OSDI 25
19th USENIX Symposium on Operating Systems Design and Implementation

OS Rendering Service Made Parallel with Out-of-Order Execution and In-Order Commit

Yuanpei Wu, Dong Du, Chao Xu, Yubin Xia, Yang Yu, Ming Fu, Binyu Zang, Haibo Chen

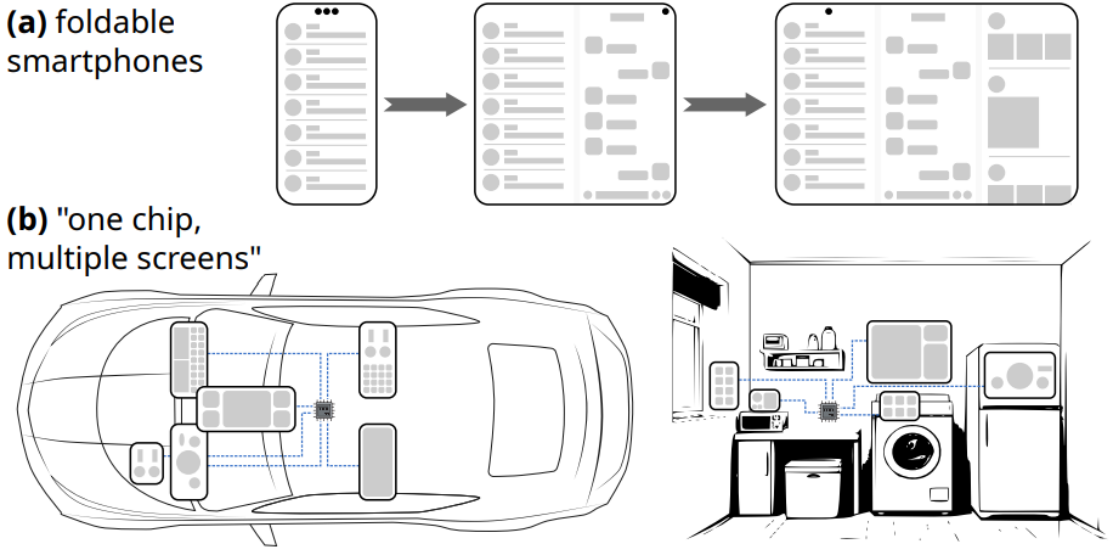
IPADS, Shanghai Jiao Tong University
Fields Lab, Huawei Central Software Institute

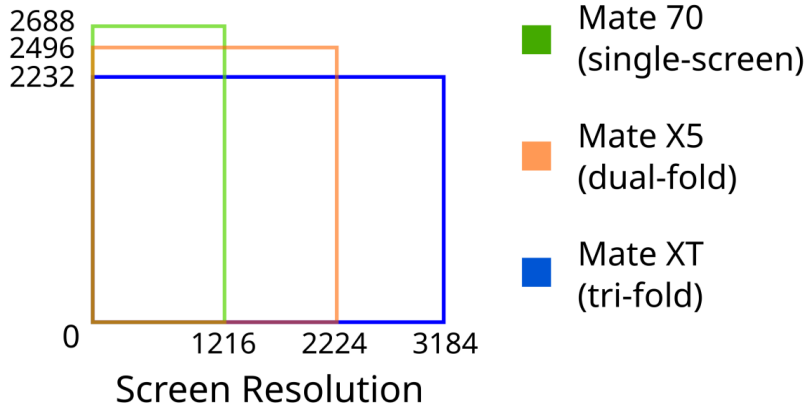
Smart-Device Graphics

OS Rendering Service powers the visually stunning graphical user interfaces.

Demands for Rendering Scalability

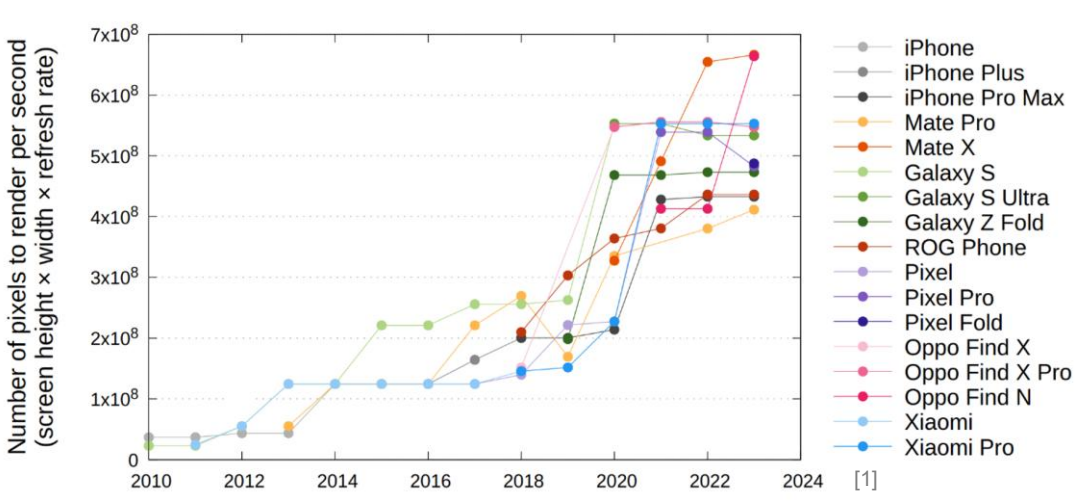
- Recent **foldable smartphones** and **one-chip-multiple-screen setups** significantly amplify the rendering workload and demand scalable rendering.
- E.g., while the Huawei Mate 70 (single-screen), X5 (dual-fold), and XT (tri-fold) use similar chipsets, the latter two need to render **70%** and **117%** more pixels per frame.



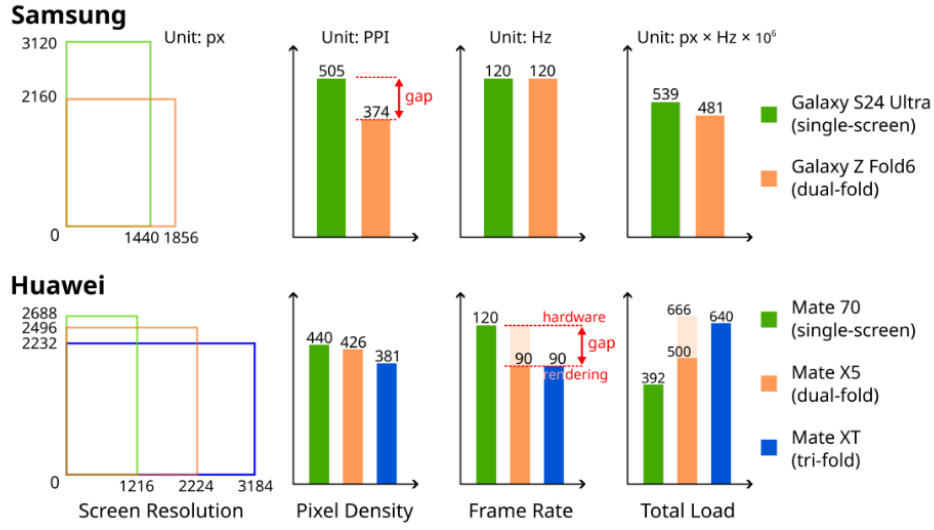


Rendering Scalability: A Continuous Trend

- Displays are continuously evolving to become **more immersive** (with larger screen areas), **clearer** (with higher pixel densities), and **smoother** (with higher frame rates).
- However, performance limits often force **trade-offs and compromises** — e.g., lower pixel densities or frame rates on foldable smartphones.



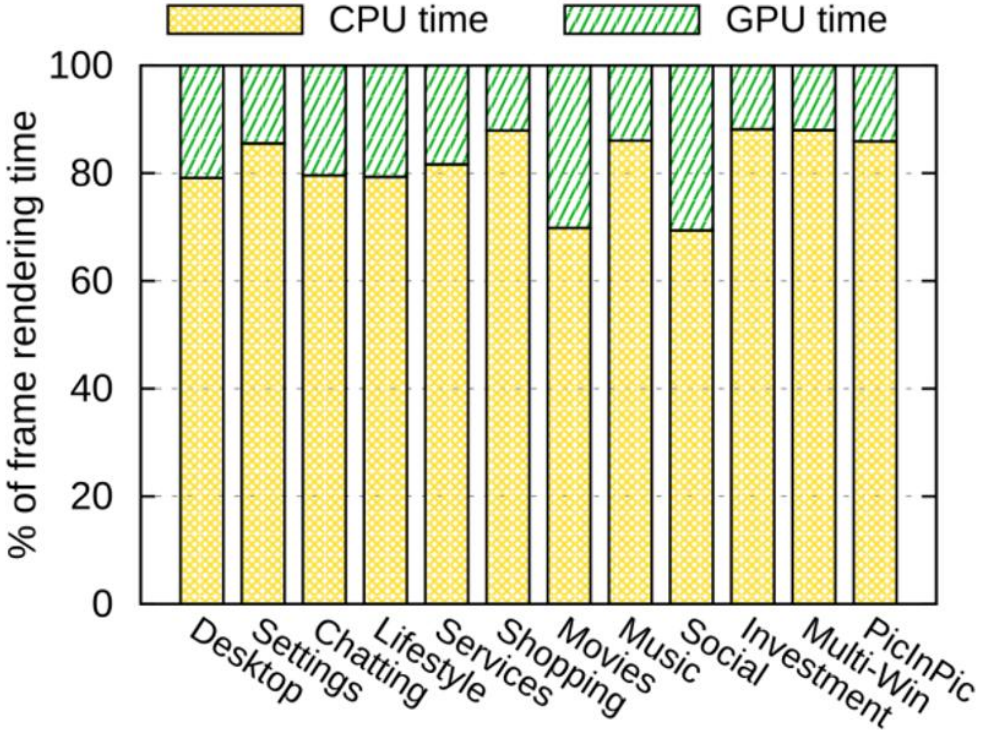
[1]

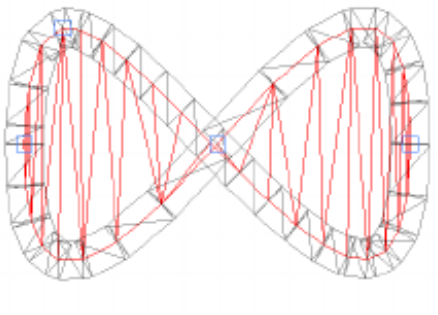


[1] Yuanpei Wu et al. D-VSync: Decoupled Rendering and Displaying for Smartphone Graphics. ASPLOS 2025.

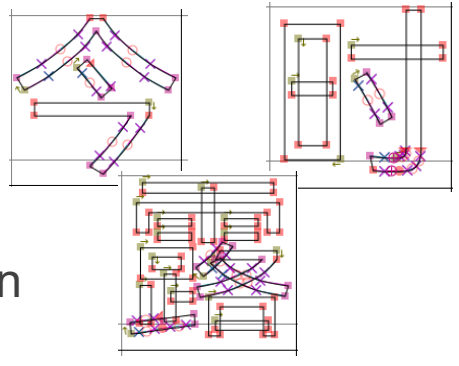
Observation-1: CPU bottlenecks OS 2D rendering

In 2D rendering, **CPU accounts for 82% of the frame rendering time**. As GPU and CPU run asynchronously, CPU is the performance bottleneck that impacts frame rate.





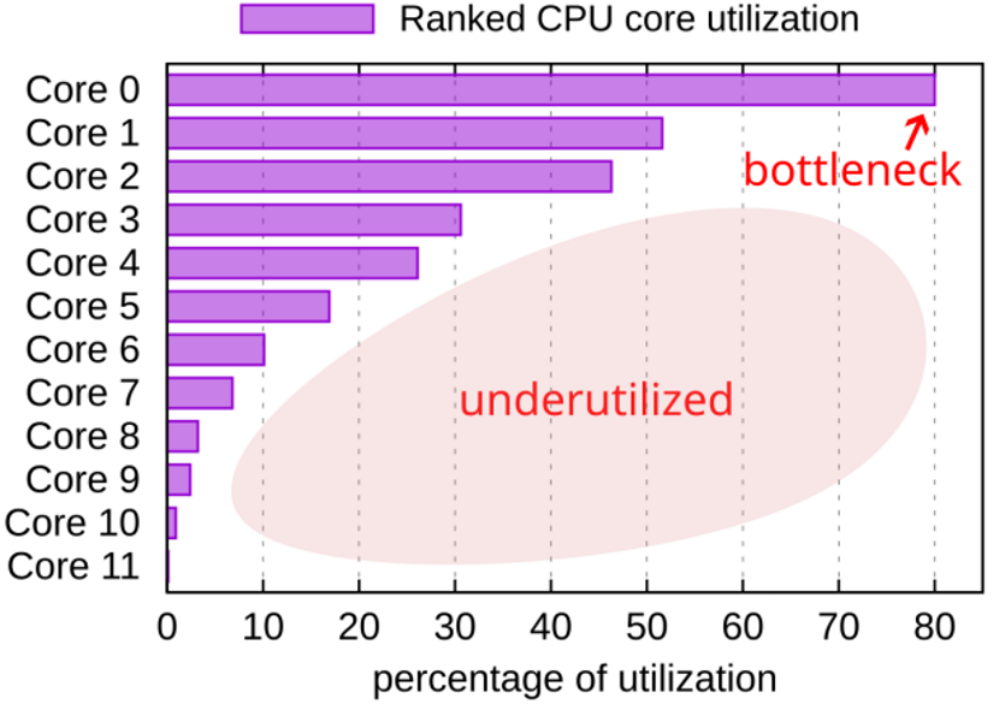
Tessellation & Triangulation



Glyph (Text) Rasterization

Observation-2: limited CPU multi-core utilization

Current state-of-the-art rendering procedure follows **a fixed-thread sequential model**, restricting the CPU multi-core utilization.



Render thread monopolizes 80% single-core utilization

While most cores (9 out of 12) are idle and underutilized

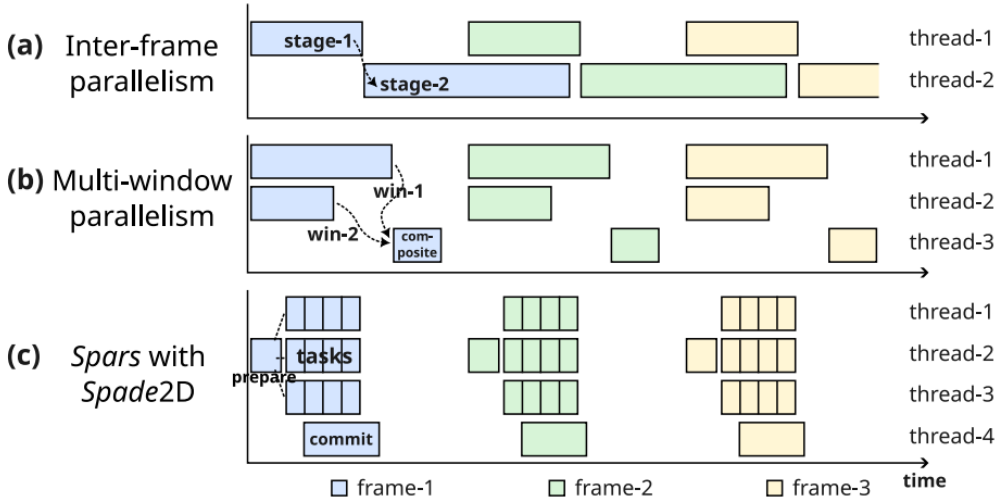
We argue that

*The next-generation rendering service should adopt a **parallel** architecture to address the scalable rendering demands.*

State-of-the-art Parallelization Efforts

Inter-frame parallelism and **multi-window parallelism** do not scale.

- Inter-frame: span at most 2~3 threads, with load imbalance and more latency.
- Multi-window: windowing abstraction required for parallelization, with load imbalance, extra memory, and composition costs.



	Applicable Scenarios	Extra Loads	Rendering Latency	Workload Fluctuation	Constant Heavy Loads
Baseline	100%	/	/	Frame Drops	Low Frame Rate
Inter-frame Parallelism	100%	Minor	Serious Influence	Frame Drops	Medium Frame Rate
Multi-Win Parallelism	<5%	Some	Some Influence	Frame Drops	Medium Frame Rate
D-VSync [1]	85%	Minor	No Influence	No Frame Drop	Low Frame Rate
Spars with *Spade*2D	100%	Minor	No Influence	Leveraging *D-VSync*	High Frame Rate

[1] Yuanpei Wu et al. D-VSync: Decoupled Rendering and Displaying for Smartphone Graphics. ASPLOS 2025.

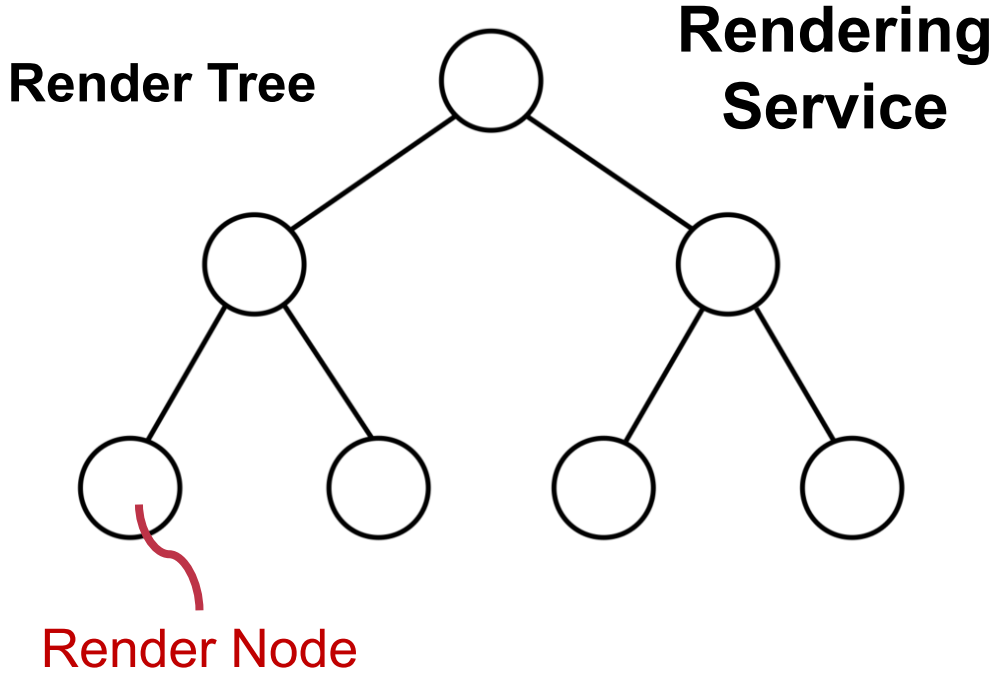
Background

Background: OS Rendering Service

Rendering service, as an OS service, renders the screen content **in a unified manner**.

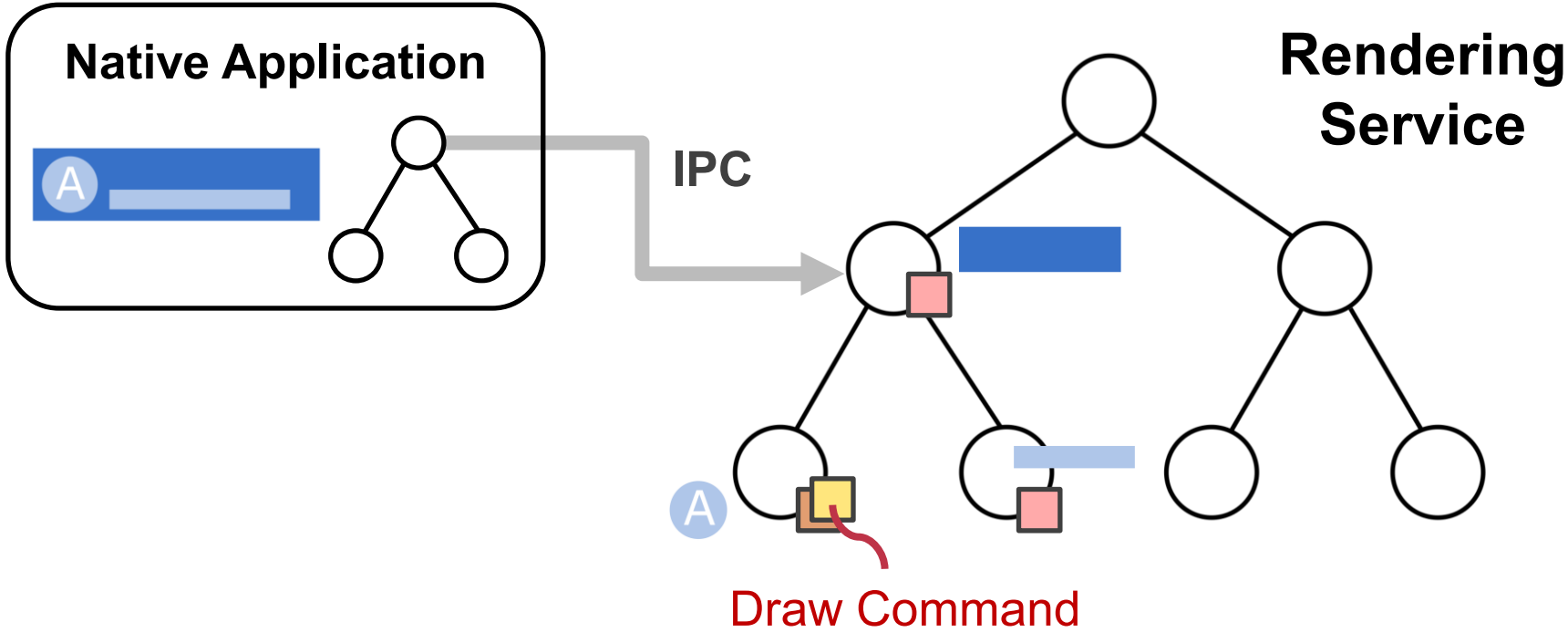
Background: Unified Render Tree

The **render tree** in the rendering service manages all the drawing information on the screen. Specifically, each render node contains the draw commands of a component.



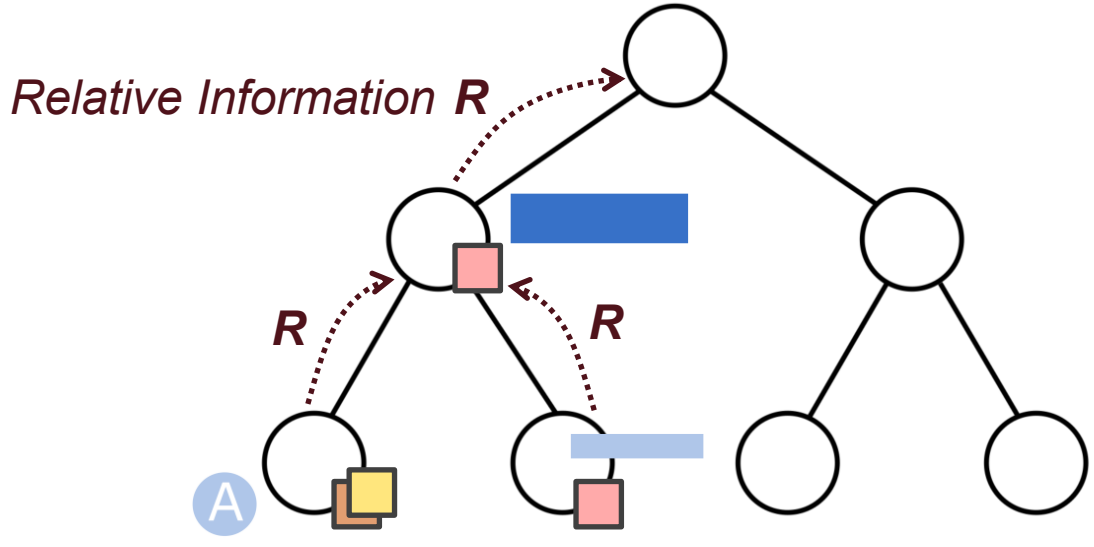
Background: Draw Command

Native applications synchronize **draw commands** to the rendering service for rendering.



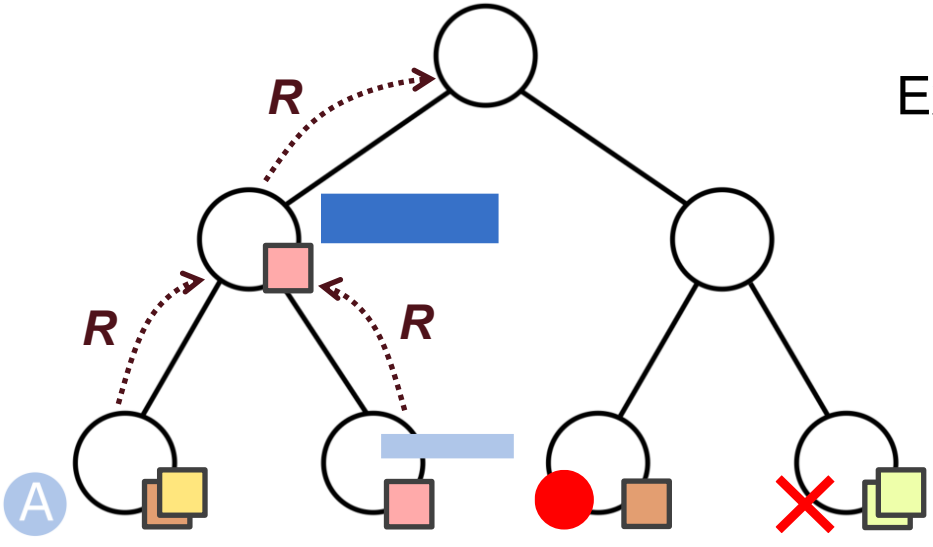
Background: Relative Information

Each render node only contains the **relative information (*R*)** with respect to its parent, such as the relative transform matrix (encoding translation, scaling, and rotation).

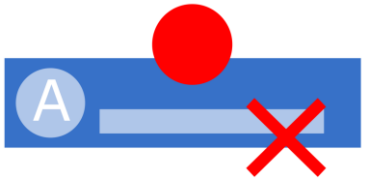


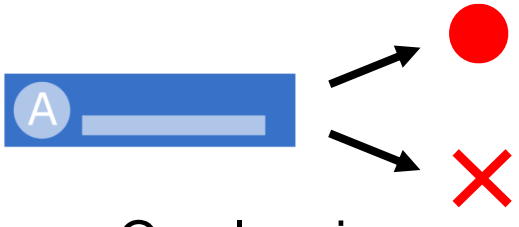
Background: Drawing Order

Each render node has its children sorted based on the **drawing order**, from back to front. Note that node drawing is not always commutative due to possible overlapping.



Expected Result:

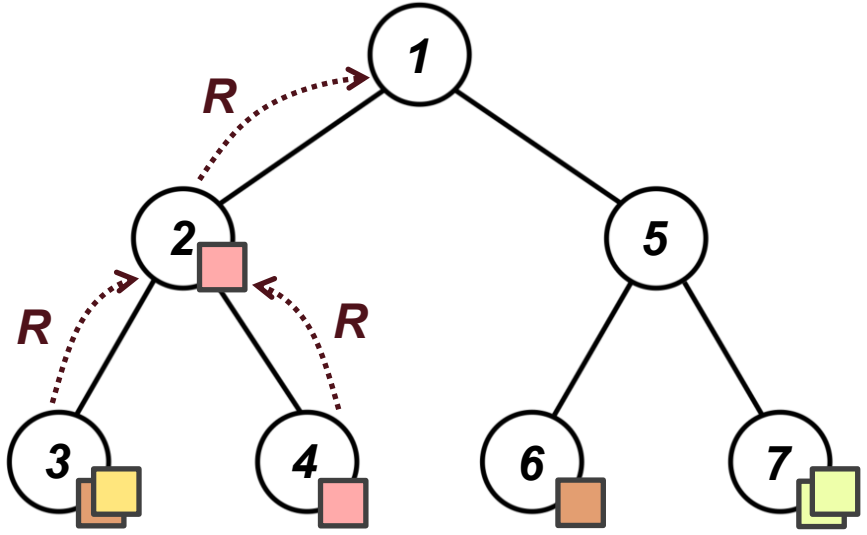




Overlapping Relations

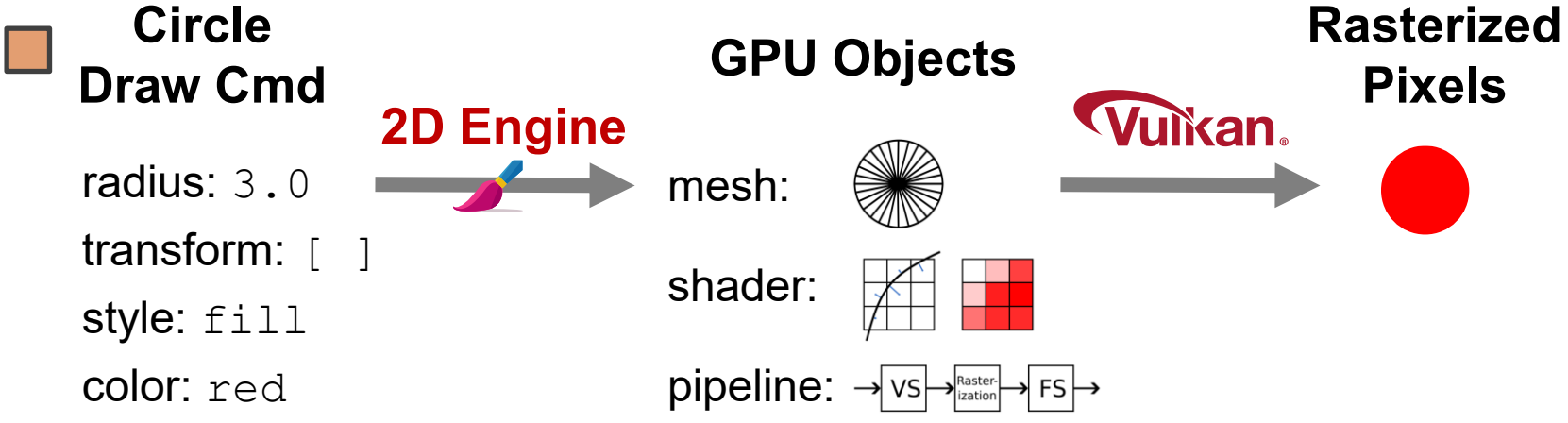
Background: Depth-First Traversal

A **sequential depth-first traversal** of the tree implicitly enforces the drawing order and invokes the 2D engine to convert a stack of relative information into absolute states.



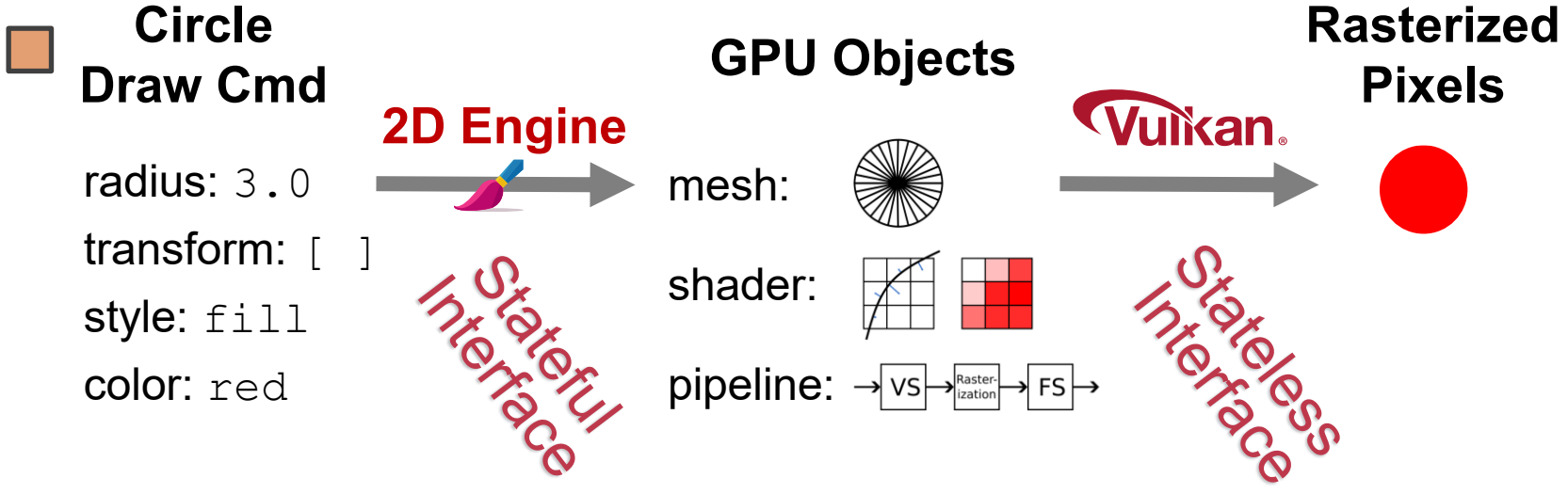
Background: 2D Engine

The **2D engine** translates the parameters of the draw commands into GPU objects like meshes, textures, and pipelines, which are later sent to the GPU for rasterization.



Background: Stateful Interface

Current 2D engines utilize **stateful interfaces**, where states can be kept internally, so new draw commands can rely on past draw commands.



Parallelization Challenges

A scalable OS rendering service with **fine-grained parallelization** faces 3 challenges:

C1: State Dependency

Parallelization necessitates **complete and absolute states** for each rendering task, while each node in the render tree stores only the relative information to its parent, requiring a depth-first traversal.

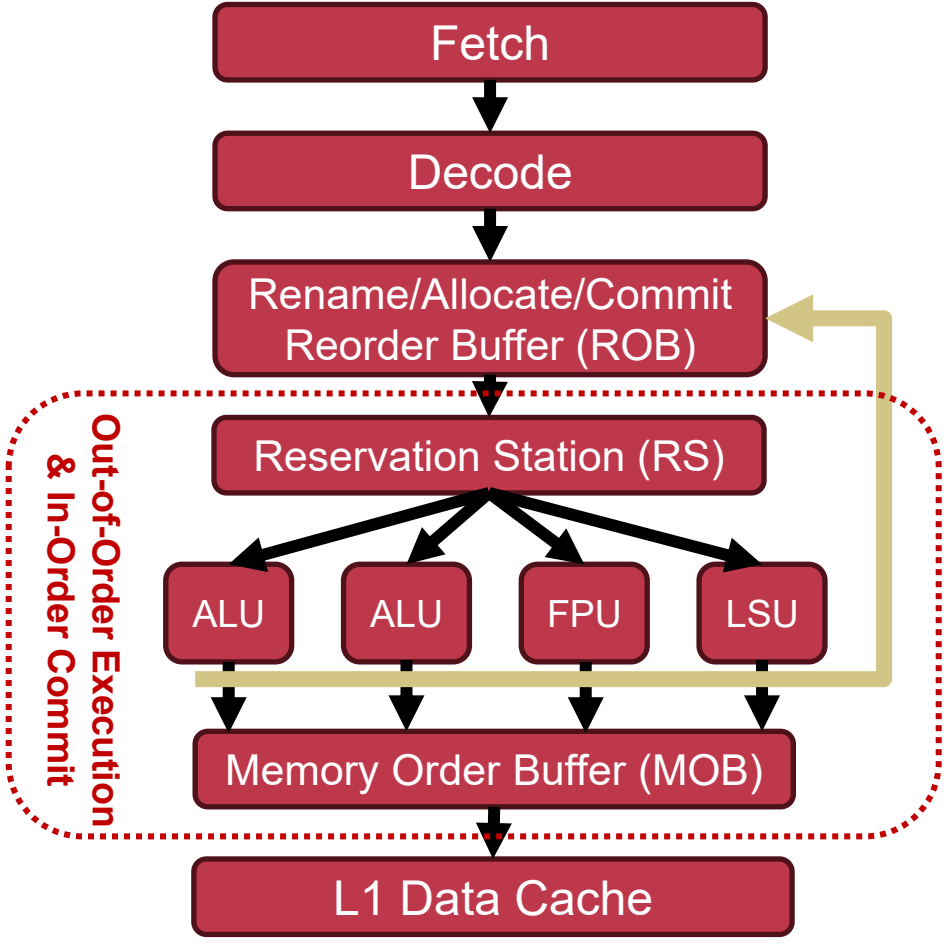
C2: Drawing Order Dependency

Parallelization must explicitly maintain **overlapping relations** for drawing-order correctness. When two graphics primitives overlap, the background must always be drawn before the foreground.

C3: Interface Dependency

Parallelization has **stateless interface**, while current 2D engines utilize a stateful API (e.g., Skia Canvas) for custom-rendering applications and internal stateful optimizations like command batching.

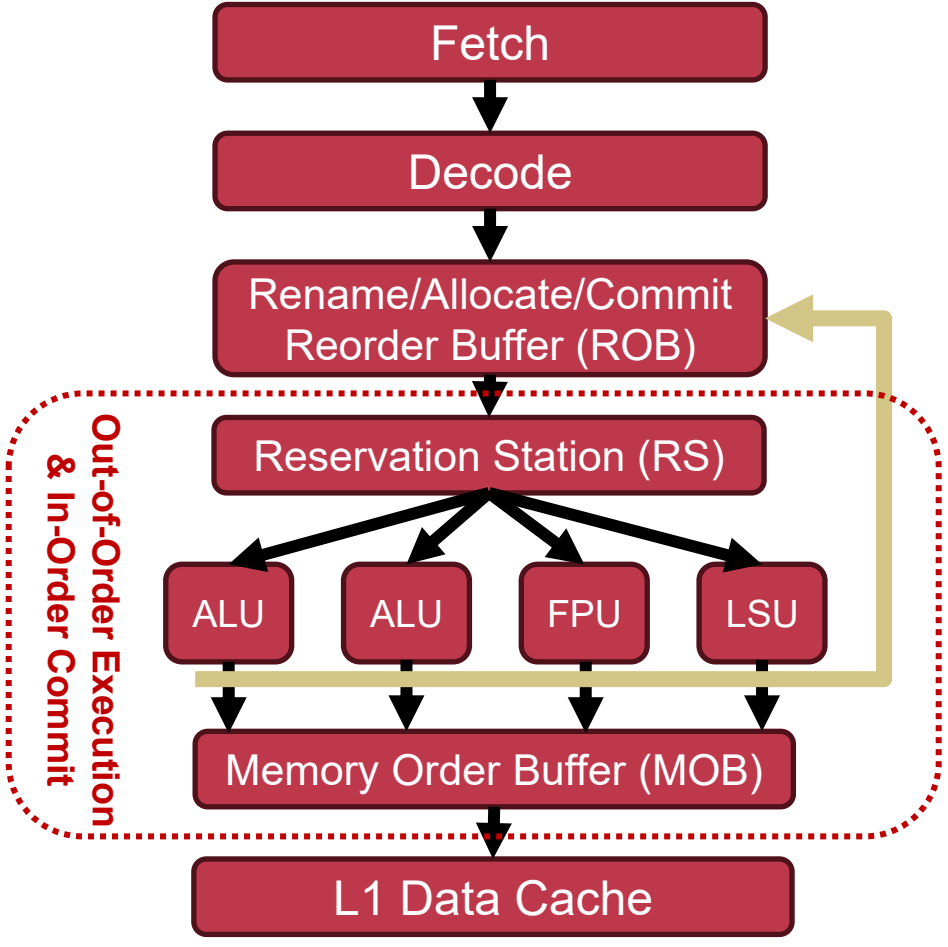
Insight: Out-of-Order Execution with In-Order Commit



Modern CPUs use **out-of-order execution with in-order commit**:

- Instruction dependencies are analyzed to identify which ones can run in parallel.
- Independent instructions are dispatched to available units (e.g., ALUs, FPUs) out of order.
- After execution, results wait in the Reorder Buffer (ROB) until all earlier instructions have been committed in program order.

Insight: Out-of-Order Execution with In-Order Commit



We observed **the analogy**:

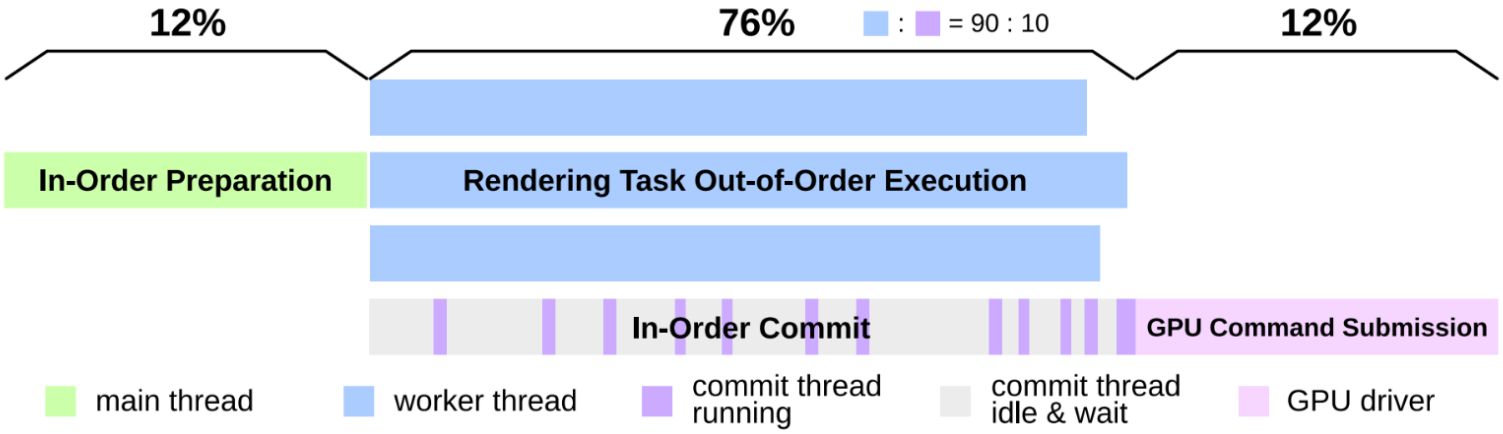
- Draw command → CPU instruction
- CPU multi-core → ALU/FPU/LSU
- State dependency → operand dep.
- Drawing order → program order

Why not adopt **a similar strategy**?

- In-Order preparation
- Out-of-Order execution
- In-Order commit

Spars: A Scalable and Parallel OS Rendering Service

The Three-Stage Design of Spars



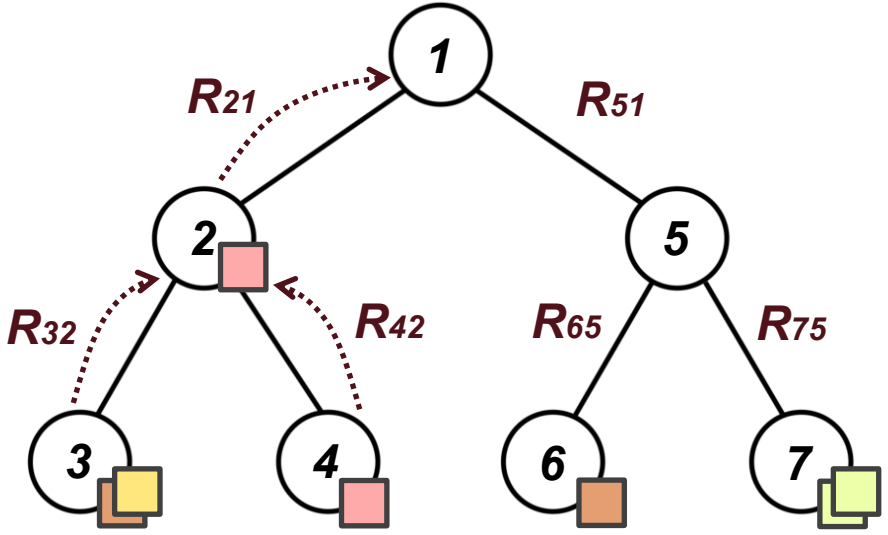
In-order preparation (main thread): the render tree is traversed once in a depth-first manner to prepare **self-contained rendering tasks** (put into a single-producer multi-consumer SPMC pool) and their **overlapping relations**.

Out-of-order execution (worker threads): execute rendering tasks retrieved from the SPMC pool. Each task can be **independently** and **OoO executed**. The generated GPU objects are packed and put into a MPSC pool.

In-order commit (commit thread): bind the objects into the GPU command in order.

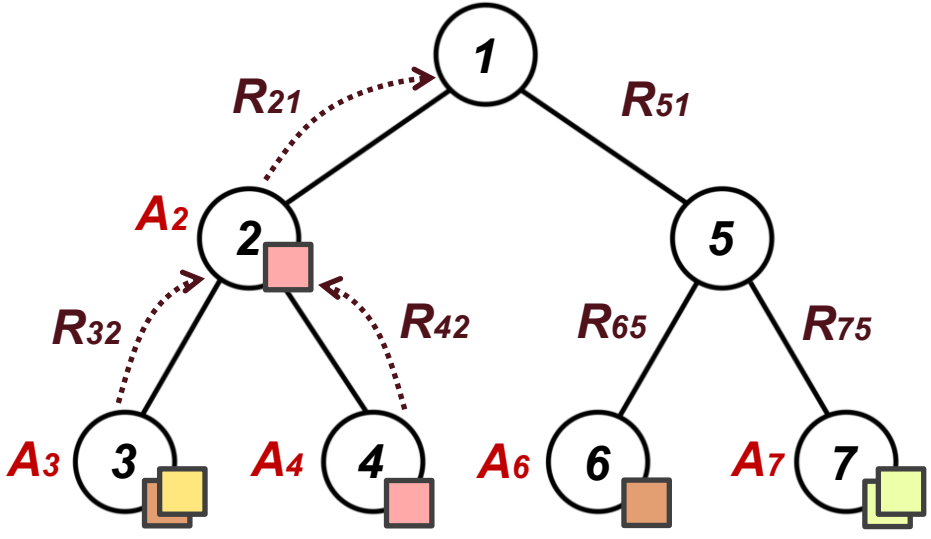
Revisit State Dependency

State dependency (C1): current nodes only store relative information ***R***.



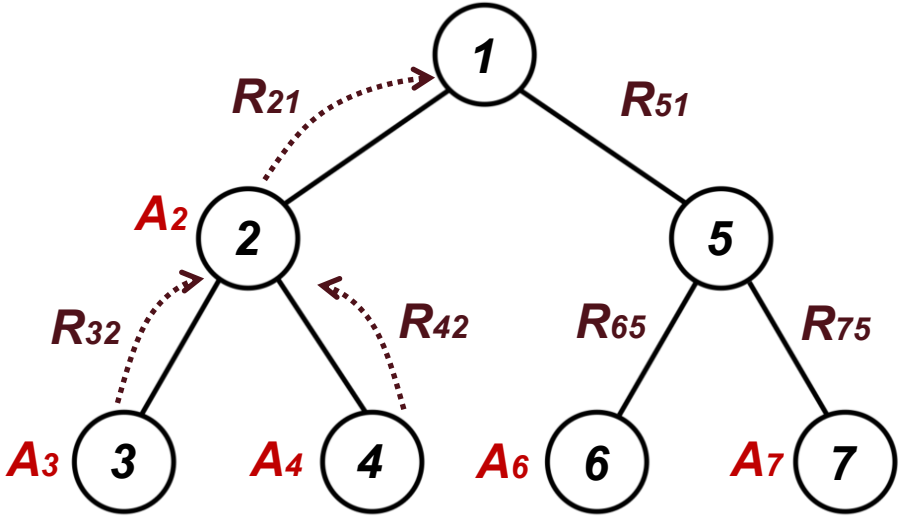
In-Order Preparation: a dry run for complete states

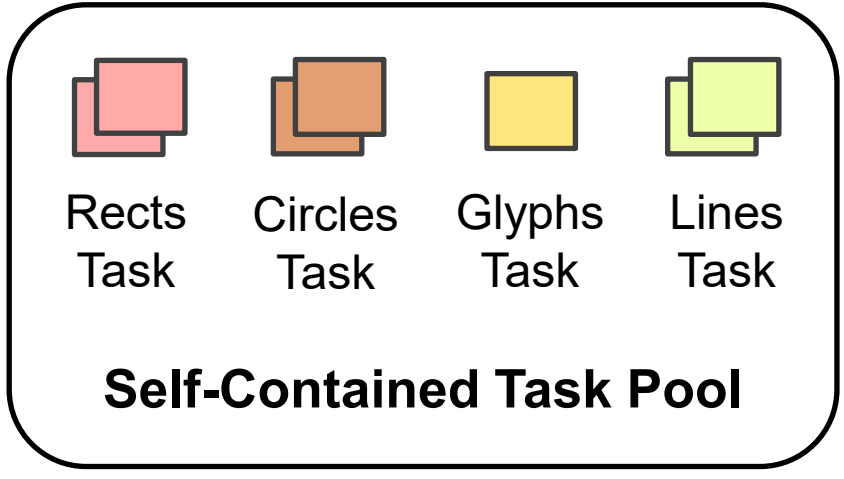
To tackle C1, during preparation, Spars traverses the render tree once **in a dry run** (without doing any rendering), and records the absolute information ***A*** — relative to the screen coordinates or the absolute zero.



In-Order Preparation: generate self-contained tasks

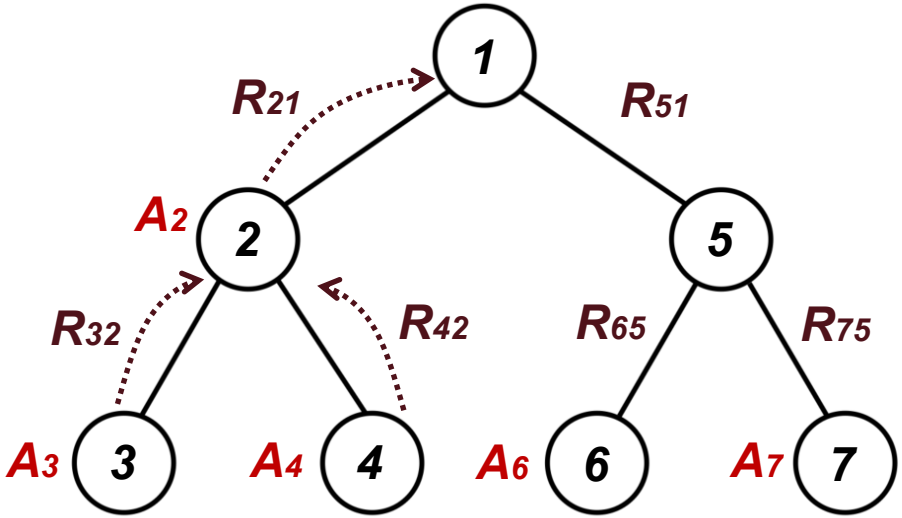
Meanwhile, the draw commands with complete states are batched and encapsulated into **self-contained rendering tasks**, put into a single-producer multi-consumer (SPMC) pool.

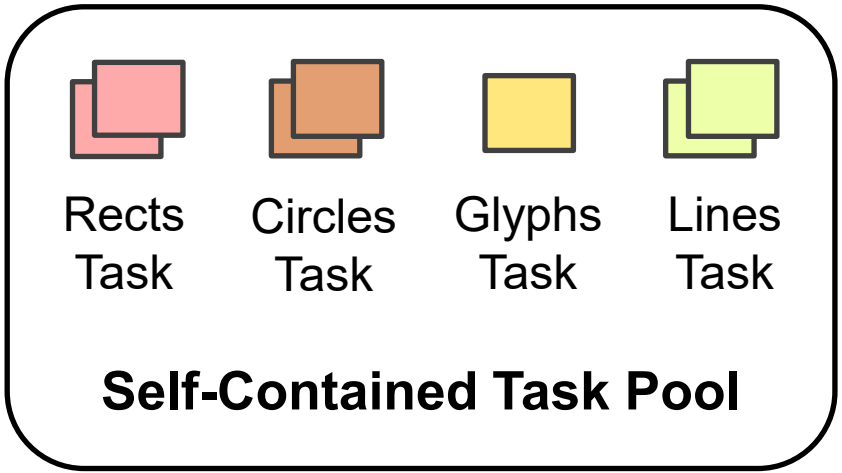




Revisit Drawing Order Dependency

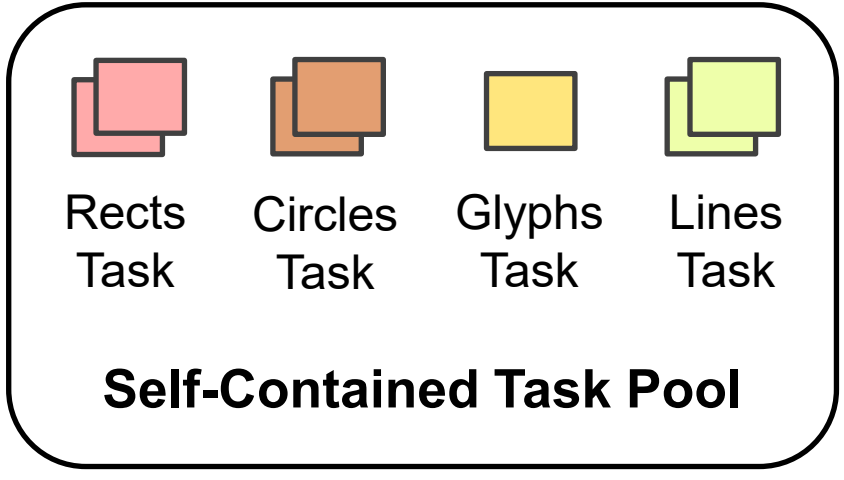
Drawing order dependency (C2): drawing order must be maintained for correctness.

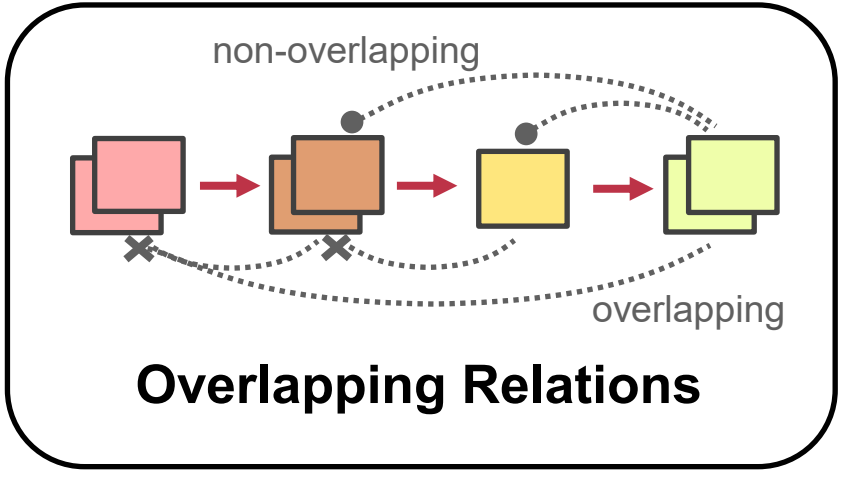




In-Order Preparation: construct overlapping relations

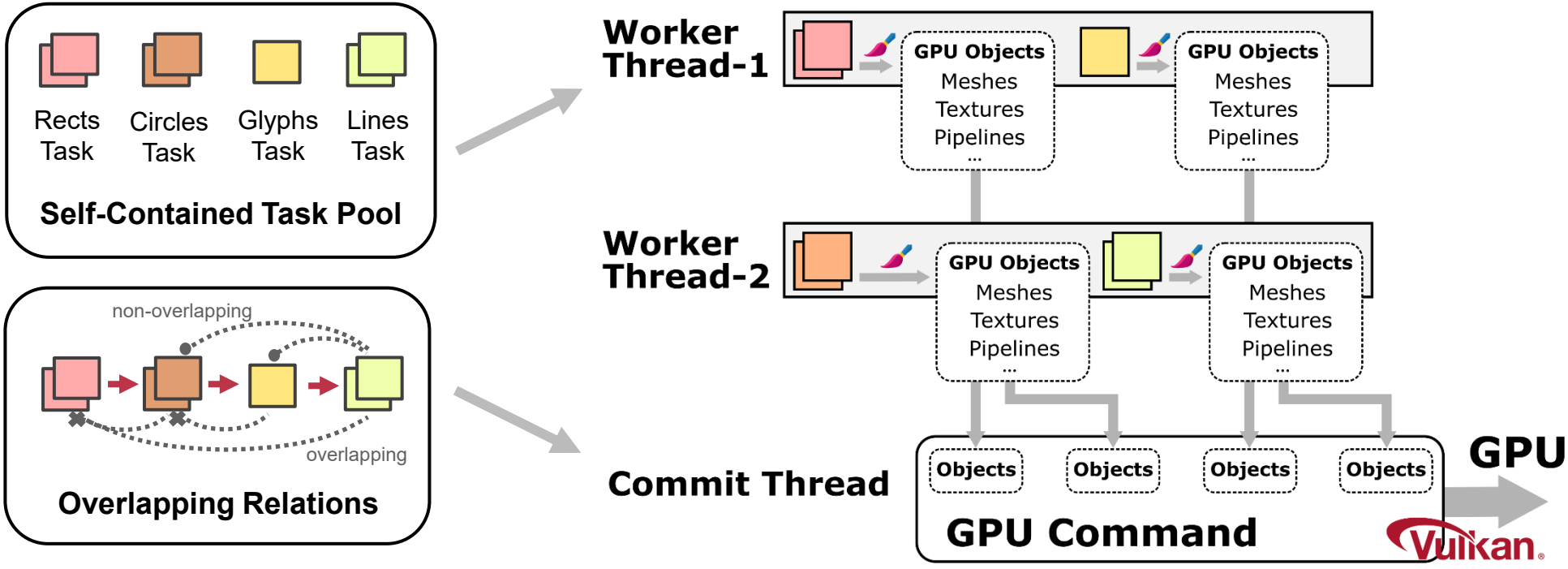
To tackle C2, Spars **chains** the rendering tasks together for a naïve drawing order and manages their **drawing regions** through axis-aligned bounding boxes (AABBs) for each task. The chain is later synchronized to the commit thread from the main thread.





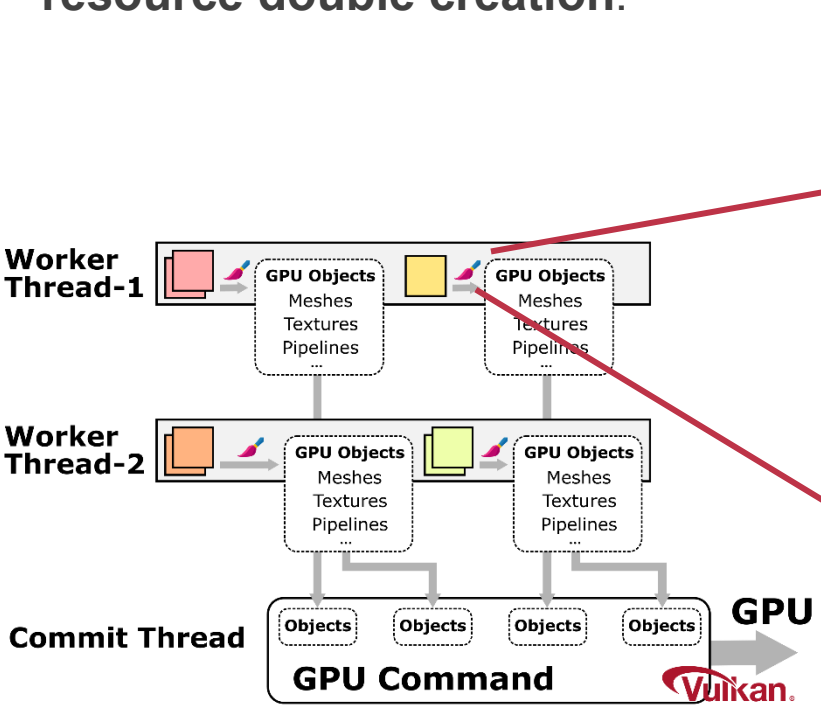
Out-of-Order Execution on multiple worker threads

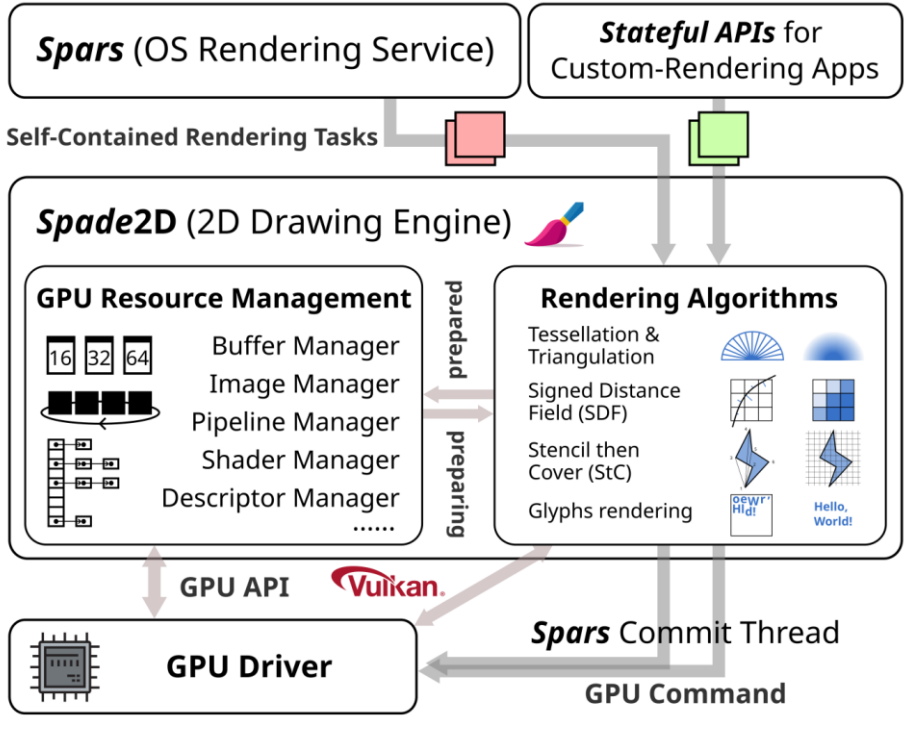
The tasks are dispatched to **multiple worker threads** for out-of-order execution. As tasks are **independent**, any thread can process any task, using a novel 2D engine.



Out-of-Order Execution with a stateless 2D engine

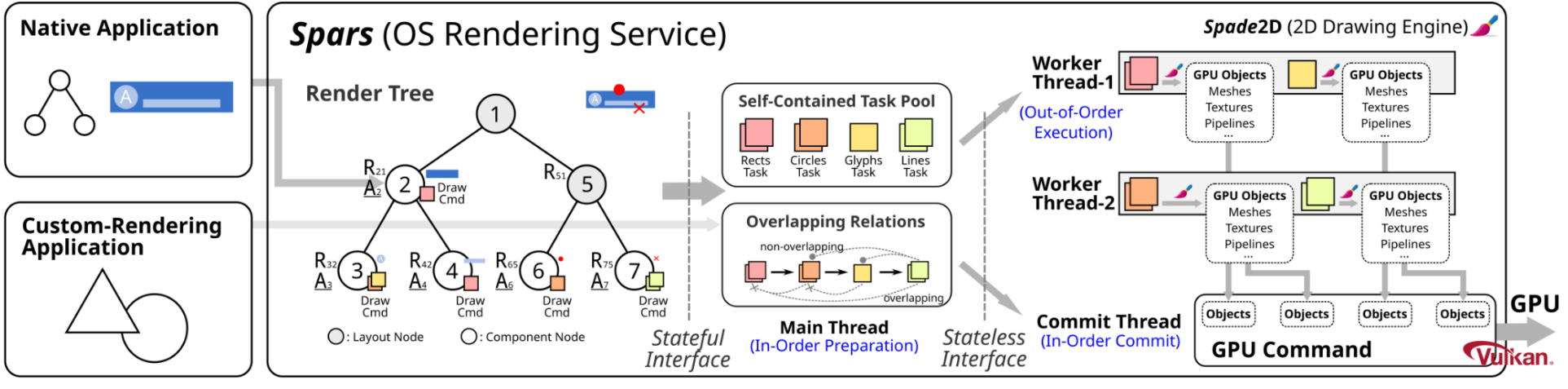
Spade2D adopts stateless APIs and accepts inputs in the form of self-contained tasks. GPU resources in Spade2D are managed by **thread-safe** managers that **prevent resource double creation**.





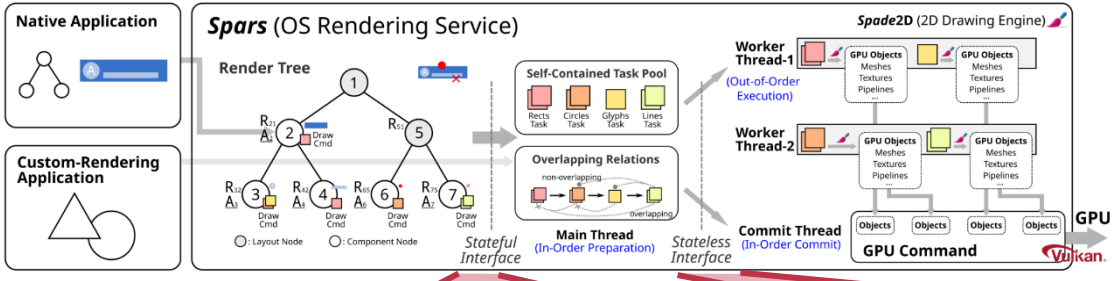
Revisit Interface Dependency

Interface dependency (C3): current rendering APIs for custom-rendering applications are stateful. 2D engines also rely on internal state-based optimizations.



Out-of-Order Execution ensuring compatibility

To tackle C3, Spars decouples the rendering procedure into **stateful and stateless phrases**: the former maintains a consistent interface for custom-rendering applications and does state-based optimizations, while the underlying Spade2D engine is stateless.

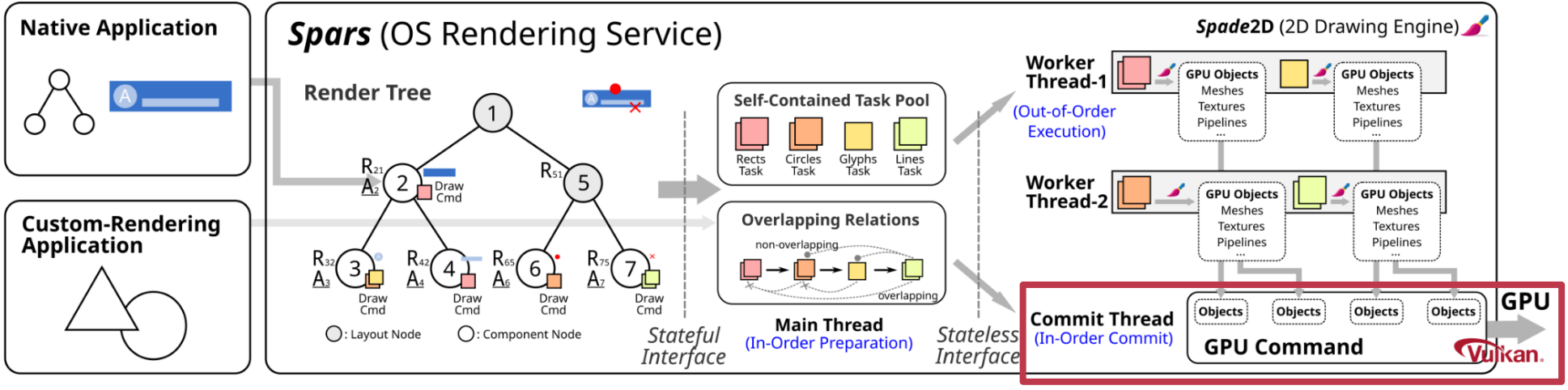



```
1  ▶Stateful APIs◀
2  /* state-update commands */
3  void Canvas::translate(float x, float y);
4  void Canvas::rotate(float degrees);
5  void Canvas::clip(Shape &shape);
6  void Canvas::save();
7  void Canvas::restore();
8  /* draw commands */
9  void Canvas::drawRect(Rect &rect);
10 void Canvas::drawCircle(Circle &circle);
```

```
11 ▶Stateless APIs◀
12 /* self-contained tasks */
13 struct RectsTask {
14    int primitiveCount;
15    Rect *rects; // a batch of rects
16    State *states; // the corresponding
      ↪ complete and absolute states
17 };
18 GpuObjects Spade2D::draw(RectsTask &task);
19 GpuObjects Spade2D::draw(CirclesTask &task);
```

In-Order Commit based on overlapping relations

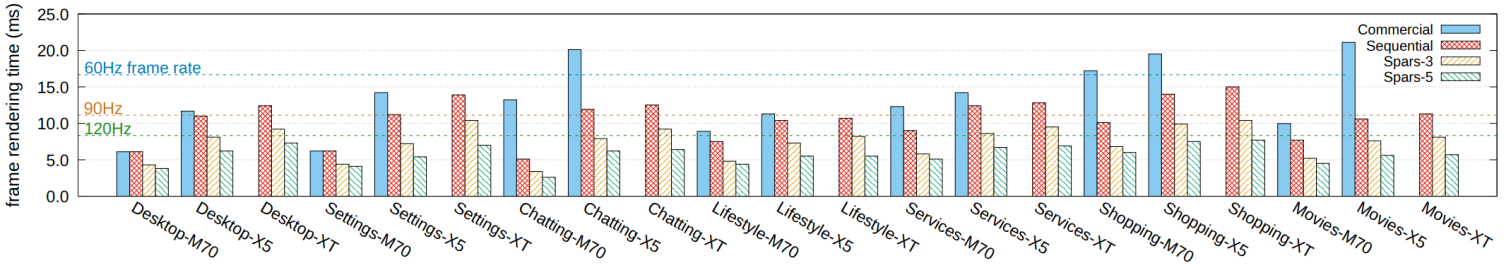
The commit thread checks if it is possible to commit (bind) the received completed task (GPU objects) into the GPU command, based on the **task chain** and **AABBs** — if the task is the current head or has no overlapping with the previous uncommitted tasks.

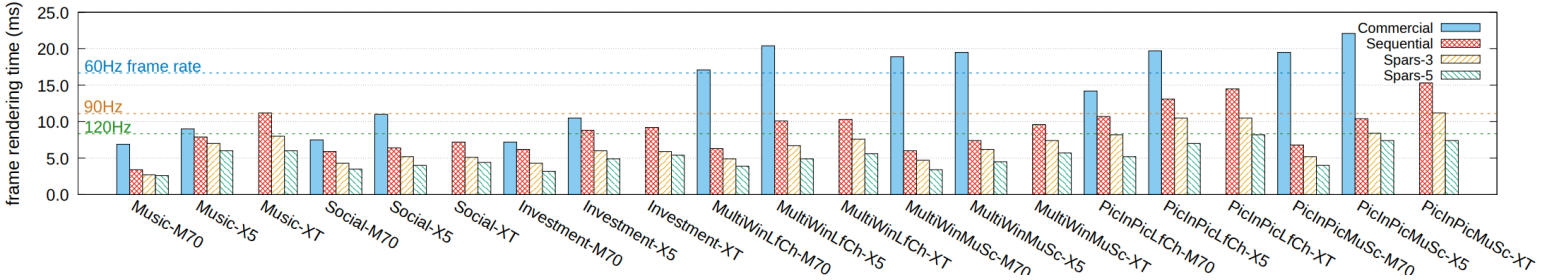


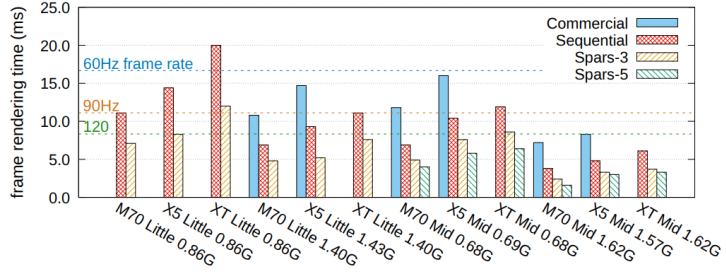
Evaluation

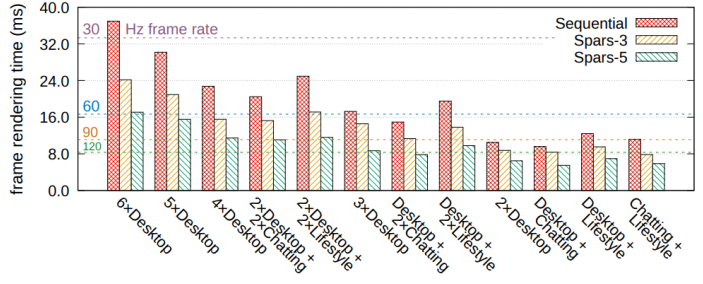
Frame rate improvement 1.76× ~ 1.91× with 5 workers

42 representative smartphone scenarios on **Mate 70, X5, XT**, and **multi-screen** configs.



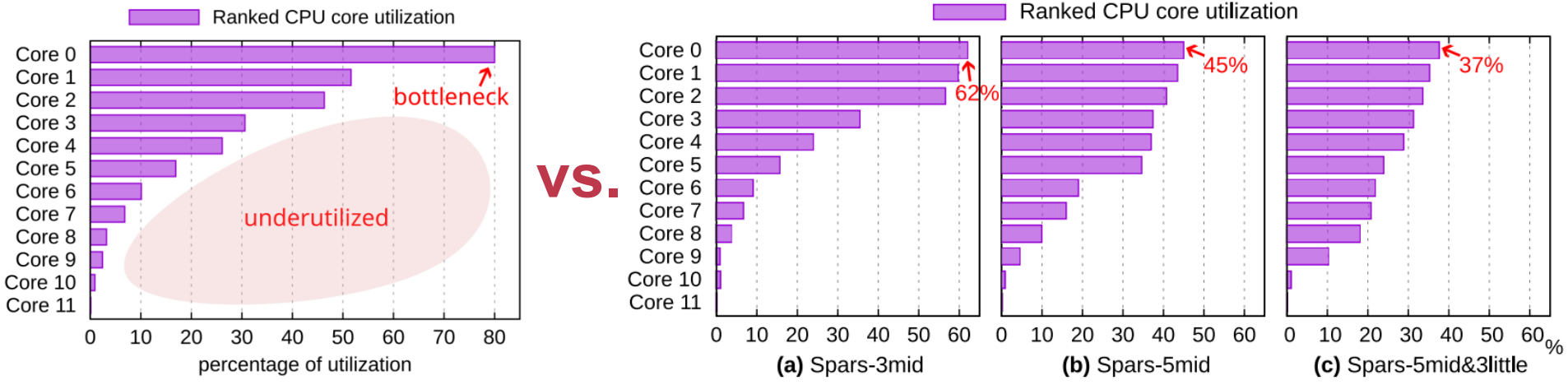






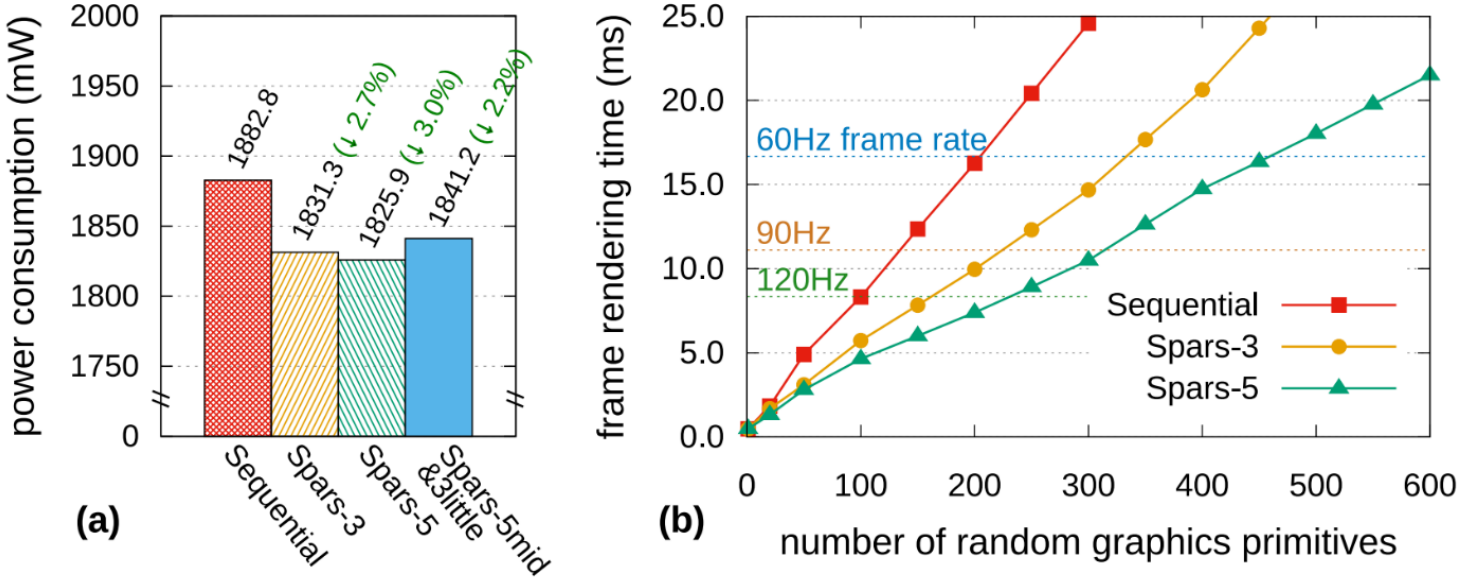
Multi-core CPU utilization is more balanced

Under the same frame rate, Spars optimizes **the utilization of multi-core hardware** in smart devices and ensures a more balanced workload across cores, preventing any single core from becoming the frame rate bottleneck.



Power Consumption & Scalability of Rendering

Under the same stable frame rate, Spars can reduce **(a) the power consumption** by 3.0%, or increase **(b) the number of graphics primitives**, i.e., visual quality, by 2.31×.



Conclusions

- We present **Spars with Spade2D**, a next-generation scalable and parallel OS rendering service & drawing engine for smart devices.
- Spars untangles **state dependency**, **drawing order dependency**, and **interface dependency**, allowing **out-of-order execution with in-order commit** of self-contained rendering tasks.
- Evaluations demonstrate that Spars **substantially enhances frame rates** on emerging foldable smartphones and multi-screen configurations, efficiently leveraging multiple CPU cores while reducing power consumption and scaling the number of graphics primitives.

IPADS
INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

HUAWEI

Thanks!