Exploiting Static Application Knowledge in a Java Compiler for Embedded Systems: A Case Study

Christoph Erhardt Michael Stilkerich Daniel Lohmann
Wolfgang Schröder-Preikschat
{erhardt,stilkerich,lohmann,wosch}@cs.fau.de
Friedrich-Alexander University Erlangen-Nuremberg, Germany

ABSTRACT

Offering many benefits in terms of productivity and reliability, Java is becoming an attractive choice for the field of embedded computing. However, its programming model that relies on the capabilities of just-in-time compilation limits the opportunities to generate highly optimized code in an ahead-of-time compiler. In the KESO project, a Java VM for statically-configured deeply embedded systems, we have previously used static application knowledge to create a tailored runtime environment. In this paper, we present and discuss how this static knowledge can further be exploited by our compiler in order to perform advanced optimizations that would otherwise not be achievable. We conducted a case study with the CD_x real-time benchmark in which we examined the peculiarities and challenges that arise, and evaluated the effectiveness of both standard and system-model-specific compiler optimizations in the context of a static embedded application model. Our results show that incorporating the available a-priori knowledge in the compiler provides significant improvements to both footprint and runtime, and can additionally help the system integrator to identify consistency problems between the code and a higher-level system specification at an early development stage.

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

Reliability, Design, Languages

Keywords

KESO, Java, embedded systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
JTRES 2011 September 26-28, 2011, York, UK
Copyright 2011 ACM 978-1-4503-0731-4I/11/09 ...$10.00.

1. INTRODUCTION

Java is widely accepted in many computing domains. For reasons of lower software maintenance costs, improved productivity over traditional languages [20, 22] and better availability of trained programmers [17], Java is becoming more attractive to fields of computing other than its traditional target domains, such as embedded and real-time systems.

For these domains, however, the system model of Java is rather unsuitable. Java has originally been designed for cross-platform interoperability and the secure, sandboxed execution of code dynamically loaded from potentially untrusted sources, for example for the execution within a web browser. Its portability goals and dynamic features make it a difficult source language for static, ahead-of-time compilers, as native code is unportable by its nature and many powerful compiler optimizations become impractical due to Java's programming model. A prominent example is method inlining, which gets limited to few special cases in the presence of a combination of method overriding and dynamic class loading. The most common technique for the efficient (and portable) execution of Java bytecode is the use of a just-in-time (JIT) compiler on the target machine that compiles portions of the Java program into native code during runtime of the application and incorporates its volatile runtime state (e.g., the current class hierarchy) to overcome the limiting impact of the Java system model on compiler optimizations. As the relevant runtime state changes (e.g., through the loading of a new class), the recompilation of parts of the program may become necessary. In the domain of embedded and real-time systems, JIT compilation is normally infeasible or not the preferred option for reasons of runtime overhead and determinism.

The system model of static embedded applications is in sharp contrast to that of Java. Most of Java's dynamic features that negatively impact optimizations in ahead-of-time compilers are not needed or undesirable, because they likewise complicate the static verification of the program. In the context of the KESO project [28], we have previously harvested the a-priori knowledge available in statically-configured embedded applications to derive the set of Java features needed by a particular application through a static code analysis. Based on this knowledge, KESO's ahead-of-time Java-to-C compiler generates a JVM variant that specifically fits the requirements of a particular application. This avoids both the overhead of unneeded features and the restrictions of fixed Java profiles, as found in traditional approaches for scaling Java to small devices such as the Java 2 Micro Edition [14].

In this paper, we analyze and evaluate how the knowl-

edge available in statically-configured embedded applications about the system model and the application itself, either explicitly expressed in the form of a system description file or extractable from the code by means of static whole-program analyses, can be leveraged for both traditional and new system-model-specific compiler optimizations that can normally only be carried out by a JIT compiler.

This paper is structured as follows: In Section 2, we give a description of the widely spread OSEK/VDX system model that serves as the base for our case study, and the KESO JVM that provides a Java runtime environment for this system. Section 3 presents the case study that we performed at the example of the CD_x real-time Java benchmark, followed by an experimental evaluation of the results in Section 4.

2. DOMAIN ANALYSIS

The class of applications in the scope of this paper are statically-configured embedded systems. In this section, we will first briefly introduce the properties of these applications and the surrounding market conditions, and then give an introduction to the OSEK/VDX system model and KESO, a JVM implementation specifically designed for the OSEK/VDX system model.

2.1 Surrounding Conditions

By statically-configured embedded applications we refer to applications where both the operating-system-level objects (threads, interrupt service routines, synchronization locks, etc.) and the entire code of the application are known ahead of time. This type of application covers many, if not most, traditional embedded applications from the electronic control units found in appliances to safety-critical tasks such as the electronic stability program (ESP) and many other electronic functions found in modern cars. These devices are normally mass products, subject to the immense cost pressure usual in this market where cost differences of few cents on the single device amount to huge values considering the whole of the produced devices. The tolerance towards added cost is particularly limited when caused by things that do not directly pose an added value to the customer, which is typically the case for infrastructure software such as the operating system (OS) or, in the case of Java, the Java runtime environment. The automotive industry has developed an own OS standard, OSEK/VDX [19], which defines a lightweight OS layer that can scale with the requirements of the application.

2.2 OSEK/VDX System Model

An OSEK/VDX application consists of a fixed number of tasks (threads) that are statically created in the system, and interrupt service routines (ISR) that are statically assigned to an interrupt source provided by the hardware. Tasks can be activated (i.e., started or restarted) and terminated, but cannot be destroyed. They are scheduled based on statically assigned priorities, with tasks of equal priority being scheduled by first-come, first-served. A stack-based priority ceiling protocol can be used to synchronize tasks. The locks are also statically created in the OS and use predetermined ceiling priorities. OSEK/VDX distinguishes two types of tasks. Basic tasks have run-to-completion semantics and can thus share a single stack. Extended tasks may block during the execution to wait for an event.

Besides the code, the OSEK/VDX application needs to provide a system description file [18] that defines the instances of OS objects (tasks, ISRs, locks), their attributes, and relationships between those objects. The attributes comprise things such as the task priorities, the type of a task (basic or extended), and the entry functions of tasks and ISRs. The relationships among OS objects define which system objects interact with each other, for example, which events a task will wait for or which locks a task will acquire.

The information contained in the system description file is used by the OSEK/VDX implementation to create an OS variant containing statically allocated instances of the defined OS objects. In addition, the OSEK/VDX standard defines four scalability classes that provide standard size/functionality trade-offs, of which the most suitable one can automatically be selected from the information in the configuration file. Besides these standard variants, many OSEK/VDX implementations ship with a code generator that performs a more sophisticated and fine-grained procedure to create an OS variant that is specifically tailored for the application in order to avoid unnecessary overhead.

2.3 Tailoring the JVM

With the KESO JVM [28], we adopted from OSEK/VDX the idea of creating a tailored version of the infrastructure software that provides only the features required by the application at hand. KESO features a Java-to-C-code ahead-of-time compiler specifically designed for the domain of statically-configured embedded systems. Besides the full source code of the entire application, KESO's compiler *jino* is provided with a system configuration file that contains all the information found in an OSEK/VDX configuration file plus KESO-specific extensions. It uses the information contained within that file in combination with static analyses of the application code to determine which JVM features of the application are needed. On the one end, this can be rather coarse-grained features explicitly expressed in the configuration file, for example that the application wants to use a garbage collector (GC). On the other end, there are features that are automatically determined from the post-reachability-analysis code of the application, for instance support for floating point, 64-bit integers or more elementary things such as the need for virtual method calls. Some features are tuned on a more fine-grained level, for example in the case of virtual method binding the dispatch tables are only generated for methods for which *jino* failed to provide a full static binding.

In this paper, we empirically investigate how this static knowledge impacts compiler optimizations on the application code itself, and additionally present some specific compiler optimizations that build on the compiler's awareness of the system model. The base system for our case study is the KESO JVM, to which we will first give a brief introduction.

2.4 An Overview of the KESO JVM

KESO is a Java runtime environment for deeply embedded, statically-configured applications. Figure 1 shows the architecture of a KESO system. Originally developed with an automotive background, KESO is based on an OSEK/VDX [19] or AUTOSAR [2] OS. KESO is, however, not limited to automotive applications and we have used it in other domains as well, for example there is a full port of the control software of a quad-rotor helicopter [30]. The OSEK/VDX OS is responsible for scheduling tasks and provides the basic synchronization and notification facilities. KESO comes with

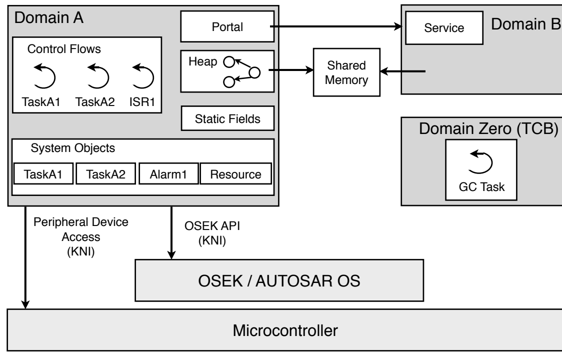


Figure 1: Architecture of KESO

a Java class library that maps the standard Java thread API as far as feasible and possible to OSEK/VDX tasks. Its class library also provides interfaces that allow low-level programming in Java (e.g. writing device drivers), in particular a `RawMemoryAccess` mechanism as defined in the Real-Time Specification for Java (RTSJ) [10] and an own mechanism that allows to map Java objects to raw memory areas.

While a number of similar AOT Java VMs for real-time embedded systems are available [26, 24, 21], KESO is particular in that it puts a strong focus on tailoring the runtime system according to the application's needs.

2.4.1 *System Architecture*

KESO supports the isolated execution of different components or applications. The realms of isolation are called *domains*. Each system object belongs to a domain, and every control flow is executed in the context of a domain that determines which Java objects the control flow is able to access. The isolation of domains in space is established by a strict separation of the object heaps of the different domains and a separate set of static fields in every domain. Domains are in many ways similar to Java Isolates [11], but there are some important differences. A differentiation of KESO domains from Java Isolates is available in a previous paper [28].

Domains may export a functional interface, a so-called *Service*, that can be invoked from other domains by using a proxy object (*Portal*) that represents the service in the foreign domain. Deep copying is used for parameters and return values in portal calls to retain the heap separation. As a copy-free alternative to the portal mechanism, KESO also provides shared memory areas that can be accessed by a controllable set of domains using the same programmatic interfaces that are available for accessing raw memory areas.

2.4.2 *Garbage Collection*

The strict separation of the heaps and static fields leads to disjoint object graphs in the different domains. This allows garbage collection to be performed individually for each domain. Garbage collection is an optional feature in KESO that can be enabled on a per-domain basis. KESO supports two flavors of garbage collection, a stop-the-world GC that pauses the application for the duration of the garbage collection and an incremental GC that can be interrupted by the application but requires additional synchronization overhead. To allow the garbage collector to scan the references on the tasks' runtime stacks, KESO uses Henderson's linked stack frames [9].

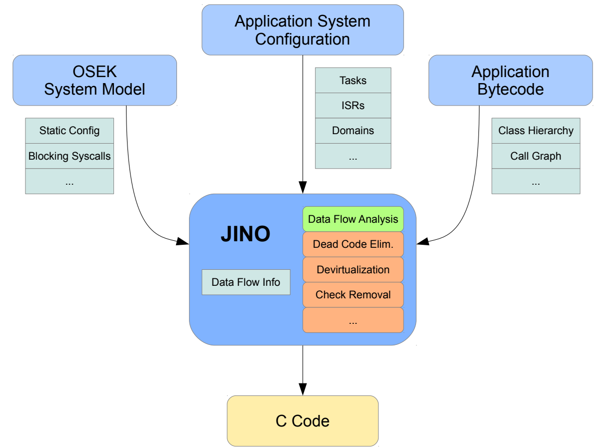


Figure 2: Information Flow into the *jino* Compiler

2.4.3 *Toolchain*

The core of KESO is *jino*, a Java-to-C ahead-of-time compiler. It acts as a driver for the `javac` compiler, then parses the resulting `.class` files and transforms the bytecode into an intermediate representation. After performing a number of analyses and applying a number of transformations and optimizations, *jino* emits a set of C code files along with an OSEK/VDX system configuration file.

Choosing C as the target language enables us to make use of the existing standard low-level optimization capabilities present in most modern C compilers such as loop transformations – preventing us from having to reinvent the wheel. However, the Java programming model, being object-oriented and tailored towards dynamic JIT compilation, demands a whole set of additional optimization techniques in order to overcome its inherent weaknesses with respect to code performance and efficiency and to yield competitive results. Such techniques rely on the availability of domain-specific high-level information: on the one hand, static knowledge that stems directly from the application configuration; on the other hand, information about the program code that is collected and inferred. Due to the dynamic nature of Java, the latter is normally only available at runtime. This is the case because most of this information is complete and predictable only as long as the program code remains unchanged, which for instance no longer applies after a new class has been loaded dynamically.

As dynamically modifying application code at runtime is not a viable option in our target domain, we deliberately omit this feature in KESO. This enables us to collect the extensive information needed to apply advanced optimization techniques in our ahead-of-time compiler that would otherwise not be possible.

In general, the static knowledge used by *jino* for tailoring and optimizing the application can be divided into three categories, as pictured in Figure 2:

1. **Meta-information.** Many structural properties and general considerations are predetermined through the use of an OSEK/VDX or AUTOSAR OS as the basis upon which the application is built. These basic "meta-properties" are hard-wired within the compiler, fundamentally affecting its design and its inner workings. The fact that the system model dictates the entire system to be configured statically

constitutes the very foundation upon which KESO is built. Other system aspects that are reflected directly within *jino* are the distinction between simple and extended tasks (see Section 2.2), the knowledge which system calls are blocking and which are non-blocking, and the priority-based task scheduling mechanism.

2. **System configuration.** The application configuration file tells *jino* about the concrete setting of the application world. Most prominently, it declares into which memory protection domains the system is split, and for each domain at which entry points the control flow starts. Both are essential for the analysis of the code.

3. **Application code.** The program bytecode itself constitutes the information that is processed and transformed by the compiler. Effectively, the configuration determines a specialized variant of the code base. Due to the system's static nature, the class hierarchy, the call graph, etc. remain constant at runtime – it is not possible to load any additional code a-posteriori. This allows us to perform a whole-program analysis and aggressive optimizations ahead of time.

Starting from these three classes of static knowledge, it is possible to collect extensive information about the program that can be exploited in order to improve on the emitted C code. To achieve this, a detailed whole-program data flow analysis is needed.

2.4.4 Whole-Program Analysis

We implemented an iterative SSA-based [4, 27] work list algorithm that combines data flow and 0CFA [25] control flow analysis. Each domain is processed separately starting at its configured entry points. The intermediate code is visited from there; information is aggregated about the values, types and validity of all instruction operands, and all instructions are evaluated statically wherever possible. This includes both arithmetic expressions and control-flow-related operations such as conditional jumps or method invocations. Because the entire code base and task configuration is known ahead-of-time, the gathered knowledge accurately reflects the actual state of the program at any point in time during its execution, and can be exploited for both tailoring the runtime system and applying aggressive optimizations.

In the following section, we will discuss the optimization opportunities that arise through the static application knowledge at the example of the CD_x benchmark.

3. A CASE STUDY WITH KESO

In our case study, we investigate to which extent the overhead added by the Java programming model, mechanisms of object-orientation and Java programmer habits can be compensated by compiler optimizations that leverage knowledge available on the system model and the application itself.

The application we examine in our case study is the Collision Detector (CD_x) benchmark [12]. CD_x is a family of real-time benchmarks with variants for Java standard virtual machines and RTSJ-compliant Java implementations. In addition to the Java version of the benchmark (CD_j), there is a C port (CD_c) of the benchmark that allows to compare the Java variants with a C implementation. The benchmark simulates an application that monitors the aircraft in a radar-controlled airspace and detects potential collisions. The two core components of the benchmark are an air traffic simulator that generates simulated radar frames from various parameters such as the size of the simulated airspace and the maximum number of planes within the airspace, and a detector component that analyzes the radar frames for aircraft that are closer to each other than a pre-configured proximity radius. For resource-constrained devices, the air traffic simulator can be replaced by a simpler online frame generation, or pre-simulated radar frames can be compiled into the application.

CD_j is interesting in that it is an application that has been originally developed in Java and subsequently ported to C, not vice versa. It may be for this reason that CD_j looks different from most traditional embedded applications. It strongly utilizes Java's standard collection classes and programming idioms such as short-lived objects. In contrast, C does not provide a class library as comprehensive as that of Java, which is one reason why embedded C applications mostly use custom data structures and only static memory allocation or specialized forms of dynamic memory management. From our own experience in the automotive domain, we argue that an application originally written in C and subsequently ported to Java would show more similar characteristics to the original code with respect to the use of standard library code and memory allocation. Interestingly, an earlier investigation [28] of the CD_x benchmark revealed that the C port of the benchmark shows a shift towards the characteristics of C applications without a compulsory technical reason, most notably the increased use of preallocated memory where the Java version allocated memory on demand. We believe that CD_j is a good representative for a software as it could have been developed by Java programmers and poses a good subject for our case study.

In the remainder of this section, we will present different optimization techniques that are based on our observations of CD_j, either by manual code analysis or as a result of profiling. We have not yet implemented all of these optimizations, however, most of those not implemented yet can be partially evaluated by hand-tuning the code of the benchmark. We will discuss the impact of the different optimizations in Section 4.

3.1 Local Optimizations

The *jino* Java-to-C compiler comes with a set of standard optimizations such as method inlining, local constant propagation and folding or copy propagation. As these techniques are very common and require no extensive static application knowledge, we will not elaborate further on them in this paper and instead take them for granted, focusing on our advanced whole-program optimizations.

3.2 Runtime Check Elimination

While the object-oriented programming paradigm makes it easier to write software in a cleaner and more structured way, its combination with strict type safety in Java initially causes a significant overhead in both execution time and footprint. This is the case because several operations cannot be fully validated statically, but have to be checked at runtime, most notably:

- For all invocations of non-static methods and accesses to object fields, the associated object reference must be valid, that is, non-`null`.
- All array accesses must be within the array's bounds.

Consequently, corresponding checks have to be inserted in the appropriate places.

Modern JVMs with JIT compilers are able to determine at runtime which checks are actually necessary and which can be

omitted. This is not possible in our case, where everything is compiled statically. However, both `null`- and bounds check optimization are well-researched [16, 8, 13]. The whole-program analysis can in many cases infer the validity and values of an instruction's operands. This is possible through a combination of inter-procedural constant propagation and further control-flow- and path-sensitive analysis techniques.

The *jino* compiler can omit `null` checks where the object reference in question is known to always be valid, and bounds checks where the index is known to be within the limits.

3.3 Dead-Code Elimination

Being a benchmark *family* rather than a single static application, CD_x offers a wide range of configuration possibilities to produce a specialized variant according to the needs of the user. While this flexibility is welcome in general – with software families being very common in the domain of embedded systems –, it results in different variants utilizing different parts of the generic code base. Consequently, a specific instance of CD_x may never take certain code paths, invoke certain methods, and even use certain classes.

Since *jino* produces C code as an intermediate stage, it is possible to use function-level linking. This by itself is however not sufficiently effective because, for instance, virtual methods that are never actually called in the program will still be referenced in the dispatch table and thus be preserved by the linker [6]. Hence, we implemented advanced dead-code elimination optimizations within *jino*.

As necessary high-level information about the application code gets lost during the stages of the compilation process, it is beneficial to detect and remove dead code at an early point in time. Picking up an idea from Wegman and Zadeck's SCCP algorithm [31], finding such pieces of code in the ahead-of-time compiler is trivial with the whole-program analysis described in Section 2.4.4: All basic blocks, methods and classes that were never visited are dead and can be removed; statically predicted conditional branches can be converted into unconditional jumps. Effectively, this optimization can be regarded as a specialization of the application's code base to its concrete configuration.

3.4 Method Devirtualization

We observed that CD_x, albeit not using polymorphic class hierarchies to an excessive degree, contains a certain amount of virtual methods, whose invocation is expensive compared to simple function calls. With the concrete callee depending on the dynamic type of the callee object, a lookup in the dispatch table is required each time a virtual method is called. If the callee candidate is known to be unique, it is however possible to omit the lookup and use a direct call instead.

This optimization is trivial when the entire class hierarchy of the callee object contains only one method with the appropriate signature – which can be determined ahead of time because the entire application code is known. Additional occurrences of virtual invocations with only one unique callee can be found thanks to the information collected by the data flow analysis: By factoring in object allocations and casts, it is possible to keep track of a variable's dynamic type during compilation. Even if the precise type cannot be identified, for example because instances of different classes flow together into a variable from different paths, it is still possible through class hierarchy analysis [5] to find the most specific of their common superclasses. In this way, the set of callee candidates coming into question is reduced for a further amount of call sites. For those method invocations where it boils down to one, a devirtualization [1, 29] can be performed. This is similar to the *treeshake* transformation implemented by Fitzgerald et al. in their Marmot compiler [7]. We are working on also incorporating the idea of *Rapid Type Analysis* [3] to use the information about instantiated classes to further reduce the set of executable virtual functions. Due to the Multi-JVM architecture of KESO, our analyses yield separate class hierarchies and call graphs for each domain. Rapid type analysis can also be performed separately for each domain. The resulting domain-specific type sets in combination with the domain-specific class hierarchies and call graphs can further support method devirtualization.

```
void foo(obj_t **llref, obj_t *thisp) {
  // 3 reference variables + frame link pointer
  obj_t *references[4] = { NULL, NULL, NULL };

  // link to previous frame
  *llref = references;

  bar(&references[3], thisp);
}

void bar(obj_t **llref, obj_t *thisp) {
  // terminate the list ...
  *llref = KESO_EOLLREF;
  // ... before calling a blocking system service
  WaitEvent(EventID);
}
```

Listing 1: Linked Stack Frames Implementation

It is worth noting that many optimizations effectively go hand in hand and affect each other. For instance, devirtualization may reveal additional dead methods, while finding and eliminating a dead code path may reveal that a certain object reference can be left unchecked because it is valid on all remaining incoming paths. The framework in *jino* combines most of the advanced analyses and transformations into one single pass, respectively, exploiting their interdependencies and simultaneously improving compilation times.

3.5 Selective Use of Linked Stack Frames

As we have mentioned in Section 2.4.2, KESO uses linked stack frames to allow the GC to scan the task stacks for references. Listing 1 illustrates how the linked stack frames are implemented in KESO in the generated C code. The local reference variables of a function are stored in an array `references` rather than individual variables to ensure their physical collocation in memory. Besides the reference variables, the last element of the array contains a link pointer that is used to link to the `references` of the next frame, or contains a marker value `KESO_EOLLREF` to let the GC detect the end of the linked list. To maintain the list, the function interface is extended by a parameter `llref` that points to the previous link pointer and is updated in the prologue of a called function. Initially, the head pointer location which is known to the GC is passed. Before calling a blocking function, the list is terminated with the marker value.

Linked frames add directly visible overhead to the function prologues and calls for maintenance work on the linked list of references. In addition, the reference values need to be initialized with `null` in case the GC scans the list before the

program has assigned a value to the variable. More severely, however, is the hidden overhead: the effect on the C compiler's optimizations, since alias analysis is more complicated when multiple variables are stored as a compound rather than individually. To reduce the overhead caused by linked frames, we only generate them for functions that are potentially active while the GC is running. In the OSEK/VDX programming model, we can limit these as follows:

- Garbage collection is performed at slack time in KESO. This means that all application tasks are either suspended (empty stack) or blocked at that time.
- Heap space exhaustion will never cause a task to block during an allocation. Instead, an exception will be generated.
- Functions reachable only from basic tasks are never active while the GC is running due to the run-to-completion semantics of basic tasks.
- Functions reachable from extended tasks can be active during garbage collection only if they (transitively) invoke a blocking system service.
- All blocking system services are known to our compiler. In OSEK/VDX, there is only a single blocking system service, `WaitEvent()`.

Based on these observations, we only use linked stack frames in blocking functions, that is, functions from which a path in the call graph to a call of the `WaitEvent()` service exists. If a non-blocking function is invoked from a blocking function, the linked frames list is not maintained in this sub-graph of the call graph. An interesting special case are dynamically bound method calls for which both blocking and non-blocking candidates exist. The function interface needs to be the same for all candidates to remain call-compatible. This issue can easily be solved by adding the `llref` as an unused parameter to the non-blocking candidates. Rafkind et al. also restricted the use of linked stack frames to allocating functions in the Magpie C source-to-source compiler [23]. We can apply this technique more aggressively due to KESO's side-stepped garbage collection that will never cause an allocation to be interleaved by a garbage collection. In our target domain, tasks are often periodically executed and only block at a shallow stack depth to wait for the next period, while the actual periodic activity is performed in a called method. CD_j is no exception in that regard. For such applications, the periodically executed code will not require the use of linked frames at all and therefore not suffer from the performance penalties in our system model.

3.6 Variant-Specific Constants

Const-correct code not only helps the compiler to detect possible programming errors in the source code, but also forms the base for constant propagation on non-literal values, which in turn assists many other optimization steps such as the dead-code elimination.

Programmers are often lazy or unaware of the importance of writing const-correct code. Sometimes, furthermore, it is not possible for the programmer to fully qualify all constant fields in a program, for instance if those fields are constant only in some variants of the program. As an example, we look at the variant configuration of the CD_j benchmark, which mostly happens in two classes containing constant value definitions similar to Listing 2. It stands out that most members of the `Constants` class are not qualified as `final`. The reason in this particular case is that some variants of

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

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

Listing 2: CD_j configuration

CD_j can be configured by command-line parameters that change the values of the members of the `Constants` class. In configurations for embedded devices, the command-line configuration of the code is not used, however, and the code that writes to these fields is not reachable.

We propose an analysis step that runs after the reachability analysis to determine which static fields initialized with a constant expression retain their initially assigned value throughout the runtime of the program. Initially, the analysis considers fields to which at most one write operation that resides in the static initializer of the field's class exists in the live code base. As opposed to programmer-declared non-blank static final fields, regular static fields that are assigned a constant expression are initialized in the static initializer. The analysis furthermore only considers fields where the initialization is not dominated by a read of the same field, be it in the static initializer itself or (transitively) in a method that is (directly or indirectly) invoked from a basic block dominating the initialization. Otherwise, a read that dominates the initialization might wrongly observe the field's initialized value in violation of the sequential consistency.

This step needs to be embedded in an iterative compiler framework, as the results of the analysis support other analyses which may in turn lead to new fields that retain a constant value due to more code having been identified as being dead.

3.7 Compile-Time Bug Detection

In addition to providing the infrastructure for optimizations that aim at increasing the performance or lowering the footprint of the generated code, the information gathered in the static code analysis can also be used to detect certain potential bugs in the program code. Warning the developer early at compile time and allowing him or her to review the situation is important especially in safety-critical systems. In the following, we discuss two examples where the knowledge gathered by the static compiler analysis can be leveraged to provide the developer or system integrator with information on the code that allows him or her to cross-check the properties of the code with the intended properties.

3.7.1 System Configuration Cross-Checking

As discussed earlier, an OSEK/VDX (and a KESO) application not only consists of code but additionally comes with a system description file. This description contains OS object configuration and instantiation, but also some properties that provide guarantees given by the programmer on the behavior of application entities such as tasks. The OSEK/VDX sys-

tem generator uses this knowledge to generate an OS variant that is specifically tuned for the given application. On the other hand, if the application violates the properties assured by the definitions in the system configuration to the system generator, the runtime behavior is undefined according to the OSEK/VDX specification [19]. This is comparable to using a `const` qualifier in C code that is subsequently overridden and violated by using a type cast. The generated code does not necessarily still function as expected.

Basic Tasks. Basic tasks are tasks with run-to-completion semantics. In OSEK/VDX, this means that a basic task must never block to wait for an event. The system description defines for each task whether it is a basic or an extended task. The OSEK/VDX scheduling model allows basic tasks to share a single stack whereas extended tasks need to be provided with own stacks. The information gathered by *jino* to restrict the use of linked stack frames allows to cross-check whether the entry functions of the basic tasks have been marked as blocking functions. In that case, *jino* is able to produce a warning that allows the developer to check whether the system description is incorrect or whether *jino* produced a false positive, as the blocking path is actually dead but was not recognized as such by the analysis.

System Object Use and Service Protection. The system description also contains information on what system objects a control flow interacts with, for example the resources (i.e., locks) it will acquire at runtime, which is used to statically compute the locks' ceiling priorities. KESO extends on the standard definitions by introducing service protection that provides a runtime mechanism to inhibit the use of system objects in violation to the definitions in the system description. AUTOSAR OS provides a similar mechanism. The control-flow-specific reachability information gathered by our analysis framework can be used to check whether a control flow accesses system objects only in the way defined by the system configuration. In addition, it can also check the service protection constraints at compile time in many cases, which eliminates the overhead of the runtime check otherwise needed. As before, this analysis may produce false positives, so *jino* only outputs a warning that needs to be checked by the developer.

3.7.2 Analysis of used raw memory areas

The `RawMemoryAccess` API [10] provides a way for Java applications to access memory regions at a specific physical memory address. This mechanism is often sufficient to write device drivers in Java if the device can be configured by memory-mapped device registers. The mechanism, however, also potentially poses a problem to Java's memory safety, if the raw memory region happens to overlap with a managed memory region of the Java runtime such as the heap. If the VM supports isolation of different components like KESO does, there may also be undesired interference of different protection domains accessing the same physical memory region. With the control-flow-specific reachability information at hand, *jino* can tell which raw memory regions are accessible from which domains[1]. This information is on the one hand used by *jino* to check the raw memory ranges for intersection with the regular RAM regions (extracted from the linker file's memory description), which is signalled as an error. On the other hand, a list of the raw memory address ranges that are reachable by each domain can be provided to the developer, who can check with the target's datasheet whether these regions match the mappings of the respective devices. Also, access by multiple domains to the same raw memory area is detected by this mechanism, aiding the developer in finding potentially problematic shared access to a device.

[1]KESO requires compile-time constants to be used when creating a raw memory region. We believe that this is sufficient for the intended purpose of writing device drivers.

4. RESULTS

We use version 1.2 of the CD_j benchmark for our evaluation. The evaluation is split in two parts: In Section 4.2, we test the effectiveness of our optimizations mainly with respect to the execution time by running a CD_j configuration on a microcontroller typical for the target domain. Due to the memory constraints of the target controller, however, we are not able to run the full simulator task on our microcontroller and therefore only use a single-domain configuration with a detector task that is provided with radar frames generated using CD_j's simple *On-the-Go* frame generator. For the evaluation of the domain- and control-flow-sensitive portions of our analysis, we test a two-domain configuration with one domain containing the simulator and the other containing the detector component on an x86 platform in Section 4.1. As the characteristics of the x86 architecture are not representative for our target domain, we will mostly focus on relative and architecture-independent metrics as far as execution time is concerned, whereas the footprint is evaluated using binaries generated for the microcontroller used in Section 4.2.

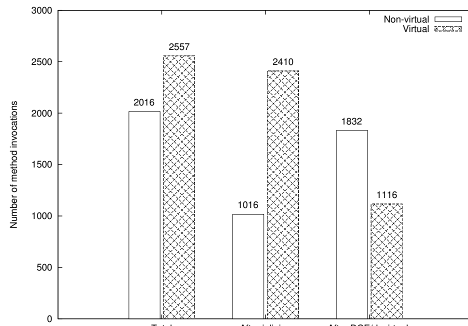
4.1 Multi-Domain Configuration

In the following, we analyze and discuss the results of our optimizations on the two-domain configuration of CD_j. The evaluation examines the concrete optimizations and the way they affect the resulting C code, as well as assesses their eventual effects on the deployment and execution of the application binary.

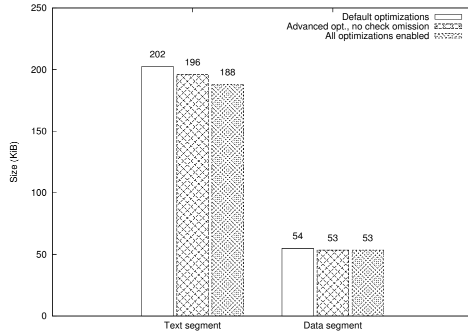
4.1.1 Dead-Code Elimination and Devirtualization

In order to measure the effectiveness of dead-code elimination and devirtualization, we counted the number of method invocations at three stages in the compiler. The results for each stage are depicted in Figure 3(a), distinguished between non-virtual and virtual method calls:

1. In the application bytecode, more than half of all invocations are virtual. This is a rather difficult starting situation for optimizations that is owed to the extensive use of the Java class library in CD_j.
2. After method inlining has been performed, the number of statically bound invocations has shrunk significantly. Of the virtual calls, on the contrary, only a very small number could be inlined because the existence of multiple candidates prevents this type of optimization.
3. The combined dead-code elimination and method devirtualization pass produces two effects: On the one hand, the total number of call sites is reduced because dead basic blocks are discarded. On the other hand, a significant amount of virtual invocations are converted into non-virtual ones. At this stage, it would be beneficial to re-execute the inlining pass because it would find several more suitable methods. This is not happening at the moment since we have yet to finish

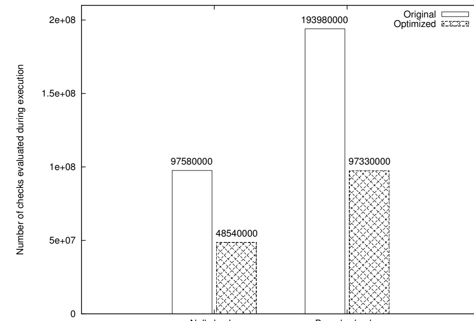


(a) DCE and Devirtualization

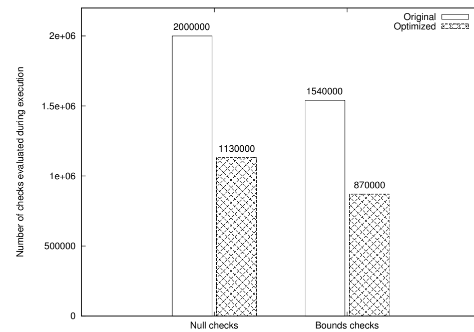


(b) Code and Data Sizes

Figure 3: Code-Size-Related Optimization Results



(a) CD_j Simulator



(b) CD_j Detector

Figure 4: Runtime Checks During Execution

adapting the compiler to the new optimization framework.

4.1.2 Runtime Check Elimination

The `null`-check omission is fairly effective: The data flow analysis proves 78 % of all object accesses to always operate on a valid reference, eliminating the need to check the operand for validity during execution. This is mainly enabled by the inter-procedural aspect of the analysis, because both actual arguments and return value at a call site are connected to the formal arguments and return value of the callee.

When it comes to array accesses, the picture is more diverse: A mere 49 % are found to be within bounds. The remainder, for which runtime checks are emitted, includes many operations on collection classes such as `ArrayList` and `HashMap` that contain a variably-sized array. In these cases, the compiler has a tough time trying to prove the legality of loads from or stores to the array.

We also counted the system object lookups (see Section 3.7.1). In the case of CD_j, 13 out of the 16 lookups could be evaluated statically and the service protection mechanism could be disabled in these places.

The omission of runtime checks has a directly correlated impact on the code size, which we will evaluate and discuss in the following section. Its effect on the actual execution speed of the program, however, is by far more complex to estimate. Even little optimizations in "hot spots" of the code can yield massive performance gains whereas huge improvements in code that is only executed once can have negligible repercussions – therefore the percentage of eliminated checks per se has only a limited expressiveness in this respect.

For this reason, we examined how many checks are evaluated when the CD_j application is executed with 10,000 radar frames (i.e., iterations). The results are shown separately for the simulator and the detector part in Figure 4 since the two parts exhibit vastly different characteristics.

It can be noticed that the overwhelming majority of all checks is performed in the simulator, which generates the radar frames for consumption by the detector. The largest portion (over 95 %) of these checks is carried out within only a few methods of the following classes: `String`, `StringBuilder`, `StringBuffer`, `Long` and `Yylex`. These classes involve the creation and manipulation of short-lived string objects for lexing and parsing aircraft information. This is a behavioral pattern that is rather uncommon in real-world embedded systems, thus the results have only limited significance for our targeted field of application.

The detector component, which is closer to typical "embedded" code, performs no advanced string operations and consequently does not contribute significantly to the number of visited checks. Of both `null`- and array bounds checks, 43–44 % are elided, respectively. Most of the remaining checks are performed in small methods that are frequently invoked. These methods include `HashMap` and `ArrayList` element accesses, the computation of hash codes and the comparison of `Aircraft` objects. Such checks are obviously hard to elide.

4.1.3 Memory Footprint

To analyze the actual effect these optimizations have on space consumption in practice, we determined the segment sizes of the ELF binaries with the aid of the `size` utility. The results are visualized in Figure 3(b).

With all optimizations enabled, the footprint of the text segment is reduced by about 7 %. The effect of the omission of redundant runtime checks lies roughly in the same order of magnitude as that of the removal of dead code and the remaining advanced optimizations. The data segment is shrunk by 2.7 % due to the elimination of unused static

fields. Further improvement opportunities to reduce the memory footprint are discussed in the following section at the example of the single-domain variant of CD_j.

4.2 Execution Time

We run our experiments on an Infineon TriCore TC1796 microcontroller clocked at 150 MHz. Our board is equipped with 2 MiB of Flash ROM and 1 MiB of SRAM. We use CiAO [15] as OS, an own AUTOSAR OS implementation. Our CD_j configuration uses six airplanes. We run the collision detector on 10,000 frames. The collision detector task is periodically released with a period of 40 ms. In addition, CD_j contains two additional tasks, which mainly perform initialization work and output the benchmarks results upon completion of the benchmark. Both of these tasks are blocked during the entire execution of the benchmark. Our GC task is configured with the lowest priority in the system and uses the slack time the detector task leaves in each period. We only use KESO's stop-the-world collector in our benchmarks. A comparison of KESO's garbage collectors using the CD_j benchmark is available in a previous paper [28].

Figure 5 shows the results of our execution time measurements. Figure 5(a) shows the result with the current default optimizations, which includes the use of the new optimization framework. We will consider the other variants relative to this baseline. Figure 5(b) shows the execution times for a variant that uses linked stack frames for all methods. The impact on both execution time (50 %) and code size (22.9 %) is significant and shows how severely the innocent looking differences affect the C compiler's code optimizations.

The analysis to identify variant-specific constant values as discussed in Section 3.6 is not yet implemented in the *jino* compiler. To get an impression of the effect that this optimization would provide for the CD_j code, we created a hand-tuned variant of the CD_j code by marking all fields of the `Constants` class as `final` and commenting out the (dead) function that writes these fields. Figure 5(c) shows that the execution time improves only marginally by about 1 % with these changes, however, the dead-code elimination is able to identify more dead code and consequently the code size is reduced by 15.9 % and the statically allocated data shrinks by 36.6 %. The latter is mostly a consequence of string constants being eliminated that are used in some output code that is disabled in the particular configuration.

5. CONCLUSIONS

In this paper, we performed a case study on the real-time embedded benchmark CD_j to determine the extent to which static application knowledge can aid both standard and system-model-specific compiler optimizations in a Java ahead-of-time compiler. The system model that we used as a basis is the widely spread OSEK/VDX operating system standard. While predominantly used in the automotive industry, we believe that the system model of OSEK/VDX is representative for many static embedded applications. We used the KESO JVM, which was specifically designed for this application domain, as the basis for our evaluation and extended it by a new iterative compiler-optimization framework. Our results show that the static knowledge could significantly improve on both the execution time and footprint of the CD_j benchmark. We did, however, also find that parts of CD_j, most notably the simulator component, contain code portions that we consider quite atypical for deeply embedded applications, such as the lexical analyzer in the simulator. The lion's share of hot runtime checks that could not be statically optimized belongs to these portions of the code. We leave as future work additional framework improvements and an evaluation of the effectiveness of these optimizations on a code base that has not originally been written in Java, but was later ported to it and shows a more static characteristic than the CD_j benchmark. Besides cost-oriented optimizations, we have also reviewed how the information gathered by the static analyses can be used to provide consistency checking between the high-level application properties denoted in an external system configuration file and the actual code base to the system integrator, which can help detecting problems at an early stage.

6. REFERENCES

[1] G. Aigner and U. Hölzle. Eliminating virtual function calls in C++ programs. In *10th Eur. Conf. on OOP (ECOOP '96)*, pages 142–166, London, UK, 1996. Springer.

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

[3] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. *SIGPLAN Not.*, 31(10):324–341, 1996.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct 1991.

[5] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *LNCS*, 952:77–101, 1995.

[6] K. Driesen and U. Hölzle. The direct cost of virtual function calls in C++. In *11th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '96)*, Oct. 1996.

[7] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An optimizing compiler for Java. *Softw. Pract. Exper.*, 30(3):199–232, 2000.

[8] R. Gupta. A fresh look at optimizing array bound checking. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '90)*, PLDI '90, pages 272–282, New York, NY, USA, 1990. ACM.

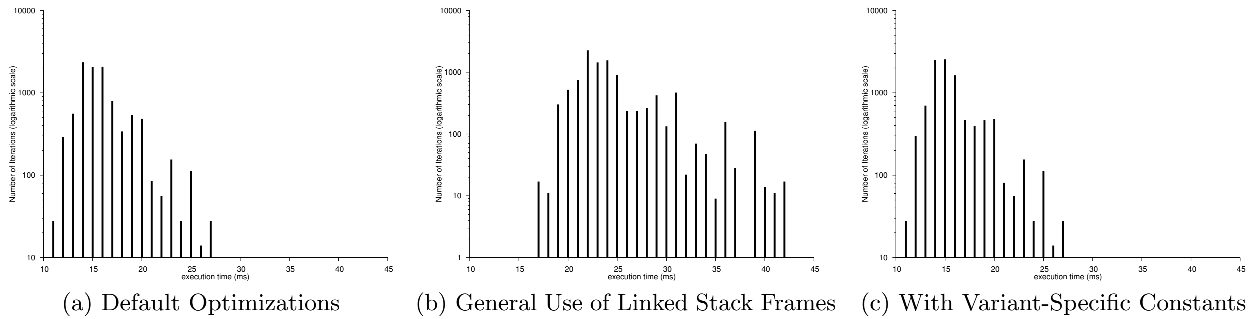
[9] F. Henderson. Accurate garbage collection in an uncooperative environment. In *ISMM '02: 3rd Int. Symp. on Memory Management*, pages 150–156, New York, NY, USA, 2002. ACM.

[10] JSR 1: Real-time Specification for Java. Sun Microsystems JCP, May 2006.

[11] JSR 121: Application Isolation API Specification. Sun Microsystems JCP, June 2006.

[12] 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.

[13] P. Kolte and M. Wolfe. Elimination of redundant array subscript range checks. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*



(a) Default Optimizations (b) General Use of Linked Stack Frames (c) With Variant-Specific Constants

	Default Optimizations	General Use of Linked Stack Frames	With Variant-Specific Constant Analysis
Median	15.9 ms	23.9 ms (50 %)	15.8 ms (-1 %)
Worst	27.2 ms	42.1 ms (55 %)	27.3 ms (0 %)
Best	11.9 ms	18.0 ms (51 %)	11.8 ms (-1 %)
Code Size	55.2 KiB	67.8 KiB (22.9 %)	46.5 KiB (-15.9 %)
Data Size	4.86 KiB	4.86 KiB (0 %)	3.08 KiB (-36.63 %)

(d) Execution Times

Figure 5: Execution Time Measurements

(PLDI '95), PLDI '95, pages 270–278, New York, NY, USA, 1995. ACM.

[14] J2ME building blocks for mobile devices — white paper on KVM and the connected, limited device configuration (CLDC), May 2000. `http://java.sun.com/products/cldc/wp/KVMwp.pdf`.

[15] 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.

[16] V. Markstein, J. Cocke, and P. Markstein. Optimization of range checking. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, SIGPLAN '82, pages 114–119, New York, NY, USA, 1982. ACM.

[17] K. Nilsen. Ada-Java middleware for legacy software modernization. In *JTRES '10: 8th Int. W'shop on Java Technologies for real-time & embedded Systems*, pages 85–94, New York, NY, USA, 2010. ACM.

[18] OSEK/VDX Group. OSEK implementation language specification 2.5. Technical report, OSEK/VDX Group, 2004. `http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf`, visited 2009-09-09.

[19] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, Feb. 2005. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`, visited 2011-08-17.

[20] G. Phipps. Comparing observed bug and productivity rates for Java and C++. *Softw. Pract. Exper.*, 29(4):345–358, 1999.

[21] 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.

[22] E. Quinn and C. Christiansen. Java Pays – Positively. IDC Bulletin W16212, May 1998.

[23] J. Rafkind, A. Wick, J. Regehr, and M. Flatt. Precise garbage collection for C. In *ISMM '09: 2009 Int. Symp. on Memory Management*, pages 39–48, New York, NY, USA, 2009. ACM.

[24] T. Schoofs, E. Jenn, S. Leriche, K. Nilsen, L. Gauthier, and M. Richard-Foy. Use of PERC Pico in the AIDA avionics platform. In *JTRES '09: 7th Int. W'shop on Java Technologies for real-time & embedded Systems*, pages 169–178, New York, NY, USA, 2009. ACM.

[25] O. Shivers. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 164–174, New York, NY, USA, 1988. ACM.

[26] F. Siebert and A. Walter. Deterministic execution of Java's primitive bytecode operations. pages 18–18, Apr. 2001.

[27] V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies, and V. Santhanam. Translating out of static single assignment form. In *Proceedings of the 6th International Symposium on Static Analysis*, SAS '99, pages 194–210, London, UK, 1999. Springer-Verlag.

[28] M. Stilkerich, I. Thomm, C. Wawersich, and W. Schröder-Preikschat. Tailor-made JVMs for statically configured embedded systems. *Concurrency and Computation: Practice and Experience*, 2011. To appear. http://dx.doi.org/10.1002/cpe.1755.

[29] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. *SIGPLAN Not.*, 35(10):264–280, 2000.

[30] P. Ulbrich, R. Kapitza, C. Harkort, R. Schmid, and W. Schröder-Preikschat. I4Copter: An adaptable and modular quadrotor platform. In *Proceedings of the 26th ACM Symposium on Applied Computing (SAC '11)*, pages 380–396, New York, NY, USA, 2011. ACM.

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