The `final` Frontier

Coping with Immutable Data in a JVM for Embedded Real-Time Systems

Christoph Erhardt, Simon Kuhnle, Isabella Stilkerich, Wolfgang Schröder-Preikschat

FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG

https://www4.cs.fau.de/Research/KESO/

Introduction

Embedded devices

- Weak CPU
- Limited memory
  - SRAM expensive, scarce
  - Flash cheaper, more ample

Introduction

Embedded devices

- Weak CPU
- Limited memory
  - SRAM expensive, scarce
  - Flash cheaper, more ample

Embedded programming in Java

- Productivity
- Safety

Introduction

Embedded devices

- Weak CPU
- Limited memory
  - SRAM expensive, scarce
  - Flash cheaper, more ample

Embedded programming in Java

- Productivity

- Safety

! Performance
  - AOT compilation
  - Dependent on effective optimisations

Introduction

Embedded devices

- Weak CPU
- Limited memory
  - SRAM expensive, scarce
  - Flash cheaper, more ample

Embedded programming in Java

- Productivity

- Safety

! Performance
  - AOT compilation
  - Dependent on effective optimisations

! Memory footprint
  - RAM usage in particular

The Trouble with Java's `final` Qualifier

- Aids optimisations, but cannot always be declared explicitly
  - *Effectively* `final`

The Trouble with Java's `final` Qualifier

- Aids optimisations, but cannot always be declared explicitly
  - *Effectively final*
- No way to mark pointees as constant/immutable

```
public class ConstArray {
    public static final int[] ARRAY = {10, 2};
}
```

The Trouble with Java's `final` Qualifier

- Aids optimisations, but cannot always be declared explicitly
  - *Effectively final*
- No way to mark pointees as constant/immutable

```
public class ConstArray {
    public static final int[] ARRAY = {10, 2};
}
```

immutable

The Trouble with Java's `final` Qualifier

- Aids optimisations, but cannot always be declared explicitly
  - *Effectively final*
- No way to mark pointees as constant/immutable

```
public class ConstArray {
    public static final int[] ARRAY = {10, 2};
}
```

immutable → ARRAY

heap-allocated → 10, 2

The Trouble with Java's `final` Qualifier

- Aids optimisations, but cannot always be declared explicitly
  - *Effectively final*
- No way to mark pointees as constant/immutable

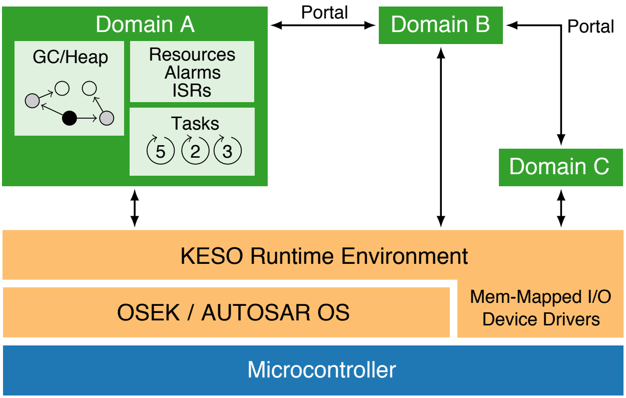
```
public class ConstArray {
    public static final int[] ARRAY = {10, 2};
}
```

immutable (ARRAY) — heap-allocated ({10, 2})

→ No statically allocated + initialised arrays
→ No programmatic flash allocation

Remedy: Compiler Analyses

Platform: KESO JVM



- Portable, scalable to low-end devices
- Static configuration
- Ahead-of-time compilation to C code

Effectively final Fields

```
public final class Constants {
    public static int MAX_FRAMES = 1000;
}

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

Effectively `final` Fields

```java
public final class Constants {
    public static int MAX_FRAMES = 1000;
}

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

unreachable

Effectively `final` Fields

```java
public final class Constants {
    public static int MAX_FRAMES = 1000;
}
```

→ effectively `final`

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

unreachable

Effectively final Fields

```
public final class Constants {
    public static int MAX_FRAMES = 1000;
}

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

→ effectively final

unreachable

Let's focus on static fields for now.

Finding Effectively final Fields

```
public final class Constants {
    public static int MAX_FRAMES;
    static {
        MAX_FRAMES = 1000;
    }
}
```

Finding Effectively `final` Fields

```java
public final class Constants {
    public static int MAX_FRAMES;
    static {
        MAX_FRAMES = 1000;
    }
}
```

Criteria

1. Field is written exactly once, in the class constructor

Finding Effectively `final` Fields

```java
public final class Constants {
    public static int MAX_FRAMES;
    static {
        if (...) {
            MAX_FRAMES = 1000;
        }
    }
}
```

Criteria

1. Field is written exactly once, in the class constructor

Finding Effectively `final` Fields

```
public final class Constants {
    public static int MAX_FRAMES;
    static {
        System.out.println("MAX_FRAMES = " + MAX_FRAMES);
        MAX_FRAMES = 1000;
    }
}
```

Criteria

1. Field is written exactly once, in the class constructor
2. No read prior to initialisation
   - Within class constructor
   - Within method called from class constructor

Finding Effectively `final` Fields

```
public final class Constants {
    public static int MAX_FRAMES;
    static {
        MAX_FRAMES = 1000;
    }
}
```

Criteria

1. Field is written exactly once, in the class constructor
2. No read prior to initialisation
   - Within class constructor
   - Within method called from class constructor

Finding Effectively `final` Fields

```java
public final class Constants {
    public static int MAX_FRAMES;
    static {
        MAX_FRAMES = 1000;
    }
}
```

Criteria

1. Field is written exactly once, in the class constructor
2. No read prior to initialisation
   - Within class constructor
   - Within method called from class constructor

Effect on optimisations

- Constant folding, check elision
- Potentially many indirect effects!

Finding Constant Arrays

```
public class ConstArray {
    public static final int[] ARRAY = {10, 2};
}
```

Finding Constant Arrays

```
public class ConstArray {
    public static final int[] ARRAY;
    static {
        int[] a = new int[2];
        a[0] = 10;
        a[1] = 2;
        ARRAY = a;
    }
}
```

Criteria

1. Array created with constant size, in class constructor

Finding Constant Arrays

```
public class ConstArray {
    public static final int[] ARRAY;
    static {
        int[] a = new int[2];
        a[0] = 10;
        a[1] = 2;
        ARRAY = a;
    }
}
```

Criteria

1. Array created with constant size, in class constructor
2. All writes are to constant indices, with constant values, not more than once per index

Finding Constant Arrays

```
public class ConstArray {
    public static final int[] ARRAY;
    static {
        int[] a = new int[2];
        a[0] = 10;
        a[1] = 2;
        ARRAY = a;
    }
}
```

Criteria

1. Array created with constant size, in class constructor
2. All writes are to constant indices, with constant values, not more than once per index
3. No reads prior to initialisation
   - Consider aliasing!

Finding Constant Arrays

```
public class ConstArray {
    public static final int[] ARRAY;
    static {
        int[] a = new int[2];
        a[0] = 10;
        a[1] = 2;
        ARRAY = a;
    }
}
```

Criteria

1. Array created with constant size, in class constructor
2. All writes are to constant indices, with constant values, not more than once per index
3. No reads prior to initialisation
   - Consider aliasing!

Multi-dimensional arrays: bottom-up approach

Static Allocation of Constant Arrays

- **Java source:**

```
public class ConstArray {
    public static final int[] ARRAY = {10, 2};
}
```

Static Allocation of Constant Arrays

- **Java source:**

```
public class ConstArray {
    public static final int[] ARRAY = {10, 2};
}
```

- **Emitted C code:**

```
typedef struct {
    uint16_t      classID;
    uint32_t      length;
    const int32_t data[2];
} int_array2_t;
```

Static Allocation of Constant Arrays

- **Java source:**

```
public class ConstArray {
    public static final int[] ARRAY = {10, 2};
}
```

- **Emitted C code:**

```
typedef struct {
    uint16_t      classID;
    uint32_t      length;
    const int32_t data[2];
} int_array2_t;

const int_array2_t ca = {
    /* .classID = */ INT_ARRAY_ID,
    /* .length  = */ 2,
    /* .data    = */ {10, 2},
};
```

Static Allocation of Constant Arrays

- **Java source:**

```java
public class ConstArray {
    public static final int[] ARRAY = {10, 2};
}
```

- **Emitted C code:**

```c
typedef struct {
    uint16_t       classID;
    uint32_t       length;
    const int32_t data[2];
} int_array2_t;

const int_array2_t ca = {
    /* .classID = */ INT_ARRAY_ID,
    /* .length  = */ 2,
    /* .data    = */ {10, 2},
};

void ConstArray__clinit_(void) {
    ConstArray_ARRAY = &ca;
}
```

Making Use of Flash Memory

Constant arrays

- Declare as `const` in emitted C code
- Emit `section` attribute
- Adapt linker script

Making Use of Flash Memory

Constant arrays

- Declare as `const` in emitted C code
- Emit `section` attribute
- Adapt linker script

Other candidates for flash allocation

- Strings from constant pools
- Runtime-system data structures
  - Type-information store
  - Dispatch table

Flash-Memory Pitfalls

Mark-and-sweep GC

! Shouldn't try to flip colour bit in flash-allocated object header

→ Don't scan flash-allocated objects

Flash-Memory Pitfalls

Mark-and-sweep GC

! Shouldn't try to flip colour bit in flash-allocated object header
→ Don't scan flash-allocated objects

AVR

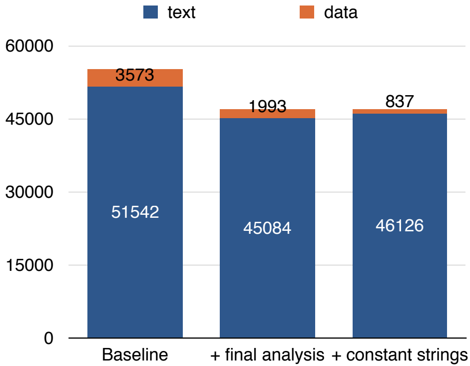
! Separate access instructions for RAM and flash (`ld` vs. `lpm`)
→ For each use: determine correct instruction through alias analysis
→ Prevent aliasing between RAM and flash objects

Evaluation

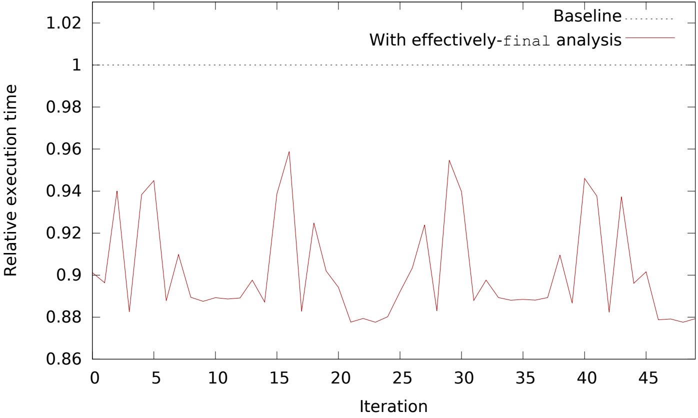
Collision Detector

CD_j 1.2

- Real-time air-traffic simulator and collision detector
- CiAO OS
- TriCore TC1796 @ 150 MHz, 2 MiB flash, 1 MiB SRAM



Collision Detector

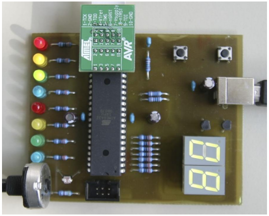


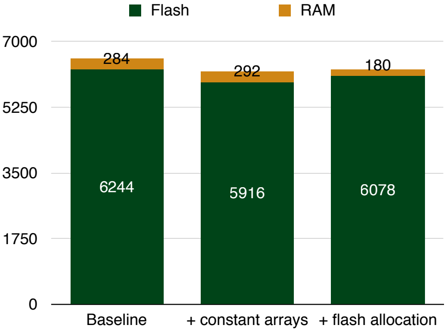
- Folded primitive constants
- Singleton objects → 30 % fewer `null`-checks

SPiCboardTest

Test application

- Evaluation board for teaching
- JOSEK OS
- AVR ATmega32 @ 1 MHz, 32 KiB flash, 2 KiB SRAM





Conclusion

Summary

- Effectively-`final` analysis
  - Can greatly support optimisations
- Static initialisation + flash allocation of arrays
  - Improves startup times
  - Conserves precious RAM

Conclusion

Summary

- Effectively-`final` analysis
  - Can greatly support optimisations
- Static initialisation + flash allocation of arrays
  - Improves startup times
  - Conserves precious RAM

Outlook

- Permit programmer intervention through annotations
  - Cross-check against code to detect contradictions
- Exploit knowledge about target platform (e.g. memory map)