The `final` Frontier: Coping With Immutable Data in a JVM for Embedded Real-Time Systems

Christoph Erhardt Simon Kuhnle Isabella Stilkerich
Wolfgang Schröder-Preikschat
{erhardt, simon.kuhnle, isa, wosch}@cs.fau.de
Friedrich-Alexander University Erlangen-Nuremberg, Germany

ABSTRACT

Managed, type-safe languages such as Java are becoming an increasingly competitive alternative for programming real-time and embedded applications, a field which has traditionally been dominated by C. However, one peculiar issue in the use of Java is the insufficient way immutable data is handled. There are some important cases, such as primitive arrays, where the `final` keyword is not expressive enough to declare data as truly constant. This leads to an unnecessary increase in both code size and runtime memory footprint. Moreover, it prevents the compiler from applying its optimisations as aggressively as would be possible. In this paper, we propose a set of compiler techniques to improve the handling of immutable data in embedded Java applications. Our approach includes (a) detecting constant program data that could not be declared as such by the programmer, (b) eliminating the overhead associated with it, and (c) providing an automated way to allocate that data in flash memory in order to save RAM.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and Objects*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time Systems and Embedded Systems*

General Terms

Design, Languages, Performance

Keywords

Java, embedded systems, real-time systems, KESO

1. INTRODUCTION

Even though C is still the vastly dominating programming language in the field of embedded and real-time systems due to its low-level semantics and its "closeness to metal", an interest in alternative, higher-level languages is on the rise. This shift is particularly motivated by a need for safety and reliability in such systems. Java, being a managed language built on type safety, prevents buffer overflows, dangling pointers and other severe faults by design. Faults are detected at once instead of possibly going unnoticed indefinitely. Multiple applications can run side by side while the runtime environment guarantees software-based isolation.

Previous work [14, 17] has demonstrated ways to access raw memory and memory-mapped registers, and ways to implement interrupt service routines in Java. Moreover, it has been shown that an optimising ahead-of-time (AOT) compiler can translate Java bytecode into low-level or machine code that is competitive with native C code, eliminating the bulk of the overhead inherent in Java's programming model [13, 6]. In terms of runtime performance, this includes omitting redundant type-safety checks such as `null`- and array-bounds checks, but also devirtualising and possibly inlining indirect calls to virtual methods. Through tailoring, the runtime system can be reduced to the minimal set of features actually required by the concrete application. In combination with a suitable memory-management mechanism such as scoped memory [12] or real-time-capable garbage collection, Java can meet all the basic requirements for developing embedded real-time applications.

However, two closely related issues arise when implementing such applications in Java. The first issue is that Java lacks the expressiveness to declare data as truly constant. Compared with the `const` qualifier of C and C++, Java's `final` keyword does not provide adequately strong semantics. For instance, it is plain impossible to declare the contents of a primitive array as constant. The second problem is that Java offers no way to explicitly control the placement of a piece of data – objects are always allocated on the heap[1]. As a consequence, the programmer has no way to tell the toolchain that a certain piece of data should be placed into flash memory rather than into RAM.

We examine and discuss the implications of these circumstances in Section 2. We then propose a set of compiler techniques aimed at alleviating these shortcomings. In Section 3, we give a brief overview of the KESO JVM, into whose AOT compiler we implemented these techniques. Our approach is described in Section 4 and evaluated in Section 5. We present and compare related work in Section 6, before con-

JTRES '14, October 13–14 2014, Niagara Falls, NY, USA

http://dx.doi.org/10.1145/2661020.2661024

[1]This is true from a high-level point of view. If the compiler determines that an object does not escape its method of creation, it may implicitly allocate it on the stack [5].

cluding the paper and providing an outlook in Section 7.

2. PROBLEM ANALYSIS: IMMUTABLE DATA IN JAVA

As stated above, Java's way of handling immutable data is inconsequent to say the least and misses one important use case. This becomes obvious when examining the semantics of the `final` keyword.

2.1 The `final` Keyword

With simplicity being one of its primary design goals, Java attempts to ease the programmer's mental burden by omitting language concepts that could be considered too complex. One instance of such a concept is the `const` qualifier in C++, along with the notion of *const-correctness*. Using `const`, it is possible to qualify pointers, the data a pointer points to, the destination of references and the contents of arrays in a very fine-grained manner. Even methods can be declared as `const`: The compiler ensures that such a method does not modify the internal state of the associated object, and prevents constant objects from being accessed via a non-`const` method. While this concept of const-correctness is a powerful asset, it also requires a skilled and mindful programmer and is at times cumbersome to handle.

Java deliberately abstains from the `const` qualifier[2] and instead specifies a keyword `final` with vastly simpler semantics. Methods declared as `final` cannot be overridden; `final` classes cannot be inherited from. In the context of variables, `final` has the following semantics:

- The variable must be initialised exactly once and cannot be written to again afterwards. A use before the initialisation is illegal.
- For object field variables, every constructor must perform an explicit initialisation (or call another constructor that does). Static class fields must be explicitly initialised by the class constructor.
- For primitive variables, `final` means that their value is indeed immutable after initialisation. For references, it only means that the *reference* itself cannot be modified – but the *contents* of the referenced object can be touched in arbitrary ways provided they are not declared as `final` themselves.

Fields that are declared as `final` allow an AOT compiler to optimise the code more aggressively because it can make more precise assumptions about their contents. For instance, `null`-checks on a reference field can be elided because the field is known to be initialised.

A field which fulfils all of the above characteristics, but was not explicitly declared `final`, is called *effectively* `final` [7]. This may be simply caused by programmer obliviousness, but it can also be due to other reasons:

- The codebase of a configurable application may contain a module where an initialised field is overwritten, but that module may be disabled in some variants of the application.
- The piece of code that overwrites an initialised field may be optimised away by the AOT compiler's dead-code elimination.

In either case, it is beneficial for the compiler to find such effectively `final` fields. We present an algorithm to achieve this in Section 4.1.

2.2 Array Constness and Initialisation

Arrays containing initialised data can be frequently found in embedded systems. For instance, a DSP application may contain lookup tables, a device control may keep encoded state-transfer information in arrays or a DECT-phone application may include ringtone data. In the following, we highlight how such arrays are handled differently in Java than in C, and explain why this behaviour is problematic for embedded systems.

```
const int ARRAY[] = {10, 2};
```

Listing 1: A constant integer array in C.

In C, constant arrays are typically defined as shown in Listing 1. The compiler will reject any attempts to write into the array (not considering type-punning casts). It will place the contents of the array in the `.rodata` section of the binary, so the data will be initialised at load time. Depending on the target architecture and the linker settings, the `.rodata` section will be mapped into either RAM or flash memory.

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

Listing 2: The closest equivalent of the above code in Java.

In Java, initialised arrays are typically defined in a very similar manner (see Listing 2), but their semantics are different. Firstly, arrays are modelled as objects with a `final length` field and a number of non-`final` members. The static field holding the reference is usually also declared as `final`, so the reference cannot be overwritten. However, the contents of the array remain writable – there is no way to prevent it from being overwritten in arbitrary manners.

Secondly, Java does not have a notion of a pre-initialised data section like C does. There is a weakly related concept called *constant pool*, but the latter can hold only strings and numerical values, not complex objects like arrays. As a result, array contents are always heap-based – that is, mutable – and have to be allocated and initialised at runtime.

Listing 3 shows the bytecode that is produced by the Java compiler for the above source code. The generated class constructor first allocates the array on the heap, then explicitly initialises every element one by one, and finally stores a reference to the object into the static field. Each element takes four bytecode instructions to initialise. When translated into machine code, these instructions consume memory. As an example, on the 32-bit TriCore microcontroller architecture, writing one member of an `int` array takes between 4 bytes in the best case[3] ($index \leq 13$; $-8 \leq value \leq 255$) and 14 bytes in the worst case ($index > 16381$; $value <$

[2] On a curious side note, `const` is actually a reserved keyword in Java, but has no function.

[3] Compared to other RISC architectures, TriCore instructions can be encoded rather efficiently.

```
static {};
  Code:
     0: iconst_2
     1: newarray   int
     3: dup
     4: iconst_0
     5: bipush    10
     7: iastore
     8: dup
     9: iconst_1
    10: iconst_2
    11: iastore
    12: putstatic #2      // Field ARRAY:[I
    15: return
```

Listing 3: Class-initialisation bytecode generated from the source code in Listing 2.

$-32768 \vee value > 32767$). Hence, every array not only consumes regular heap memory, but its initialisation code takes up an additional multiple of that size, wasting considerable space when compared to C. Also, the explicit initialisation increases startup times.

This scheme comes with an additional undesirable effect: Since Java methods compiled to bytecode cannot be larger than 64 KiB, the size of an array initialised in that manner is effectively limited to a maximum of about 9,000–11,000 elements, depending on the concrete element values. In classes containing multiple static array fields, the sum of their lengths cannot exceed that limit because all of them are initialised within the static class constructor. Small applications for 8-bit microcontrollers will never hit this limitation, but more complex programs running on 32-bit platforms may be affected.

```
public class UnpackedArray {

    public static final int[] ARRAY
            = unpack("\u0000\u000a\u0000\u0002");

    private static int[] unpack(String s) {
        int[] array = new int[s.length() / 2];
        for (int i = 0; i < s.length(); i += 2) {
            array[i / 2] = (s.charAt(i) << 16)
                         | s.charAt(i + 1);
        }
        return array;
    }
}
```

Listing 4: The same example as in Listing 2, but with the integer array being unpacked from a constant string.

A common workaround for this problem, as seen in Listing 4, is to encode the array values into a constant string (which will be put into the constant pool) and then to decode that string when the class is initialised. The footprint of the initialisation code is much lower in this case, at an additional cost of higher startup times. On the other hand, the same data will still reside in memory *twice* – once as a string and once as the unpacked array.

In Section 4.2, we propose a compiler analysis to find initialised singleton arrays whose contents are not modified. Futhermore, we present a mechanism to give such arrays a simplified allocation and initialisation behaviour equivalent to that of constant arrays in C.

2.3 Data Placement

As mentioned above, Java gives no control to the programmer to specify where a piece of data should be allocated. Since RAM is an expensive and scarce resource in embedded microcontrollers, it makes sense to place constant data into flash memory, which is usually available in higher quantities. This includes a number of runtime-system data structures on the one hand. On the other hand, it includes the contents of previously found immutable array objects.

The effort required to place data into flash memory depends on the concrete target architecture. It is trivial on a platform where flash memory is mapped into the physical address space and can be accessed using regular load/store instructions. On systems where flash memory and RAM are accessed via separate buses, extra work is needed and caution is required in case of aliasing. We describe the respective steps taken by the AOT compiler in Section 4.4.

In summary, this paper covers three compiler techniques related to immutable data in embedded Java applications:

1. Finding effectively `final` fields with the aim of aiding existing compiler optimisations.

2. Finding constant singleton arrays and purging their expensive initialisation code.

3. Automatically placing constant data into flash.

We have implemented these techniques into the AOT compiler of the KESO JVM[4].

3. OVERVIEW OF THE KESO JVM

KESO is a Java Virtual Machine designed for statically configured embedded applications running on top of an AUTOSAR OS [1]. In statically configured systems, all relevant entities of the application as well as the system software – that is, the program code and all OS objects such as tasks and alarms – are known at compile time. KESO does not permit dynamically loading code at runtime or modifying existing code via reflection. This allows its AOT compiler *jino* to produce efficient code and to create a slim runtime system tailored to the needs of the application. Programs pay only for the features they actually use.

The architecture of a KESO application is shown in Figure 1. Applications can be partitioned or isolated from each other by assigning them to protection *domains*. Spatial isolation is constructively ensured by a strict logical separation of all global data (heap, static class fields, etc.). The runtime system provides control-flow abstractions such as threads and interrupt service routines (ISRs), along with their respective activation and synchronisation mechanisms such as alarms and locks. The Java thread API is mapped to the thread abstraction layer of the underlying OS. Inter-domain communication is possible through so-called *portals*, using a remote-procedure call mechanism.

KESO's ahead-of-time compiler *jino* takes the application's Java bytecode as input and translates it into plain C code. It then relies on a C compiler to generate machine

[4]https://www4.cs.fau.de/Research/KESO/

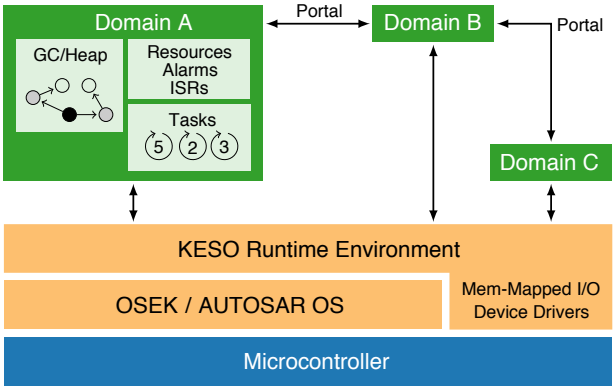

Figure 1: Architecture of a KESO application.

code for the respective target platform. The compiler comprises a set of analysis and transformation passes. The static nature of the entire application system allows it to assume a *closed world*, making the program code good to analyse and optimise. A number of high-level optimisations are included in *jino*, among them:

- Constant propagation and folding
- Method-call devirtualisation
- Method inlining
- Stack allocation
- Dead-code elimination
- Runtime-check elision

Even though most modern C compilers already implement these techniques, it makes sense to also perform them in the Java AOT compiler. The AOT compiler possesses high-level type, application and platform knowledge that cannot be represented in C and gets lost in translation. Hence, *jino*'s high-level optimisations are complemented by the lower-level optimisations performed later on by the C compiler.

KESO has been shown to generate very efficient code suitable even for resource-constrained low-end systems [16].

4. IMPLEMENTATION

In this section, we present our solutions to the problems identified in Section 2: identifying immutable initialised fields, finding constant arrays and allocating data in flash memory. Our approaches are built upon a number of compiler techniques already present in *jino*.

Data-flow analysis.

The data-flow analysis in *jino* is derived from Wegman and Zadeck's SSA-based *Sparse Conditional Constant Propagation* algorithm [18]. In a nutshell, it keeps track of the values and types of all variables and expressions in the intermediate code and constitutes the basis for a range of optimisations. Most importantly, it permits intra- and inter-procedural constant propagation and folding, allowing programmers to initialise their constant arrays with arbitrary expressions as long as the compiler is able to statically evaluate and fold them into constants.

Dominator analysis.

Dominator information is used in various parts of *jino*, e.g. during SSA construction. Since our control-flow graph consists of *maximal* basic blocks, we need to define a relation DOMINATES for a pair of statements (s_1, s_2) as follows: s_1 dominates s_2 iff (a) the basic block containing s_1 strictly dominates the block containing s_2 or (b) both statements are located in the same block and s_1 does not come after s_2.

Alias analysis.

For a potentially constant array, we need a list of all possible references to it in order to make sure we miss no reads and writes. Due to Java's type safety and since KESO does not permit dynamic code-loading or reflection, no hidden aliases are possible.

4.1 Finding Effectively `final` Fields

Fields with `final` properties give the compiler more opportunities for optimisation than regular fields: On the one hand, the compiler knows that the field is initialised and does not carry its default value (0 or `null`). In the case of a reference field, this eliminates the need to insert `null`-checks for operations on that field. On the other hand, since the field is only written once, the initialisation value can be propagated to all reads. If the propagated values can be folded, this possibly benefits many other optimisations. For example, a conditional branch depending on a field might now be folded into unconditional ones, leaving parts of the code dead, which in turn creates an "optimisation avalanche" effect. Hence, our goal is to detect all fields that fulfil the properties of being effectively `final`.

For a static field, the following criteria must be met:

1. The field must be written exactly once, in the class constructor of its class.
2. The write must dominate the class constructor's exit – that is, all possible code paths through that function must go through the write.
3. If there are any reads of the field within the class constructor (or recursively in methods called by the class constructor), they too must be dominated by the write.

The algorithm to find out if these conditions are met for a given field f is shown in Listing 5. It first makes sure that there is only a single write, which is performed in the field's class constructor and dominates its exit. It then searches the code of the class constructor for reads of the field to check if they are dominated by the write. If a method invocation is encountered, all possible callees are recursively searched as well. To protect the algorithm from infinite recursion due to loops in the call graph, a stack of the methods being processed is maintained. In case a method to be searched is already found on the stack, processing is defensively aborted.

It must be noted that, under normal circumstances, this algorithm would not suffice to determine the third criterion. This is due to Java's default lazy class-loading approach: Every class is not loaded and initialised until it is being accessed for the first time. As a consequence, we would not only have to consider explicit method calls, but also implicit calls to the class constructors of all classes that may not have been initialised yet. This set is far from trivial to determine.

However, KESO makes two assumptions which allow it to implement a simplified class-initialisation scheme:

```
procedure isEffectivelyFinal(f)

    w ← Writes[f]                                   ▷ Criterion 1
    if |w| ≠ 1 then
        return false
    end if
    m ← Class[f].<clinit>
    if w ∉ Code[m] then
        return false
    end if

    if ¬dominates(w, Exit[m]) then                  ▷ Criterion 2
        return false
    end if

    for all statements s ∈ Code[m] do               ▷ Criterion 3
        if readsField(s, f) ∧ ¬dominates(w, s) then
            return false
        end if
    end for
    return true

end procedure

procedure readsField(s, f)
    if s ∈ Reads[f] then
        return true
    else if s is an invocation then
        for all methods c ∈ Callees[s] do
            if call-graph loop detected then
                return true                         ▷ Be conservative
            end if
            for all statements s_c ∈ Code[c] do
                if readsField(s_c, f) then
                    return true
                end if
            end for
        end for
    end if
    return false
end procedure
```

Listing 5: Algorithm to determine whether a field f is effectively `final`.

1. In each protection domain, the runtime system invokes all class constructors sequentially at startup, before execution begins at the entry point of the first task. The execution order is unspecified, but it is guaranteed that no class is used before it has been initialised.

2. A class constructor must have run-to-completion semantics; it must not block or call a blocking method.

To determine the order in which the class constructors are called, the compiler analyses their code, builds a dependency graph and sorts it topologically. This imposes the additional constraint that there must be no cyclic cross-dependencies between class constructors as shown in Listing 6. If the application violates this constraint, the compilation is aborted. We argue that code should not rely on such cyclic dependencies because its behaviour is hard to impossible to follow – a limitation that is also imposed by the SCJ specification [8].

```
class A {
    public static int field = B.foo;
}

class B {
    public static int foo = 42;
    public static int bar = A.field;
}
```

Listing 6: Example of a class-initialisation cross-dependency. If `A` is loaded before `B`, `A.field` and `B.bar` will both be 0; otherwise, they will both be 42.

The simplified class-initialisation mechanism has the advantage that it eliminates the need to perform initialisation checks at runtime whenever a class is accessed, and it makes the algorithm in Listing 5 sufficient to find all read accesses to potentially uninitialised data.

For a non-static field to be marked as effectively `final`, it must be initialised in all constructors – either directly or through a call to another constructor. This is more complex and currently not yet implemented in *jino*.

4.2 Finding Constant Arrays

In this section, we describe how *jino* detects constant arrays and how these arrays are converted for static initialisation. Our approach is limited to singleton arrays which are initialised in class constructors. That is, we assume that programmers define constant arrays in the same manner as exemplified in Listing 2 and that the resulting bytecode looks like Listing 3. From our experience, this is also the most common, obvious and convenient manner. For an array to be marked as constant, it must fulfil a number of conditions:

1. It must be created with a constant size within in a class constructor, and the allocation statement must dominate the class constructor's exit. This ensures that there is exactly one instance of the array.

2. All writes to the array or any of its aliases must be inside the class constructor and dominate its exit. Moreover, each write statement must have a constant index and a constant value, and every index must appear at most once. This guarantees that the entire array is initialised with constant values. Elements which are never written can be assumed to be zero or `null`.

3. If there are any reads of the array or its aliases within the class constructor (or recursively in methods called by it), they too must be dominated by all writes.

These criteria are very similar to the ones for effectively `final` fields described in Section 4.1. Essentially, the goal is to find arrays whose members are all effectively `final` – with the extra requirement that they be initialised with compile-time constants.

Our algorithm to determine if an array is constant is shown in Listing 7. We first check that the array is allocated with a fixed length inside a class constructor and dominates its exit. We then make sure that all writes into the array have a constant index and value, and that no member is written more than once. We check the dominance criteria in the same fashion as in the effectively-`final` analysis, again exploiting KESO's simplified class-loading mechanism.

procedure isConstantArray(a)

  if ¬isConstant($Length[a]$) **then**
    return $false$
  end if
  $m \leftarrow Method[a]$
  if $m \neq$ `<clinit>` $\vee$ ¬dominates($a, Exit[m]$) **then**
    return $false$
  end if
  $indices \leftarrow \emptyset$

  for all array writes $w \in ArrayWrites[Aliases[a]]$ **do**
    $i \leftarrow Index[w]$
    if ¬isConstant(i) $\vee$ $i \in indices$ **then**
      return $false$
    end if
    $indices = indices \cup i$
    if ¬isConstant($Value[w]$) **then**
      return $false$
    end if

    if $w \notin Code[m]$ $\vee$ ¬dominates($w, Exit[m]$) **then**
      return $false$
    end if
    for all statements $s \in Code[m]$ **do**
      if readsArray(s, a)
        $\wedge$ ¬dominates(w, s) **then**
        return $false$
      end if
    end for
  end for
  return $true$

end procedure

Listing 7: Algorithm to determine if an array (denoted by its allocation a) is initialised with constant values. The procedure readsArray() is analogous to readsField() from Listing 5, but also takes the aliases of a into account.

Multi-dimensional arrays are handled in the same manner, provided the isConstant function also returns *true* for arrays previously tagged as constant and a fixed-point iteration is built around the above algorithm. A two-dimensional primitive array (the base case) is represented in Java as an array of references to a number of primitive arrays. In the first iteration, we mark all the primitive sub-arrays as constant. In the next iteration, isConstantArray determines that the super-array contains only references to constant arrays, and marks it as well. By repeating the process, multi-dimensional arrays of arbitrary order can be handled.

In the following section, we describe how to convert the arrays into statically allocated data.

4.3 Static Allocation of Constant Arrays

The process of turning a dynamically allocated Java array into static data is done by *jino*'s C backend. Since we already possess all relevant information, it is straightforward:

1. Declare a C `struct` for holding the data of the array (see the explanation in the following section), with the correct type and number of elements.
2. Emit a global variable of that type, filled with the values gathered from the analysed write operations or zero where no write was found. For reference arrays, fill it with pointers to the previously emitted sub-arrays or `null`, respectively.
3. Generate the C code as normal, but omit all the previously identified writes and replace the allocation with a simple assignment of the global object's address.

Revisiting our original example from Section 2, Listing 8 shows the C code that is emitted by *jino* for the Java code in Listing 2. The class constructor is now boiled down to the bare minimum.

```
// Declaration of the array type
typedef struct {
    uint8_t       gcinfo;
    uint16_t      class_id;
    uint32_t      length;
    const int32_t data[2];
} int_array2_t;

// Initialisation of the constant array
const int_array2_t const_arr0 = {
    1,
    INT_ARRAY_ID,
    2,
    {10, 2},
};

// Class constructor of the ConstArray class
void c7_ConstArray_m1__clinit_(void) {
    object_t *obj0_0 = (object_t *) &const_arr0;
    SC7_CONSTARRAY_C7F1_ARRAY(&dom1_DDesc) = obj0_0;
}
```

Listing 8: C code emitted by *jino* for the array in Listing 2 (slightly cleaned up for readability).

4.4 Making Use of Flash Memory

When RAM is scarce, it is a common option for embedded applications to move constant data into flash memory to trade faster access times for more available space. KESO applications generally have three categories of constant, statically initialised data that is a candidate for flash allocation:

- Runtime-system data structures, namely the *class store* containing size and layout information about all class types, and the *dispatch table* holding function pointers for virtual-call lookups.
- Strings from constant pools in the `.class` files.
- Constant arrays, as previously determined by the analysis in Section 4.2.

Since *jino*'s backend emits C code, we can make automated use of the regular mechanisms available to C programmers. This is straightforward on platforms which have one unified address space, but more complex on specific architectures that exhibit a different addressing model.

4.4.1 Von Neumann Platform (Single Address Space)

In the easy case, for instance on the 32-bit TriCore platform, RAM and flash memory are both mapped into the

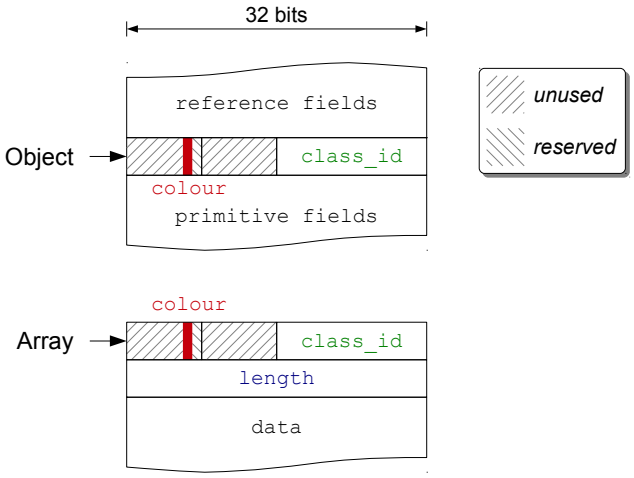

Figure 2: Layout of regular objects and arrays in KESO on a 32-bit platform. On small systems, object headers can be reduced to 16 bits.

same physical address space and can be accessed using the same instructions. The linker normally places the `.rodata` section into flash memory by default – so the relevant data merely has to be declared as `const` in the C code.

To understand how constant objects are handled, we must first take a look at the object layout in KESO. As illustrated in Figure 2, an object consists of a header and the actual data. The header contains the ID of the instantiated class plus a number of management bits for the garbage collector (GC) if the application is configured accordingly. References to an object are represented as pointers to its header.

To simplify garbage collection, the data is laid out around the header in a bi-directional manner: Primitive fields lie in memory directly after the header, reference fields directly before it. Arrays are structured like regular objects, but have an additional length field followed by its members.

KESO's mark-and-sweep GC runs in a separate task and scans the object graph, starting at the root set (task stacks and static fields) and marking the objects visited as live. When scanning an object, it uses the class ID to look up the number of reference fields from the class-store table, and then recursively processes the references adjacent to the header. To avoid infinite loops, the GC maintains a "colour bit" in the header of each object which indicates if the object has already been visited during the current GC cycle.

Flipping this colour bit can be an issue if the object resides in non-writable memory. Depending on the architecture and configuration of the hardware, a write may be silently ignored, but it may also cause a trap. Hence, the GC needs to skip scanning constant objects. A flash-allocated object cannot contain references to heap objects (allocated at runtime) because those references would be unknown at compile time, preventing the object from being put into flash memory in the first place. Consequently, skipping flash-allocated objects cannot lead to falsely freed heap memory. Two implementations are possible:

- The GC can perform an address-range check and only follow references which do not point into `.rodata`[5].

[5]Only following references into the heap is not an option because objects may be allocated on the stack.

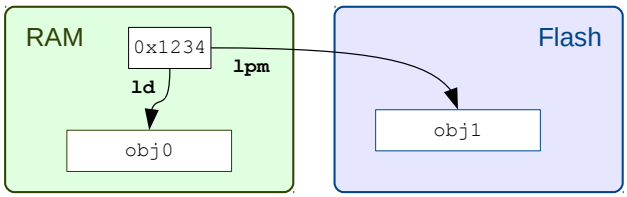

Figure 3: Aliasing between different address spaces if the mutable object `obj0` is allocated in RAM and the immutable object `obj1` lies at the same address in flash memory.

- The header of a constant object can be marked with a special bit (e.g. bit 2). The GC then skips all objects whose "constant"-bit is set.

Constant arrays are allocated as already shown in Listing 8. Strings consist of two parts: a `char` array containing the data and the actual `String` object which encapsulates that array. Constant `String` objects must have the same layout as a regular, heap-allocated `String` object, so the array and the wrapper object are emitted as separate entities.

4.4.2 *Harvard Architecture (Multiple Address Spaces)*

Some lower-end microcontrollers such as the 8-bit AVR platform have a fundamentally different architecture which maintains strictly separated address spaces for RAM (*data memory*) and flash (*program memory*). Values can be loaded from program memory using special instructions. Flash-allocated data cannot be handled transparently on AVR. C developers must use special macros to access program memory, and it is their own responsibility to make sure that addresses are not used in the wrong manner.

Since the address spaces of RAM and flash both begin at address `0x0`, it is impossible to determine the target of a pointer simply by looking at its contents. Hence, the information which pointer is associated with which address space must either be explicitly carried along with the pointer, or it must be implicitly expressed in the data flow of the program. A pointer into flash memory must never be used to access RAM or vice versa. In particular, flash pointers and RAM pointers must not be aliases. An exemplary case of improper aliasing on the AVR architecture is shown in Figure 3. Depending on the access instruction, the address `0x1234` is either used to load a word from data memory (`ld`) or from program memory (`lpm`). Preventing such aliasing and using the correct operations is the programmer's responsibility.

In Java, we want the use of flash memory to be transparent to the programmer. The compiler must ensure that code like in Listing 9 causes no aliasing across address spaces[6].

To find such potentially problematic accesses, we create a *flash-allocation candidate set* – the set of arrays previously marked as constant. We track the aliases of these arrays and examine all instructions in the code that access either the members or the object header via one of the aliases. This applies to the following bytecode instructions:

- `[abcdfils]aload`
- `arraylength`

[6]This code is for illustration. In practice, the compiler will inline `printFirst()`, thus eliminating problematic aliasing.

```
public class Aliasing implements Runnable {

    private static final int[] WC = {54, 74, 90};

    public void run() {
        java.util.Random r = new java.util.Random();
        int[] a = {r.nextInt(), r.nextInt()};
        printFirst(WC);
        printFirst(a);
    }

    private static void printFirst(int[] array) {
        System.out.println(array[0]); // !!!
    }
}
```

Listing 9: Aliasing between a constant and a non-constant array in Java.

- `checkcast`
- `instanceof`

If the operand of any of these instructions has a points-to set which contains an array that is *not* a flash-allocation candidate, we have detected cross-address-space aliasing. We remove the affected constant arrays from the candidate set and repeat the process until reaching a fixed point.

After termination of the algorithm, we distinguish between two categories of constant arrays:

- Arrays which are in the candidate set – that is, arrays not participating in problematic aliasing: We mark these arrays and all accesses to them as *flashable*. Based on this tag, the compiler backend later emits the appropriate macros (`PROGMEM`, `pgm_read_word()`, etc.) instead of regular C code.
- Arrays not in the candidate set: These arrays cannot be placed into flash memory, but they still benefit from static initialisation and allocation (in RAM).

Placing the class store and dispatch table of KESO's runtime system into flash memory is trivially achieved by emitting the respective C macros.

One downside of our approach is that mark-and-sweep garbage collection is not possible in the presence of flash-allocated objects. Since the runtime system has no information about the address space a reference is associated with, neither of the solutions proposed in Section 4.4.1 are feasible on AVR – the GC would not know which reference has to be used in which manner. AVR applications that contain flash-allocated arrays consequently need to fall back to a simpler memory-management mechanism such as scoped memory. We argue that this is a reasonable trade-off because full-blown garbage collection is often not the best choice for a small, deeply embedded system anyway.

5. EVALUATION

In this section, we evaluate the effectiveness of our optimisations. We use two different applications and platforms and break the results down to analyse the effects of the optimisations individually. The first benchmark primarily demonstrates the capability of the effectively-`final` analysis and the handling of constant strings, whereas the second highlights the allocation of constant arrays in flash memory.

Variant	**text**	**data**
Baseline	51542	3573
+ `final` analysis	45084 *(–13 %)*	1993 *(–44 %)*
+ constant strings	46126 *(–11 %)*	837 *(–77 %)*

Table 1: Section sizes of the CD_j benchmark (in bytes).

5.1 Collision Detector

As our first application, we use version 1.2 of the real-time *Collision Detector* benchmark (CD_x) [9], which comes in a C (CD_c) and a Java (CD_j) variant. CD_x is an aircraft monitor that detects potential collisions from simulated radar frames. A collision is reported whenever two aircraft are closer than a configured proximity radius. To bound the computation, the detection is performed in two phases: In the first phase, only the x- and y-coordinates of aircraft are considered, so impossible collisions can quickly be ruled out. In the second stage, a full three-dimensional collision detection is performed for the remaining candidates.

We deploy the slimmed *onthego* variant of CD_j that uses pre-generated radar frames on an Infineon TriCore TC1796 board (150 MHz CPU clock, 75 MHz system clock, 1 MiB SRAM). We use a 600-KiB heap managed by a mark-and-sweep GC. The application is compiled with GCC 4.5.2 and bundled with KESO and the C*i*AO operating system [11]. Code and constant data are allocated in internal flash.

Table 1 shows the difference in binary size with multiple combinations of the optimisations proposed in Section 4. The *text* section comprises the contents of flash memory, i.e. code and constant data, the *data* section contains the initialised data located in RAM.

The effectively-`final` analysis found 71 static fields in total. 30 of these fields stem from a class called `Constants` containing configuration parameters of the air-traffic generator. In some application variants, CD_j can be configured via the command line, in which case some of the values in the `Constants` class are overwritten. These fields cannot be marked as `final`. Since the KESO application is statically configured, the command-line code is dead. Most of the now-constant fields are folded and then removed by the compiler, shrinking the size of the data section by 44 %.

The optimisation also leads to a considerable reduction in code size. This is on the one hand caused by constant folding, including the folding of conditional branches and the subsequent occurrence of dead basic blocks. On the other hand, many reference fields are now known to be initialised and no longer have to be checked upon access. For instance, the CD_j code contains several singleton objects which are created in the class constructor of the respective class and stored in a static field. In total, the number of emitted `null` checks is reduced by 30 %.

The optimisations accompanying the effectively-`final` analysis also improve the performance of the application. Figure 4 compares the iteration times of a CD_j run with the analysis enabled against the times measured without it. The optimisations yield a mean performance improvement of 10 %.

The CD_j application contains no constant arrays, but 1 KiB of constant strings. Moving these strings into flash

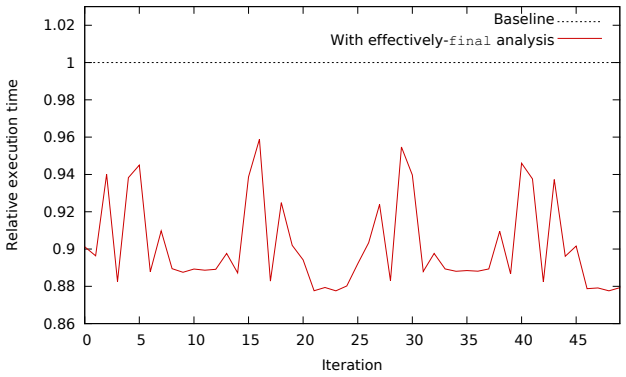

Figure 4: Relative execution times of the collision detector with the effectively-`final` optimisation enabled.

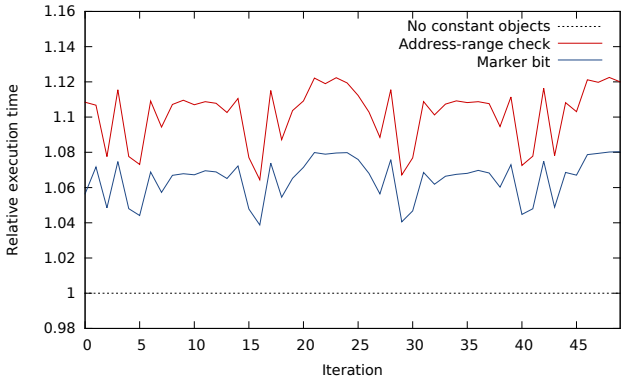

Figure 5: Comparison of CD_j execution times with constant strings allocated in flash memory and marked in two ways.

memory further shrinks the data section, freeing up RAM. In Figure 5, we compare the two methods of marking constant objects for the GC presented in Section 4.4. The measurements show that using a marker bit in the object headers has a smaller impact on the execution time, with a mean of slowdown by 6 % (most of which is caused by the longer access times of flash memory) compared to a 10 % runtime penalty when performing address-range checks.

5.2 SPiCboardTest

Our second application is a showcase program for an AVR-based evaluation board called *SPiCboard*, which is primarily used for teaching embedded C programming (and soldering) to students. The board is populated with LEDs, seven-segment displays, buttons, a potentiometer and a photoresistor, all of which are controlled by an Atmel ATmega32 MCU. The microcontroller runs at 1 MHz and provides 32 KiB flash memory and 2 KiB SRAM. The primary purpose of the application is to provide a testbed for students to verify that the board was populated and soldered correctly. We ported the SPiCboardTest application from C to Java, using the KESO JVM running on top of a JOSEK[7] operating system. Since the application only allocates memory during its initialisation, it runs without garbage collection.

To compile the C code emitted by *jino* into machine code, we used GCC 4.8.2. The resulting binary sizes can be seen in Table 2. The baseline version makes use of all existing default optimisations, but has the new optimisations disabled. Since we already declared all possible static fields as `final`, the effectively-`final` analysis did not yield any results.

[7]https://www4.cs.fau.de/Research/KESO/josek/

Variant	text	data	heap
Baseline	6244	140	144
+ constant arrays	5916 *(−5 %)*	220 *(+57 %)*	72 *(−50 %)*
+ RT data in flash	5964 *(−4 %)*	188 *(+34 %)*	72 *(−50 %)*
+ arrays in flash	6078 *(−3 %)*	108 *(−23 %)*	72 *(−50 %)*

Table 2: Section sizes of the SPiCboardTest application (in bytes).

The application code contains three arrays with primitive constant data: a mapping from LED numbers to I/O pins and two fonts for the seven-segment displays (one for digits, one for alphabetical characters). All three arrays, with a total size of 80 bytes (including headers), are detected by our analysis and converted. The data section consequently grows by the same size and the heap can be halved. Most importantly, the initialisation code of the arrays – whose size is more than four times the actual payload – is discarded.

The runtime system's class-store structure takes 32 bytes; the dispatch table is empty because all virtual calls were devirtualised. Moving class store and constant array data into program memory incurs some overhead in the text section because the instructions to load from program memory are less compact than regular loads. As Listing 10 shows, loading a byte from program memory takes 9 cycles as opposed to 6, so the maximum overhead for a theoretical application that performs only bytewise loads is at 50 %. In practice, the performance impact will be significantly lower than that.

```
movw r30, r16 ; 1 cycle
ldd  r22, Z+5 ; 2
mov  r28, r17 ; 1
std  Y+2, r22 ; 2
```

```
movw r30, r28  ; 1 cycle
adiw r30, 0x05 ; 2
lpm  r28, Z+   ; 3
movw r30, r16  ; 1
std  Z+2, r28  ; 2
```

Listing 10: Loading one byte from data memory (left) vs. loading one byte from program memory (right) on AVR.

Since our test application handles only a relatively small amount of data, there is no extreme pressure to move constant data from RAM to flash memory in our case. Nevertheless, half of the data is constant, and we expect the majority of more complex embedded applications to have a significantly higher data-to-code ratio. In these cases, our optimisations will pay off more visibly.

6. RELATED WORK

Several proposals [2, 4] have been made to add "read-only" qualifiers with stronger semantics than `final` to Java. As of yet, none of these suggestions have found their way into the programming language, and it is not forseeable that any of them will in the near future.

The RTSJ [3] supports accessing arbitrary pieces of physical memory through the `RawMemoryAccess` class. The addressed storage can be of any memory type, so it is also possible to access flash memory. The data residing in the raw-memory area cannot be defined directly in the Java program, but must be defined separately and then linked with

the application. Data can be accessed in the form of an individual primitive load or by copying memory chunks into a heap-allocated array. The first way is cumbersome to use, whereas the second way does not allow for memory savings.

There are several other JVMs for embedded and real-time systems, including Fiji VM [13], JamaicaVM [15] and HVM [10]. To our knowledge, only the HVM has special handling for constant arrays and allows allocating their contents in flash memory. The HVM leaves this task to the programmers, requiring them to explicitly annotate such arrays. Mistaken writes into a constant array are not detected until runtime, when an exception will be thrown. The aliasing problem on Harvard architectures is solved by a run-time switch, which is somewhat costly.

In contrast, our approach requires no programmer intervention, prevents mistaken writes by design (such arrays are not made constant in the first place) and incurs no additional runtime cost.

7. CONCLUSION AND FUTURE WORK

In this paper, we highlighted a number of shortcomings in the Java programming language that have the potential of limiting the feasibility of employing Java on small, resource-constrained devices. These shortcomings are caused by the Java's insufficient means to cope with immutable data. Building upon an AOT compiler, we presented a number of techniques to automatically detect immutable, initialised static fields and constant arrays and to allocate program and runtime-system data in flash memory in order to free up RAM.

Our techniques are currently fully automatic and neither require nor permit programmer intervention. However, developers may want to prevent certain constant data from ending up in flash memory, for example because of its higher access times. This could be achieved using annotations like in the HVM and checking their validity using our analyses.

Further future work includes an alternative approach to solving the issues related to the Harvard architecture: On MCUs where the RAM is mapped into the lower part of the address space (e.g. `0x0060-0x085f` on the ATmega32), flash objects could be allocated at addresses above the RAM's upper bound. This would make it possible to distinguish between RAM and flash references by means of a simple address comparison. Consequently, it would eliminate the limitation that no GC can be used. Also, when detecting conflicting accesses, we would not have to prevent the flash allocation of the arrays affected, but could insert HVM-style runtime switches in the critical locations instead.

8. REFERENCES

[1] AUTOSAR. Specification of operating system (version 4.0.0). Technical report, Automotive Open System Architecture GbR, Dec. 2009.

[2] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. *SIGPLAN Not.*, 39(10):35–49, Oct. 2004.

[3] G. Bollella, B. Brosgol, J. Gosling, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. AW, 1st edition, Jan. 2000.

[4] J. Boyland. Why we should not add readonly to Java (yet). In *In FTfJP*, pages 5–29, 2005.

[5] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *14th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '99)*, pages 1–19, New York, NY, USA, 1999. ACM.

[6] C. Erhardt, M. Stilkerich, D. Lohmann, and W. Schröder-Preikschat. Exploiting static application knowledge in a Java compiler for embedded systems: A case study. In *JTRES '11: 9th Int. W'shop on Java Technologies for real-time & embedded systems*, pages 96–105, New York, NY, USA, Sept. 2011. ACM.

[7] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification*, Java SE 8 edition, Mar. 2014.

[8] JSR-302: Safety-critical Java technology specification (version 0.94). Oracle JCP, June 2013.

[9] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek. CD_x: A family of real-time Java benchmarks. In *JTRES '09: 7th Int. W'shop on Java Technologies for real-time & embedded systems*, pages 41–50, New York, NY, USA, 2009. ACM.

[10] S. Korsholm. Flash memory in embedded Java programs. In *JTRES '11: 9th Int. W'shop on Java Technologies for real-time & embedded systems*, pages 116–124, New York, NY, USA, 2011. ACM.

[11] D. Lohmann, W. Hofer, W. Schröder-Preikschat, J. Streicher, and O. Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *2009 USENIX ATC*, pages 215–228, Berkeley, CA, USA, June 2009. USENIX.

[12] F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek. Real-time Java scoped memory: Design patterns and semantics. In *7th IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '04)*, pages 101–110, Los Alamitos, CA, USA, 2004. IEEE.

[13] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek. High-level programming of embedded hard real-time devices. In *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2010 (EuroSys '10)*, pages 69–82, New York, NY, USA, Apr. 2010. ACM.

[14] M. Schoeberl, S. Korsholm, C. Thalinger, and A. P. Ravn. Hardware objects for Java. In *11th IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '08)*, pages 445–452, Washington, DC, USA, 2008. IEEE.

[15] F. Siebert. Realtime garbage collection in the JamaicaVM 3.0. In *JTRES '07: 5th Int. W'shop on Java Technologies for real-time & embedded systems*, pages 94–103, New York, NY, USA, 2007. ACM.

[16] H. Søndergaard, S. E. Korsholm, and A. P. Ravn. Safety-critical Java for low-end embedded platforms. In *JTRES '12: 10th Int. W'shop on Java Technologies for real-time & embedded systems*, JTRES '12, pages 44–53, New York, NY, USA, 2012. ACM.

[17] M. Stilkerich, I. Thomm, C. Wawersich, and W. Schröder-Preikschat. Tailor-made JVMs for statically configured embedded systems. *Concurrency and Computation: Practice and Experience*, 24(8):789–812, 2012.

[18] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13:181–210, Apr. 1991.