Compiling Java for Low-End Embedded Systems

Ulrik Pagh Schultz
Center for Pervasive Computing, Univ. of Aarhus
Aabogade 34
DK-8200 Aarhus N, Denmark
ups@daimi.au.dk

Kim Burgaard
Systematic Software Engineering A/S
Søren Frichs Vej 39
DK-8000 Aarhus C, Denmark
kib@systematic.dk

Flemming Gram Christensen
Jørgen Lindskov Knudsen
Mjølner Informatics A/S
Helsingforsgade 27
DK-8200 Aarhus N, Denmark
{fgc,jlk}@mjolner.dk

ABSTRACT

The production of embedded systems is continuously increasing, but developing reusable software for such systems is notoriously difficult, in particular in the case of low-end embedded systems based on 16-bit or 8-bit processors. We have developed a compilation system for executing Java byte code on low-end embedded systems, and we demonstrate how this system permits object-oriented programming techniques to be used on devices with only a few hundred bytes of RAM and a few kilobytes of ROM.

We analyze the execution overheads of using object-oriented programming on low-end embedded systems. Based on the conclusion that memory consumption is the major obstacle, we show how the configuration features and optimizations integrated into our compiler can be used to significantly reduce memory requirements. In particular, we use a novel approach based on Java interfaces to control integration of Java programs with the hardware, and demonstrate how aggressive whole-program optimization can significantly reduce the size of the compiled program.

Categories and Subject Descriptors

D.3.4 [**Software**]: Processors—*compilers*; D.3.2 [**Software**]: Language Classifications—*object-oriented languages*

General Terms

Performance, Languages

Keywords

Embedded systems, compilers, Java, interfaces

1. INTRODUCTION

The production of embedded systems is continuously increasing, and improved device connectivity enables the construction of pervasive computing systems composed of heterogeneous collections of devices. Today, systems equipped with microprocessors range from advanced set-top boxes to simple coffee machines. But even though the price/performance ratio of microprocessors continues to decrease, and full-fledged 32-bit processors are used in embedded systems, small 16-bit and 8-bit microprocessors remain attractive for low-end embedded systems due to their low price and low power consumption characteristics paired with relatively high performance.

There is a shift in embedded systems from programming the required functionality in hardware to using general-purpose hardware and supplying the required functionality in software, thus obtaining reuse at the hardware level. However, software for low-end embedded systems is often written in C or perhaps assembly language; development in such languages is tedious, and reuse is difficult. Moreover, in a heterogeneous system with embedded systems and external control systems, a different language will often be used when programming the control systems.

Object-oriented languages facilitate fast development of robust, reusable software, and would thus appear to be an obvious evolution path. Nevertheless, object-oriented programming usually relies on expensive implementation features such as virtual dispatches and dynamic memory allocation, features not needed when programming assembly or C on an embedded system. Moreover, reusable object-oriented software often contains functionality not needed in all applications, a situation which is inappropriate when memory is a scarce resource. Last, the feasibility of developing object-oriented software for low-end embedded system has simply not been investigated.

We have developed an optimizing compiler and class library for a restricted version of the Java language, named JEPES, specifically targeted to low-end embedded systems as small as 512 bytes RAM and 4KB ROM. JEPES complements existing Java technologies by allowing the same programming language (Java) and object model to be used in implementing all parts of a pervasive computing system, with components ranging from desktop systems to low-end embedded devices. This paper concerns the feasibility of running object-oriented programs written in Java on low-end embedded systems, and we argue that reducing memory consumption is the critical issue here; studying run-time efficiency is considered future work.

This paper concerns compilation to low-end embedded systems. For experiments, we use two different hardware platforms, based on Atmel and Hitachi chips. The Atmel platform is a board with an 8-bit Atmel AVR processor (the AT90S8515 variant), 8KB flash ROM and 512 bytes RAM.

The Hitachi platform is the Lego Mindstorms RCX brick, which contains a 16-bit Hitachi H8/300L processor equipped with 32KB RAM and 16KB EEPROM; I/O is performed using the LegOS operating system [25]. The Atmel platform was the initial inspiration for the development of JEPES: in the context of an industrial project, the question was raised whether Java programs could be made to run on systems with only 512 bytes of RAM and 4KB of ROM. The Hitachi platform served as validation that JEPES could be made to work with larger systems containing an operating system.

The rest of this paper is organized as follows. Section 2 describes low-end embedded systems with a focus on object-oriented and Java-based technologies. Then, Section 3 describes JEPES, our solution for reconciling object-oriented programming with low-end embedded systems, Section 4 presents our solution for non-intrusive program configuration, and Section 5 reports our experiments with JEPES. Last, Section 6 presents related work, and Section 7 presents our conclusions and outlines perspectives for future work.

2. LOW-END EMBEDDED SYSTEMS VS. JAVA

Java is used in computer systems ranging from PDAs to servers, although with some variations. Nonetheless, in this section we argue that existing Java-based approaches are inappropriate for low-end embedded systems. We first describe low-end embedded systems, then give an overview of Java for embedded systems, and last analyze the potential overheads of using Java for low-end embedded systems.

2.1 Low-end embedded systems

Embedded systems have gradually been put to use everywhere. The ability to replace complex hardwired circuitry and expensive custom made processors with simple generic components provides compelling opportunities to manufacturers of electronic devices and appliances.

Appliances with embedded processors are often produced in large quantities, so the unit price is important in determining the profit that can be made from selling the appliance. One way of lowering the unit price is by choosing cheaper and more cost effective parts — a classical engineering challenge, which the industry already is accustomed to. Devices with 8-bit processors and a few kilobytes of RAM and ROM are still significantly cheaper than more resourceful 16- and 32-bit counterparts because they require less advanced technology to produce and they are produced as generic solutions in large quantities. Moreover, 8-bit processors often have additional advantages in terms of low power consumption and higher tolerance toward electrical interference.

Although appliances with embedded processors often are produced in families of products with similar traits and functionalities, software is often developed specifically for each product without any reuse across the product line. Moreover, the software is typically written in a mix of C and assembly language, and the design often reflects the actual hardware entities such as CPU pins rather than the application domain. This lack of code reuse increases development costs and time-to-market. Thus, any technology that improves the reuse of design and implementation would also be beneficial to commercial enterprises engaged in developing and producing software for such devices.

2.2 Java

Java is an attractive choice of platform for a number of reasons. First, object-oriented programming is known to offer advantages in terms of code reuse, and the platform independence of Java facilitates reusing implementations across different hardware platforms. Second, the lack of pointer-related errors combined with checked exceptions fits well with the requirements for reliable operation in embedded systems. Last, Java is rapidly becoming the language of choice for many developers, so using Java in the embedded systems helps to leverage their expertise.

For all of its advantages in terms of software development, Java does have a number of significant disadvantages compared to more traditional languages such as C++. First and foremost, the Java virtual machine requires additional resources in terms of memory and processor speed. Second, close interaction with the hardware level is unavoidable when programming interrupt handlers or manipulating raw bytes, but Java has limited support for such operations. Last, C and C++ are simply much more widely accepted in the industry for use in embedded systems.

The Java standards that relate to embedded systems are Java Card, Java 2 Micro Edition with *Connected, Limited Device Configuration* (J2ME / CLDC) and *Connected Device Configuration* (J2ME / CDC), and last the Real-Time Specification for Java (RTSJ).

Java Card is targeted at smart cards with 8/16/32 bit processors and as little as 1K of RAM, 8K EEPROM, and 16K of ROM. The class library is mostly specific to Java Card, and a special byte code format is used [32, 33, 34]. Garbage collection is not supported, so objects must be statically allocated.

J2ME/CLDC is designed for PDAs, mobile phones and similar devices with at least 160KB RAM and at least a 16-bit processor. The class libraries are a scaled down version of the J2SE core class libraries extended with functionality specific to embedded devices.

J2ME/CDC is designed for systems with a 32-bit processor and at least 2MB of total memory. The class libraries includes the core class libraries of J2SE.

RTSJ is not specifically targeted to embedded systems, but embedded systems are often subject to real-time requirements. Among other things, RTSJ extends the Java programming language with real-time scheduling, interruptible threads, and physical memory access. However, the complexity of the RTSJ makes it likely that it will not be useful in low-end embedded systems.

Of these standards, Java Card and J2ME are most relevant to low-end embedded systems; we return to these specifications later in the paper. The Personal Java standard is also seen on older PDA designs but has been replaced by CDC.

2.3 Object overheads in embedded systems

In spite of the well-known software development advantages of object-orientation, object-oriented programs are usually less efficient than their imperative counterparts. On a standard computer system, the most significant run-time overheads to take into account when executing Java programs are interpretation/dynamic compilation, virtual dispatches, memory management, and synchronization [1, 3, 14, 17, 19, 31].

Interpretation and dynamic compilation

Java programs are normally either interpreted or compiled dynamically, due to the requirement that dynamic class loading be supported. However, embedded devices are generally slower than conventional computers and workstations, and there is little or no extra memory available for dynamically generated code. Moreover, neither approach is suitable for embedded devices with strict timing requirements and low response times on external input data. For these reasons, ahead-of-time (AOT) compilation of Java to native code has caught on in the embedded systems world, usually at the cost of disabling dynamic class loading [10, 15, 18, 27, 36].

Virtual dispatches

A virtual dispatch is implemented using an indirect jump, which causes unpredictable control flow that inhibits compiler optimizations and may interfere with hardware optimizations such as pipelining and instruction pre-fetching. Low-end embedded systems however do not normally rely on hardware optimization for speed, and thus it would appear that virtual dispatches could be considered inexpensive in terms of runtime performance. Nevertheless, when compiling Java for the Atmel AVR processor, we need 9 instructions to implement a virtual dispatch, whereas a direct jump only takes 3 instructions. The additional 6 instructions may not be an problematic overhead in terms of execution speed, but are unavoidably a significant overhead in terms of code size.

Memory management

Dynamic memory allocation is considered central to most object-oriented languages, and automatic memory management in the form of garbage collection is common to most modern object-oriented languages. However, on a low-end embedded system, the amount of memory may be so limited that dynamic memory allocation becomes infeasible. Moreover, memory for storing the implementation of the garbage collector is a scarce resource, and real-time constraints are difficult to ensure in the presence of garbage collection.

The solution in JavaCard is to allow dynamic memory allocation, but not guarantee garbage collection. (Thus, a faulty program that instantiates too many objects during its execution may fail at some point.) As an alternative to the JavaCard solution of static allocation, stack allocation can be used, which is for example possible in Realtime Java.[1] However, restrictions must be imposed on stack allocated objects to ensure that they do not escape the method in which they are allocated.

In a low-end embedded system with enough memory to make garbage collection feasible, real-time constraints must often be taken into account to ensure responsiveness. However, real-time constraints are difficult to satisfy using standard garbage collection algorithms, since a collection can be triggered at any point where objects are instantiated, and can potentially need to traverse the entire heap to copy live data or free dead data. Incremental collection allows garbage collection to be done step-by-step, but typically results in a significant space overhead [13], as exemplified by the Treadmill real-time collector [2]. Recent work on the Treadmill collector has produced more space-efficient algorithms, but all objects still need to be kept in doubly-linked list, and this algorithm can still result in fragmentation [21]. Alternatively, more efficient incremental algorithms can be made deterministic by employing constant-time root scanning [30]

[1]Stack allocation can also be performed as an optimization by a compiler, for example based on an escape analysis [8, 35, 4], but does not normally guarantee that no objects are allocated in the heap.

Synchronization

In a language such as Java that directly integrates multithreading primitives, thread synchronization can be a significant overhead. Synchronization usually involves accessing a per-object lock using either atomic operations (in the non-contended case) or operating system calls (in the contended case) [1, 19]. Multiple threads implies multiple execution stacks and additional bookkeeping, which can be inappropriate for a low-end embedded system. Rather, programmable interrupts are used to handle events as they arrive, and timer-based interrupts can be used for scheduling periodic tasks. The interrupt counterpart to synchronization is disabling interrupts in critical zones, which for example is an inexpensive operation on our Atmel hardware.

Synthesis

The overhead of interpretation can be eliminated by ahead-of-time compilation, and thread synchronization is not an issue on low-end embedded systems. Memory management, on the other hand, is a central issue, one that is essential to address for enabling the execution of object-oriented programs on low-end embedded systems. Virtual dispatches may be an issue in terms of speed and size, but thorough experiments similar to those performed for workstation-size systems would be needed to reveal if this is the case (e.g., [14]). In this paper, we address the memory management issue, and leave the investigation of virtual dispatches to future work.

3. JEPES

We now present the JEPES platform. This section gives an overview of the system, including the JEPES programming language and the JEPES compiler, and discusses various issues pertaining to predictability of the compilation and execution of JEPES programs.

3.1 An overview of the JEPES platform

JEPES is a high-performance, customizable platform for the execution of Java bytecode in embedded systems. JEPES is targeted towards low-end embedded systems with as little as 512 bytes of RAM and 4KB of ROM, but is scalable up to KVM-sized systems (32-bit processor with 1MB of RAM/ROM). The JEPES platform consists of a Java-like language (see Section 3.2) and an optimizing compiler for Java bytecode (see Section 3.3). JEPES is intended for use in conjunction with standard Java development environments.

Figure 1 presents an overview of how the JEPES compiler is used to compile programs. Programs are developed in Java using a standard development environment, and compiled to class files. In addition, *interface-directed configuration* is used to instrument the program with domain-specific

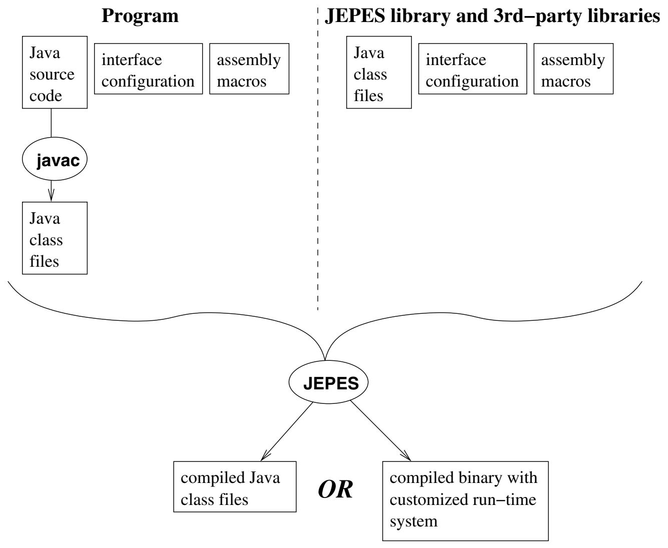


Figure 1: Overview of using the JEPES compiler

information. Interface-directed configuration is a key feature of JEPES that allows the programmer to annotate a program with information using standard Java interfaces; interface-directed configuration is described in detail in Section 4. The JEPES equivalent of native methods is assembly macros, which must also be supplied to the compiler.

The output of JEPES is either a set of optimized class files or a binary, self-contained program. During compilation to binary code, the JEPES runtime environment is automatically customized to and integrated with the application at hand in order to reduce memory usage. Thus, the application is produced as a self-contained, monolithic entity suitable for running directly on the hardware or integrating with an operating system. JEPES is configurable with regards to exception handling, threads, and garbage collection: exceptions are supported only if they are used in the application, and the other features are controlled by interface-directed configuration.[2]

The JEPES native interface is based on inlining assembly macros directly into the compiled Java code, which permits direct and efficient access to the hardware as well as existing native software such as operating system routines and legacy application code (see Section 4 for an example).[3] This feature enables JEPES to e.g. efficiently use the thread model of the underlying operating system. The tight integration with existing native software allows object orientation to be introduced incrementally into existing products during their evolution by gradually replacing components, which greatly reduces the initial design costs.

[2]The current implementation of JEPES does not support garbage collection and has no default API for threads; introducing basic support for these features is however straightforward. Exceptions are supported by the analysis framework, but are not currently supported in the runtime environment.

[3]Calls to external native methods can still be generated.

3.2 The JEPES language

JEPES is based on the Java programming languages, and the compiler reads Java class files in the standard class format. Nonetheless, JEPES has a different standard API, a different set of primitive types, and imposes restrictions on object operations when garbage collection is not used. In addition, object finalization is not supported, and all classes are initialized prior to execution of the main method (the standard Java semantics dictate that the static fields of a class are initialized only when the class is used for the first time).

In order to reduce the overhead of using 32-bit numbers on 8-bit and 16-bit processors, integers have been redefined such that `int` is 16 bit and `long` is 32 bit. Furthermore, `char` is 8-bit and `String` does not support Unicode characters, which enhances the interoperability with C-based, existing code. Support for `float` and `double` is platform dependent.

When garbage collection is not used, which is typically the case for low-end embedded systems, objects must be either statically allocated or stack allocated. Objects that are allocated within a `static` block of a class are statically allocated before the execution of the program (but are initialized when their constructor is run). All other objects must be stack allocatable, meaning that they must not be used after returning from the method in which they are instantiated. To ensure memory consistency, stack allocated objects can only be stored in objects that are allocated in the same stack frame or higher on the stack. This invariant is enforced by the compiler.

3.3 The JEPES compiler

The JEPES compiler takes Java bytecode as input and has backends for the Atmel AVR processor, the Hitachi H8/300L processor (used in the Lego Mindstorms series of products), the i386 architecture, and Java bytecode. For the native backends, virtual dispatches and `switch` instructions are generated as binary search trees, due to the significant cost of indirect memory access on the target hardware (moreover, class hierarchies are usually not deep in the programs we consider).

The JEPES compiler uses a context-insensitive, whole-program dataflow analysis which incorporates constant propagation, sign analysis, CHA [12], and escape analysis [8]. The flow of values through object fields is also traced by the analysis, but only on a per-class basis. The analysis results are used to perform standard optimizations such as constant folding, branch prediction, etc., as well more specific optimizations such as virtual dispatch elimination, inlining[4], stack allocation of objects, and elimination of object field operations that manipulate constant values. After dead code elimination, all fields, methods and classes that are detected as being unused are eliminated from the program. The lack of pointers in Java facilitates safely performing aggressive optimizations that would be difficult to perform in languages such as C or C++.

When the analysis can globally determine the state of an object and where it is used, the object can be completely eliminated from the program; we refer to such objects as *ghost objects*. Ghost objects can be used in the program without any overhead. However, the JEPES compiler can

[4]Inlining is only used when the program size can be reduced, e.g. when a method can be called from only one call site.

currently only eliminate ghost objects with invariant state. Nevertheless, ghost object elimination does for example allow the stream abstraction in the JEPES API (which covers serial ports and similar devices) to be used without any overhead, subject to the analysis results derived.

3.4 Predictability

Predictable behavior is a critical issue in embedded systems, in particular in the case of real-time systems. JEPES programs can execute without garbage collection and hence with predictable execution times: static object allocation is performed during initialization, and stack allocation of objects takes a fixed number of instructions. The worst-case cost of virtual dispatches and class cast checks depends on the depth of the class hierarchy, which is a constant for a given program.

When a JEPES program is modified by adding new classes, it cannot be guaranteed to preserve its current timing characteristics. Virtual dispatches that can be eliminated in one version of the program may need to be preserved in an extended version of the program (e.g., one that relies on a higher degree of polymorphism). Similarly, stack allocation and ghost object elimination may no longer be possible in a modified version of a program. In the case of stack allocation and ghost object elimination, the interface-directed configuration mechanism can be used to ensure that a compile-time error is generated when this is the case, but there is currently no similar mechanism for virtual dispatches.

4. INTERFACE-DIRECTED CONFIGURATION

4.1 The basic idea

Interface-directed configuration provides a flexible and powerful mechanism to enable per-class configuration for governing the compilation process. The solution is non-intrusive, because it does not introduce new syntax to the Java programming language, and it does not rely on "magic", hard-coded class names to be recognized in the compiler. Instead, the compiler matches Java interfaces with externally supplied configuration files. The configuration files can then be used to attach additional semantic meanings to the classes that implement these interfaces, and to enable special behavior of the compiler on selected methods and fields.

As an example of interface-directed configuration, consider the JEPES program fragment shown in Figure 2. The class `InputProcessor` defines a method `handle` which can perform interrupt-driven input. Interrupt handlers require special entry and exit code, and the interface `InterruptHandler` directs the compiler to generate such code. The semantic meaning of the interface, that the `handle` method is an interrupt handler, is defined in a separate configuration file (`InterruptHandler.jid`).

```
class InputProcessor implements InterruptHandler {
  public static void handle() {
    ... read input from physical memory address ...
  }
}
```

InputProcessor.java

```
interface InterruptHandler {
  // empty marker interface
}
```

InterruptHandler.java

```
InterruptHandler {
  methods {
    public static void handle() {
      interrupt-handler {
        vector = 0x0E;
      };
    }
  }
}
```

InterruptHandler.jid

Figure 2: Interrupt handling in JEPES.

4.2 Configuration directives

The JEPES interface configuration mechanism supports the following kinds of directives: assembly macros, interrupt handlers, external access to Java classes, stack allocation and ghost allocation.

Substitution of methods with inlined assembly macros

Assembly macros provide a high-performance alternative to native methods. By constraining what can be written in the macros, the compiler can inline the macros into the Java code, which would not be possible with standard Java native methods. The macros are written in the native assembly language of the target platforms with additional macro syntax for symbolic register names and reservation of "locally used" registers, which allows compiler-generated parameter passing and save/restore of registers.

As an example, consider the code fragments shown in Figure 3. The serial port encapsulation of the JEPES library implements most of its functionality in Java (3a), but the methods for reading and writing bytes are written in assembly macros, as indicated by the interface configuration file (3c). The body of a Java method marked as a macro is ignored by the compiler and can thus contain arbitrary Java code; this feature is used in simulated testing environments where the native code needs to be implemented in pure Java. Assembly macros for the Atmel AVR platform are also shown (3d and 3e). The register parameters `<@R>` and `<b>` in the assembly macros are substituted with real register names during compilation. The parameter `<@R>` maps to the register where the caller expects to find the (primitive typed) return value, and the parameter `<b>` maps to the register that holds the formal parameter of `writeByte()`. Alternatively, the implementation could have been done by writing accessor macros for all of the ports and the special purpose registers on the Atmel AVR port, enabling even more of the serial port code to be written in pure Java.

Method entry and exit code for interrupt handlers

JEPES makes it possible to write interrupt handlers in pure Java, as was shown in the example of Figure 2. The compiler extends all methods with code that manages the call stack and saves and restores the contents of registers. However, interrupt handler methods often require an alternative call semantics, because the calls takes place out of the ordinary flow of control.

```
package jepes.io.bus;
public class SerialImp implements jepes.io.bus.ISerialImp {
  public int readByte() {
    return 0; // default return value; macro
  }

  public void writeByte(byte b) {
     // macro
  }
}
```

(a) File `jepes/io/bus/SerialImp.java`

```
package jepes.io.bus;
interface ISerialImp {
  public int readByte();
  public void writeByte(byte b);
}
```

(b) File `jepes/io/bus/ISerialImp.java`

```
ISerialImp {
  methods {
    public int readByte() {
      macro;
    }
    public void writeByte(byte b) {
      macro;
    }
  }
}
```

(c) File `jepes/io/bus/ISerialImp.jid`

```
; Atmel AVR macro for readByte()
; wait until UART is ready
Receive:
        SBI     UCR,RXEN
        SBIS    USR,RXC
        RJMP    Receive

; write data to return register
        IN      <@R>,UDR

; clear the receive flag on UART
        CBI     UCR,RXEN
```

(d) File `jepes/io/bus/ISerialImp/readByte__B.asm`

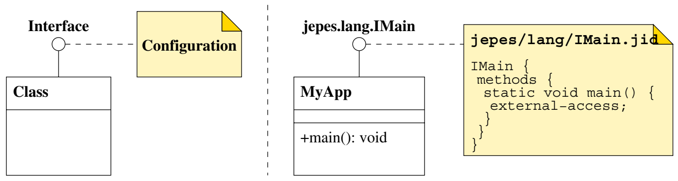
```
; Atmel AVR macro for writeByte()
; wait until UART is ready
Send:
        SBI     UCR,TXEN
        SBIS    USR,UDRE
        RJMP    Send

; write the output byte
        OUT     UDR,<b>

; clear the send flag on UART
        CBI     UCR,TXEN
```

(e) File `jepes/io/bus/ISerialImp/writeByte_B_V.asm`

Figure 3: Serial port implementation using assembly macros



```
IMain {
 methods {
  static void main() {
   external-access;
  }
 }
}
```

Figure 4: The relation between Java interfaces and interface configuration files

Externally accessible classes, methods, and fields

The optimizer needs to be told where the whole-program analyses of the Java code cannot be guaranteed to provide all of the program state information. Otherwise, overly optimistic optimizations would be applied to e.g. interrupt handler routines, and fields that map to ports or memory mapped addresses.

Stack allocation of objects

The stack allocation directive provides a mechanism for guaranteeing stack allocation of objects even when garbage collection is used. It is a compile-time error if an object instantiated from a stack-allocatable class cannot be stack allocated. Thus, a method that only uses stack-allocatable objects is guaranteed not to be interrupted by a garbage collector, and hence has a fixed execution time, which enables real-time programming.

Defining stack allocation as a per-class attribute is coarse-grained, and a per-allocation-site granularity would probably be more appropriate in some cases. This limitation has so far not been a problem, but this may change as we experiment with compiling larger JEPES programs. We note that it would be convenient if Java's facility for defining anonymous classes could be used for providing per-allocation-site granularity, e.g.:

```
Iterator i = new VectorIterator(vector)
                 implements StackAllocation {};
```

However, Java only allows such anonymous classes to add new members, not to implement new interfaces.

Ghost allocation

Ghost allocation is used similarly to stack allocation of objects, but provides a mechanism for guaranteeing the elimination of objects with immutable contents, referred to as "ghost objects." It is a compile-time error if an object that was marked as ghost-allocatable could not be eliminated from the program, ensuring that such objects never incur any overheads in the compiled program.

4.3 Implementation

Interface-directed configuration is implemented as follows. The compiler matches Java interfaces with configuration files based on the package and interface name, as shown in Figure 4. A configuration thus has a "per-interface" granularity, but any matched configuration file has effect on all implementing classes. Each configuration file contains one or more compiler directives that either may be general to the class, or to a specific method or field in implementing classes. The method- and field-specific directives both cover non-static as well as static entities, in contrast to Java where interfaces only contain non-static method declarations.

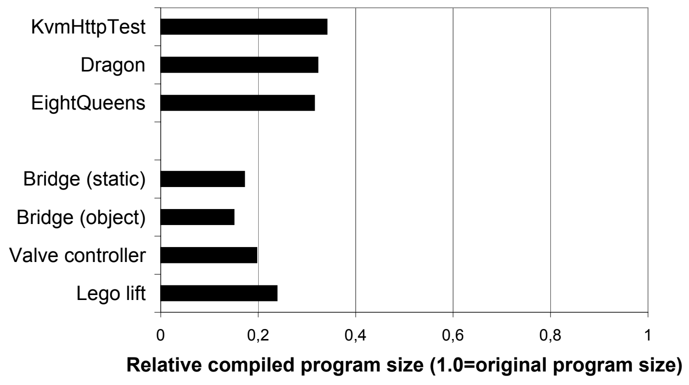


Figure 5: Relative size reduction

5. EXPERIMENTS

To evaluate the size reduction performed by the JEPES compiler, we have applied JEPES to two sets of programs: a number of demo programs included with the KVM virtual machine, and a number of programs written specifically for JEPES on a low-end embedded system. In all experiments, the size before optimization is reported based on the transitive closure of the classes referenced from the program (as opposed to including all classes from the entire class library). The results are summarized in Figure 5 by the relative size reduction (25% means that the program only takes up a quarter of the space that it did before). We now discuss the results in detail.

The KVM demo programs are KvmHttpTest, Dragon, and EightQueens. These programs are not appropriate for execution on low-end embedded systems, as they rely on a graphical user interface. On the other hand, they are relatively large Java programs written in an object-oriented style of programming, which makes them interesting for evaluating the effectiveness of the optimizations applied by JEPES. Indeed, JEPES can be applied to these programs as a bytecode to bytecode compiler, which significantly reduces their size before loading them onto e.g. the KVM virtual machine.[5]

On average, JEPES reduces the size of the KVM demo programs to 32.6% of their original size. Table 1 provides more details. This table compares the size of the class files ("J-size") and their contents (number of fields, methods, and bytecode instructions) before optimizations, after using CHA to remove unneeded classes and methods, and after performing all optimizations supported by JEPES. In addition, the number of heap allocation sites that could be converted into stack allocation is reported. As can be seen, the CHA provides a large advantage in size compared to the transitive closure, and should probably be considered minimum for compilers for low-end embedded systems. Moreover, the full selection of optimizations performed by JEPES provides a significant advantage over the result obtained by applying CHA.

The native JEPES programs are two different versions of a CAN-bus [5, 7] to serial connection bridge, a valve controller, and an RCX application. The static version of the CAN-bus to serial bridge is written in C-style using static methods, whereas the object version relies on ghost allocation to use objects to represent e.g. streams without any significant overhead in terms of size. The valve controller monitors and controls a fictional valve over a serial line, and the Lego RCX application controls a lift built from Lego bricks. The Lego lift program is compiled to native Hitachi H8 instructions, the other programs are compilated to native Atmel AVR instructions. On average, JEPES reduces the size of these programs to 18.9% of their original size, when compiling from Java bytecode to native code. Table 2 provides more details, including the size of the compiled program both as Java bytecode and as native code. For these programs, the majority of the size reduction was obtained using CHA. As for the size of the native code, we have found that straight-line C code compiled using the commercial IAR Embedded Workbench C++ compiler for Atmel AVR is up to 2.5 times more compact than that generated by JEPES for equivalent Java code. We believe this difference is due to the immaturity of the JEPES code generator.

JEPES currently provides no automated way of computing the total memory footprint of an application. Manual inspection of the assembly code generated for the static version of the CAN–serial bridge reveals that this program uses roughly 50 bytes of RAM (excluding registers). Thus, the total footprint of this application is 1511 bytes of ROM and 50 bytes of RAM, which is several orders of magnitude smaller than what is required by an interpretive apporach such as Sun's KVM.

6. RELATED WORK

The work that most closely relates to the application domain of JEPES is the existing JavaCard and J2ME specifications from Sun, described in Section 2.2. Unlike JEPES, both JavaCard and J2ME permit dynamic loading of classes, which makes ahead-of-time compilation with all of its associated optimizations impossible. Nonetheless, size-based optimizations can be applied to Java systems with dynamic loading, for example by compacting the run-time representation of the program as in JavaCard or performing bytecode factorization [9]. JEPES already uses a highly compact class representation, but code factorization could perhaps be useful, depending on the characteristics of the target embedded system.

Compilation of Java for low-end embedded systems can also be done by compiling to C as an intermediate language. This approach leverages existing, mature compiler technology ensuring that high-quality low-level code is generated, as shown by Nilsson, Nilsson and Ekman [23, 24]. Here, the primary focus is on threads and real-time garbage collection, which is supported by a run-time system that takes up only 10KB of ROM and 1KB of RAM on an Atmel-based system with 128KB of ROM and 64KB of RAM (leaving 118KB of ROM and 63KB of RAM for the application). JEPES currently does not support threads and garbage collection, but can be used on much smaller systems. We expect that similar tecniques could be used in JEPES to implement garbage collection and multithreading for larger systems. The decision in JEPES to directly generate native code was motivated primarily by the need for providing debugging information at the source code level, which is difficult when using C as an intermediate language.

The optimization techniques used in the JEPES compiler are mostly standard, although they are focused on decreas-

[5]Optimizations such as stack allocation and virtual dispatch elimination cannot be represented in the bytecode format, but could be output as annotations in the class files and used by the virtual machine for performing optimizations [26].

	unoptimized Java					CHA-optimized Java			fully optimized Java					
Program	J-size	fields	meth.	J-ins.	h-alloc	fields	meth.	J-ins.	J-size	fields	meth.	J-ins.	h-alloc	s-alloc
KvmHttpTest	67779	119	528	7400	32	119	209	2198	23121	31	93	1048	6	26
Dragon	69264	145	546	7655	32	145	163	1383	22345	45	88	994	6	26
EightQueens	67304	128	539	7402	32	128	184	1587	21227	32	89	943	6	23

Table 1: JEPES bytecode to bytecode compilation results

	unoptimized Java				fully optimized Java					native	
Program	J-size	fields	methods	J-ins.	J-size	fields	methods	J-ins.	ghost	N-ins.	N-size
CAN–serial bridge (static)	8784	8	106	336	3850	0	7	94	0	234	1511
CAN–serial bridge (object)	10653	19	139	371	4561	1	10	107	1	244	1601
Valve controller	6773	10	88	252	2758	10	6	78	3	221	1335
Lego lift	6547	4	69	286	3377	3	15	134	0	497	1562

Table 2: JEPES bytecode to native code compilation results

ing program size rather than increasing performance. Compilation with code-size constraints can be expressed as an integer linear programming problem covering the whole program, which allows highly compact code to be generated, as shown by Naik and Palsberg [22]. JEPES has no similar optimization of low-level code, but rather performs high-level, space saving optimizations at earlier stages in the compiler, which would be complementary. In particular, we note that ghost allocation, which more closely resembles the interprocedural propagation of complex values performed by program specialization rather than a standard compiler technique, is guaranteed not to increase program size (unlike standard program specialization techniques) [28].

JEPES uses interface-directed configuration to instrument the Java program with additional semantic information at the class and method level. Such instrumentation of Java programs is typically done using Javadoc-style comments. Javadoc-style comments have been used for expressing invariants (e.g, [11, 20]), but are inappropriate for annotations that modify the semantics of the program, such as those used in JEPES.

7. CONCLUSION AND FUTURE WORK

Traditionally, applications for embedded devices have been programmed in C (and perhaps assembler) using a procedural programming style. However, many other areas of the software industry have taken a step towards object-oriented methods, which has resulted in significant boosts in productivity. With the introduction of JEPES, the use of object-oriented programming in embedded devices has now become possible. JEPES makes Java technology available on even the smallest devices, enabling the construction of devices with advanced built-in functionality that are both cost and power efficient. The use of Java as a language facilitates aggressive, space-saving optimizations: on our JEPES-specific programs, the compiler reduced the program size to 18.9% of the original on average, which allowed the programs to be stored in approximately 1.5KB of ROM, including runtime support. For the KVM-based Java programs, the size was reduced to 32.6%. Moreover, our experiments clearly demonstrate that CHA should be considered a baseline optimization when compiling programs for low-end embedded systems.

In terms of future work, our primary interest is investigating the use of object-oriented frameworks and design patterns on low-end embedded systems. Frameworks permit significant reuse of design and implementation [29], but often with an overhead in terms of overly general code. For JEPES to efficiently compile programs based on frameworks to low-end embedded systems, more precise analyses and more aggressive optimizations may be needed, but larger experiments are needed to reveal if this is the case. Design patterns are reusable micro-architectures that increase reusability and adaptability of code [16]. However, the programming style imposed by JEPES for low-end embedded systems (e.g., only static and stack allocation of objects) may not be appropriate for implementing most design patterns.

Availability

JEPES is a commercial product of Mjølner Informatics, and is currently available on request as a binary distribution. Detailed information on JEPES can be found in [6].

8. REFERENCES

[1] D.F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for java. In *Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation*, pages 258–268. ACM Press, 1998.

[2] H.G. Baker. The treadmill: real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, 1992.

[3] S.M. Blackburn, S. Singhai, M. Hertz, K.S. McKinely, and J.E.B. Moss. Pretenuring for java. In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications*, pages 342–352. ACM Press, 2001.

[4] B. Blanchet. Escape analysis for object-oriented languages: application to java. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 20–34. ACM Press, 1999.

[5] Bosch. Bosch CAN homepage. `<URL:http://www.can.bosch.com>`.

[6] K. Burgaard. Extending the reach of java to low-end

embedded systems. Master's thesis, DAIMI, University of Aarhus, 2001.

[7] CAN-CiA.org. CAN in automation. <URL:http://www.can-cia.org.

[8] J.-D. Choi, M. Gupta, M. Serrano, V.C. Sreedhar, and S. Midkiff. Escape analysis for java. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19. ACM Press, 1999.

[9] L. Clausen, U.P. Schultz, C. Consel, and G. Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(3):471–489, May 2000.

[10] Hewlett-Packard Company. TurboChai compiler. Web site; http://www.hp.com/products1/embedded/products/devtools/turbochai.html.

[11] J.C. Corbett, M.B. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *SPIN*, pages 205–223, 2000.

[12] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Walter G. Olthoff, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, Århus, Denmark, August 1995. Springer-Verlag.

[13] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 21(11):966–975, 1978.

[14] K. Driesen and U. Hölzle. The direct cost of virtual function calls in C++. In *OOPSLA'96 Conference Proceedings*, volume 31, 10 of *ACM SIGPLAN Notices*, pages 306–323, New York, NY, USA, October 1996. ACM Press.

[15] Esmertec. Products by esmertec, inc. Web site; http://www.esmertec.com/.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[17] M.W. Hicks, J.T. Moore, and S.M. Nettles. The measured cost of copying garbage collection mechanisms. In *Proceedings of the ACM SIGPLAN international conference on Functional programming*, pages 292–305. ACM Press, 1997.

[18] IBM. IBM VisualAge micro edition: The J9 virtual machine. Available at http://www.embedded.oti.com/download/mkt/j9.pdf.

[19] I.H. Kazi, H.H. Chen, B. Stanley, and D.J. Lilja. Techniques for obtaining high performance in Java programs. *ACM Computing Surveys (CSUR)*, 32(3):213–240, 2000.

[20] G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.

[21] T.F. Lim, P. Pardyak, and B.N. Bershad. A memory-efficient real-time non-copying garbage collector. In *Proceedings of the international symposium on Memory management*, pages 118–129. ACM Press, 1998.

[22] M. Naik and J. Palsberg. Compiling with code-size constraints. In *Joint Conferences on Languages, Compilers and Tools for Embedded Systems & Software and Compilers for Embedded Systems*, pages 120–129, June 2002.

[23] A. Nilsson and T. Ekman. Deterministic Java in tiny embedded systems. In *The Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, pages 60–68. IEEE Computer Society, May 2001.

[24] A. Nilsson, T. Ekman, and K. Nilsson. Real Java for real time — gain and pain. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (SCOPES'2002)*, pages 304–311. ACM Press, 2002.

[25] Markus L. Noga. About legOS. <URL:http://www.noga.de/legOS>.

[26] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing java using attributes. In *Compiler Construction, 10th International Conference (CC 2001)*, pages 334–554, 2001.

[27] Silicomp RI. Turboj java to native compiler. Web site; http://www.ri.silicomp.fr/adv-dvt/java/turbo/index-b.htm.

[28] U.P. Schultz, , J.L. Lawall, and C. Consel. Automatic program specialization for Java. *TOPLAS*. Accepted for publication.

[29] M. Shaw and D. Garlan. *Software Architecture.* Prentice Hall, 1996.

[30] F. Siebert. Constant-time root scanning for deterministic garbage collection. In *Compiler Construction, 10th International Conference (CC 2001)*, volume 2027 of *LNCS*, pages 308–318, Genova, Italy, April 2001. Springer-Verlag.

[31] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a java just-in-time compiler. In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications*, pages 180–195. ACM Press, 2001.

[32] Sun Microsystems. *JavaCard 2.1.1 Application Programming Interface*, May 2000. Available at http://java.sun.com/products/javacard/.

[33] Sun Microsystems. *JavaCard 2.1.1 Runtime Environment Specification*, May 2000. Available at http://java.sun.com/products/javacard/.

[34] Sun Microsystems. *JavaCard 2.1.1 Virtual Machine Specification*, May 2000. Available at http://java.sun.com/products/javacard/.

[35] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 187–206. ACM Press, 1999.

[36] WindRiver. WindRiver Systems Java overview. Web site; http://www.windriver.com/internet/html/java.html.