Certified Preemptive OS Kernels

Xinyu Feng
University of Science and Technology of China
Why OS Kernel Verification?

Computer Systems
Why OS Kernel Verification?

Correctness of OS is crucial for safety and security of the whole system
Why OS Kernel Verification?

• Fundamental, but also simpler to verify! (comparing to applications)
  • Less domain knowledge required
    – every programmer knows OS
  • Stable specifications
  • Slow evolution
  • Specs validated by application-level verification
OS Kernel Verification: Challenges

• Low-level programs
  • C + inline assembly, interrupts, task management, ...
• Larger code base (than algorithm verification)
• Code at different abstraction layers
  • E.g., threads vs. schedulers
• Involves both libraries (sys. calls) and runtime (scheduler)
  • What is a proper specification?
• Rich concurrency
  • Multi-tasking, multi-core, interrupts
Preemption and nested interrupts

Task A

Handler 1

Handler 0

nested multi-level interrupts
Preemption and nested interrupts
Preemption and nested interrupts

Preemptions and multi-level interrupts are crucial for real-time systems. They also make system highly concurrent and complex.

Not fully supported in existing work
Concurrency & Preemption in Previous work

• seL4 [Klein et al. 2009 ...]
  • Mostly sequential
  • Limited support of interrupts at fixed program points

• Verisoft [Rieden et al. 2007 ...]
  • Kernel is sequential

• Verve [Yang & Hawblitzel. PLDI 2010]
  • Allows preemption, but no nested interrupts
  • Mostly about safety, limited functionality verification

• CertiKOS [Gu et al. 2015, Chen et al. 2016, Gu et al. 2016]
  • Evolving: sequential $\rightarrow$ limited interrupts $\rightarrow$ multicore
  • Still no preemption
Challenges for Verifying Preemptive OS Kernels

- Verifying concurrent programs is difficult
  - Non-deterministic interleaving

[Brookes & O’Hearn 2016], courtesy of Ilya Sergey
Challenges for Verifying Preemptive OS Kernels

• Verifying concurrent programs is difficult

• Verifying concurrent kernels is even more challenging
  • More difficult to establish refinement with concurrency
    • Theories not fully developed until recently
      [Turon et al. POPL’13, ICFP’13] [Liang et al. PLDI’13, CSL-LICS’14]

• Kernel-level preemption can be more complex than multi-tasking/multi-processor concurrency

A natural correctness spec. for OS kernels
Kernel-level preemption can be more complex than multi-tasking/multi-processor concurrency.
Kernel-level preemption can be more complex than multi-tasking/multi-processor concurrency

Interrupt management is now a verification target:
lower abstraction layer and non-uniform concurrency model

More low-level details:
e.g., can context switch only when there are no nested interrupts
This talk

- Verification framework for preemptive OS kernels
  - Refinement reasoning about concurrent kernels
  - Multi-Level nested interrupts and preemption

- Verification of (key modules of) commercial OS kernels
  - μC/OS-II
  - SpaceOS
On Certified Programs

- Mathematical model of Programs
- Specifications (Mathematical Properties)

Verification

Proofs (as certificates)
Outline

• OS Correctness Specification

• Verification Framework
  • System modeling
  • CSL-R: Program logic for refinement & multi-level interrupts
  • Coq tactics

• Verifying real systems
Outline

• OS Correctness Specification

• Verification Framework
  • System modeling
  • CSL-R: Program logic for refinement & multi-level interrupts
  • Coq tactics

• Verifying real systems
Outline

• OS Correctness Specification

• Verification Framework
  • System modeling
  • CSL-R: Program logic for refinement & multi-level interrupts
  • Coq tactics

• Verifying real systems
Outline

• OS Correctness Specification

• Verification Framework
  • System modeling
  • CSL-R: Program logic for refinement & multi-level interrupts
  • Coq tactics

• Verifying real systems
OS Correctness

• **OS provides abstraction for programmers**
  • Hides details of the underlying hardware
  • Provides an abstract programming model

• **OS Correctness** : refinement between high-level abstraction and low-level concrete implementation
OS Correctness

Applications

High-Level Language

C + Abstract primitives

Low-Level Language

C + Assembly

High-Level Abstract Primitives

Low-Level Concrete Implementations
Refinement

Applications

High-Level Abstract Primitive

Low-Level Concrete Kernel Impl.

System Call

System Call

System API
Low-Level Concrete Kernel Impl.

Contextual Refinement

For all applications

System Call

IU

High-Level Abstract Primitive

System Call

Low-Level Concrete Kernel Impl.

System API
Contextual Refinement as OS API Correctness

\[ O \subseteq_{\text{ctxt}} S \text{ iff } \forall A. \text{ ObsBeh}(A[O]) \subseteq \text{ObsBeh}(A[S]) \]

With some assumptions about A

The set of observable behaviors

A: Application   O: Concrete Impl. of OS API

S: Abstract Prim.
Contextual Refinement as OS API Correctness

But OS correctness is more than API correctness:

Correctness of runtime services, e.g., scheduler
(not exported as an API)

Whole system properties,
  e.g., isolation and security, real-time properties, ...

Cannot be specified as abstract API primitives!

How to specify their correctness?
Runtime services and Sys. Props

**Runtime:** specified as part of the high-level language semantics (e.g., scheduling)

Verified through refinement
Runtime services and Sys. Props

**Whole system properties:** specified as trace properties of all apps (with high-level views)

Proved at high-level only, propagated to low-level through contextual refinement!
Outline

• OS Correctness Specification

• Verification Framework
  • System modeling
  • CSL-R: Program logic for refinement & multi-level interrupts
  • Coq tactics

• Verifying μC/OS-II
Our Verification Framework

A. Modeling of OS Kernels

Relational Assertion Entailment

Verification Condition Generator

Domain-Specific Solvers

C. Coq Tactics

B. Refinement-Based Verification

High-Level Language

High-Level Operational Semantics with Configurable Schedulers

High-Level Spec. Language

C Subset

Low-Level Operational Semantics with Context Switch and Interrupts

Low-Level Language

CSL-Style Refinement-Based Program Logic

Contextual Refinement

Relational Assertion Entailment
OS Correctness

Applications

High-Level Language

IU

Low-Level Language

High-Level Abstract Primitives

Low-Level Concrete Implementations
Our Verification Framework

A. Modeling of OS Kernels
   - High-Level Language
     - High-Level Operational Semantics with Configurable Schedulers
   - Domain-Specific Solvers
   - C. Coq Tactics
   - Relational Assertion Entailment

B. Refinement-Based Verification
   - CSL-Style Refinement-Based Program Logic
   - Low-Level Language
     - Context Switch and Interrupts
     - Assembly Primitives

C. Coq Tactics

Refinement-Based Verification Framework
The Low-Level Language

L ::= C | Pr | L;L | …

C ::= while e { C } | if e { C1 } { C2 } | f(e) | e=e | …
e ::= &e | *e | e[e] | e.id | …
The Low-Level Language

```
OSCtxSw:    # Task switching from task level
  pushfl    # Save current task’s context
  pushal
  mov OSTCBCur,%ebx
  mov %esp,(%ebx) # OSTCBCur->OSTCBStkPtr = ESP
  call OSTaskSwHook    # Call user defined task switch hook
  mov OSTCBHighRdy,%eax # OSTCBCur <= OSTCBHighRdy
  mov %eax,OSTCBCur
  mov OSPrioHighRdy,%al # OSPrioCur <= OSPrioHighRdy
  mov %al,OSPrioCur
  mov OSTCBHighRdy,%ebx # ESP - OSTCBHighRdy->OSTCBStkPtr
  mov (%ebx),%esp
  popfl
  ret    # Return to new task
```

**Pr ::= encrt | excrt | switch | ...**

Explicit interrupts management and context switch

```c
#define OS_ENTER_CRITICAL() __asm__ ("pushf \n\tcli") /* Disable interrupts*/
#define OS_EXIT_CRITICAL() __asm__("popf") /* Enable interrupts*/
```
Semantics

Small-step, even for expressions: Try to be faithful to the granularity of machine-code

Semantics similar to CompCertTSO [Sevcik et al. 2011] (but interleaving semantics instead of TSO model)
Our Verification Framework

A. Modeling of OS Kernels

- High-Level Language
  - High-Level Operational Semantics with Configurable Schedulers

- Domain-Specific Language

- C Subset

B. Refinement-Based Verification

- CSL-Style Refinement-Based Program Logic
- Contextual Refinement

C. Coq Tactics

- Relational Assertion Entailment
- Verification Condition Generator
- Domain-Specific Solvers
- Assembly Primitives

Low-Level Operational Semantics with Context Switch and Interrupts

Refinement-Based Verification Framework
High-Level Language

\[ H ::= C | S | H;H | ... \]

\[ S ::= \text{sched} | \gamma(v) | S;S | S+S | ... \]

- C subset
- High-level API specification language
- explicit scheduling points
High-Level Language

\[ H ::= C \mid S \mid H;H \mid \ldots \]

\[ S ::= \text{sched} \mid \gamma(v) \mid S;S \mid S+S \mid \ldots \]

abstract \textbf{atomic} transitions
(over the abstract kernel states)
Example

```c
void OSTimeDly (INT16U ticks)
{
    if (ticks > 0) {
        OS_ENTER_CRITICAL();
        .......
        OS_EXIT_CRITICAL();
        OS_Sched();
    }
    return;
}
```
Example

```c
void OSTimeDly (INT16U ticks)
{
    if (ticks > 0) {
        OS_ENTER_CRITICAL();
        ...
        OS_EXIT_CRITICAL();
        OS_Sched();
        return;
    }
}
```

Suspend the current thread, and remove it from the READY thread queue

call scheduler
Example

Low-level Code vs. High-Level Spec

```c
void OSTimeDly (INT16U ticks)
{
    if (ticks > 0) {
        OS_ENTER_CRITICAL();
        ....
        OS_EXIT_CRITICAL();
    }

    OS_Sched();
    return;
}
```

ticks <= 0
+ ticks>0;
\( \gamma_{dly}(ticks) \);
sched
System Model

• Low-level impl. $O \equiv (\eta_a, \theta, \eta_i)$
  • $\eta_a$: API implementations
  • $\theta$: Interrupt handlers
  • $\eta_i$: Internal functions

• High-level spec. $S \equiv (\phi, \varepsilon, \chi)$
  • $\phi$: API specs. (high-level primitives for APIs)
  • $\varepsilon$: Abstract events (high-level primitives for int. handlers)
  • $\chi$: Abstract scheduler
    • Scheduling policy can be customized by instantiating $\chi$
System Model

Runtime services and Sys. Props

**Runtime**: specified as part of the high-level language semantics (e.g., scheduling)

- **runtime**: API implementations
- **θ**: Interrupt handlers
- **η**: Internal functions

**High-level spec. S**:
- **φ**: API specs. (high-level primitives for APIs)
- **ε**: Abstract events (high-level primitives for int. handlers)
- **χ**: Abstract scheduler
  - Scheduling policy can be customized by instantiating **χ**
  - Shows abstractions for runtime
System Model

• Low-level impl. $O \equiv (\eta_a, \theta, \eta_i)$
  - $\eta_a$: API implementations
  - $\theta$: Interrupt handlers
  - $\eta_i$: Internal functions

Verification goal:
$$(\eta_a, \theta, \eta_i) \subseteq_{ctxt} (\varphi, \varepsilon, \chi)$$

• High-level spec. $S \equiv (\varphi, \varepsilon, \chi)$
  - $\varphi$: API specs. (high-level primitives for APIs)
  - $\varepsilon$: Abstract events (high-level primitives for int. handlers)
  - $\chi$: Abstract scheduler
    - Scheduling policy can be customized by instantiating $\chi$
    - Shows abstractions for runtime
Outline

• OS Correctness Specification

• Verification Framework
  • System modeling
  • CSL-R: Program logic for refinement & multi-level interrupts
  • Coq tactics

• Verifying μC/OS-II
Our Verification Framework

A. Modeling of OS Kernels

B. Refinement-Based Verification

C. Coq Tactics

Relational Assertion Entailment

Verification Condition Generator

Domain-Specific Solvers

Relational Assertion Entailment

High-Level Language

High-Level Operational Semantics with Configurable Schedulers

High-Level Language

C Subset

Low-Level Language

Low-Level Operational Semantics with Context Switch and Interrupts

Contextual Refinement

CSL-Style Refinement-Based Program Logic

B. Refinement-Based Verification Framework

C. Coq Tactics
Program Logic for Refinement and Multi-Level Interrupts

• Relational program logic for simulation/refinements
  [Liang et al. PLDI’13, CSL-LICS’14]

• Ownership-Transfer semantics for interrupts
  [Feng et al. PLDI’08]

• Combining the two: CSL-R for refinement reasoning with multi-level interrupts
Refinement Verification via Simulation

High ($A[S]$):


$S$

$e$

$O$
Simulation with Interrupts & Multitasking

High ($A[S]$):


Interrupt handler:

Another task:

How to do compositional verification?
Simulation with Interrupts & Multitasking

High \((A[S])\):

Low \((A[O])\):

Interrupt handler:

Another task:

Use invariant “\(I\)” to specify non-deterministic interference
Simulation with Interrupts & Multitasking

High ($A[S]$):


Adapted from RGSim [Liang et al. POPL’12] and the relational program logic [Liang et al. PLDI’13, CSL-LICS’14]
Program Logic for Simulation

Low-Level concrete states

High-Level abstract states

High-Level abstract primitive

Low-Level concrete code

S

VI

C

X

Z
Program Logic for Simulation

- Judgement

\[ I \vdash \{ p^* \{ |S| \} \} \ c \ \{ q^* \{ \text{|end|} \} \} \]

Remaining high-level code that needs to be refined
Program Logic for Simulation

• Judgement

\[ I \vdash \{ p^* [|S|] \} C \{ q^* [|\text{end}|] \} \]

No remaining high-level code (refinement is done)
Program Logic for Simulation

• Judgement

\[ I \vdash \{ p^* \models |S| \} \mathcal{C} \{ q^* \, |\text{end}| \} \]

Relational assertions for pre-/post-condition

High-Level abstract primitive \( S \)

High-Level abstract states

Low-Level concrete code \( C \)

Low-Level concrete states

\[ I \vdash \{ p^* \models |S| \} \mathcal{C} \{ q^* \, |\text{end}| \} \]
Program Logic for Simulation

- Judgement

\[ I \vdash \{ p \ast \{ |S| \} \} \subseteq \{ q \ast \{ |\text{end}| \} \} \]

Relational Invariants

High-Level abstract primitive \( S \)

High-Level abstract states

Low-Level concrete code \( C \)

Low-Level concrete states

\[ x \rightarrow \cdots \rightarrow z \]
Soundness

If

$$I \models \{ p^* [ \langle S \rangle ] \} \ C \ \{ q^* [ \langle \text{end} \rangle ] \}$$

then $C$ is simulated by $S$, ...
An Example

```c
void Add() {
    OS_ENTER_CRITICAL();
    Count ++ ;
    OS_EXIT_CRITICAL();
}

I ::= ∃v. Count→ v * CNT=v
```
An Example

OS_ENTER_CRITICAL();

Count ++ ;

{[<CNT++>]*I}

OS_EXIT_CRITICAL();

Ownership transfer

Unfold I

Execute high-level code

Fold I

Ownership transfer
An Example

```c
{ [<CNT++>] }
OS_ENTER_CRITICAL();

Count ++ ;

The code refines
<CNT++>

OS_EXIT_CRITICAL();

{ [<end>] }
```
An Example

```c
{ [<CNT++>] }
OS_ENTER_CRITICAL();

{ [<CNT++>] } * Count→ v * CNT=v
Count ++ ;

{ [<CNT++>] } * Count→ v+1 * CNT=v

{ [<end>] } * Count→ v+1 * CNT=v+1

OS_EXIT_CRITICAL();

{ [<end>] }
```
An Example

```c
{ [<CNT++ > ] } 

OS_ENTER_CRITICAL();

Count ++ ;

{ [<CNT++ > ] * Count→v+1 * CNT=v}

Execute high-level code

{ [<end>] * Count→v+1 * CNT=v+1}

Abstract consequence rule:

\[ p^*[S] \implies r^*[S'] \quad \vdash \{ r^*[S'] \} C \{ q \} \]

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \vdash \{ p^*[S] \} C \{ q \} \]

p \implies q \iff \forall (\sigma, \Sigma, S) \models p,
\exists (\Sigma', S'). (\Sigma, S) \rightarrow^* (\Sigma', S')
\land (\sigma, \Sigma', S') \models q,
```

OS_EXIT_CRITICAL();

{ [<end>] ]
An Example

OS_ENTER_CRITICAL();

Count ++ ;

{ [<CNT++>] * I }

Ownership transfer

{ [end] * I }

OS_EXIT_CRITICAL();

{ [end] }

Interrupt reasoning

Ownership transfer
Interrupt Reasoning

Program invariant  [O'Hearn CONCUR’04]

There is always a **partition** of resource among concurrent entities, and each concurrent entity only accesses its own part.

But note:

The partition is dynamic: ownership of resource can be dynamically transferred.

Interrupt operations can be modeled as operations that trigger resource ownership transfer.  [Feng et al. PLDI’08]
Ownership-Transfer Semantics for Single-Level Interrupt

Interrupt enabled

IF = 1

Resource

B1

B0

Task

Handler 0

cli

IF = 0

Interrupt disabled

Resource

B1

B0

{p}

cli

{p * I0}

sti

{p * I0}

sti

{p}

I0

I0

I0
Memory Model for Multi-Level Interrupts

- Higher-priority handler has priority to select its required resource.
- N blocks are assigned to N interrupt handlers.
- Each well-formed resource block is specified by a resource invariant.
Ownership-Transfer Semantics for Multi-Level Interrupts

[8259A interrupt controller]
Inference Rules for Interrupt Operations

\[ I \vdash \{ [\text{ISR}, 1, k] \ast p \} \text{cli} \{ [\text{ISR}, 0, k] \ast p \ast I \ [0... k-1] \} \]

\[ \text{ISR}(k) = 1 \]
Top Rule for Proving $O \subseteq_{\text{ctxt}} S$

Diagram:

```
O \subseteq_{\text{ctxt}} S
```

```
(\eta_a, \theta, \eta_i) \quad (\varphi, \varepsilon, \chi)
```
Top Rule for Proving $O \subseteq_{\text{ctxt}} S$

- kernel APIs
- interrupt handlers
- abstract primitives for interrupt handlers
- internal functions
- abstract primitives for kernel APIs
- abstract scheduler
Top Rule for Proving \( \Omega \subseteq_{\text{ctxt}} S \)

- Verifying internal functions
- Verifying kernel APIs
- Verifying interrupt handlers

\[
\begin{align*}
\chi, I & \vdash \eta_i : \Gamma \\
\Gamma, \chi, I & \vdash \eta_a : \phi \\
\Gamma, \chi, I & \vdash \theta : \varepsilon
\end{align*}
\]

\( (\eta_a, \theta, \eta_i) \) \( \subseteq_{\text{ctxt}} \) \( (\phi, \varepsilon, \chi) \)
Outline

• OS Correctness Specification

• Verification Framework
  • System modeling
  • CSL-R: Program logic for refinement & multi-level interrupts
  • Coq tactics

• Verifying μC/OS-II
Our Verification Framework

A. Modeling of OS Kernels
   - High-Level Language
     - High-Level Operational Semantics with Configurable Schedulers
   - Domain-Specific Language
     - C Subset
   - Low-Level Operational Semantics with Context Switch and Interrupts

B. Refinement-Based Verification
   - CSL-Style Refinement-Based Program Logic
   - Contextual Refinement

C. Coq Tactics
   - Verification Condition Generator
   - Domain-Specific Solvers
   - Relational Assertion Entailment

Refinement-Based Verification Framework
Coq Tactics for Automation Support

• Verification condition generator: **hoare forward**
  • Automatically select and apply the inference rules

• Assertion entailment prover: **sep auto**
  • Automatically prove “p => q”

• Domain specific solvers: **mauto ...**

\[
\forall x. x < 64 \rightarrow x \geq 3 < 8 ; \quad \forall x. x < 8 \rightarrow (x \ll 3) \& 7 = 0
\]
Coq Tactics for Automation Support

• To reduce the proof efforts

• To hide the underlying details of the verification framework

• To prove domain specific propositions

The ratio of Coq scripts to the verified C is around 27:1

lots of space for improvement
Outline

• OS Correctness Specification

• Verification Framework
  • System modeling
  • CSL-R: Program logic for refinement & multi-level interrupts
  • Coq tactics

• Verifying real systems
  • μC/OS-II
  • SpaceOS
μC/OS-II

• A commercial preemptive real-time multitasking OS kernel developed by Micrium.

• 6,316 lines of C & 316 lines of assembly code.

• Multitasking & Multi-Level interrupts & Preemptive priority-based scheduling & Synchronization mechanism

• Deployed in many real-world safety critical applications
  • Avionics and medical equipments, etc.
Verifying μC/OS-II

Refinement-Based Verification Framework

A. Modeling of OS Kernels

B. Refinement-Based Verification

C. Coq Tactics

D. Verifying key modules of μC/OS-II

- Multi-level Interrupts
- Priority-Based Scheduler
- Task Mana.
- Message Queue
- Mutex
- Semaphore
- Mail Box

Synchronization Mechanisms

- Message Queue
- Mutex
- Semaphore
- Mail Box
Verified APIs:

covers 68% of the frequently used APIs
Proving Priority Inversion Freedom

A. Modeling of OS Kernels

B. Refinement-Based Verification Framework

Relational Assertion Entailment
Verification Condition Generator
Domain-Specific Solvers
C. Coq Tactics

High-Level Language
High-Level Operational Semantics with Configurable Schedulers

Low-Level Language
Low-Level Operational Semantics with Context Switch and Interrupts

C Subset
Assembly Primitives

CSL-Style Refinement-Based Program Logic

Priority inversion freedom of mutex in μC/OS-II
Proving Priority Inversion Freedom

Runtime services and Sys. Props

Whole system properties: specified as trace properties of all apps (with high-level views)

Proved at high-level only, propagated to low-level through contextual refinement!

Priority inversion freedom of mutex in \( \mu C/OS-II \)
Bugs found in μC/OS-II

• Priority Inversion Freedom in Mutex
  • Use a simplified priority ceiling protocol
Limitation of Mutex

- Mutual exclusion semaphores with built-in priority ceiling protocol to prevent priority inversions
- Delivered with complete, clean, consistent 100% ANSI C source code with in-depth documentation.
- Mutual exclusion semaphores with built-in priority ceiling protocol to prevent priority inversions
- Timeouts on ‘pend’ calls to prevent deadlocks
- Up to 254 application tasks (1 task per priority level), and unlimited number of kernel objects
- Highly scalable (6K to 24K bytes code space, 1K+ bytes data space)
- Very low interrupt disable time
- Third party certifiable
Bugs found in μC/OS-II

• Priority Inversion Freedom in Mutex
  • Use a simplified priority ceiling protocol
  • **May cause priority inversion with nested use of mutex!**
  • Fixed in μC/OS-III

• Concurrency bug (atomicity violation)
  INT8U **OSTaskChangePrio** (INT8U oldprio, INT8U newprio)
  • **May lead to access of invalid pointers**
  • Found in μC/OS-II v2.52 (the version we verified)
  • Fixed in μC/OS-II v2.9
Coq Implementations

- CertiOS
  - Framework
  - Tactics
  - Certiuicos
    - Code
    - Spec
    - Proofs
  - Machine
  - Simulation
  - Logic
  - Theory

- Refinement-Based Verification Framework
  - A. Modeling of OS Kernels
  - B. Refinement-Based Verification
  - C. Coq Tactics
  - D. Verifying key modules of uC/OS-II

- Numbers:
  - 55,000
  - 20,000
  - 225,000
  - 150,000
Time cost: 6 person-years

<table>
<thead>
<tr>
<th>Components</th>
<th>Cost (py)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic framework design and impl.</td>
<td>4</td>
</tr>
<tr>
<td>First module: message queue (750 lines of C)</td>
<td>1</td>
</tr>
<tr>
<td>the other modules (3000 loc)</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>6</td>
</tr>
</tbody>
</table>

Debugging and fixing framework, specifications, tactics, etc.

Verification can be much faster with stable framework, tools and libraries

http://staff.ustc.edu.cn/~fuming/research/certiucos/
SpaceOS

• Developed by Beijing Inst. of Control Eng. (BICE)

• Deployed in the central computer of the Chang'e-3 lunar exploration mission
  • and other aerospace systems
Verification of SpaceOS

• Ongoing work

• Adaptation of the verification framework
  • Map SPARCv8 interrupt primitives to the framework
    • Current setting based on the 8259A controller for x86
    • Requires little adaptation

  • Different scheduling
    • The framework parameterize over the scheduling policy
    • Requires almost no change of framework

• Memory management lib.
  • Memory ownership transfer between clients/kernels
  • Ongoing work to extend the framework
Conclusion

• Contextual refinements: a natural correctness formulation for OS kernels

• Verification framework for preemptive kernels
  • CSL-R: Concurrency refinement + hardware interrupts

• Verification of real systems
  • μC/OS-II and SpaceOS
  • Commercial system independently developed
Limitations & Future Work

• No termination proofs
  • Relatively simple, can be done in logic or using tools

• Assembly and compiler are not verified
  • Ongoing work

• No separate addr. space and isolation

• No real-time properties

• More whole-system properties, in addition to PIF

• Improvements for automation (better tools and libs)
Acknowledgments: Group Members

Ming Fu  
Fengwei Xu  
Xiaoran Zhang  
Zhaohui Li  
Hui Zhang  
Ding Ma  
Haibo Gu  
Junpeng Zha  

Alumni: Jingyuan Cao, Jiebo Ma
Thank you!