Component-based OS Design for Dependable Cyber-Physical Systems

Gabriel Parmer, Runyu Pan, Yuxin Ren, Phani Kishore Gadepalli, Wenyuan Shao

The George Washington University
gparmer@gwu.edu
Gravitational Pull in CPSes

**Embedded System** Priorities
- Safety
- Dependability
- Predictability
Gravitational Pull in CPSes

**Embedded System** Priorities
- Safety
- Dependability
- Predictability

**Complex Software** Priorities
- Functionality
- Development Speed
- Throughput
- Security
Gravitational Pull in CPSes

**Embedded System** Priorities
- Safety
- Dependability
- Predictability

**Complex Software** Priorities
- Functionality
- Development Speed
- Throughput
- Security

Cyber-Physical Systems
Gravitational Pull in CPSes

**Embedded System** Priorities
- Safety
- Dependability
- Predictability

**Complex Software** Priorities
- Functionality
- Development Speed
- Throughput
- Security
POSIX++

• Enduring foundation
  - Pervasive for high-functionality libraries/apps
  - Extended in many directions

• Dominates high-functionality SW development
  - Move fast and break things
Beyond POSIX

• Lot of *warts*
  – fork, signals, allocation, copies...

• Practical systems: *fast moving targets*
  – ...or vulnerable

• *Insufficient*
  – Linux-specific extensions commonplace

• Implied *complexity*
  – Little ability to scale down
RTOSes

• Simple(-ish) implementations
  - Some support for more complicated standards

• Standards:
  ARINC 653
Beyond RTOSes

- Limited software availability
- Inefficient use of multi-core
- Large ecosystem
  - With difficult code reuse story
- Silos
Choose your system

High Assurance
Predictable
Simple

ARINC 653

Lower Assurance
Tenuous WCET bounds
Complex
Choose your system

Pervasive problem:

A single

- buggy line of code
- bit flip in a ds
- cache-line bouncing
- unbounded loop

threatens full-system dependability

High Assurance
Predictable
Simple

Lower Assurance
Tenuous WCET bounds
Complex
Isolation:
Still the remedy for complexity

• Temporal Isolation
  – Core partitioning, Threading
  – TDMA, Rate-limiting Servers
• Spatial Isolation
  – HW Memory Access Cntl
• Abstract Resource Isolation
  – Access control
Isolation: Only a Partial Solution

• Temporal Isolation
  – Shared resources
  – Shared HW

• Spatial Isolation
  – Shared Services

• Abstract Resource Isolation
  – Discretionary Access Cntl
Modern “Standardization”

Productivity via package/dependency mgrs

- **OS Distributions**
  - apt, rpm/yum, brew, ...

- **Language ecosystems**
  - npm, pip, **cargo**, ...

```
[package]
name = "mkimg"
version = "0.1.0"
authors = ["Gabe Parmer <gparmer@gwu.edu>"]

[dependencies]
xmas-elf = "0.6.0"
toml = "0.4"
serde = "1.0"
serde_derive = "1.0"
pipers = "1.0.0"
tar = "0.4"
```

Diagram:
- Application
  - lib₀
  - lib₁
  - lib₂
  - lib₃
  - lib₄
Goal: Functionality + Dependability + Productivity

Component-based system design

- Code, data
- Export APIs ($E_1 = \{ \text{fn, } \ldots \} )$
- Explicit dependencies
- Unit of reuse & isolation
Goal: Functionality + Dependability + Productivity

Component-based system design

- Code, data
- Export APIs ($E_1 = \{ \text{fn}, \ldots \}$)
- Explicit dependencies
- Unit of reuse & isolation

- Minimize functionality for the necessary APIs
- Strong, fine-grained isolation
- Libraries of shared functionality
Goal:
Functionality + Dependability + Productivity

- Components *compose* system

- Limit *scope* of
  - compromise
  - fault
  - unpredictability
Goal: Functionality + Dependability + Productivity

- Components compose system

- Limit scope of
  - compromise
  - fault
  - unpredictability
Cross Cutting Concerns

Adversarial economics of existing systems

Security
- A single line of code $\rightarrow$ full system compromise

Predictability
- A single unbounded loop $\rightarrow$ unpredictable exec

Dependability
- A single fault/bit flip $\rightarrow$ full system failure

Scalability
- A single bouncing cache-line $\rightarrow$ scalability
Requirements for CPSOSes

• Efficient Isolation
  – Constrain scope of impact of any one line/byte
  – Spatial, temporal, I/O

• Policy customization*
  – Various assurance-levels, functional requirements
  – Multiple policies co-exist with designed interference

• Composability WRT cross-cutting concerns
  – Components must compose

* See Bjorn's talk: we have no opinions ;-)
Example Challenge I: \textit{IPC}

- Communication between components
  - Efficient?
  - Predictable?
  - Policy interactions with kernel?
Asynchronous IPC

Scheduling analysis is dependency-aware

- Holistic scheduling
- Self-interference
- $f$(scheduling policy)
Asynchronous IPC

Predictable component + predictable component = predictable system

Scheduling analysis is dependency-aware
- Holistic scheduling
- Self-interference
- $f$(scheduling policy)
Synchronous IPC between Threads

Scheduling analysis is dependency-aware
• Resource sharing
• Requires in-kernel priority inheritance machinery
Synchronous IPC between Threads

Predictable component + predictable component = predictable system?

Scheduling analysis is dependency-aware
- Lock-aware scheduling
- Requires in-kernel priority inheritance machinery
Thread Migration

\[ \text{prio}_x \]
Thread Migration

$prio_x +$ predictable component $=$ predictable system
Example Challenge II: *Kern Sync*

Can kernel APIs

- Limit the scalability of components?
- Cause interferences between components?
Example Challenge II: *Kern Sync*

Can kernel APIs

- Limit the scalability of components?
- Cause interferences between components?
Example Challenge II: *Kern Sync*

Can kernel APIs

- Limit the scalability of components?
- Cause interferences between components?
Example Challenge II: *Kern Sync*

Possible to enable *composability* WRT *predictability/scalability*?

- Kernel must be lock-*free*
- Kernel must use *controlled IPI facilities*
"...a concept is tolerated inside the μ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system’s required functionality."

- Liedtke '95
Guidance for CPSOS Design I

Kernel mechanisms must enable the definition of resource management policies in user-level components, and the additive construction of high-level behaviors from components.
Guidance for CPSOS Design II

The kernel should include minimal but strong facilities for component-centric resource isolation, while enabling cross cutting concern-constrained composability.
The **FUTURE**

- **Trust** in a service should be **chosen & designed**
- Services/abstractions are **mutually isolated**
- Systems are **composed** of these components
- **Components** that are $x$, **compose** to be $x$
  
  $x \in \{ \text{security, dependability, predictability, scalability} \}$
The Skeptic

- IPC is slow
- This cannot scale down
- You cannot support legacy
- This is too much work
? || /* */

composite.seas.gwu.edu
Functional composition

Do component *functionalities* compose?

- Interface algebras (e.g. Henzinger)
- Hierarchical scheduling compositions
- ...

Questions:

- How to test this with generic compositions?
- For system-level code?