Decaf: Moving Device Drivers to a Modern Language

Matthew J. Renzelmann and Michael M. Swift
University of Wisconsin–Madison
{mjr, swift}@cs.wisc.edu

Abstract

Writing code to interact with external devices is inherently difficult, and the added demands of writing device drivers in C for kernel mode compounds the problem. This environment is complex and brittle, leading to increased development costs and, in many cases, unreliable code. Previous solutions to this problem ignore the cost of migrating drivers to a better programming environment and require writing new drivers from scratch or even adopting a new operating system.

We present Decaf Drivers, a system for incrementally converting existing Linux kernel drivers to Java programs in user mode. With support from program-analysis tools, Decaf separates out performance-sensitive code and generates a customized kernel interface that allows the remaining code to be moved to Java. With this interface, a programmer can incrementally convert driver code in C to a Java *decaf driver*. The Decaf Drivers system achieves performance close to native kernel drivers and requires almost no changes to the Linux kernel. Thus, Decaf Drivers enables driver programming to advance into the era of modern programming languages without requiring a complete rewrite of operating systems or drivers.

With five drivers converted to Java, we show that Decaf Drivers can (1) move the majority of a driver's code out of the kernel, (2) reduce the amount of driver code, (3) detect broken error handling at compile time with exceptions, (4) gracefully evolve as driver and kernel code and data structures change, and (5) perform within one percent of native kernel-only drivers.

1 Introduction

Our research is motivated by three factors. First, writing quality device driver code is difficult, as evidenced by the many books and conferences devoted to the subject. The net result of this difficulty is unreliability and unavailability [45]. Second, device drivers are a critical part of operating systems yet are developed by a broad community. Early versions of Unix had only a handful of drivers, totaling a few kilobytes of code, that were written by a single developer–Dennis Ritchie [39]. In contrast, modern versions of Linux include over 3200 driver versions in the kernel source tree, developed by over 300 people and entailing 3 million lines of code [46]. Similarly, Windows Vista was released with over 30,000 available device drivers [2]. As a result, the difficulty of writing drivers has a wide impact. Third, despite attempts to change how drivers are developed, they continue to be written as they have been: in the kernel and in C. A glance at early Unix source code [39] shows that, despite decades of engineering, driver code in modern versions of Linux bears a striking resemblance to drivers in the original versions of Unix.

Efforts at providing a cross-platform driver interface [25, 38], moving driver code to user mode [14, 21, 25, 26, 33, 49] or into type-safe languages or extensions to C [5, 9, 25, 42, 51] have had niche successes but have not seen widespread adoption. While we cannot be sure of the reason, we speculate that use of unfamiliar programming languages and the lack of a migration path have stifled use of these approaches. Prior efforts at re-using existing driver code relied on C extensions not in widespread use, such as CCured [9]. In contrast, systems using popular languages generally require that drivers be written from scratch to gain any advantage [25, 51]. However, many drivers are written by copying and pasting existing code [15, 28]. Thus, it may still be easier for a driver developer to modify an existing C driver than to write a new driver from scratch, even if the environment is simpler to program.

Decaf Drivers takes a best-effort approach to simplifying driver development by allowing most driver code to be written at user level in languages other than C. Decaf Drivers sidesteps many of the above problems by leaving code that is critical to performance or compatibility in the kernel in C. All other code can move to user level and to another language; we use Java for our implementation, as it has rich tool support for code generation, but the architecture does not depend on any Java features. The Decaf architecture provides common-case performance comparable to kernel-only drivers, but reliability and programmability improve as large amounts of driver code can be written in Java at user level.

The goal of Decaf Drivers is to provide a clear migration path for existing drivers to a modern programming language. User-level code can be written in C initially and converted entirely to Java over time. Developers can also implement new user-level functionality in Java.

We implemented Decaf Drivers in the Linux 2.6.18.1 kernel by extending the Microdrivers infrastructure [19]. Microdrivers provided the mechanisms necessary to convert existing drivers into a user-mode and kernel-mode component. The resulting driver components were written in C, consisted entirely of preprocessed code, and offered no path to evolve the driver over time.

The contributions of our work are threefold. First, Decaf Drivers provides a mechanism for converting the user-mode component of microdrivers to Java through cross-language marshaling of data structures. Second, Decaf supports incremental conversion of driver code from C to Java on a function-by-function basis, which allows a gradual migration away from C. Finally, the resulting driver code can easily be modified as the operating system and supported devices change, through both editing of driver code and modification of the interface between user and kernel driver portions.

We demonstrate this functionality by converting five drivers to decaf drivers, and rewriting either some or all of the user-mode C code in each into Java. We find that converting legacy drivers to Java is straightforward and quick.

We analyze the E1000 gigabit network driver for concrete evidence that Decaf simplifies driver development. We find that using Java exceptions reduced the amount of code and fixed 28 cases of missing error handling. Furthermore, updating the driver to a recent version predominantly required changes to the Java code, not kernel C code. Using standard workloads, we show while decaf drivers are slower to initialize, their steady-state performance is within 1% of native kernel drivers.

In the following section we describe the Decaf architecture. Section 3 provides a discussion of our Decaf Drivers implementation. We evaluate performance in Section 4, following with a case study applying Decaf Drivers to the E1000 driver. We present related work in Section 6, and then conclude.

2 Design of Decaf Drivers

The primary goal of Decaf Drivers is to simplify device driver programming. We contend that the key to dramatically improving driver reliability is to simplify their development, and that requires:

- User-level development in a modern language.
- Near-kernel performance.
- Incremental conversion of existing drivers.
- Support for evolution as driver and kernel data structures and interfaces change.

User-level code removes the restrictions of the kernel environment, and modern languages provide garbage collection, rich data structures, and exception handling. Many of the common bugs in drivers relate to improper memory access, which is solved by type-safe languages; improper synchronization, which can be improved with language support for mutual exclusion; improper memory management, addressed with garbage collection; and missing or incorrect error handling, which is aided by use of exceptions for reporting errors.

Moving driver code to advanced languages within the kernel achieves some of our goals, but raises other challenges. Support for other languages is not present in operating system kernels, in part because the kernel environment places restrictions on memory access [51]. Notably, most kernels impose strict rules on when memory can be allocated and which memory can be touched at high priority levels [11, 32]. Past efforts to support Java in the Solaris kernel bears this out [36]. In addition, kernel code must deal gracefully with low memory situations, which may not be possible in all languages [6].

The Decaf architecture balances these conflicting requirements by *partitioning* drivers into a small kernel portion that contains performance-critical code and a large, user-level portion that can be written in any language. In support of the latter two goals, Decaf provides tools to support migration of existing code out of the kernel and to generate and re-generate marshaling code to pass data between user mode and the kernel.

2.1 Microdrivers

We base Decaf Drivers on *Microdrivers*, a user-level driver architecture that provides both high performance and compatibility with existing driver and kernel code [19]. Microdrivers partition drivers into a kernel-level *k-driver*, containing only the minimum code required for high-performance and to satisfy OS requirements, and a user-level *u-driver* with everything else. The k-driver contains code with high bandwidth or low-latency requirements, such as the data-handling code and driver code that executes at high priority, such as interrupt handlers. The remaining code, which is often the majority of code in a driver, executes in a user-level process. While the kernel is isolated from faults in the user-level code, systems such as SafeDrive [52] or XFI [16] can be used to isolate and recover from faults in the kernel portion.

To maintain compatibility with existing code, the *DriverSlicer* tool can create microdrivers from existing driver code. This tool identifies high-priority and low-latency code in drivers that must remain in the kernel and creates two output files: one with functions left in the kernel (the k-driver), and one with everything else (the u-driver). With assistance from programmer annotations, DriverSlicer generates RPC-like stubs for communication between the k-driver and u-driver. The kernel invokes microdrivers normally by either calling into the k-driver or into a stub that passes control to the u-driver.

The Microdrivers architecture does not support several features necessary for widespread use. First, after split-

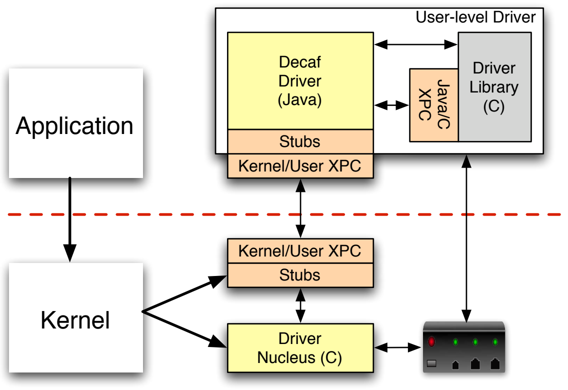


Figure 1: The Decaf Drivers architecture. The OS kernel invokes driver nucleus code or stubs that communicate with the decaf driver via an extension procedure call (XPC).

ting the driver, Microdrivers produces only preprocessed C output, which is unsuitable for evolving the driver once split. Second, Microdrivers only supports C in the u-driver, and provides no facility for moving to any other language.

2.2 Decaf Drivers Overview

Decaf Drivers extends Microdrivers by addressing the deficiencies outlined previously: Decaf Drivers supports (1) writing user-level code in a language other than C, (2) incremental conversion of a legacy driver to a new driver architecture, and (3) evolving the driver as interfaces and data structures change.

Decaf Drivers partitions drivers into two major components: the *driver nucleus*[1] that executes in the kernel for performance and compatibility; and the user-level *decaf driver* written in any language that supports marshaling/unmarshaling of data structures. However, user-level driver code may need to perform actions that are not expressible in all languages, such as directly controlling the hardware with instructions such as *outb*. This code resides in the user-level *driver library*, which executes normal C code. The driver library also provides a staging ground when migrating C code out of the kernel, where it can execute before being converted to another language.

While the architecture supports any language, our implementation is written for Java and we refer to code in the decaf driver as being written in Java. Using Java raises the issue of communicating data structures between languages, in contrast to C++. We believe that other languages that provide mechanisms for invoking native C code, such as Python, would also work well with the Decaf Drivers architecture.

At runtime, all requests to the driver enter through the kernel. The kernel directly invokes functionality implemented by the driver nucleus. Functionality implemented at user level enters through a stub that transfers control to user level and dispatches it to the driver library or the decaf driver. The user-level components may invoke each other or call back into the kernel while processing a request.

The Decaf architecture consists of two major components:

1. *Extension Procedure Call (XPC)* for communication between kernel/user level and between C and Java.
2. *DriverSlicer* to generate marshaling code for XPC.

We next discuss these two components in detail.

2.3 Extension Procedure Call

Extension procedure call, created as part of the Nooks driver isolation subsystem [45], provides procedure calls between protection domains. XPC in Decaf Drivers provides five services to enable this cooperation: *control transfer* to provide procedure call semantics (*i.e.*, block and wait); *object transfer* to pass language-level objects, such as structures, between domains; *object sharing* to allow an object to exist in multiple domains; and *synchronization* to ensure consistency when multiple domains access a shared object. *Stubs* invoke XPC services for communication between domains.

The two primary domains participating in driver execution are the driver nucleus and the decaf driver. However, driver functionality may also exist in the driver library, both when migrating code to another language or for functionality reasons. For example, code shared across operating systems may be left in C. Thus, the Decaf architecture also provides XPC between the decaf driver and the driver library to provide access to complex data structures requiring conversion between languages. The decaf driver may directly invoke code in the driver library for simple library calls.

Cross-Domain Control Transfer. The control-transfer mechanism performs the actions of the runtime in an RPC system [4] to pass control from the calling thread to a thread in the target domain. If the decaf driver and the driver library execute in a single process, the control transfer mechanism can be optimized to re-use the calling thread rather than scheduling a new thread to handle the request.

Cross-Domain Object Transfer. XPC provides customized marshaling of data structures to copy only those fields actually accessed at the target. Thus, structures defined for the kernel's internal use but shared with drivers are passed with only the driver-accessed fields. In addition, XPC provides cross-language conversion, converting structures making heavy use of C language features

[1]We re-christened the k-driver and u-driver from Microdrivers to more descriptive names reflecting their purpose and implementation, not just their execution mode.

for performance (*e.g.*, bit fields) to languages without such control over memory layout.

Object Sharing. Driver components may simultaneously process multiple requests that reference the same object. If two threads are accessing the same object, they should work on a single copy of this object, as they would in the kernel, rather than on two separate copies. Similar to Nooks [45], Decaf Drivers XPC uses an object tracker that records each shared object, extended to support two user-level domains. When transferring objects into a domain, XPC consults the object tracker to find whether the object already exists. If so, the existing object can be updated, and if not, a new object must be allocated.

Synchronization. Synchronized access to data is a challenging problem for regular device drivers because they are reentrant. For example, a device may generate an interrupt while a driver is processing an application request, and the interrupt handler and the request handler may access the same data. To prevent corruption, driver writers must choose from a variety of locking mechanisms based on the priority of the executing code and of potential sharers [29].

The Decaf Drivers synchronization mechanism provides regular locking semantics. If code in one domain locks an object, code in other domains must be prevented from accessing that object while the lock is held. Furthermore, Decaf ensures that the holder of a lock has the most recent version of the objects it protects.

Stubs. Similar to RPC stubs, XPC stubs contain calls specific to a single remote procedure: calls into marshaling code, object tracking code, and control transfer code. These can be written by hand or generated by the DriverSlicer tool. Calls to native functions must be replaced with calls to stubs when the function is implemented in another domain.

2.4 DriverSlicer

The XPC mechanism supports the execution of Decaf Drivers, but does little on its own to simplify the writing of drivers. This task is achieved by the DriverSlicer tool, which enables creation of decaf drivers from existing kernel code written in C. DriverSlicer provides three key functions: (1) partitioning, to identify code that may run outside the kernel, (2) stub generation to enable communication across language and process boundaries, and (3) generation of the driver nucleus and user-level C code to start the porting process. Furthermore, DriverSlicer can regenerate stubs as the set of supported devices, driver data structures, and kernel interfaces change.

Partitioning. Given an existing driver, DriverSlicer automatically partitions the driver into the code that must remain in the kernel for performance or functionality reasons and the code that can move to user level. This feature is unchanged from the Microdrivers implementation of DriverSlicer. As input, it takes an existing driver and type signatures for *critical root functions*, *i.e.*, functions in the kernel-driver interface that must execute in the kernel for performance or functionality reasons. DriverSlicer outputs the set of functions reachable from critical root functions, all of which must remain in the kernel. The remaining functions can be moved to user level. In addition, DriverSlicer outputs the set of entry-point functions, where control transfers between kernel mode and user mode. The user-mode entry points are the driver interface functions moved to user mode. The kernel entry points are OS kernel functions and critical driver functions called from user mode.

Stub Generation. DriverSlicer creates stubs automatically based on the set of kernel and user entry points output from the partitioning stage. With the guidance of programmer annotations [19], DriverSlicer automatically generates marshaling code for each entry-point function. In addition, DriverSlicer emits code to marshal and unmarshal data structures in both C and Java, allowing complex data structures to be accessed natively in both languages.

Driver Generation. DriverSlicer emits C source code for the driver nucleus and the driver library. The driver library code can be ignored when functions are rewritten in another language. The source code produced is a partitioning of the original driver source code into two source trees. Files in each tree contain the same include files and definitions, but each function is in only one of the versions, according to where it executes.

To support driver evolution, DriverSlicer can be invoked repeatedly to generate new marshaling code as data structures change. The generated driver files need only be produced once since the marshaling code is segregated from the rest of the driver code.

2.5 Summary

The Decaf architecture achieves our four requirements. The decaf driver itself may be implemented in any language and runs at user level. The driver nucleus provides performance near that of native kernel drivers. DriverSlicer provides incremental conversion to C through automatic generation of stubs and marshaling code both for kernel-user communication and C-Java communication. Finally, DriverSlicer supports driver evolution through regeneration of stubs and marshaling code as the driver changes.

3 Implementation

We implemented the Decaf Drivers architecture for the Linux 2.6.18.1 kernel and re-wrote five device drivers into decaf drivers. We use a modified version of DriverSlicer from Microdrivers [19] to generate code for XPC stubs and marshaling, and implemented extensions to generate similar code between the driver library and decaf driver.

The driver nucleus is a standard Linux kernel module and the decaf driver and driver library execute together as a multithreaded Java application. Our implementation relies on Jeannie [22] to simplify calling from C into Java and back. Jeannie is a compiler that allows mixing Java and C code at the expression level, which simplifies communication between the two languages. Languages with native support for cross-language calls, such as C♯ [34], provide the ability to call functions in different languages, but do not allow mixing expressions in different languages.

Decaf Drivers provides runtime support common to all decaf drivers. The runtime for user-level code, the *decaf runtime*, contains code supporting all decaf drivers. The kernel runtime is a separate kernel module, called the *nuclear runtime*, that is linked to every driver nucleus. These runtime components support synchronization, object sharing, and control transfer.

3.1 Extension Procedure Call

Decaf Drivers uses two versions of XPC: one between the driver nucleus and the driver library, for crossing the kernel boundary; and another between the driver library and the decaf driver, for crossing the C-Java language boundary. XPC between kernel and user mode is substantially similar to that in Microdrivers, so we focus our discussion on communication between C and Java.

The Decaf implementation always performs XPCs to and from the kernel in C, which allows us to leverage existing stub and marshaling support from Microdrivers. An upcall from the kernel always invokes C code first, which may then invoke Java code. Similarly, downcalls always invoke C code first before invoking the kernel. While this adds extra steps when invoking code in the decaf driver, it adds little runtime overhead as shown by the experiments in Section 4.

3.1.1 Java–C Control and Data Transfer

Decaf Drivers provides two mechanisms for the decaf driver to invoke code in the driver library: direct cross-language function calls and calls via XPC. Direct calls may be used when arguments are scalar values that can be trivially converted between languages, such as arguments to low-level I/O routines. XPC must be used when arguments contain pointers or complex data structures to provide cross-language translation of data types. In addition, downcalls from the decaf driver to the driver nucleus require XPC.

In both cases, Decaf Drivers relies on the Jeannie language [22] to perform the low-level transfer between C and Java. Jeannie enables C and Java code to be mixed in a source file at the granularity of a single expression. The backtick operator (`) switches between languages. From a combined source file, the Jeannie compiler produces a C file, a Java file, and Java Native Interface code allowing one to call the other. Jeannie provides a clean syntax for invoking a Java function from C and vice versa. When invoking simple functionality in C, the decaf driver can inline the C code right into a Java function.

When invoking a function through XPC, Decaf Drivers uses RPC-style marshaling to transfer complex objects between Java and C. While Jeannie allows code in one language to read variables declared in the other, it does not allow modifications of those variables. Instead, Decaf uses the XDR marshaling standard [13] to marshal data between the driver library and the decaf driver, which we discuss in Section 3.2.3.

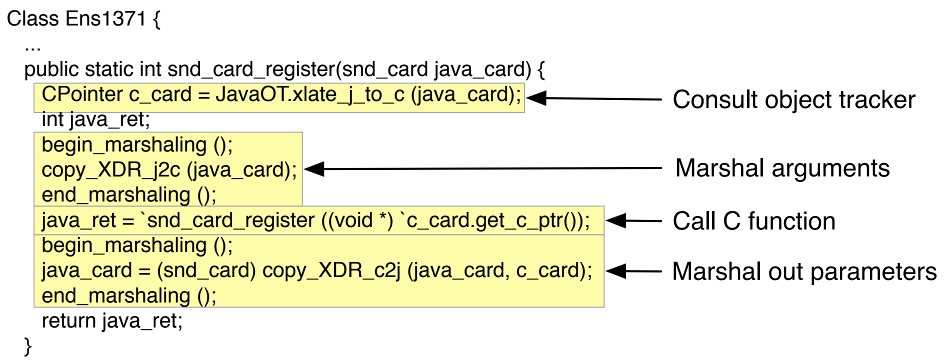
We write Decaf stubs in Jeannie to allow pure Java code to invoke native C code. The stubs invoke XDR marshaling code and the object tracker. Figure 2 shows an example of a stub in Jeannie. As shown in this figure, the following steps take place when calling from the decaf driver to the driver nucleus:

1. The decaf driver calls the Jeannie stub.
2. The stub invokes the object tracker to translate any parameters to their equivalent C pointers.
3. The stub, acting as an XPC client, invokes an XDR routine to marshal the Java parameters.
4. While marshaling these parameters, the XDR code uses inheritance to execute the appropriate marshaling routine for the object.
5. The same stub then acts as an XPC server, and unmarshals the Java objects into C.
6. While unmarshaling return values, the C stubs call specialized functions for each type.

We write stubs by hand because our code generation tools can only produce pure Java or C code, but the process could be fully automated.

3.1.2 Java-C Object Sharing

Object sharing maintains the relationship between data structures in different domains with an *object tracker*. This service logically stores mappings between C pointers in the driver library, and Java objects in the decaf driver. Marshaling code records the caller's local addresses for objects when marshaling data. Unmarshaling code checks the object tracker before unmarshaling each object. If found, the code updates the existing object with its new contents. If not found, the unmarshaling code allocates a new object and adds an association to the object tracker. For kernel/user XPC, the unmarshaling code in the kernel consults the object tracker with a simple procedure call, while unmarshaling code in the driver library



```
Class Ens1371 {
  ...
  public static int snd_card_register(snd_card java_card) {
    CPointer c_card = JavaOT.xlate_j_to_c (java_card);
    int java_ret;
    begin_marshaling ();
    copy_XDR_j2c (java_card);
    end_marshaling ();
    java_ret = `snd_card_register ((void *) `c_card.get_c_ptr());
    begin_marshaling ();
    java_card = (snd_card) copy_XDR_c2j (java_card, c_card);
    end_marshaling ();
    return java_ret;
  }
```

Figure 2: Sample Jeannie stub code for calling from Java to C. The backtick operator ` switches the language for the following expression, and is needed only to invoke the C function.

must call into the kernel.

The different data representations in C and Java raise two difficulties. First, Java objects do not have unique identifiers, such as the address of a structure in C. Thus, the decaf runtime uses a separate user-level object tracker written in Java, which uses object references to identify Java objects. C objects are identified by their address, cast to an integer.

Second, a single C pointer may be associated with multiple Java objects. When a C structure contains another structure as its first member, both inner and outer structures have the same address. In Java, however, these objects are distinct. This difference becomes a problem when a decaf driver passes the inner structure as an argument to a function in the driver library or driver nucleus. The user-level object tracker disambiguates the uses of a C pointer by additionally storing a *type identifier* with each C pointer.

When an object is initially copied from C to Java, marshaling code adds entries to the object tracker for its embedded structures. When an embedded structure is passed back from Java to C, the marshaling code will search for a C pointer with the correct type identifier. The object tracker uses the address of the C XDR marshaling function for a structure as its identifier.

Once an object's reference is removed from the object tracker, Java's garbage collection can free it normally. We have not yet implemented automatic collection of shared objects, so decaf drivers must currently free shared objects explicitly. Implementing the object tracker with weak references [20] and finalizers would allow unreferenced objects to be removed from the object tracker automatically.

3.1.3 Synchronization

Decaf Drivers relies on kernel-mode *combolocks* from Microdrivers to synchronize access to shared data across domains [19]. When acquired only in the kernel, a combolock is a spinlock. When acquired from user mode, a combolock is a semaphore, and subsequent kernel threads must wait for the semaphore. Combolocks also provide support for multiple threads in the decaf driver and allow these threads to share data with the driver nucleus and driver library.

However, combolocks alone do not completely address the problem. The driver nucleus must not invoke the decaf driver while executing high priority code or holding a spinlock. We use three techniques to prevent high-priority code from invoking user-level code. First, we direct the driver to avoid interrupting itself: the nuclear runtime disables interrupts from the driver's device with `disable_irq` while the decaf driver runs. Since user-mode code runs infrequently, we have not observed any performance or functional impact from deferring code.

Second, we modify the kernel in some places to not invoke the driver with spinlocks held. For example, we modified the kernel sound libraries to use mutexes, which allowed more code to execute in user mode. In its original implementation, the sound library would often acquire a spinlock before calling the driver. Driver functions called with a spinlock held would have to remain in the kernel because invoking the decaf driver would require invoking the scheduler. In contrast, mutexes allow blocking operations while they are held, so we were able to move additional driver functions into the decaf driver.

Third, we deferred some driver functionality to a worker thread. For example, the E1000 driver uses a watchdog timer that executes every two seconds. Since the kernel runs timers at high priority, it cannot call up to the decaf driver when the timer fires. Instead, we convert timers to enqueue a work item, which executes on a separate thread and allows blocking operations. Thus, the watchdog timer can execute in the decaf driver.

3.2 DriverSlicer Implementation

DriverSlicer automates much of the work of creating decaf drivers. The tool is a combination of OCaml code written for CIL [35] to perform static analysis and generate C code, Python scripts to post-process the generated

C code, and XDR compilers to produce cross-language marshaling code. DriverSlicer takes as input a legacy driver with annotations to specify how C pointers and arrays should be marshaled and emits stubs, marshaling routines, and separate user and kernel driver source code files.

3.2.1 Generating Readable Code

A key goal of Decaf Drivers is support for continued modification to drivers. A major problem with DriverSlicer from microdrivers is that it only generated preprocessed driver code, which is difficult to modify. The Decaf DriverSlicer instead patches the original source, preserving comments and code structure. It produces two sets of files; one set for the driver nucleus and one set for the driver library, to be ported to the decaf driver. This patching process consists of three steps.

First, scripts parse the preprocessed CIL output to extract the generated code (as compared to the original driver source). This code includes marshaling stubs and calls to initialize the object tracker. Other preprocessed output, such as driver function implementations, are ignored, as this code will be taken from the original driver source files instead.

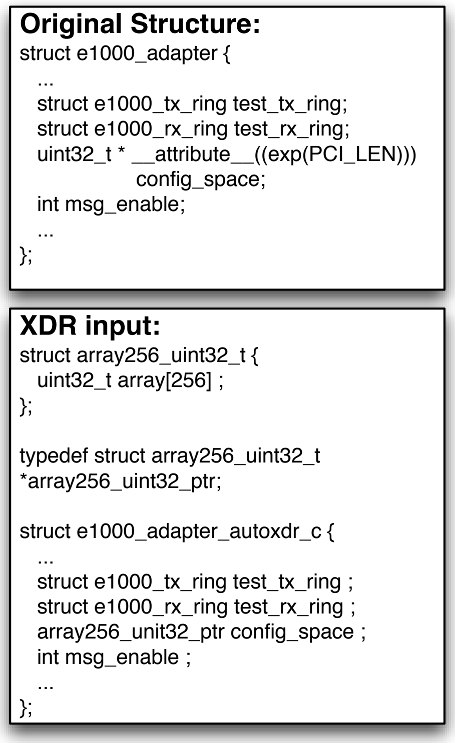
Second, DriverSlicer creates two copies (user and kernel) of the original driver source. From these copies, the tool removes function implemented by the other copy. Any functions in the driver nucleus source tree that are now implemented in the driver library and any functions in the driver library source tree that are implemented in the driver nucleus or the kernel are either replaced with stubs or removed entirely. The stubs containing marshaling code are placed in a separate file to preserve the readability of the patched driver.

Finally, DriverSlicer makes several other minor modifications to the output. It adds `#include` directives to provide definitions for the functions used in the marshaling code, and adds a function call in the driver nucleus `init_module` function to provide additional initialization.

3.2.2 Generating XDR Interface Specifications

The decaf driver relies on XDR marshaling to access kernel data structures. DriverSlicer generates an XDR specification for the data types used in user-level code from the original driver and kernel header files. The existing annotations needed for generating kernel marshaling code are sufficient to emit XDR specifications.

Unfortunately, XDR is not C and does not support all C data structures, specifically strings and arrays. DriverSlicer takes additional steps to convert C data types to compatible XDR types with the same memory layout. First, DriverSlicer discards most of the original code except for `typedefs` and structure definitions. DriverSlicer then rewrites these definitions to avoid functionality that XDR does not support. For example, a driver data structure may include a pointer to a fixed length array. DriverSlicer cannot output the original C definition because XDR would interpret it as a pointer to a single element. Instead, DriverSlicer generates a new structure definition containing a fixed length array of the appropriate type, and then substitutes pointers to the old type with a pointer to the new structure type.



Original Structure:

```
struct e1000_adapter {
  ...
  struct e1000_tx_ring test_tx_ring;
  struct e1000_rx_ring test_rx_ring;
  uint32_t * __attribute__((exp(PCI_LEN)))
                config_space;
  int msg_enable;
  ...
};
```

XDR input:

```
struct array256_uint32_t {
  uint32_t array[256] ;
};

typedef struct array256_uint32_t
*array256_uint32_ptr;

struct e1000_adapter_autoxdr_c {
  ...
  struct e1000_tx_ring test_tx_ring ;
  struct e1000_rx_ring test_rx_ring ;
  array256_unit32_ptr config_space ;
  int msg_enable ;
  ...
};
```

Figure 3: Portions of a driver data structure above, and the generated XDR input below. The names have been shortened for readability. The annotation in the original version is required for DriverSlicer to generate marshaling code between kernel and user levels.

As shown in Figure 3, DriverSlicer converts pointers to an array into a pointer to a structure, allowing XDR to produce marshaling code. This transformation does not affect the in-memory layout. In this way, the generated marshaling code will properly marshal the entire contents of the array. After generating the C output, a script runs which makes a few syntactic transformations, such as converting C's `long long` type to XDR's `hyper` type. The result is a valid XDR specification.

3.2.3 Generating XDR Marshaling Routines

DriverSlicer incorporates modified versions of the `rpcgen` [43] and `jrpcgen` [1] XDR interface compilers to generate C and Java marshaling code respectively. These modifications to the original tools support object tracking and recursive data structures.

As previously mentioned in Section 3.1.2, the tools emit calls into the object tracker to locate existing versions of objects passed as parameters. The generated unmarshaling code consults the object tracker before allocating memory for a structure. If one is found, the existing structure is used.

The DriverSlicer XDR compilers support recursive data structures, such as circular linked lists. The marshaling code checks each object against a list of the objects that have already been marshaled. When the tool encounters an object again, it inserts a reference to the existing copy instead of marshaling the structure again. This feature extends also across all parameters to a function, so that passing two structures that both reference a third results in marshaling the third structure just once.

The output of DriverSlicer is a set of functions that marshal or unmarshal each data structure used by the functions in the interface. It also emits a Java class for each C data type used by the driver. These classes are containers of public fields for every element of the original C structures. The generated classes provide a useful starting point for writing driver code in Java, but do not take advantage of Java language features. For example, all member variables are public. We expect developers to rewrite these classes when doing more development in Java.

3.2.4 Regenerating Stubs and Marshaling Code

As drivers evolve, the functions implemented in the driver nucleus or the data types passed between the driver nucleus and the decaf driver may change. Consequently, the stubs and marshaling code may need to be updated to reflect new data structures or changed use of existing data structures. While this could be performed manually, DriverSlicer provides automated support for regenerating stubs and marshaling code. Simply re-running DriverSlicer may not produce correct marshaling code for added fields unless it observes code in the user-level partition accessing that field. If this is Java code, it is not visible to CIL, which only processes C code.

When the decaf driver requires access to fields not previously referenced, whether they are new or not, a programmer must inform DriverSlicer to produce marshaling code. DriverSlicer supports an annotation to the original driver code to indicate that a field may be referenced by the decaf driver. A programmer adds the annotation `DECAF_XVAR(y);` where *X* is an `R`, `W`, or `RW` depending on whether the Java code will read, write, or read and write the variable, and `y` is the variable name. These annotations must be placed in entry-point functions through which new fields are referenced.

Thus, the new annotations ensure that DriverSlicer generates marshaling code to allow reading and/or writing the new variables in the decaf driver. A programmer can add new functions to the user/kernel interface with similar annotations. In the future, we plan to automatically analyze the decaf driver source code to detect and marshal these fields. In addition, we plan to produce a concise specification of the entry points for regenerating marshaling code, rather than relying on the original driver source.

Source Components	**# Lines**
Runtime support	
Jeannie helpers	1,976
XPC in Decaf runtime	2,673
XPC in Nuclear runtime	4,661
DriverSlicer	
CIL OCaml	12,465
Python scripts	1,276
XDR compilers	372
Total number of lines of code	23,423

Table 1: The number of non-comment lines of code in the Decaf runtime and DriverSlicer tools. For the XDR compilers, the number of additional lines of code is shown.

3.3 Code Size

Table 1 shows the size of the Decaf Drivers implementation. The runtime code, consisting of user-level helper functions written in Jeannie and XPC code in user and kernel mode, totals 9,310 lines. This code, shared by all decaf drivers, is comparable to a moderately sized driver.

DriverSlicer consists of OCaml code for CIL, Python scripts for processing the output, and XDR compilers. As the XDR compilers are existing tools, we report the amount of code we added. In total, DriverSlicer comprises 14,113 lines.

4 Experimental Results

The value of Decaf Drivers lies in simplified programming. The cost of using Decaf Drivers comes from the additional complexity of partitioning driver code and the performance cost of communicating across domain boundaries. We have converted four types of drivers using Decaf Drivers, and report on the experience and the performance of the resulting drivers here. We give statistics for the code we have produced, and answer three questions about Decaf Drivers: how hard is it to move driver code to Java, how much driver code can be moved to Java, and what is the performance cost of Decaf Drivers?

We experimented with the five drivers listed in Table 2. Starting with existing drivers from the CentOS 4.2 Linux distribution (compatible with RedHat Enterprise Linux 4.2) with the 2.6.18.1 kernel, and we converted them to Decaf Drivers using DriverSlicer. All our experiments except those for the E1000 driver are run on a 3.0GHz Pentium D with 1GB of RAM. The E1000 experiments are run on a 2.5GHz Core 2 Quad with 4GB of RAM. We used separate machines because the test devices were not all available on either machine individually.

4.1 Conversion to Java

Table 2 shows for each driver, how many lines of code required annotations and how many functions were in the driver nucleus, driver library, and decaf driver. After splitting code with DriverSlicer, we converted to Java all the functions in user level that we observed being called. Many of the remaining functions are specific to other devices served by the same driver. The column "Lines of Code" reports the quantity of code in the original driver. The final column, "Orig. LoC" gives the amount of C code converted to Java.

The annotations affect less than 2% of the driver source on average. These results are lower than for Microdrivers because of improvements we made to DriverSlicer to more thoroughly analyze driver code. In addition to the annotations in individual drivers, we annotated 25 lines in common kernel headers that were shared by multiple drivers. These results indicate that annotating driver code is not a major burden when converting drivers to Java. We also changed six lines of code in the `8139too` and `uhci-hcd` driver nuclei to defer functions executed at high priority to a worker thread while the decaf driver or driver library execute; the code is otherwise the same as that produced by DriverSlicer.

While we converted the `8139too` and `ens1371` to Java during the process of developing Decaf Drivers, we converted the other two drivers after its design was complete. For `uhci-hcd`, a driver previously converted to a microdriver, the additional conversion of the user-mode code to a decaf driver took approximately three days. The entire conversion of the `psmouse` driver, including both annotation and conversion of its major routines to Java, took roughly two days. This experience confirms our goal that porting legacy driver code to Java, when provided with appropriate infrastructure support, is straightforward.

In four of the five drivers, we were able to move more than 75% of the functions into user mode. However, we were only able to convert 4% of the functions in `uhci-hcd` to Java because the driver contained several functions on the data path that could potentially call nearly any code in the driver. We expect that redesigning the driver would allow additional code to move to user level. In the `psmouse` driver, we found that most of the user-level code was device-specific. Consequently, we implemented in Java only those functions that were actually called for our mouse device.

The majority of the code that we converted from C to Java is initialization, shutdown, and power management code. This is ideal code to move, as it executes rarely yet contains complicated logic that is error prone [40].

4.2 Performance of Decaf Drivers

The Decaf architecture seeks to minimize the performance impact of user-level code by leaving critical path code in the kernel. In steady-state behavior, the decaf driver should never or only rarely be accessed. However, during initialization and shutdown, the decaf driver executes frequently. We therefore measure both the latency to load and initialize the driver and its performance on a common workload.

We measure driver performance with workloads appropriate to each type of driver. We use netperf [12] sending and receiving TCP/IP data to evaluate the `8139too` and `E1000` network drivers with the default send and receive buffer sizes of 85 KB and 16 KB. We measure the `ens1371` sound driver by playing a 256Kbps MP3 file. The `uhci-hcd` driver controls low-bandwidth USB 1.0 devices and requires few CPU resources. We measure its performance by untaring a large archive onto a portable USB flash drive and record the CPU utilization. We do not measure the performance of the mouse driver, as its bandwidth is too low to be measurable, and we measure its CPU utilization while continuously moving the mouse for 30 seconds. For all workloads except netperf, we repeated the experiment three times. We executed the netperf workload for a single 600-second iteration because it performs multiple tests internally.

We measure the initialization time for drivers by measuring the latency to run the `insmod` module loader. While some drivers perform additional startup activities after module initialization completes, we found that this measurement provides an accurate representation of the difference between native and Decaf Drivers implementations.

Table 3 shows the results of our experiments. As expected, performance and CPU utilization across the benchmarks was unchanged. With `E1000`, we also tested UDP send/receive performance with 1 byte messages. The throughput is the same as the native driver and CPU utilization is slightly higher.

To understand this performance, we recorded how often the decaf driver is invoked during these workloads. In the `ens1371` driver, the decaf driver was called 15 times, all during playback start and end. A watchdog timer in the decaf driver executes every two seconds in the `E1000` driver. The other workloads did not invoke the decaf driver at all during testing. Thus, the added cost of communication with Java has no impact on application performance.

However, the latency to initialize the driver was sub-

Driver		Lines of	DriverSlicer	Driver nucleus		Driver library		Decaf driver		
Name	Type	code	Annotations	Funcs	LoC	Funcs	LoC	Funcs	LoC	Orig. LoC
8139too	Network	1,916	17	12	389	16	292	25	541	570
E1000	Network	14,204	64	46	1715	0	0	236	7804	8693
ens1371	Sound	2,165	18	6	140	0	0	59	1049	1068
uhci-hcd	USB 1.0	2,339	94	68	1537	12	287	3	188	168
psmouse	Mouse	2,448	17	15	501	74	1310	14	192	250

Table 2: The drivers converted to the Decaf architecture, and the size of the resulting driver components.

Driver	Workload	Relative	CPU Utilization		Init. Latency		User/Kernel
Name		Performance	native	Decaf	native	Decaf	Crossings
8139too	netperf-send	1.00	14 %	13 %	0.02 sec.	1.02 sec.	40
	netperf-recv	1.00	17 %	15 %	–	–	–
E1000	netperf-send	0.99	2.8 %	3.7 %	0.42 sec.	4.87 sec.	91
	netperf-recv	1.00	20 %	21 %	–	–	–
ens1371	mpg123	–	0.0 %	0.1 %	1.12 sec.	6.34 sec.	237
uhci-hcd	tar	1.03	0.1 %	0.1 %	1.32 sec.	2.67 sec	49
psmouse	move-and-click	–	0.1 %	0.1 %	0.04 sec.	0.40 sec.	24

Table 3: The performance of Decaf Drivers on common workloads and driver initialization.

stantially higher, averaging 3 seconds. The increase stems from cross-domain communication and marshaling driver data structures. Table 3 includes the number of call/return trips between the driver nucleus and the decaf driver during initialization. We expect that optimizing our marshaling interface to transfer data directly between the driver nucleus and the decaf driver, rather than unmarshaling at user-level in C and re-marshaling in Java, would significantly reduce the cost of invoking decaf driver code.

5 Case Study: E1000 Network Driver

To evaluate the software engineering benefits of developing drivers in Java, we analyze the Intel E1000 gigabit Ethernet decaf driver. We selected this driver because:

- it is one of the largest network drivers with over 14,200 lines of code.
- it supports 50 different chipsets.
- it is actively developed, with 340 revisions between the 2.6.18.1 and 2.6.27 kernels.
- it has high performance requirements that emphasize the overhead of Decaf Drivers.

With this case study, we address three questions:

1. What are the benefits of writing driver code in Java?
2. How hard is it to update driver code split between the driver nucleus and the decaf driver?
3. How difficult is it to mix C and Java in a driver?

We address these questions by converting the E1000 driver from the Linux 2.6.18.1 kernel to a decaf driver. Overall, we converted 236 functions to Java in the decaf driver and left 46 functions in the driver nucleus. There are no E1000-specific functions in the driver library. Of the 46 kernel functions, 42 are there for performance or functionality reasons. For example, many are called from interrupt handlers or with a spinlock held.

The four remaining functions are left in the driver nucleus because of an explicit data race that our implementation does not handle. These functions, in the `ethtool` interface, wait for an interrupt to fire and change a variable. However, the interrupt handler changes the variable in the driver nucleus; the copy of the variable in the decaf driver remains unchanged, and hence the function waits forever. This could be addressed with an explicit call into the driver nucleus to wait for the interrupt.

5.1 Benefits from Java

We identified three concrete benefits of moving E1000 code to Java, and one potential benefit we plan to explore.

Error handling. We found the biggest benefit of moving driver code to Java was improved error handling. The standard practice to handle errors in Linux device drivers is through `goto` statements to a set of labels based on when the failure occurred. In this idiom, an `if` statement checks a return value, and jumps to a label near the end of the function on error. The labels are placed such that only the necessary subset of cleanup operations are performed. This system is brittle because the developer can easily jump to an incorrect label or forget to test an error condition [44].

Using exceptions to signal errors and nested handlers to catch errors, however, ensures that no error conditions are ignored, and that cleanup operations take place in the proper order. We rewrote 92 functions to use *checked exceptions* instead of integer return codes. The compiler requires the program to handle these exceptions. In this process, we found 28 cases in which error codes were ig-

```
Decaf driver code:
public static void e1000_open(net_device netdev)
      throws E1000HWException {
  e1000_adapteradapter = netdev.priv;
  int err;
    try {
        /* allocate transmit descriptors */
        e1000_setup_all_tx_resources(adapter);

        try {
           /* allocate receive descriptors */
           e1000_setup_all_rx_resources(adapter);

           try {
              e1000_request_irq(adapter);
              e1000_power_up_phy(adapter);
              e1000_up(adapter);
              ...
           } catch (E1000HWException e) {
              e1000_free_all_rx_resources(adapter);
              throw e;
           }
        } catch (E1000HWException e) {
           e1000_free_all_tx_resources(adapter);
           throw e;
        }
     } catch (E1000HWException e) {
        e1000_reset(adapter);
        throw e;
     }
  }
```

Figure 4: Code converted to nested exception handling.

nored or handled incorrectly. Some, but not all, of these have been fixed in recent Linux kernels.

Figure 4 shows an example from the `e1000_open` function. This code catches and re-throws the exceptions; using a `finally` block would either incorrectly free the resources under all circumstances, or require additional code to ensure the resources are freed only in the face of an error.

Checked exceptions also reduce the amount of code in the driver. Figure 5 shows an example. By switching to exceptions instead of integer return values, we cut 675 lines of code, or approximately 8%, from `e1000_hw.c` by removing code to check for an error and return. We anticipate that converting the entire driver to use exceptions would eliminate more of these checks.

Object orientation. We found benefits from object orientation in two portions of the E1000 driver. In `e1000_param.c`, functions verify module parameters using range and set-membership tests. We use three classes to process parameters during module initialization. A base class provides basic parameter checking, and the two derived classes provide additional functionality. The appropriate class checks each module parameter automatically. The resulting code is shorter than the original C code and more maintainable, because the programmer is forced by the type system to provide ranges and sets when necessary.

In addition, we restructured the hardware accessor functions as a class. In the original E1000 driver,

```
Original Code:
if(hw->ffe_config_state == e1000_ffe_config_active) {
   ret_val = e1000_read_phy_reg(hw, 0x2F5B,
                                           &phy_saved_data);
   if(ret_val) return ret_val;

   ret_val = e1000_write_phy_reg(hw, 0x2F5B, 0x0003);
   if(ret_val) return ret_val;

   msec_delay_irq(20);
   ret_val = e1000_write_phy_reg(hw, 0x0000,
                    IGP01E1000_IEEE_FORCE_GIGA);
   if(ret_val) return ret_val;
```

```
Decaf driver Code:
if(hw.ffe_config_state.value == e1000_ffe_config_active) {
    e1000_read_phy_reg(0x2F5B, phy_saved_data);
    e1000_write_phy_reg((short) 0x2F5B, (short) 0x0003);
    e1000_write_phy_reg((short) 0x2F5B, (short) 0x0003);
     DriverWrappers.Java_msleep (20);
     e1000_write_phy_reg((short) 0x0000,
           (short) IGP01E1000_IEEE_FORCE_GIGA);
```

Figure 5: Code from `e1000_config_dsp_after_link_change` in `e1000_hw.c`. The upper box shows the original code with error handling code. The lower box shows the same code rewritten to use exceptions.

these functions all required a parameter pointing to an `e1000_hw` structure. Just rewriting this code as a class removed 6.5KB of code that passes this structure as a parameter to other internal functions. This syntactic change does not affect code quality, but makes the resulting code more readable.

Standard libraries. In comparison to the Java collections library, the Linux kernel and associated C libraries provide limited generic data-structure support. We found that the Java collections library provides a useful set of tools for simplifying driver code. In addition to rewriting the parameter code to use inheritance, we also used Java hash tables in the set-membership tests.

Potential Benefit: Garbage collection. While the E1000 decaf driver currently manages shared objects manually, garbage collection provides a mechanism to simplify this code and prevent resource leaks. When allocating data structures shared between the driver nucleus and decaf driver, the decaf drivers use a custom constructor that also allocates kernel memory at the same time and creates an association in the object tracker.

Rather than relying on the decaf driver to explicitly release this memory, we can write a custom finalizer to free the associated kernel memory when the Java garbage collector frees the object. This approach can simplify exception-handling code and prevent resource leaks on error paths, a common driver problem [31].

Category	Lines of Code Changed
Driver nucleus	381
Decaf driver	4690
User/kernel interface	23

Table 4: Statistics for patches applied to E1000: the lines changed in the driver nucleus, in the decaf driver, and to shared data structures requiring new marshaling code.

5.2 Driver Evolution

We evaluate the ability of Decaf Drivers to support driver evolution by applying all changes made to the E1000 driver between kernel versions 2.6.18.1 and 2.6.27. Because we continue to use the 2.6.18.1 kernel, we omitted the small number of driver changes related to kernel interface updates. We applied all 320 patches in two batches: those before the 2.6.22 kernel and those after. Overall, we found that modifying the driver was simple, and that the driver nucleus and decaf driver could be modified and compiled separately.

The changes are summarized in Table 4. The vast majority of code changes were at user level. Thus, the bulk of the development on E1000 since the 2.6.18.1 kernel would have been performed in Java at user level, rather than in the kernel in C. Furthermore, only 23 changes affected the kernel-user interface, for example by adding or removing fields from shared structures.

In these cases, we modified the kernel implementation of the data structure, and re-split the driver to produce updated versions of the Java data structures. To ensure that new structure fields are marshaled between the driver nucleus and decaf driver, we added one additional annotation for each new field to the original driver. These annotations ensure that DriverSlicer generates marshaling code to allow reading and/or writing the new variables in the decaf driver.

5.3 Mixing C and Java

A substantial portion of the Decaf architecture is devoted to enabling a mix of Java and C to execute at user level. We have found two reasons to support both languages. First, when migrating code to Java, it is convenient to move one function at a time and then test the system, rather than having to convert all functions at once (as required by most user-level driver frameworks). This code is temporary and exists only during the porting process. We initially ran all user-mode E1000 functions in this mode and then incrementally converted them to Java, starting with leaf functions and then advancing up the call graph. Our current implementation has no driver functionality implemented in the driver library.

Jeannie makes this transition phase simple because of its ability to mix Java and C code without explicitly using the Java Native Interface. The ability to execute either Java or C versions of a function during development greatly simplified conversion, as it allowed us to eliminate any new bugs in our Java implementation by comparing its behavior to that of the original C code.

Second, and more important, there may be functionality necessary for communicating with the kernel or the device that is not possible to express in Java. These functions are *helper routines* that do not contain driver logic but provide an escape from the limits of a managed language. Some examples we have observed include accessing the `sizeof()` operator in C, which is necessary for allocating some kernel data structures, and for performing programmed I/O with I/O ports or memory-mapped I/O regions. While some languages, including C♯, support unsafe memory access, Java does not. However, we found that none of these helper routines are specific to the E1000 driver, and as a result placed them in the decaf runtime to be shared with other decaf drivers. As before, Jeannie makes using these helper routines in Java a straightforward matter.

Jeannie also simplifies the user-level stub functions significantly. These stubs include a simple mixture of C and Java code, whereas using JNI directly would significantly complicate the stubs.

6 Related Work

Decaf Drivers differs from past work on driver reliability and type-safe kernel programming in many respects. Unlike past approaches that are either compatible or transformative, we desire both compatibility with existing code and the opportunity to completely rewrite drivers.

Driver reliability. Driver reliability systems focus on tolerating faults in existing drivers with hardware memory protection [45, 50], language-based isolation [52], or private virtual machines [17, 18, 27]. However, these systems all leave driver code in C and in the kernel and thus do not ease driver programming.

Driver safety. Another approach to improving reliability is to prevent drivers from executing unsafe actions. Safety can be achieved by executing drivers in user mode [21, 26], with type safety in the kernel [9], or by formally verifying driver safety [41]. However, these approaches either require writing a completely new driver, or rewriting the entire kernel. With Decaf Drivers, drivers may be incrementally converted to any language because C is still available for helper routines.

Simplifying driver code. Many projects promise to simplify driver programming through new driver interfaces [3, 33, 38, 25, 51, 42, 30]. These systems offer advanced languages [51, 42]; domain-specific languages for hardware access [30] and for common driver logic [7, 10]; simplified programming interfaces at user-level [3, 8, 33]; and cross-platform interfaces [38, 25]. Like Decaf Drivers, Linux UIO drivers leave part of the driver in the kernel, while the bulk executes at user level [47]. Coccinelle [37] simplifies patching a large set of drivers at once. The features offered by these systems are complementary to Decaf Drivers, and the ability to gradually rewrite driver code in a new language may provide a route to their use. However, these systems either require writing new drivers for a new interface, or they simplify existing drivers but not enough: drivers are left in C and in the kernel.

Type-safe kernel programming. SPIN [23], the J Kernel [48], and Singularity [24] have kernels written in type safe languages. More recently, a real-time JVM was ported to the Solaris kernel [36]. In contrast to these systems, Decaf Drivers enables the use of modern languages for drivers without rewriting or substantially adding to the OS kernel.

7 Conclusion

Device drivers are a major expense and cause of failure for modern operating systems. With Decaf Drivers, we address the root of both problems: writing kernel code in C is hard. The Decaf architecture allows large parts of existing drivers to be rewritten in a better language, and supports incrementally converting existing driver code. Drivers written for Decaf retain the same kernel interface, enabling them to work with unmodified kernels, and can achieve the same performance as kernel drivers. Furthermore, tool support automates much of the task of converting drivers, leaving programmers to address the driver logic but not the logistics of conversion.

Writing drivers in a type-safe language such as Java provides many concrete benefits to driver programmers: improved reliability due to better compiler analysis, simplified programming due to richer runtime libraries, and better error handling with exceptions. In addition, many tools for user-level Java programming may be used for debugging.

Acknowledgments. We would like to thank our shepherd, Robert Grimm, for his useful comments and for his help, along with Martin Hirzel, in resolving Jeannie bugs. This work was supported in part by NSF grant CSR 0745517. Swift has a financial interest in Microsoft Corp.

References

[1] H. Albrecht. Remote Tea. http://remotetea.sourceforge.net/.

[2] J. Allchin. Windows Vista team blog: Updating a brand-new product, Nov. 2006. http://windowsvistablog.com/blogs/windowsvista/archive/2006/11/17/updating-a-brand-new-product.aspx.

[3] F. Armand. Give a process to your drivers! In *Proc. of the EurOpen Autumn 1991*, Sept. 1991.

[4] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.

[5] E. Brewer, J. Condit, B. McCloskey, and F. Zhou. Thirty years is long enough: getting beyond c. In *Proc. of the Tenth IEEE HOTOS*, 2005.

[6] A. Catorcini, B. Grunkemeyer, and B. Grunkemeyer. CLR inside out: Writing reliabile .NET code. *MSDN Magazine*, Dec. 2007. http://msdn2.microsoft.com/en-us/magazine/cc163298.aspx.

[7] P. Chandrashekaran, C. Conway, J. M. Joy, and S. K. Rajamani. Programming asynchronous layers with CLARITY. In *Proc. of the 15th ACMFSE*, Sept. 2007.

[8] P. Chubb. Get more device drivers out of the kernel! In *Ottawa Linux Symp.*, 2004.

[9] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *Proc. of the ACM SIGPLAN '03 ACM Conference on Programming Language Design and Implementation*, June 2003.

[10] C. L. Conway and S. A. Edwards. NDL: a domain-specific language for device drivers. In *Proc. of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, June 2004.

[11] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Associates, Feb. 2005.

[12] I. N. Division. Netperf: A network performance benchmark. http://www.netperf.org.

[13] M. Eisler. XDR: External data representation standard. RFC 4506, Internet Engineering Task Force, May 2006.

[14] J. Elson. FUSD: A Linux framework for user-space devices, 2004. User manual for FUSD 1.0.

[15] D. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proc. of the 18th ACM SOSP*, Oct. 2001.

[16] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: software guards for system address spaces. In *Proc. of the 7th USENIX OSDI*, 2006.

[17] Ú. Erlingsson, T. Roeder, and T. Wobber. Virtual environments for unreliable extensions. Technical Report MSR-TR-05-82, Microsoft Research, June 2005.

[18] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, 2004.

[19] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *Proc. of the Thirteenth ACM ASPLOS*, Mar. 2008.

[20] B. Goetz. Plugging memory leaks with weak references. http://www.ibm.com/developerworks/java/library/j-jtp11225/index.html, 2005.

[21] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Failure resilience for device drivers. In *Proc. of the 2007 IEEE DSN*, June 2007.

[22] M. Hirzel and R. Grimm. Jeannie: Granting Java native interface developers their wishes. In *Proc. of the ACM OOPSLA '07*, Oct. 2007.

[23] W. Hsieh, M. Fiuczynski, C. Garrett, S. Savage, D. Becker, and B. Bershad. Language support for extensible operating systems. In *Proc. of the Workshop on Compiler Support for System Software*, Feb. 1996.

[24] G. Hunt, J. Larus, M.Abadii, M. A. andPP. Barham, M. Fähdrich, C. Hawblitzel, O. Hodson, S. L. andNi. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, Oct. 2005.

[25] Jungo. Windriver cross platform device driver development environment. Technical report, Jungo Corporation, Feb. 2002. http://www.jungo.com/windriver.html.

[26] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Gotz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Jour. Comp. Sci. and Tech.*, 20(5), 2005.

[27] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proc. of the 6th USENIX OSDI*, 2004.

[28] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proc. of the 6th USENIX OSDI*, 2004.

[29] R. Love. Kernel locking techniques. *Linux Journal*, Aug. 2002. http://www.linuxjournal.com/article/5833.

[30] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Proc. of the 4th USENIX OSDI*, Oct. 2000.

[31] Microsoft Corp. PREfast for drivers. http://www.microsoft.com/whdc/devtools/tools/prefast.mspx.

[32] Microsoft Corporation. Windows Server 2003 DDK. http://www.microsoft.com/whdc/DevTools/ddk/default.mspx, 2003.

[33] Microsoft Corporation. Architecture of the user-mode driver framework. http://www.microsoft.com/whdc/driver/wdf/UMDF-arch.mspx, May 2006. Version 0.7.

[34] Microsoft Corporation. Interoperating with unmanaged code. http://msdn.microsoft.com/en-us/library/sd10k43k(VS.71).aspx, 2008.

[35] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Intl. Conf. on Compiler Constr.*, 2002.

[36] T. Okumura, B. R. Childers, and D. Mosse. Running a Java VM inside an operating system kernel. In *Proc. of the 4th ACM VEE*, Mar. 2008.

[37] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In *Proc. of the 2008 EuroSys Conference*, apr 2008.

[38] Project UDI. Uniform Driver Interface: Introduction to UDI version 1.0. http://udi.certek.cc/Docs/pdf/UDI_tech_-white_paper.pdf, Aug. 1999.

[39] D. Richie. The Unix tree: the 'nsys' kernel, Jan. 1999. http://minnie.tuhs.org/UnixTree/Nsys/.

[40] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *Proc. of the 2009 EuroSys Conference*, Apr. 2009.

[41] L. Ryzhyk, I. Kuz, and G. Heiser. Formalising device driver interfaces. In *Proc. of the Workshop on Programming Languages and Systems*, Oct. 2007.

[42] M. Spear, T. Roeder, O. Hodson, G. Hunt, and S. Levi. Solving the starting problem: Device drivers as self-describing artifacts. In *Proc. of the 2006 EuroSys Conference*, Apr. 2006.

[43] Sun Microsystems. UNIX programmer's supplementary documents: rpcgen programming guide. http://docs.freebsd.org/44doc/psd/22.rpcgen/paper.pdf.

[44] M. Susskraut and C. Fetzer. Automatically finding and patching bad error handling. In *Proceedings of the Sixth European Dependable Computing Conference*, 2006.

[45] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1), Feb. 2005.

[46] L. Torvalds. Linux kernel source tree. http://www.kernel.org.

[47] L. Torvalds. UIO: Linux patch for user-mode I/O, July 2007.

[48] T. von Eicken, C.-C. Chang, G. Czajkowski, C. Hawblitzel, D. Hu, and D. Spoonhower. J-Kernel: a capability-based operating system for Java. In *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume 1603 of *LNCS*, pages 369–393. Springer-Verlag, 1999.

[49] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *Proc. of the 8th USENIX OSDI*, Dec. 2008.

[50] E. Witchel, J. Rhee, and K. Asanovic. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *Proc. of the 20th ACM SOSP*, 2005.

[51] H. Yamauchi and M. Wolczko. Writing Solaris device drivers in Java. Technical Report TR-2006-156, Sun Microsystems, Apr. 2006.

[52] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *Proc. of the 7th USENIX OSDI*, 2006.