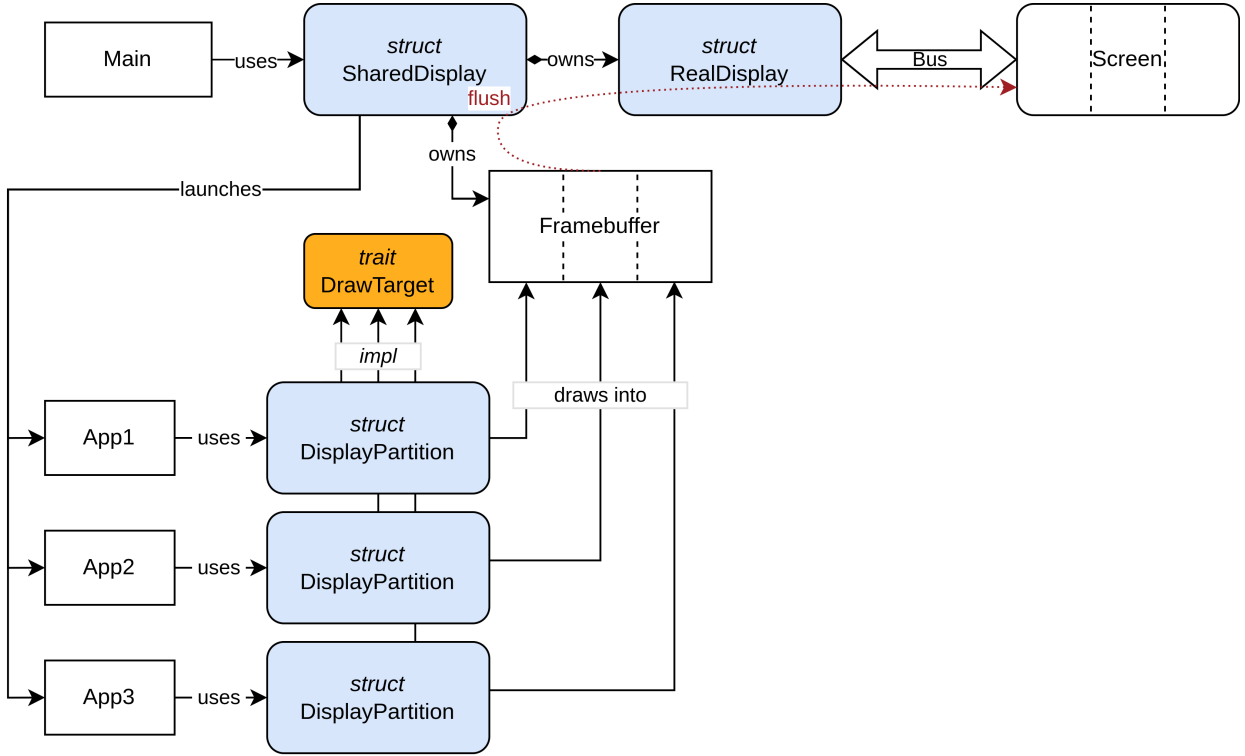
# System Design

# Graphical Applications without `asydis`



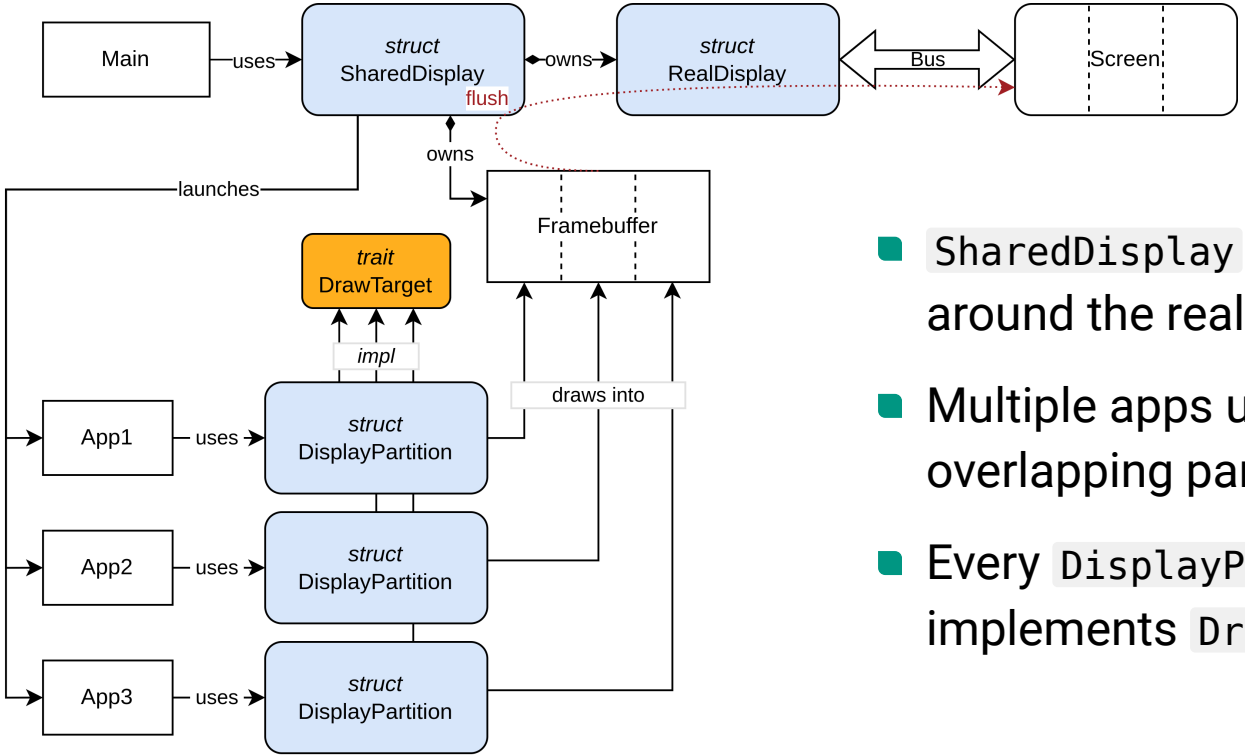

An application not using the toolkit.

# Graphical Applications with `asydis`




An application using the `asydis` toolkit for screen sharing.

# Graphical Applications with `asydis`

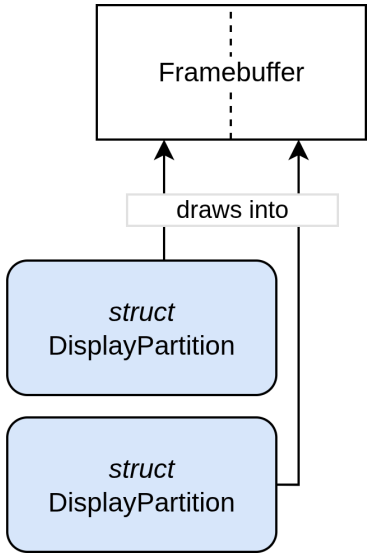



- `SharedDisplay` wraps around the real display
- Multiple apps use a non-overlapping partition each
- Every `DisplayPartition` implements `DrawTarget`

An application using the `asydis` toolkit for screen sharing.

# Concurrent Buffer Access

- Concurrent write access is disallowed in Rust
- Using the `unsafe` keyword, unchecked shared writes are possible
- Memory safety has to be manually checked by the programmer
- `asydis` filters out pixels outside the partition, guaranteeing safe access



# Evaluation

# Evaluation

Do these contributions improve embedded concurrent apps?
In terms of

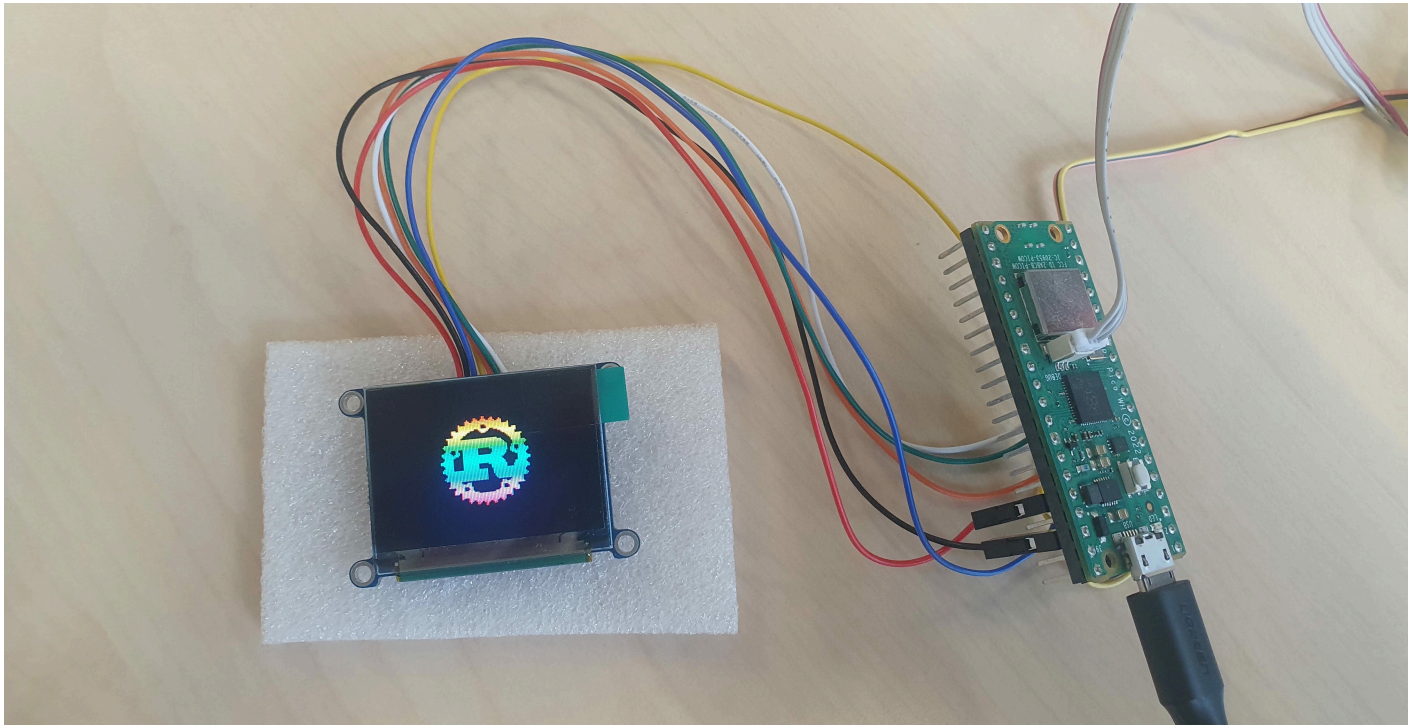
- Ease of Development
- Memory Safety
- Concurrency Effectiveness

# Evaluation

Do these contributions improve embedded concurrent apps?
In terms of

- Ease of Development (✓)
- Memory Safety (✓)
- Concurrency Effectiveness

# Experimental Setup



# Evaluation

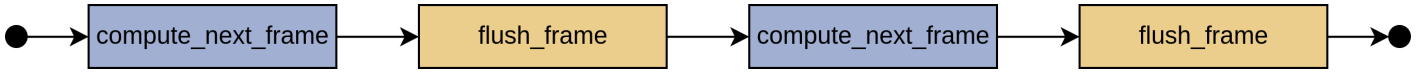
Run Benchmarks to investigate

- How much time can be gained from compute overlap?
- Does the `asydis` toolkit provide effective concurrency?
- Are there significant synchronization costs?

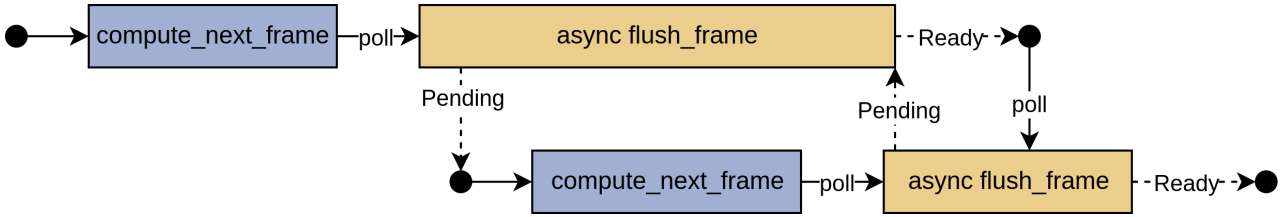
No standardised benchmarks exist, we created new ones.

# Potential Computation Overlap

- How does asynchronous programming increase runtime performance?




(a) Synchronous Flushing.




(b) Asynchronous Flushing, distributed over 2 tasks.

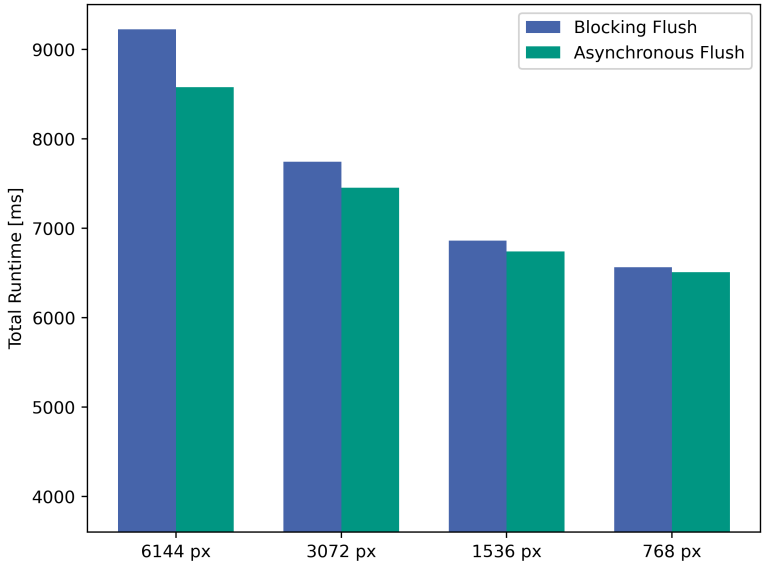
- Compare runtime of blocking vs asynchronous flush

# Potential Computation Overlap – Results

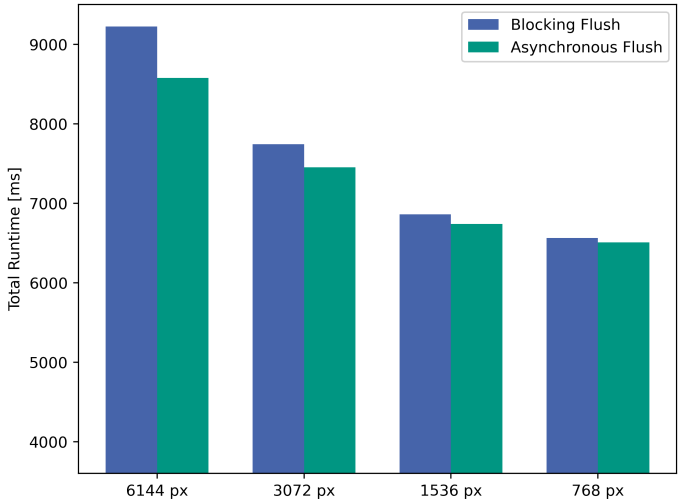
- 6 custom benchmarks consisting of drawing different shapes
- different screen sizes (simulated)

# Potential Computation Overlap – Results

- 6 custom benchmarks consisting of drawing different shapes
- different screen sizes (simulated)



# Potential Computation Overlap – Results




## Takeaway

The bigger the screen, the more overlap is possible

# How `asydis` enables effective Concurrency

- Concurrency can be achieved without the `asydis` toolkit
- How do `asydis` apps compare to concurrency with naive resource management with simple mutex?
- Same setup: 6 custom benchmarks, consisting of draw, flush, sleep
- Screen partitioned into 4 applications
- Without the toolkit, mutex needs to be locked for every draw, flush

# How `asydis` achieves Overlap – Results

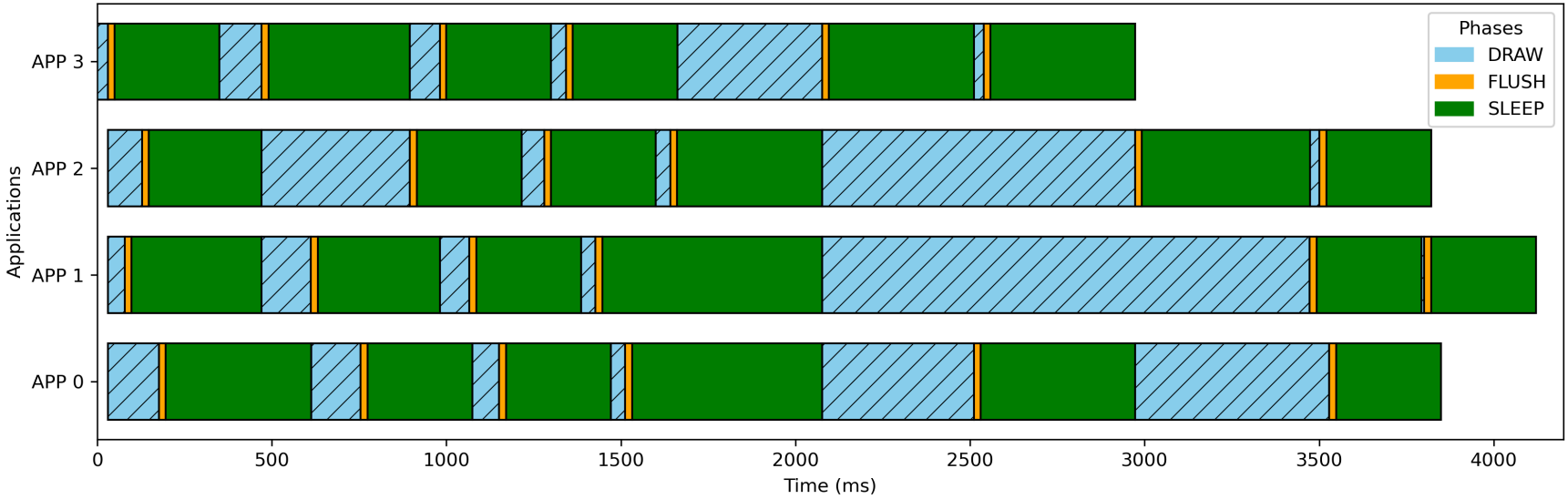



Figure 10: Naive version – mutex locked display reference

# How `asydis` achieves Overlap – Results

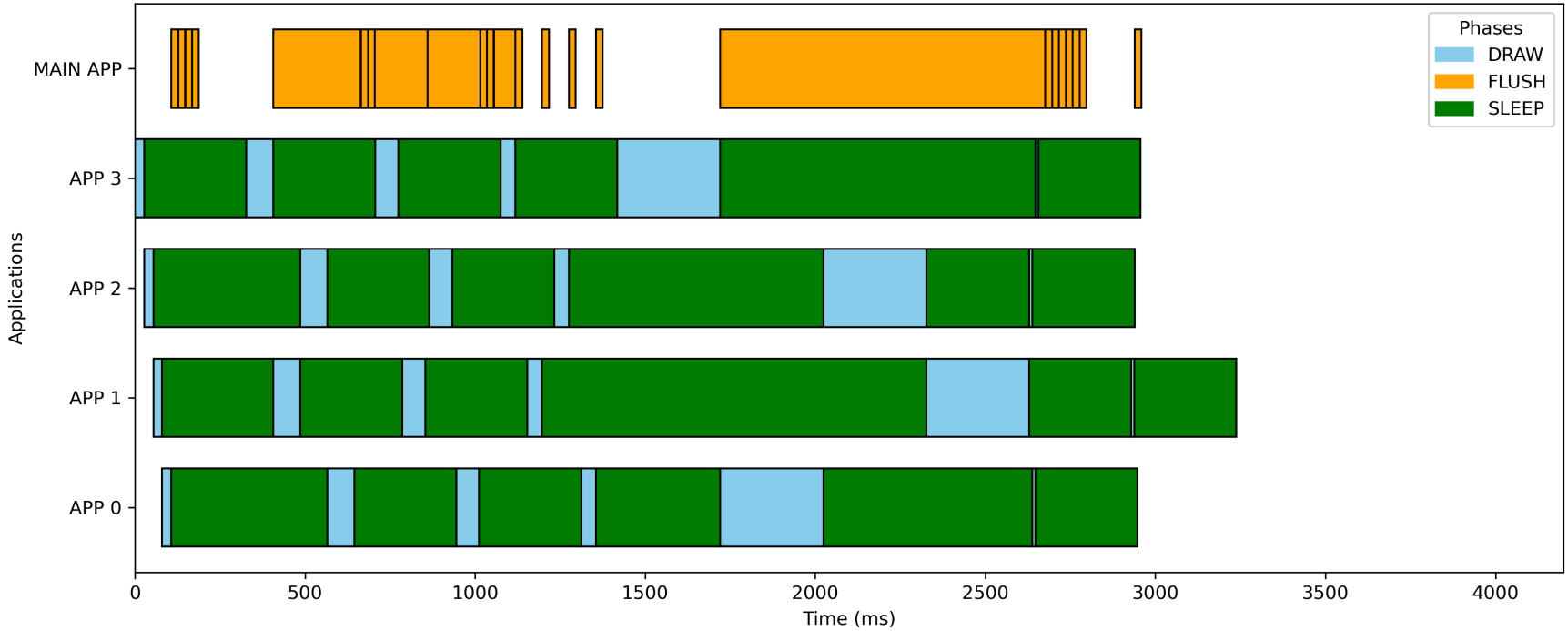
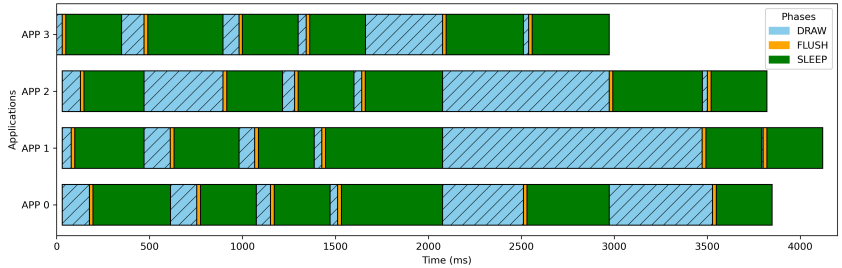
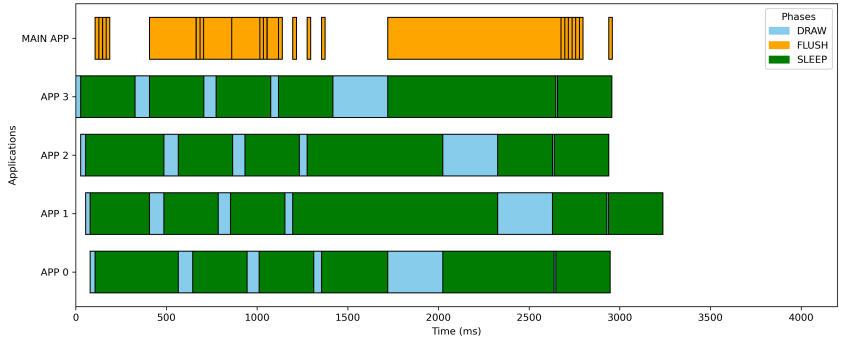



Figure 11: Toolkit version – checked `unsafe` buffer access

# How `asydis` achieves Overlap – Results



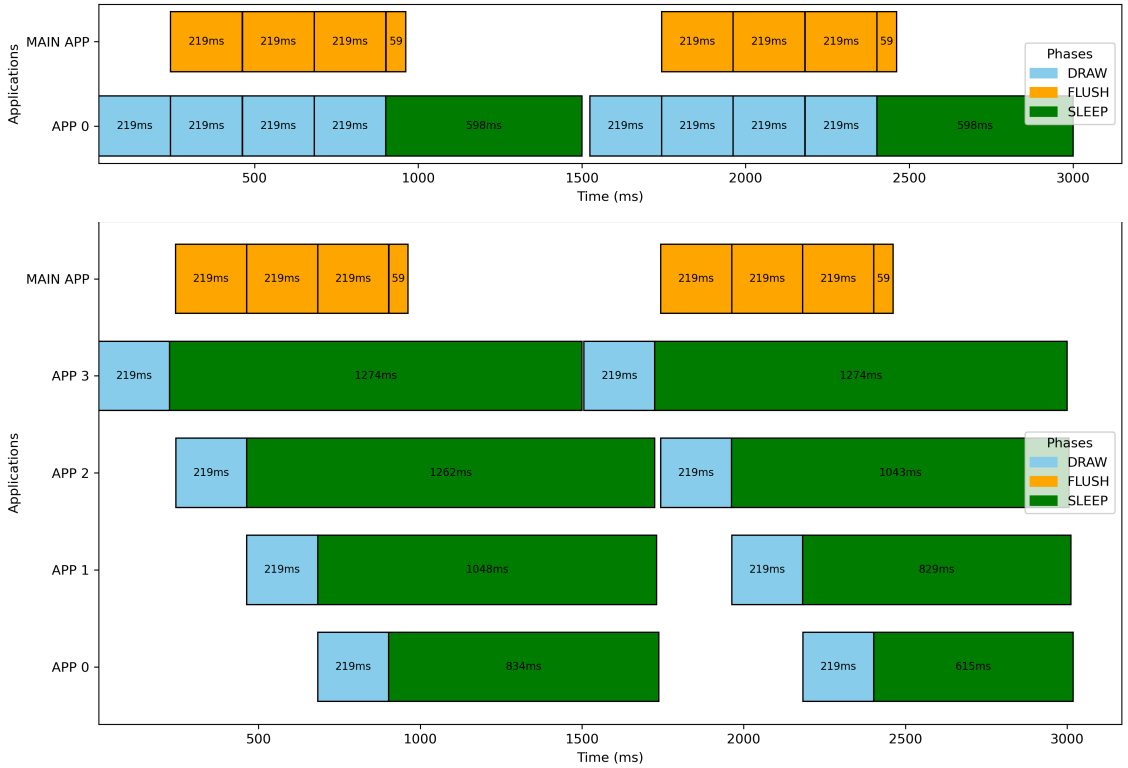




- Toolkit draw phases are significantly shorter – mutex eliminated
- Flushes are handled in a separate task (main app)
- Overall Runtime reduced from 4.12 to 3.24 seconds, 1.27× speedup

# Synchronisation Cost

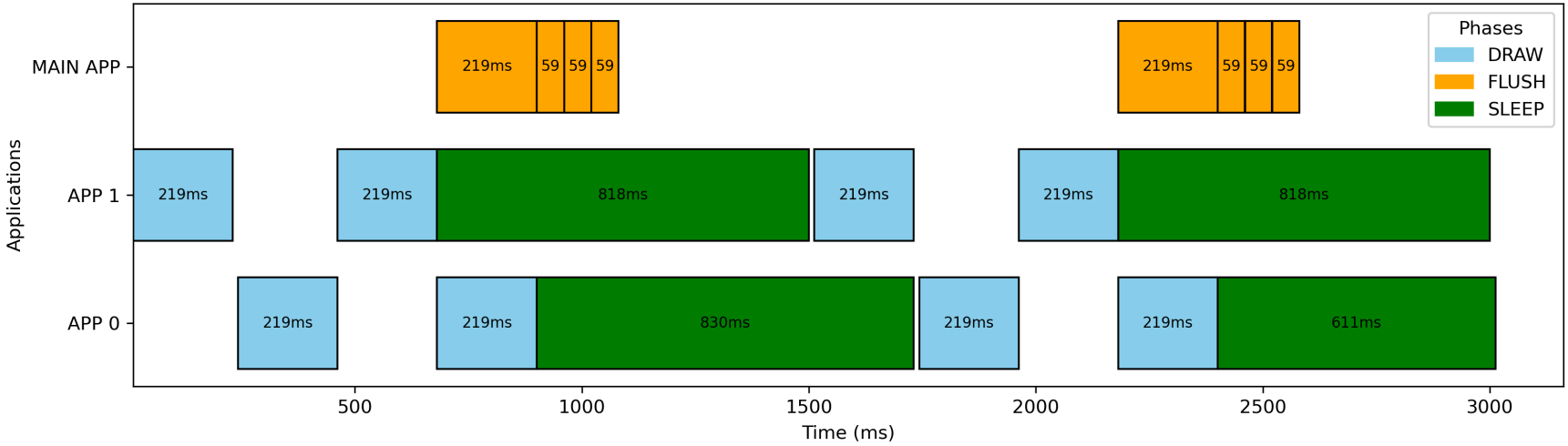
- How efficient is async task switching with the toolkit?
- Are there observable overheads when distributing the same workload on more apps?
- Experiment setup: draw, flush and sleep with a fixed framerate
- Divide each operation into 4 quarters of the screen
- Compare 1 app, 2 apps, 4 apps with the same workload

# Synchronisation Cost – Results

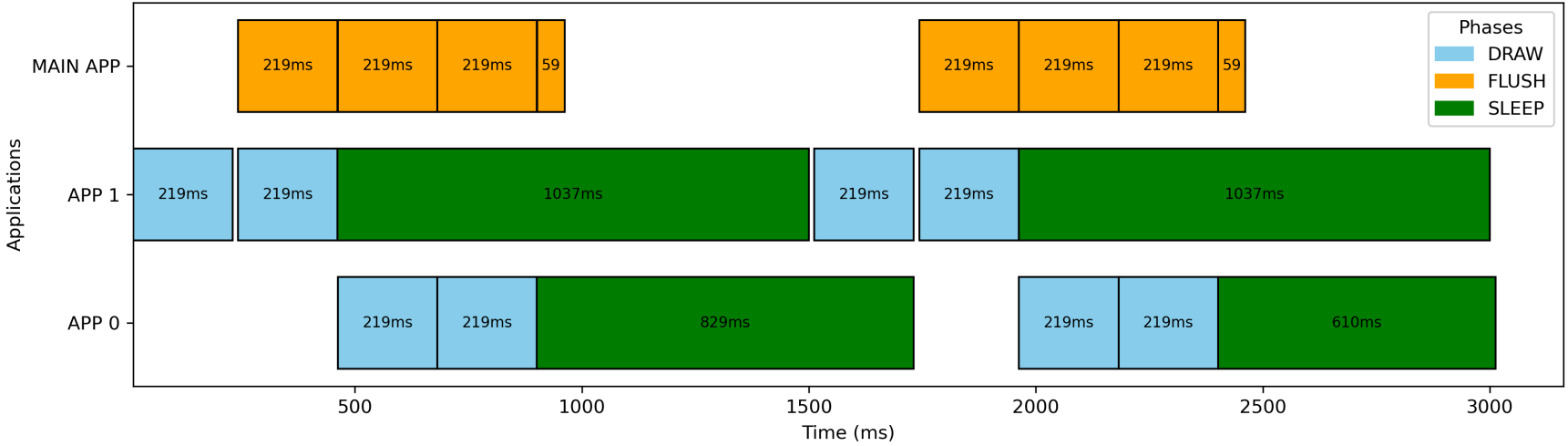



- After the last flush, the same sleep time remains
- No observable runtime overhead for task switching here

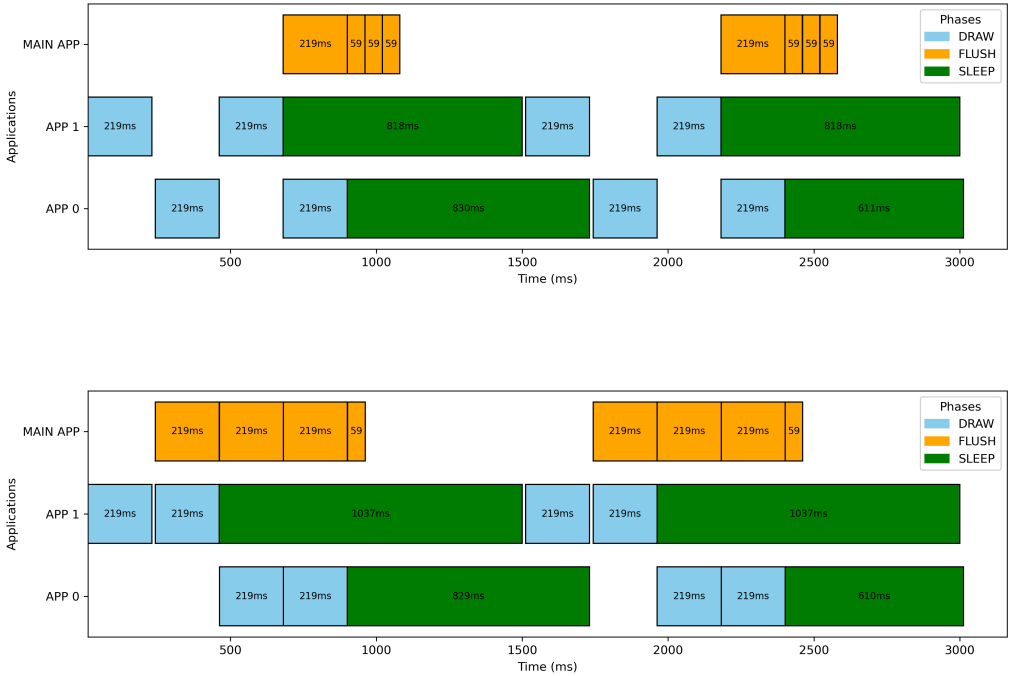
# Overlap Problems – Naive version



# Overlap Problems – Hacky Solution



# Synchronisation Cost – Results




- In a naive implementation, almost no overlap is achieved
- We prioritize the flush with manual synchronisation
- The async *runtime* schedules tasks non-preemptively

# Evaluation Recap

## Custom Benchmarks investigated

- How much time can be gained from compute overlap?
  - Simple swap for async flush gives 1.076× speedup
- Does the `asydis` toolkit provide effective concurrency?
  - Checked `unsafe` buffer access is much more efficient than naive mutex
- Are there significant synchronization costs for async apps?
  - At ms precision there is no observable cost for 4 tasks vs 1 task

# Conclusion

Does `asydis` make an operating system obsolete?

- For graphical applications, `asydis` can introduce concurrency and manage resource access without an OS
- Computation overlap can decrease runtime, task switching is efficient, **but**

# Conclusion

Does `asydis` make an operating system obsolete?

- For graphical applications, `asydis` can introduce concurrency and manage resource access without an OS
- Computation overlap can decrease runtime, task switching is efficient, **but**

Limitations:

- No direct comparison to OS-based concurrency
- No standardised benchmarks
- An OS is more than a thread scheduler and a MMU

# References


[1] N. D. Matsakis and F. S. Klock, “The rust language,” *Ada Lett.*, vol. 34, no. 3, pp. 103–104, Oct. 2014, doi: 10.1145/2692956.2663188.

[2] W. Bugden and A. Alahmar, “Rust: The programming language for safety and performance,” *arXiv preprint arXiv:2206.05503*, 2022.

[3] “Embassy (Project Website).” Accessed: May 14, 2025. [Online]. Available: https://embassy.dev/

[4] J. Waples and R. Fuest, “embedded-graphics: Embedded graphics library for small hardware displays.” [Online]. Available: https://github.com/embedded-graphics/embedded-graphics

[5] “Espruino The Bangle.js Support Library.” Accessed: May 16, 2025. [Online]. Available: https://www.espruino.com/Reference#Graphics

[6] “Light and Versatile Graphics Library (LVGL).” Accessed: May 14, 2025. [Online]. Available: https://lvgl.io/

# Conclusion

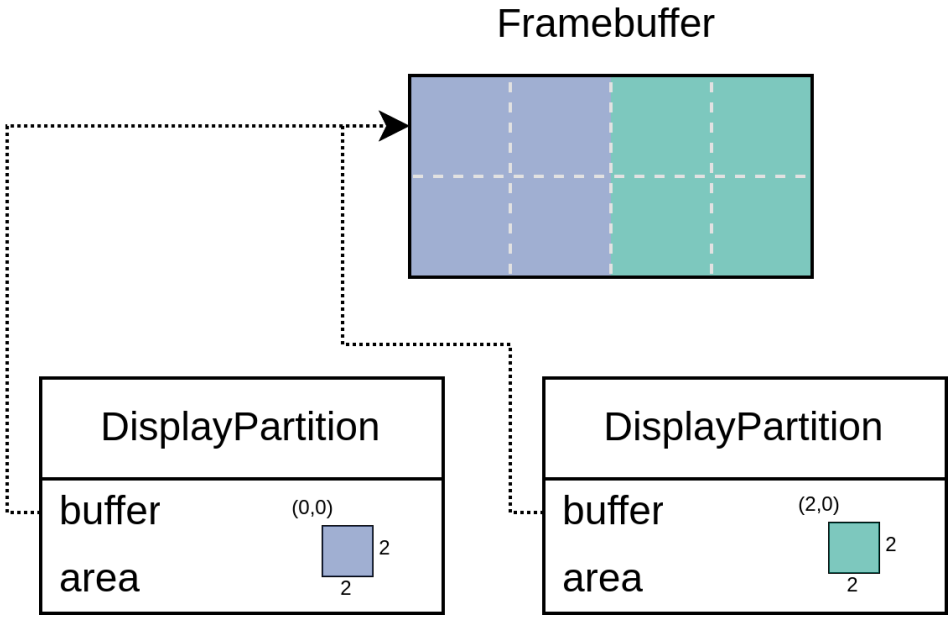
Does async Rust make an operating system obsolete?

- For graphical applications, `asydis` can introduce concurrency and manage resource access without an OS
- Computation overlap can decrease runtime, task switching is efficient, **but**

Limitations:

- No direct comparison to OS-based concurrency
- No standardised benchmarks
- An OS is more than a thread scheduler and a MMU

# Filtering for draw operations `unsafe`



# Filtering for draw operations `unsafe`