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

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

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

Embedded devices

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

SPEED

erhardt@cs.fau.de
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

erhardt@cs.fau.de  The final Frontier (JTRES 2014)  Introduction
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

```java
public class ConstArray {
    public static final int[] ARRAY = {10, 2};
}
```
Aids optimisations, but cannot always be declared explicitly
- *Effectively final*

No way to mark pointees as constant/immutable

```java
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

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

immutable → heap-allocated
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

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

→ No statically allocated + initialised arrays
→ No programmatic flash allocation
Remedy: Compiler Analyses
1.2 Motivation

Microcontroller
OSEK / AUTOSAR OS
KESO Runtime Environment
Mem-Mapped I/O
Device Drivers
Domain A
---
Domain B
---
Domain C
---

KESO Runtime Environment
OSEK / AUTOSAR OS
Mem-Mapped I/O
Device Drivers

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

Due to the reduction of structure sizes in modern computing chips, dealing with transient soft errors such as bit flips is mandatory for critical applications. Software-based mechanisms for isolation are at a disadvantage compared to microcontroller units (MCUs) with hardware-based memory protection such as MPUs and MMUs, which offer protection against errors caused by this problem class. Previous work on KESO attempts to compensate this [TSK+11, SSE+13].

1.2 | Motivation

Manual memory management using library functions has been the de-facto standard method of dealing with dynamic memory needs in C and C++. It provides fine-grained control over applications' memory allocation behavior, but comes with a downside: Programming mistakes can lead to leaks and dangling pointers, which in turn can lead to security vulnerabilities or crashes. As a consequence, developers need to be careful while writing code that uses manual memory management, in particular when used in safety-critical components.

In order to address these drawbacks, automatic memory management techniques,
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]);
        // ...
    }
}
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]);
        // ...
        }
    }

}
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;
}
```

```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]);
        // ...
    }
}
```

Let’s focus on static fields for now.
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 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
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
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
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
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!
public class ConstArray {
    public static final int[] ARRAY = {10, 2};
}
Finding Constant Arrays

```java
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

```java
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
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!

erhardt@cs.fau.de
Finding Constant Arrays

```java
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:

```java
public class ConstArray {
    public static final int[] ARRAY = {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;
  ```
  ```c
  const int_array2_t ca = {
      /* .classID = */ INT_ARRAY_ID,
      /* .length = */ 2,
      /* .data = */ {10, 2},
  };
  ```

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

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 \texttt{const} in emitted C code
- Emit \texttt{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

CDj 1.2

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

![Bar chart showing comparison of text and data sizes for different configurations.]
Collision Detector

```java
public class Clock {

    static Clock singleton = new Clock();

    public static Clock getRealtimeClock () {
        return singleton;
    }

    public AbsoluteTime getTime () {
        long nanos = System.nanoTime();
        return new AbsoluteTime(nanos / 1000000 L, (int) (nanos % 1000000 L));
    }
}
```

Listing 4.2: Example of the singleton design pattern. The instance of the class can only be acquired via the `getRealtimeClock()` method.

![](image)

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

![Graph showing Flash and RAM allocations](image)

- Baseline: 6244 bytes, 284 bytes
- + constant arrays: 5916 bytes, 292 bytes
- + flash allocation: 6078 bytes, 180 bytes

Example email: erhardt@cs.fau.de
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)