KESO
An Open-Source Multi-JVM for Deeply Embedded Systems

Isabella Thomm, Michael Stilkerich, Christian Wawersich, Wolfgang Schröder-Preikschat

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Department of Computer Science 4
Distributed Systems and Operating Systems
Friedrich-Alexander University Erlangen-Nuremberg

http://www4.cs.fau.de/Research/KESO
http://www4.cs.fau.de/~mike
stilkerich@cs.fau.de
Statically-Configured Embedded Systems

- comprehensive a priori knowledge
  - code
  - system objects (tasks/threads, locks, events)
  - system object relationships (e.g., which tasks access which locks)
- example: electronic control units (ECU) in cars
  - mass product ➞ immense cost pressure
  - programming languages: C, C++, Assembler
  - operating system: OSEK/VDX, AUTOSAR
    - static configuration
    - scalability classes
    - tailored to the application
- ECU consolidation
  - application isolation
Java and Static Embedded Systems

- **benefits**
  - more robust software (cf., MISRA-C)
  - software-based spatial isolation
  - infrastructure for new technologies (e.g., state migration)

- **problems**
  - low-level programming inconvenient/expensive
  - dynamic code loading
    - fully-featured Java runtimes (e.g., J2ME configurations)
  - overhead
    - code is interpreted or JIT compiled (execution time)
    - dynamic linking (footprint)
    - reflection (footprint, analyzability)
Outline

- Motivation: Statically-Configured Embedded Systems
- KESO: Overview
- Inter-Domain Communication with Portals
- Evaluation
KESO: Overview

- JVM tailoring (instead of fixed configurations)
  - static applications, no dynamic class loading
  - no Java reflection
  - ahead-of-time compilation to Ansi C, VM bundled with application

- scheduling/synchronization provided by underlying OS
  - currently AUTOSAR/OSEK OS
  - accustomed programming model remains

- current limitations
  - simple error hook instead of Java exception handling
  - garbage collector does not bound external fragmentation
  - no support for Java monitor concept
Domains: realms of \{memory, service\} protection

- containers for control flows and system objects
- appear as a separate JVMs to the application
Domain Zero

- trusted control flows of KESO’s runtime environment
- currently only the garbage collector
OSEK / AUTOSAR OS

- provides threading/scheduling facilities
- temporal isolation, hardware-based spatial isolation
KESO: Architecture

Domain A
- Control Flows: Task A → Task B → ISR A
- System Objects: Task A, Task B, Resource A

Domain B
- Control Flows: Task C
- System Objects: Task C

Domain Zero (TCB)
- GC

OSEK / AUTOSAR OS

Microcontroller

Typical Targets
- low-end: 8-bit AVR (ATmega8535, 8K ROM, 512b RAM)
- higher-end: 32-bit Tricore (TC1796, 2M ROM, 256K RAM)
KESO: Architecture

Domain A
- Control Flows
- Task A → Task B → ISR A → Task B
- System Objects
  - Task A
  - Task B
  - Resource A

Domain B
- Control Flows
- Task C
- System Objects
  - Task C

Domain Zero
- (TCB)
- GC

OSEK / AUTOSAR OS

Microcontroller

KESO Native Interface (KNI)
- aspect-oriented mechanism for unsafe interactions
- full access to the internal state of Java-to-C compiler
KESO: Architecture

Domain A

Control Flows
Task A  Task B  ISR A

System Objects
Task A  Task B  Resource A

Domain B

Control Flows
Task C

System Objects
Task C

Domain Zero (TCB)

OSEK / AUTOSAR OS

Microcontroller

OSEK Java API
- access to OSEK / AUTOSAR system services
- language-based service protection
KESO: Architecture

Domain A

Control Flows
Task A → Task B → ISR A

System Objects
Task A → Task B → Resource A

Peripheral Device Access (KNI)
OSEK API (KNI)

OSEK / AUTOSAR OS
Microcontroller

Domain B

Control Flows
Task C

System Objects
Task C

Domain Zero (TCB)
GC

Peripheral Device Access
- RawMemory (similar to RTSJ)
- Memory-mapped objects (similar to C structs)
Spatial Isolation

- inhibit shared data among different domains
- own set of static fields in each domain
- logical heap separation (no cross-domain references)
  - current implementation: heaps physically separated
Spatial Isolation

- inhibit shared data among different domains
- own set of static fields in each domain
- logical heap separation (no cross-domain references)
  - current implementation: heaps physically separated
- Inter-domain Communication

![Diagram showing spatial isolation between Domain A and Domain B with separate static fields and heaps.](attachment:diagram.png)
Spatial Isolation

- inhibit shared data among different domains
- own set of static fields in each domain
- logical heap separation (no cross-domain references)
  - current implementation: heaps physically separated

Inter-domain Communication
- Shared Memory ($\approx$ RawMemory with reference counting)
Spatial Isolation

- inhibit shared data among different domains
- own set of static fields in each domain
- logical heap separation (no cross-domain references)
  - current implementation: heaps physically separated

Inter-domain Communication

- Shared Memory (≈ RawMemory with reference counting)
- Portals (RMI-like mechanism)
Inservice domain: exports interface as named service

- **Domain Srv**
  - **Service**
    - name: *TickerService*
    - interface: *TickerService*
    - class: *TickerServiceImpl*

- **TickerService**
  - `+roundtrip(): void`

- **TickerServiceImpl**
  - `+roundtrip(): void`
  - `-foo(int): void`
service domain: exports interface as named service
  - service object is allocated in the service domain
Inter-Domain Communication with Portals

- service domain: exports interface as named service
  - service object is allocated in the service domain
  - proxy object (portal) is statically allocated for the client domains
Inter-Domain Communication with Portals

- service domain: exports interface as named service
  - service object is allocated in the service domain
  - proxy object (portal) is statically allocated for the client domains

- client domains: statically import this service
  - client domains acquire proxy object reference via name service
Inter-Domain Communication with Portals

- service domain: exports interface as named service
  - service object is allocated in the service domain
  - proxy object (portal) is statically allocated for the client domains

- client domains: statically import this service
  - client domains acquire proxy object reference via name service
  - other domains cannot access the service at runtime
Portals: Parameter Passing

- strictly call-by-value
  - retain logical heap separation

- reference parameters
  - deep copy to the service domain’s heap
  - GC needed in the service domain

- marker interface NonCopyable to prevent copying
  - reference replaced by null
  - used for system objects
public void foo() {
    TickerService srv = (TickerService)
        PortalService.lookup("TickerService");

    srv.roundtrip();
}
public void foo() {
    TickerService srv = (TickerService)
        PortalService.lookup("TickerService");

    srv.roundtrip();
}
public void foo() {
    TickerService srv = (TickerService)
        PortalService.lookup("TickerService");
    srv.roundtrip();
}
public void foo() {
    TickerService srv = (TickerService) PortalService.lookup("TickerService");
    srv.roundtrip();
}

public void roundtrip_portal(object_t *proxy) {
    domain_t prev_domain = CURRENT_DOMAIN;
    CURRENT_TASK->effective_domain = DomainSrv_ID;
    CURRENT_DOMAIN = DomainSrv_ID;
    PUSH_STACK_PARTITION(DomainSrv_ID);
    roundtrip_impl(&tickerservice_srvobj);
    POP_STACK_PARTITION();
    CURRENT_DOMAIN = prev_domain;
    CURRENT_TASK->effective_domain = prev_domain;
}

Switch Protection Context
- backup current domain on stack
- change current execution context
- migrate task to service domain
- restore original domain on return
Portals: Implementation and Overhead

public void foo() {
    TickerService srv = (TickerService)
            PortalService.lookup("TickerService");
    srv.roundtrip();
}

public void roundtrip_portal(object_t *proxy) {
    domain_t prev_domain = CURRENT_DOMAIN;
    CURRENT_TASK->effective_domain = DomainSrv_ID;
    CURRENT_DOMAIN = DomainSrv_ID;
    PUSH_STACK_PARTITION(DomainSrv_ID);
    roundtrip_impl(&tickerservice_srvobj);
    POP_STACK_PARTITION();
    CURRENT_DOMAIN = prev_domain;
    CURRENT_TASK->effective_domain = prev_domain;
}

Partition Stack

- enables GC to skip irrelevant partitions
- only if service method potentially blocks
public void foo() {
    TickerService srv = (TickerService)
        PortalService.lookup("TickerService");
    srv.roundtrip();
}

Invoke Service Method
- statically bound call
- service object is passed as this reference
- primitive parameters are passed through
- references are deep copied

public void roundtrip_portal(object_t *proxy) {
    domain_t prev_domain = CURRENT_DOMAIN;
    CURRENT_TASK->effective_domain = DomainSrv_ID;
    CURRENT_DOMAIN = DomainSrv_ID;
    PUSH_STACK_PARTITION(DomainSrv_ID);
    roundtrip_impl(&tickerservice_srvobj);
    POP_STACK_PARTITION();
    CURRENT_DOMAIN = prev_domain;
    CURRENT_TASK->effective_domain = prev_domain;
}
Portals: Implementation and Overhead

public void foo() {
    TickerService srv = (TickerService)
        PortalService.lookup("TickerService");

    srv.roundtrip();
}

Cost

- primitive parameters: same order of magnitude
- reference parameters: next order(s) of magnitude
## Domains compared to Java Isolates (JSR-121)

<table>
<thead>
<tr>
<th></th>
<th>Domains</th>
<th>Isolates</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>isolation concept</strong></td>
<td>realm-local static fields</td>
<td>logical heap separation</td>
</tr>
<tr>
<td><strong>instantiation</strong></td>
<td>ahead-of-time</td>
<td>at runtime</td>
</tr>
<tr>
<td><strong>runtime representation</strong></td>
<td>none (to the application)</td>
<td>instances</td>
</tr>
<tr>
<td></td>
<td></td>
<td>isolate status</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lifecycle messages</td>
</tr>
<tr>
<td><strong>communication</strong></td>
<td>portals</td>
<td>message links</td>
</tr>
<tr>
<td></td>
<td>shared memory</td>
<td>inter-JVM mechanisms</td>
</tr>
</tbody>
</table>
Performance: KESO vs. C

Flight Attitude Control Algorithm of the I4Copter
- http://www4.cs.fau.de/Research/I4Copter
- C-Code generated from a Simulink Model
- Input: sensor and steering data
- Output: engine thrust levels
- period 9ms
- Tricore TC1796b MCU @150 MHz (1 MiB MRAM)

Java port close to the C version

Recorded trace of inflight sensor and steering data
- Verified that C and Java version output the same actuator values
- Replayed 200 data samples to measure execution time
Performance: KESO vs. C

- Flight Attitude Control Algorithm of the I4Copter
  - http://www4.cs.fau.de/Research/I4Copter
  - C-Code generated from a Simulink Model
  - Input: sensor and steering data
  - Output: engine thrust levels
  - period 9ms
  - Tricore TC1796b MCU @150 MHz (1 MiB MRAM)

- Java port close to the C version

- Recorded trace of inflight sensor and steering data
  - Verified that C and Java version output the same actuator values
  - Replayed 200 data samples to measure execution time
Attitude Controller - Execution Times

- 380 µs
- 400 µs
- 420 µs
- 440 µs
- 460 µs
- 480 µs
- 500 µs

execution time
data set

original C
KESO
KESO (no checks)
KESO (CMath)
KESO (CMath, no checks)

23%
Attitude Controller - Execution Times

- 5% checks
- 23%

Execution time vs. data set graph:
- Orange line: original C
- Red line: KESO
- Blue line: KESO (no checks)
- Black line: KESO (CMath)
- Green line: KESO (CMath, no checks)
Attitude Controller - Execution Times

- 5% checks
- 18% math
- 23%

Data set:
- original C
- KESO
- KESO (no checks)
- KESO (CMath)
- KESO (CMath, no checks)
Attitude Controller - Execution Times

- 5% checks
- 18% math
- 23%

execution time

data set

original C
KESO
KESO (no checks)
KESO (CMath)
KESO (CMath, no checks)
Attitude Controller - Execution Times

constant overhead factor

Data Set

KESO
KESO (no checks)
KESO (CMath)
KESO (CMath, no checks)

overhead factor

0.9x
1x
1.1x
1.2x
1.3x

1.04x
1.18x
1.23x
0.98x
Attitude Controller - Code Size

- original C: 21,866 bytes
- KESO: 17,196 bytes (930b (5.4%) KESO runtime)

- no checks
- CMath
- CMath, no checks

code size (bytes)
Attitude Controller - Code Size

- **original C**: 21,866 bytes
- **KESO**: 17,196 bytes, 930b (5.4%) KESO runtime
- **no checks**: 16,610 bytes, 586b (3.5%) checks
- **CMath**:
- **CMath, no checks**
Attitude Controller - Code Size

- original C: 21,866 Byte
- KESO: 17,196 Byte, 930b (5.4%) KESO runtime
- no checks: 16,610 Byte, 586b (3.5%) checks
- CMath: 23,364 Byte, 1.5k (6%) overhead to C

code size (bytes)
Attitude Controller - Code Size

- Original C: 21,866 bytes
- KESO: 17,196 bytes, 930b (5.4%) KESO runtime
- No checks: 16,610 bytes, 586b (3.5%) checks
- CMath: 23,364 bytes, 1.5k (6%) overhead to C
- CMath, no checks: 22,778 bytes
## Attitude Controller - RAM Usage

<table>
<thead>
<tr>
<th>RAM Usage (bytes)</th>
<th>original C</th>
<th>KESO</th>
<th>KESO (no checks)</th>
<th>KESO (CMath)</th>
<th>KESO (CMath, no checks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>977</td>
<td>101</td>
<td>101</td>
<td>997</td>
<td>997</td>
</tr>
<tr>
<td>bss</td>
<td>1,283</td>
<td>2,311</td>
<td>2,311</td>
<td>2,311</td>
<td>2,311</td>
</tr>
<tr>
<td></td>
<td>896b impure data</td>
<td></td>
<td></td>
<td>1k heap</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20b system objects</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conclusions

- KESO targets the domain of static embedded systems

- VM tailoring
  - small overhead of the runtime environment
  - few fixed restrictions to the VM feature set
  - you pay what you need/use

- performance and footprint competitive to C
  - ahead-of-time compilation
  - static analyses

- software-based spatial isolation