Exploiting Static Application Knowledge in a Java Compiler for Embedded Systems

A Case Study

Christoph Erhardt, Michael Stilkerich, Daniel Lohmann, Wolfgang Schröder-Preikschat

FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG

http://www4.cs.fau.de/~erhardt/

Motivation

Automotive industry

- High robustness requirements
- But: mass production, immense cost pressure
- {Computational, memory} efficiency is key!

Challenge for the deployment of Java applications

Motivation

Automotive industry

- High robustness requirements
- But: mass production, immense cost pressure
- {Computational, memory} efficiency is key!

Challenge for the deployment of Java applications

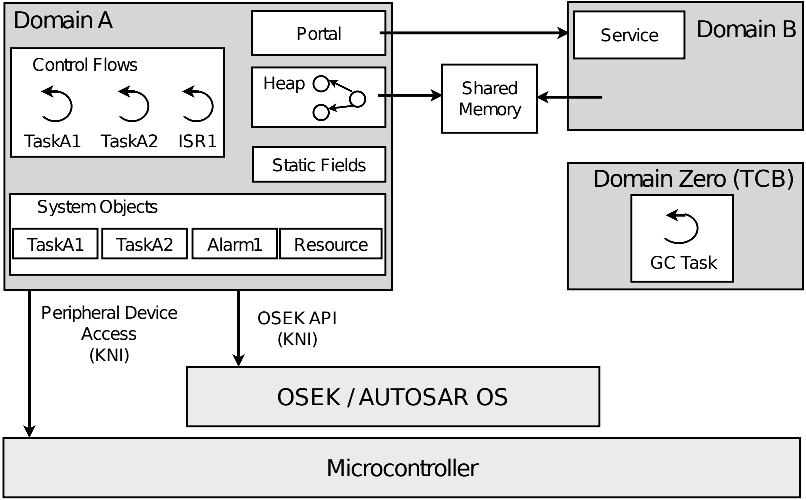
System model of the OSEK/AUTOSAR OS

- OS tailoring: only enable features needed by the application
- Completely static configuration of the application world:
  - Fixed number of tasks
  - No dynamic code loading
  - Fixed-priority scheduling

Let's build a Java VM for this system model!

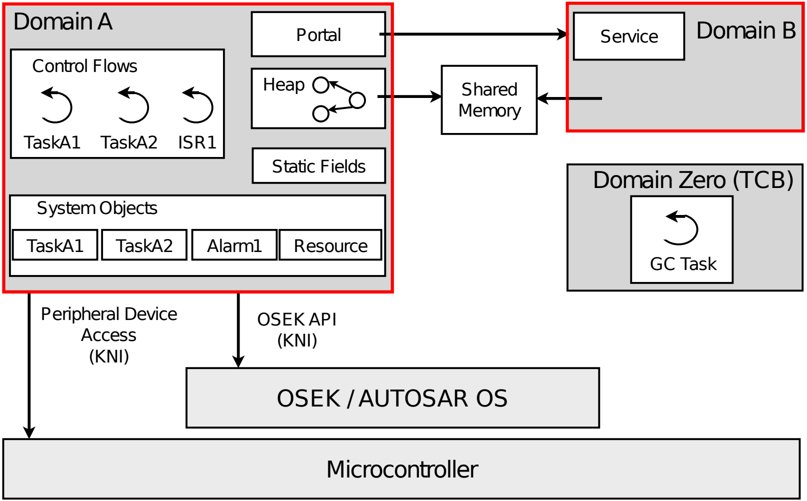
The KESO Multi-JVM

- Java-to-C ahead-of-time compiler
- VM tailoring, static configuration



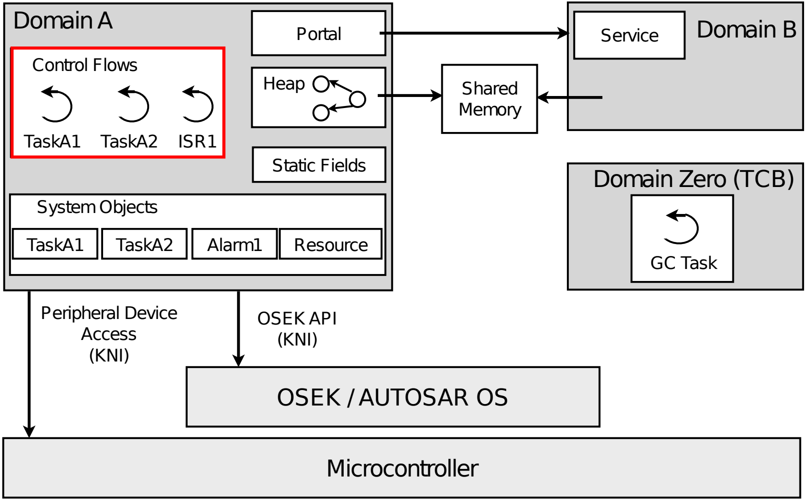
The KESO Multi-JVM

- Java-to-C ahead-of-time compiler
- VM tailoring, static configuration



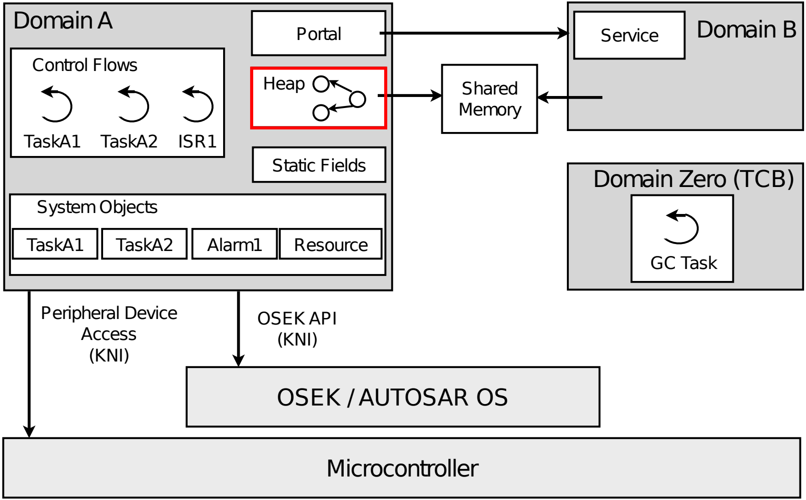
The KESO Multi-JVM

- Java-to-C ahead-of-time compiler
- VM tailoring, static configuration



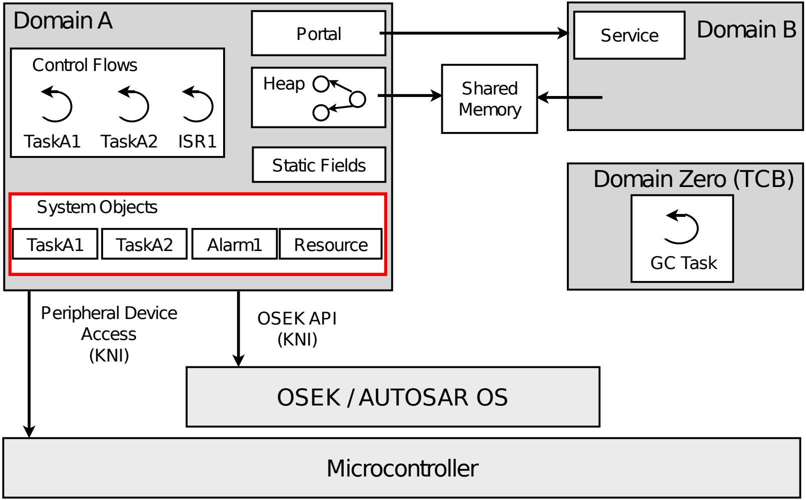
The KESO Multi-JVM

- Java-to-C ahead-of-time compiler
- VM tailoring, static configuration



The KESO Multi-JVM

- Java-to-C ahead-of-time compiler
- VM tailoring, static configuration



Static knowledge

KESO relies upon the availability of extensive application knowledge at compile-time.

Sources of knowledge

1. System model
   - Abandon the aspects of the Java programming model that don't fit the OSEK/AUTOSAR system model!
   - "Everything is static"

Static knowledge

KESO relies upon the availability of extensive application knowledge at compile-time.

Sources of knowledge

1. System model
   - Abandon the aspects of the Java programming model that don't fit the OSEK/AUTOSAR system model!
   - "Everything is static"
2. Static application system configuration
   - Isolation domains
   - Entry points
   - System objects

Static knowledge

KESO relies upon the availability of extensive application knowledge at compile-time.

Sources of knowledge

1. System model
   - Abandon the aspects of the Java programming model that don't fit the OSEK/AUTOSAR system model!
   - "Everything is static"
2. Static application system configuration
   - Isolation domains
   - Entry points
   - System objects
3. Application bytecode
   - Analyze to gain additional knowledge
   - Optimize aggressively using the aggregated knowledge

Optimizations

Static whole-program (or rather: whole-domain) analysis

- Starting at each domain's entry points
- Combined control flow, data flow and class hierarchy analysis

Since the application's static nature enables us to collect extensive information ahead of time, we can apply aggressive optimizations.

Optimizations

Static whole-program (or rather: whole-domain) analysis

- Starting at each domain's entry points
- Combined control flow, data flow and class hierarchy analysis

Since the application's static nature enables us to collect extensive information ahead of time, we can apply aggressive optimizations.

Improved standard optimizations

- Method inlining
- Constant propagation & folding
- Escape analysis & stack allocation
- Runtime check elimination (`null`-, bounds checks)
- Dead-code elimination
- Devirtualization

Selective use of linked stack frames

- Local references stored as compounds, stack frames linked in order to enable GC scanning
- Induces significant overhead!
- But: GC will only run in slack time

Optimization: Only enable Henderson frames in methods that can possibly block – i.e., methods directly or indirectly invoking the `WaitEvent()` OS function

Variant-specific constants

- The `final` qualifier for field variables allows better optimizations
- However...
  - ... programmers may be lazy or unaware
  - ... declaring a field as `final` is impossible in some cases:

```
public final class Constants {
    public static int MAX_FRAMES = 1000;
    // ... more similar constants...
}

public class Main {
    private static void parse(final String[] v) {
        // ...
        if (v[i].equals("MAX_FRAMES"))
            Constants.MAX_FRAMES = Integer.parseInt(v[i + 1]);
        // ...
    }
}
```

Variant-specific constants

- The `final` qualifier for field variables allows better optimizations
- However...
  - ... programmers may be lazy or unaware
  - ... declaring a field as `final` is impossible in some cases:

```
public final class Constants {
    public static int MAX_FRAMES = 1000;
    // ... more similar constants...
}

public class Main {
    private static void parse(final String[] v) {
        // ...
        if (v[i].equals("MAX_FRAMES"))
            Constants.MAX_FRAMES = Integer.parseInt(v[i + 1]);
        // ...
    }
}
```

unreachable

Variant-specific constants

- The `final` qualifier for field variables allows better optimizations
- However...
  - ... programmers may be lazy or unaware
  - ... declaring a field as `final` is impossible in some cases:

```
public final class Constants {
    public static int MAX_FRAMES = 1000;
    // ... more similar constants...
}
```

→ constant

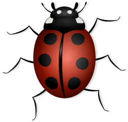
```
public class Main {
    private static void parse(final String[] v) {
        // ...
        if (v[i].equals("MAX_FRAMES"))
            Constants.MAX_FRAMES = Integer.parseInt(v[i + 1]);
        // ...
    }
}
```

unreachable

Idea: deduce `final` qualifiers where possible and legal

Bug detection at compile-time

System configuration



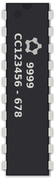
- Does a task have run-to-completion semantics according to the configuration, but invokes `WaitEvent()` anyway?
- Service protection: Does a task or ISR access system objects other than those specified in the configuration?

Bug detection at compile-time

System configuration

- Does a task have run-to-completion semantics according to the configuration, but invokes `WaitEvent()` anyway?
- Service protection: Does a task or ISR access system objects other than those specified in the configuration?

Used raw-memory areas



- Which address ranges are accessed from a specific domain?
- Would a raw-memory area intersect with the regular application memory?
- Which memory-mapped I/O ports are used by which domains?

Results

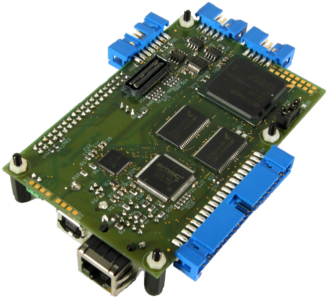
Test system

Software

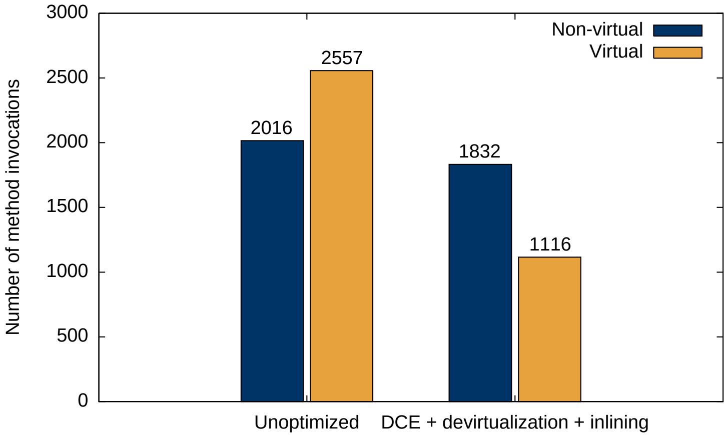
- CD_j 1.2, ported to KESO:
  - Real-time air traffic simulator and collision detector
  - Community-accepted benchmark
- CiAO OS (AUTOSAR programming interface)

Hardware

- TriCore TC1796 @ 150 MHz
- 2 MiB ROM
- 1 MiB SRAM



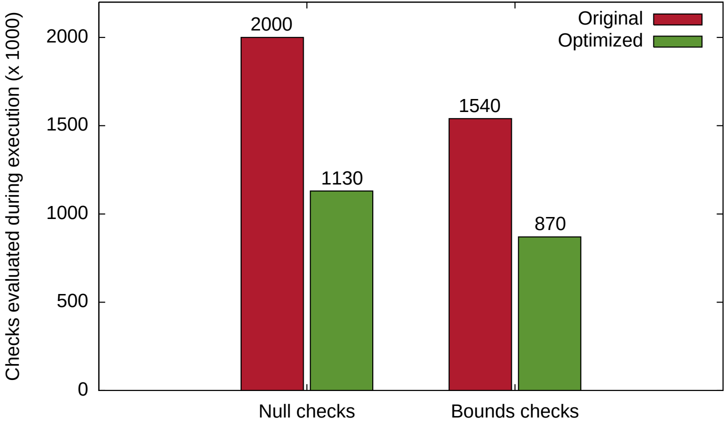
Dead code elimination & devirtualization



More than half of all virtual method invocations are either removed or bound statically.

How many checks are performed during execution?

CD_j detector, 10,000 radar frames:



Over 40 % of all runtime checks are elided.

Effectiveness of advanced optimizations

	Default	+ Optimized frames[1]	+ Variant-specific constants
Exec. time[2]	23.9 ms	15.9 ms (−33.5 %)	15.8 ms (−33.9 %)
Code size	67.8 KiB	55.2 KiB (−18.6 %)	46.5 KiB (−31.4 %)
Data size	4.86 KiB	4.86 KiB (±0 %)	3.08 KiB (−36.6 %)

- Using Henderson frames selectively improves both execution time and code size significantly
- Inferring the `final` qualifier for CD_j's `Configuration` fields would drastically reduce the memory footprint

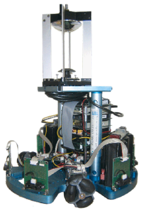
[1] Use of Henderson frames only where necessary
[2] For detector run with 10,000 radar frames; median

Conclusion

Conclusion

Summary

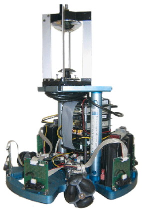
- KESO abandons the aspects of the Java programming model that don't fit the static OSEK/AUTOSAR system model...
  - ... permitting aggressive whole-program optimizations
  - ... bringing the benefits of Java to deeply embedded systems
- Our smallest system to date: Robertino
  - Autonomous robot navigating around obstacles
  - Control software running on ATmega8535 (8-bit AVR, 8 KiB Flash, 512 B SRAM)



Conclusion

Summary

- KESO abandons the aspects of the Java programming model that don't fit the static OSEK/AUTOSAR system model...
  - ... permitting aggressive whole-program optimizations
  - ... bringing the benefits of Java to deeply embedded systems
- Our smallest system to date: Robertino
  - Autonomous robot navigating around obstacles
  - Control software running on ATmega8535 (8-bit AVR, 8 KiB Flash, 512 B SRAM)



Future work

- Implementation of more advanced optimizations
- Extensive evaluation of real-life embedded applications