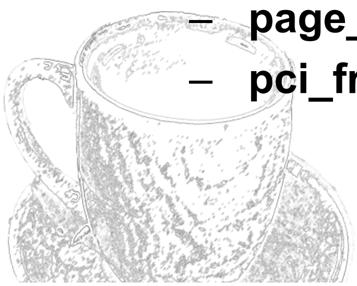
Decaf: Moving Device Drivers to a Modern Language

Matthew Renzelmann Michael Swift
University of Wisconsin-Madison

Driver Programming Is Not Easy

- __free_pages
- argv_free
- blk_queue_free_tags
- dma_free_coherent
- free_all_bootmem
- free_page_and_swap_cache
- free_pages
- free_pages_exact
- free_swap_and_cache
- hci_free_dev
- kfree
- kfree_skb
- mempool_kfree
- page_table_free
- pci_free_consistent
- release_and_free_resource
- rpc_free_iostats
- sctp_ootb_pkt_free
- selinux_xfrm_policy_free
- skb_free_datagram
- snd_device_free_all
- snd_dma_free_pages
- snd_info_free_entry
- snd_free_pages
- snd_soc_dapm_free
- snd_util_mem_free
- snd_util_memhdr_free
- ssp_free
- try_to_free_swap
- vfree

Many, many more



What About Java?

<This slide intentionally left blank>

Kernel vs. Java Development

Feature	Kernel	Java
Memory management	Manual	Garbage collection
Type safety	Limited	Extensive
Debugging	Few tools / difficult	Many tools / easier
Data structure library	Subset of libc	Java class library
Error handling	Return values	Exceptions

Motivation

- Kernel programming is difficult and leads to driver unreliability
- Existing approaches
  - Isolating drivers (Nooks [Swift04], SafeDrive [Zhou06])
  - User-level drivers (Nexus [Williams08])
  - New driver design (Dingo [Ryzhyk09], User-mode Driver Framework [Microsoft06], Singularity [Hunt05])

Decaf Drivers

- *Decaf Drivers* execute most driver code in user mode Java
  - Performance critical code left in kernel
- The *Decaf System* provides support for
  1. migrating driver code into a modern language (Java)
  2. executing drivers with high performance
  3. evolving drivers over time

Outline

- Introduction
- Overview
  - Goals
  - Architecture
- Design and Implementation
- Evaluation
- Conclusion

Goals: Making Decaf Practical

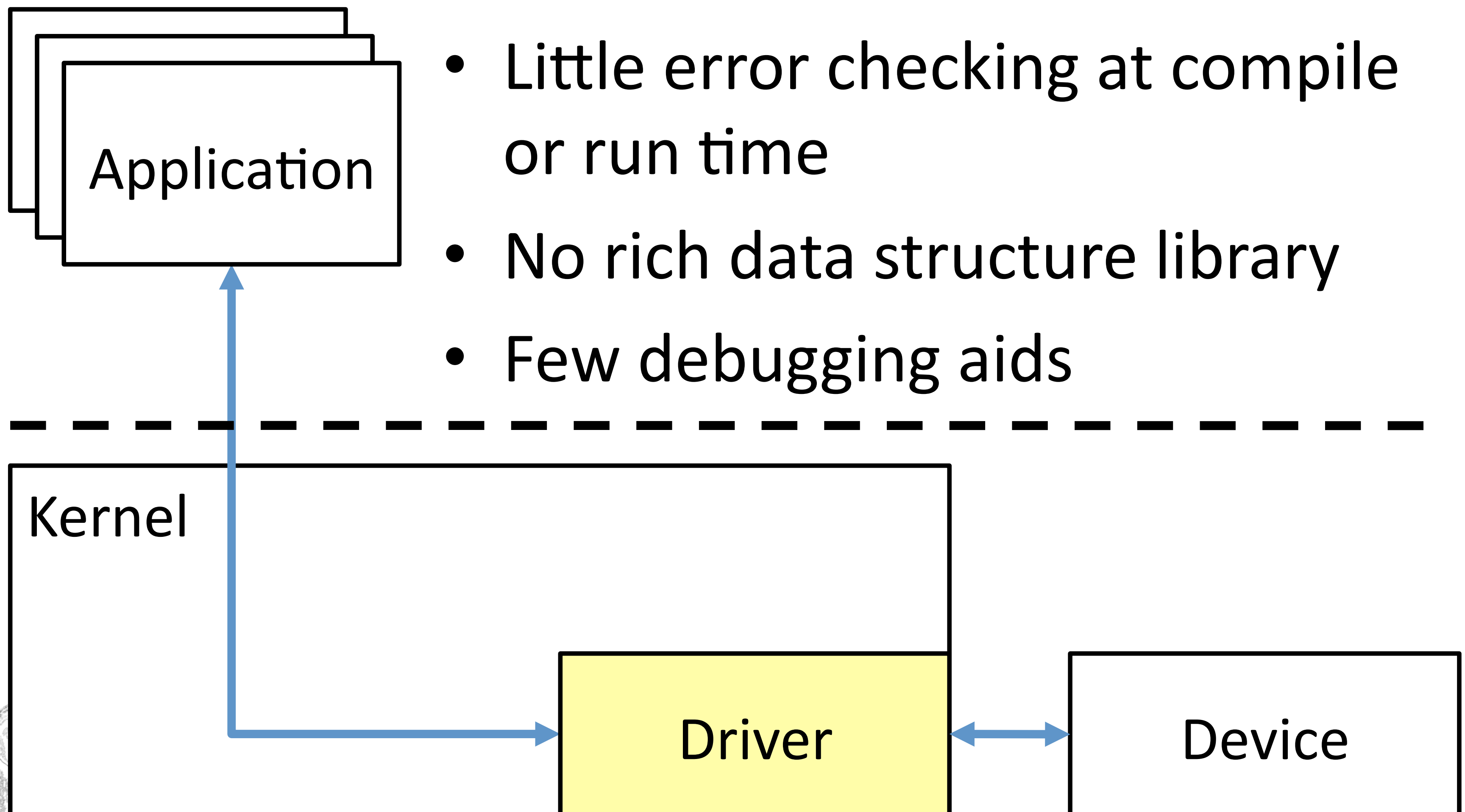
1. Compatibility with existing kernels/drivers

2. A migration path from existing drivers to decaf drivers

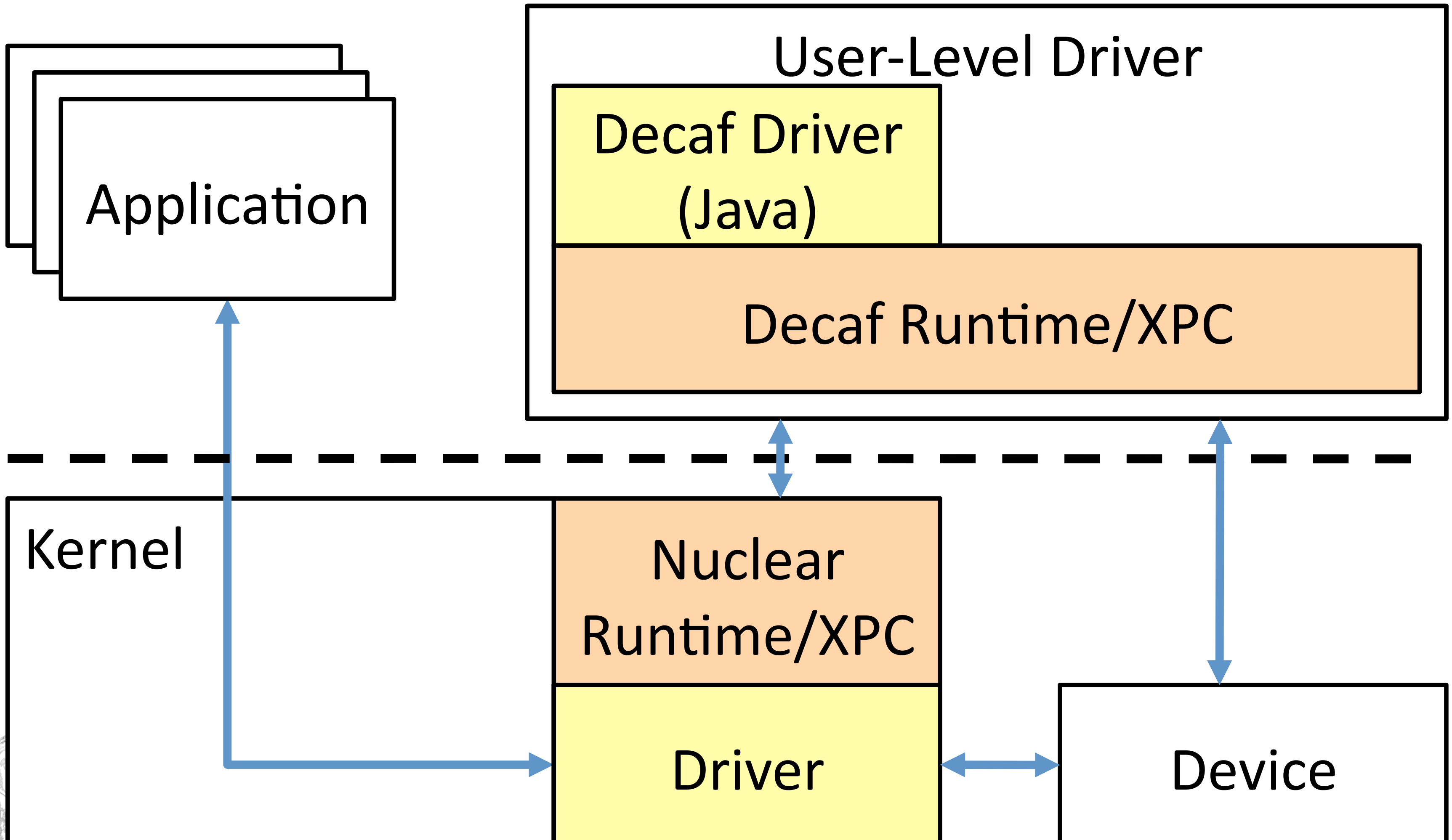
3. Support for evolution as drivers, devices, and kernels change

Existing Driver Architecture

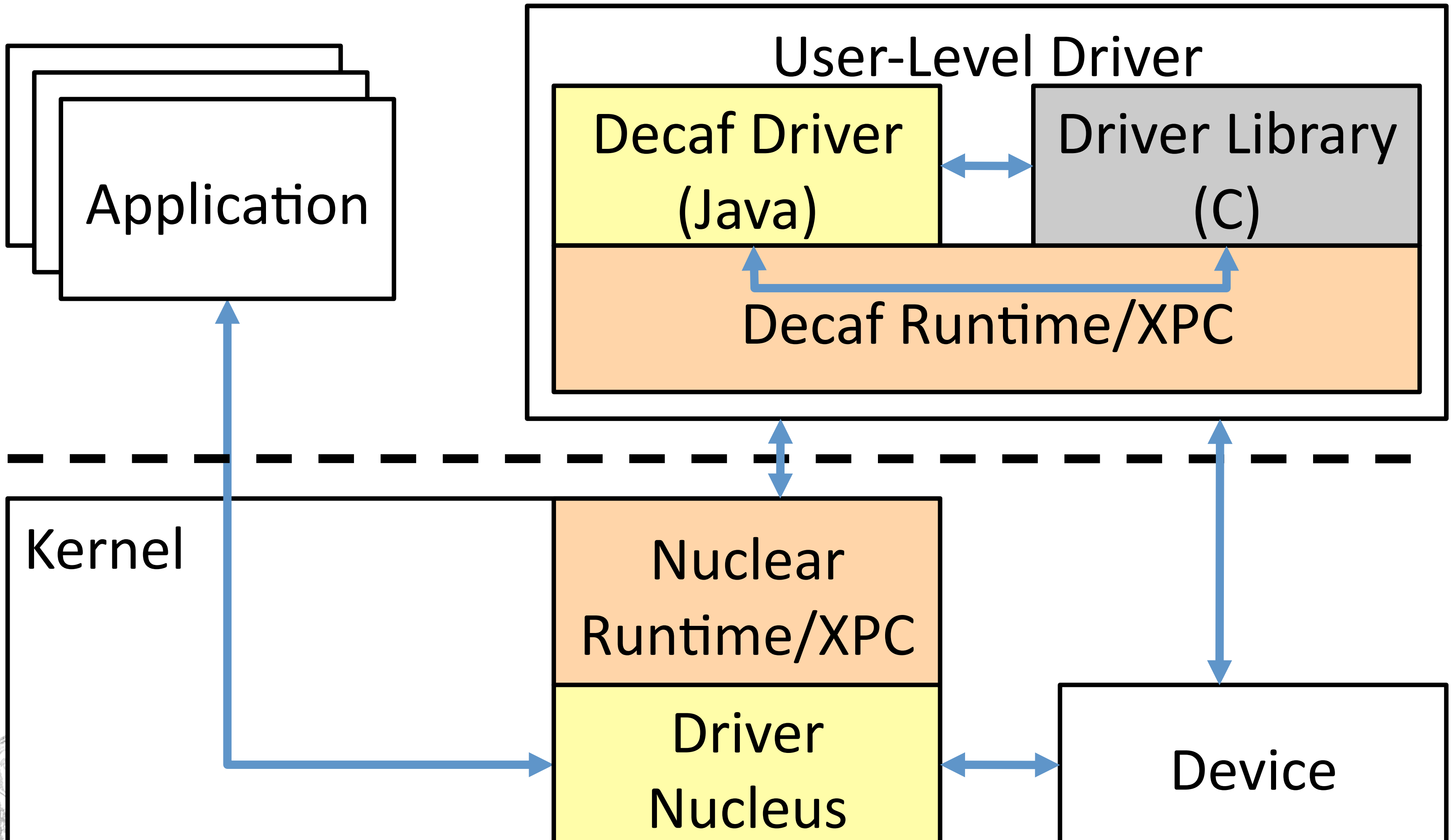
- Little error checking at compile or run time
- No rich data structure library
- Few debugging aids



Decaf Architecture



Decaf Architecture

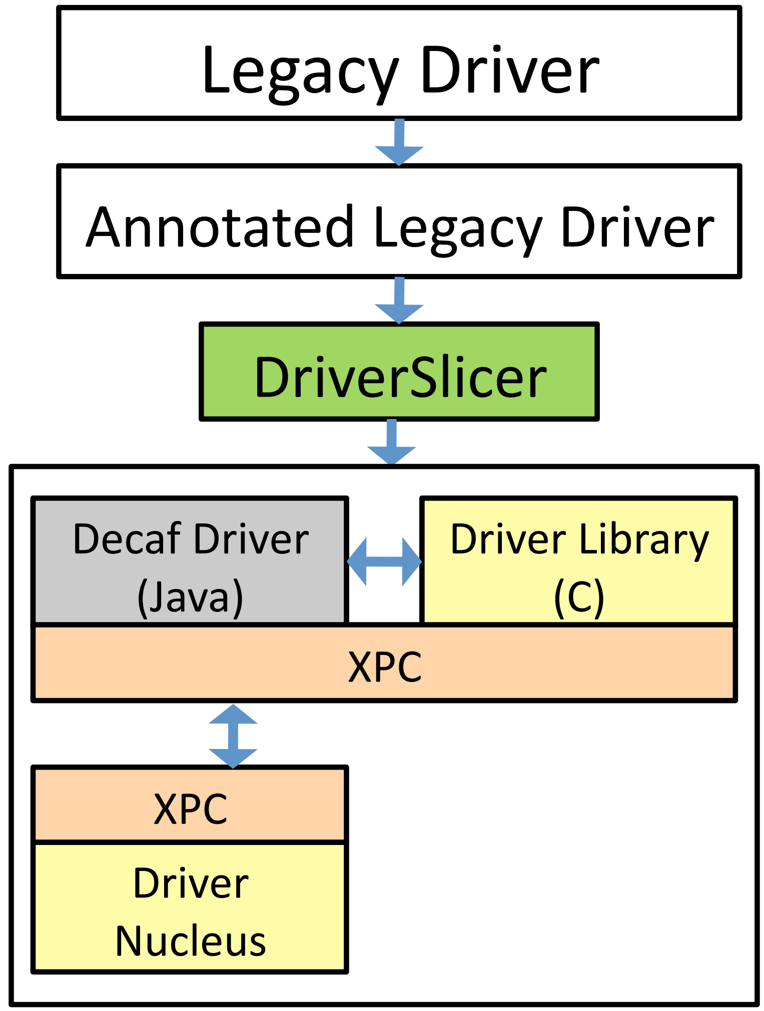


Creating Decaf Drivers

1. From scratch
2. By migrating existing kernel drivers
   – *DriverSlicer* provides tool support to move driver code out of the kernel

Creating Decaf Drivers

1. Annotate it
2. Run DriverSlicer to split the driver into a Driver Nucleus and Library
3. Migrate code from the Driver Library into the Decaf Driver



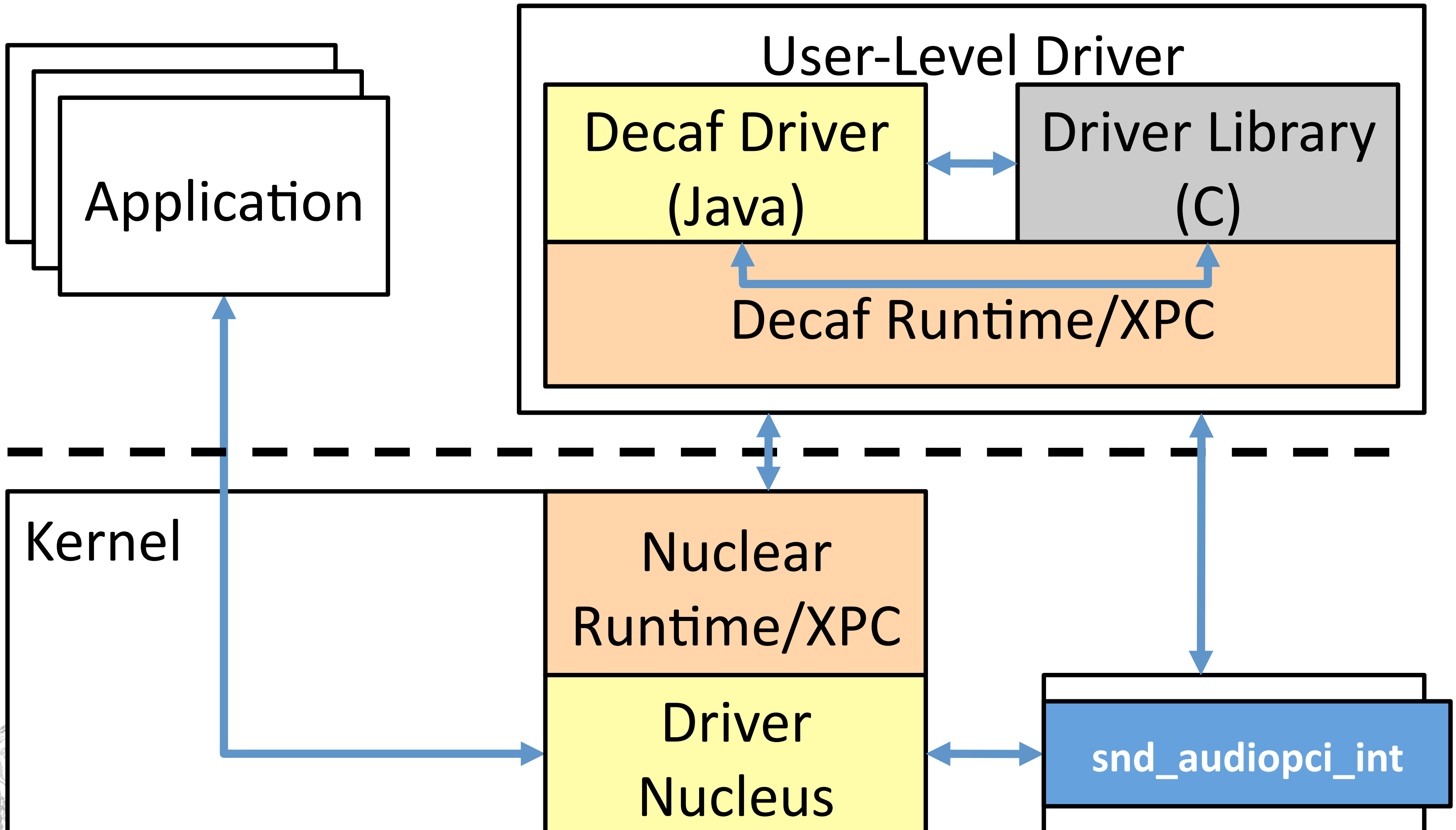
Outline

- Introduction
- Overview
- Design and Implementation
  - Communication
  - Creation
- Evaluation
- Conclusion

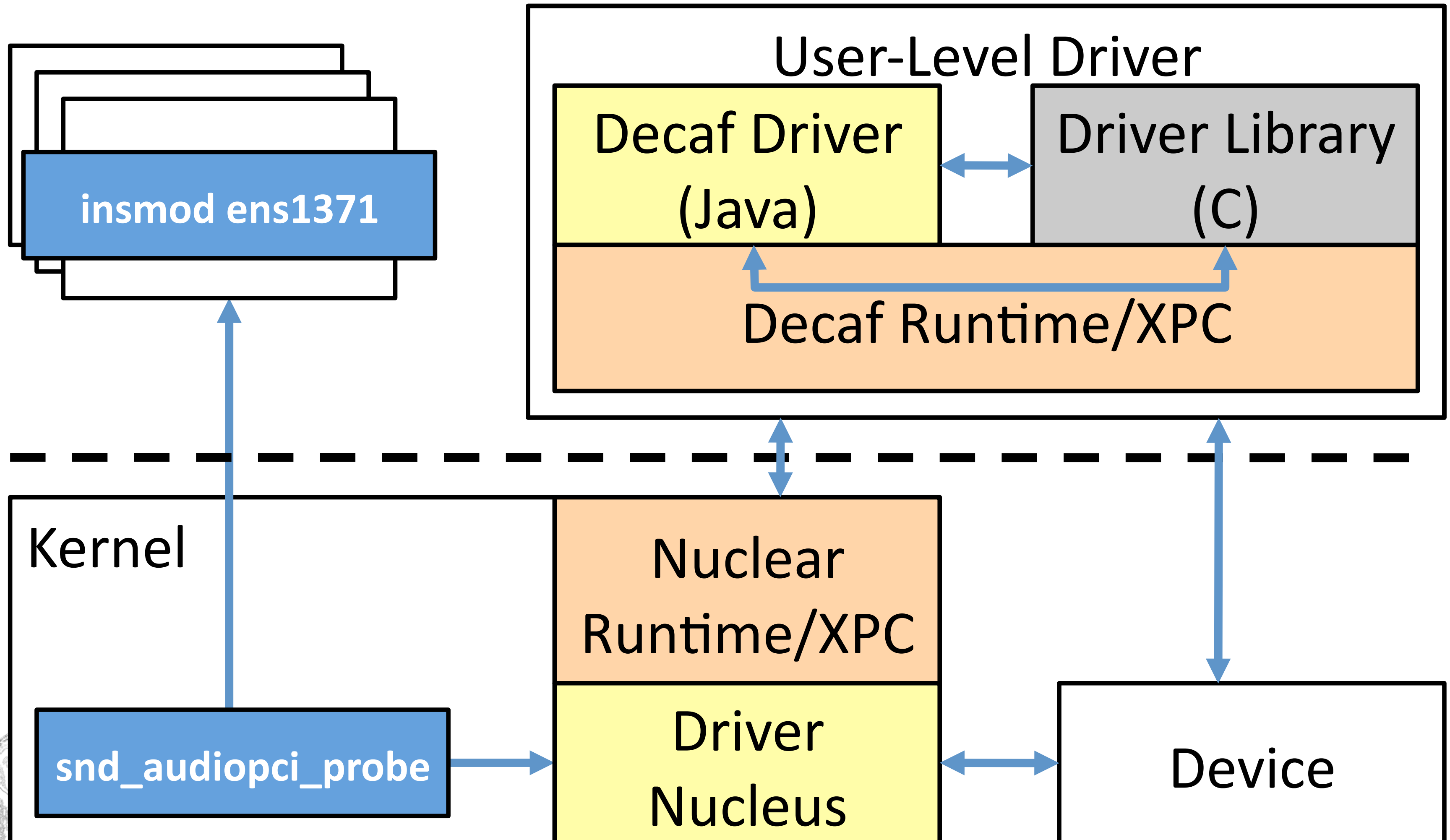
Design: Runtime Components

- Locking/Synchronization: ComboLocks
- Sharing: Object Tracker
- Communication: Extension Procedure Call (XPC)
  - Kernel/User upcalls and downcalls
  - Java/C calls

ENS1371 Communication Example



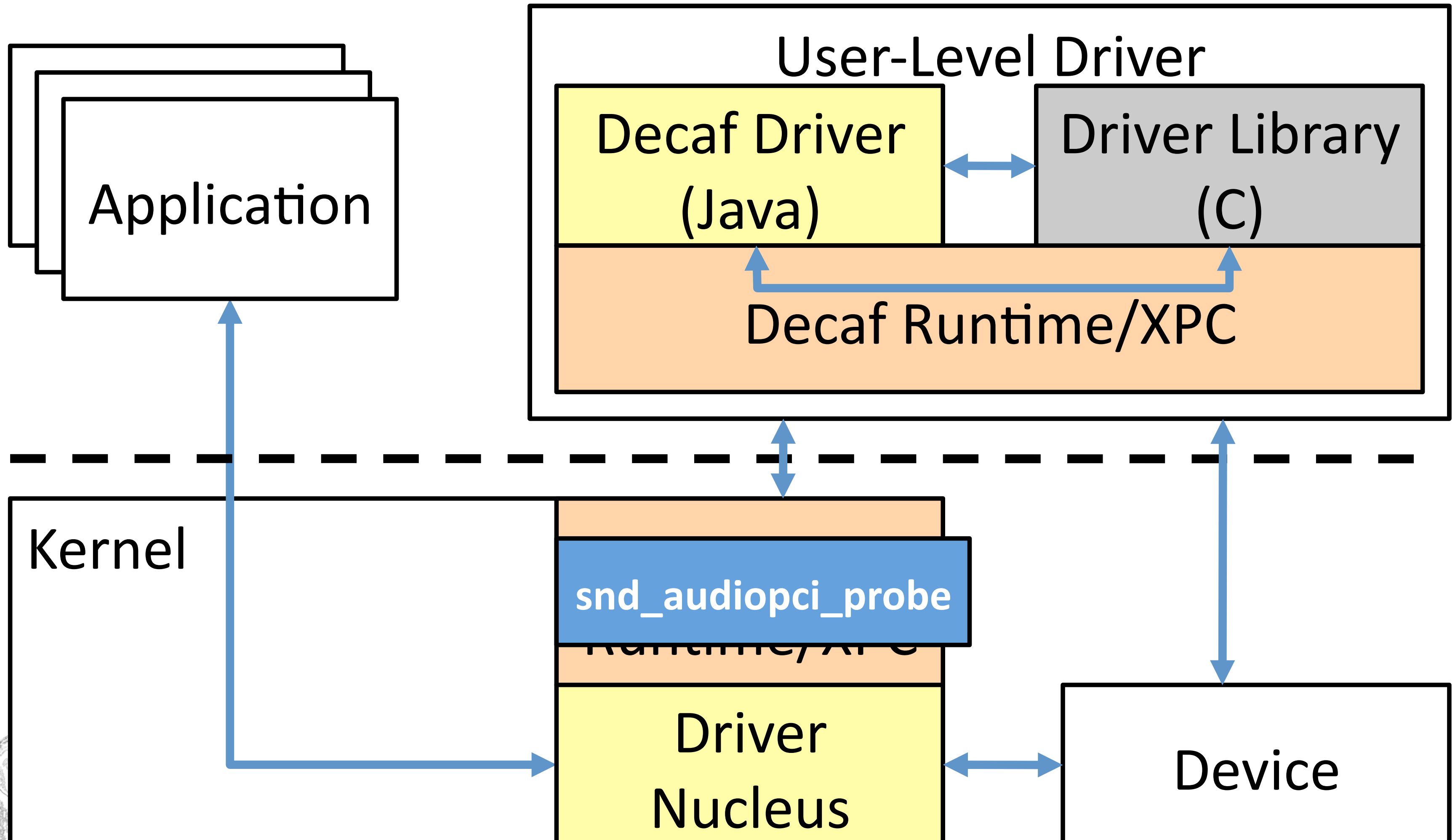
ENS1371 Communication Example



Kernel/User XPC

- Challenges
  - Minimizing data copied
  - Communicating complex data structures
- Solutions
  - Copying only structure fields that are used
  - Detecting recursion and linked data structures

Kernel/User XPC



Java/C Communication

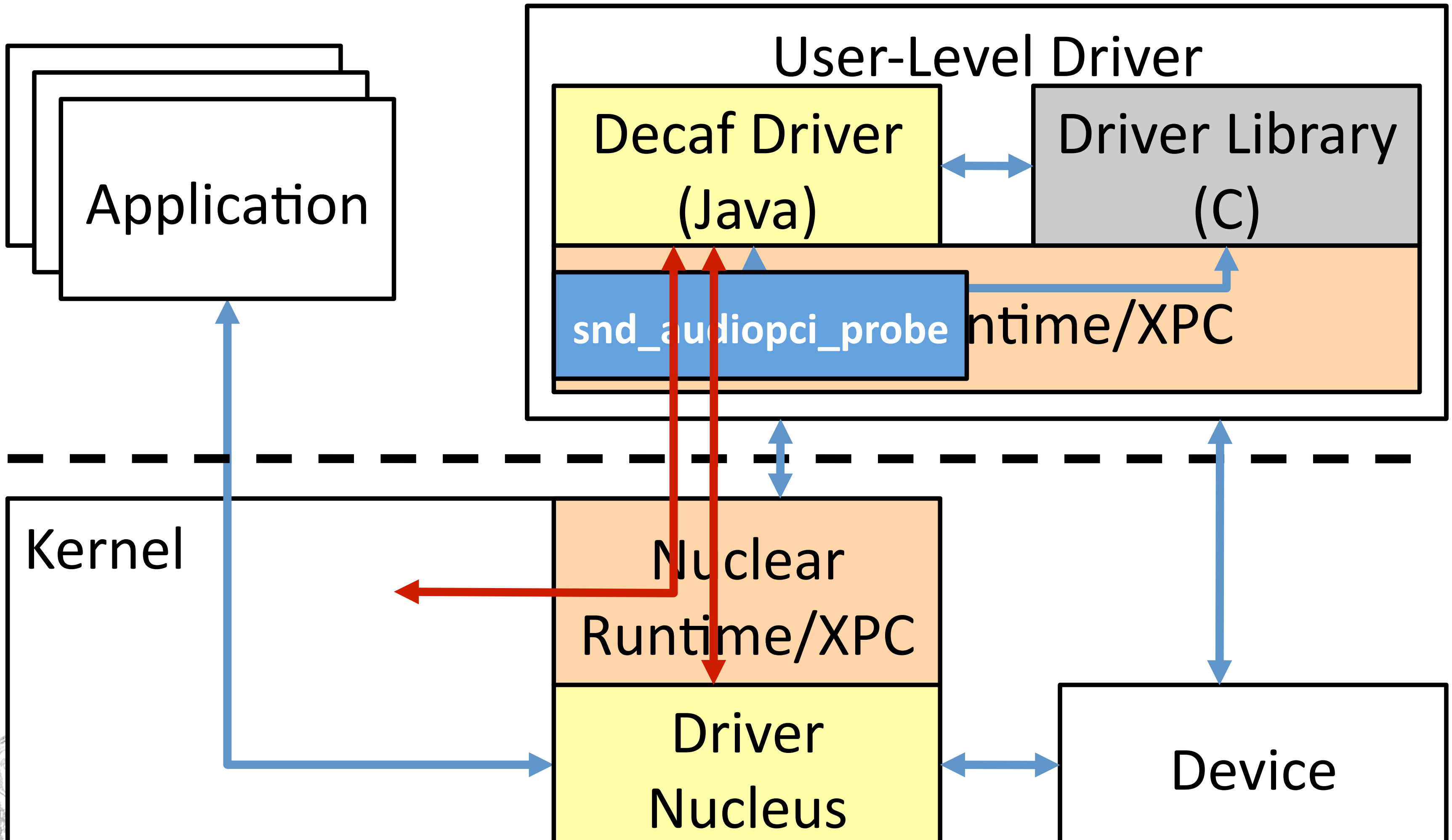
- Solution: Use Jeannie [Hirzel, OOPSLA ’07]
  - Allows C and Java code to be mixed at the expression level
  - Uses the back tick operator (`) to switch from Java to C
  - No need to write Java Native Interface code

```
public static void outb(int val, int port) `{
   outb (`val, `port); // No XPC necessary
}
```

Complex Java/C Transfer: XPC

- Example: invoking the Java implementation of **snd_audiopci_probe** from C
- Complex data structures are communicated via Java/C XPC
  - XPC uses marshaling and demarshaling to transfer data structures
  - Wrappers implemented using Jeannie simplify control and data transfer

Java/C XPC


Application
User-Level Driver
Decaf Driver (Java)
Driver Library (C)
snd_audiopci_probe
ntime/XPC
Kernel
Nuclear Runtime/XPC
Driver Nucleus
Device

Creation: DriverSlicer

- Goal: Migrate code in existing driver to Java
- DriverSlicer features
  - Splits drivers into a driver nucleus and library
  - Provides access to kernel data and functions from Java

Access to Kernel Data and Functions

- Phase one: CIL-based tool
  - Extracts all data structure definitions and typedefs
  - Converts these definitions to an XDR specification
- Phase two: Enhanced existing **rpcgen** and **jrpcgen** tools
  - Create Java classes with public fields
  - Support features like recursive data structures

Phase 1: Example

From e1000.h

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

Original C code

```
typedef unsigned int uint32_t;

struct uint32_256_t {
   uint32_t array_256[256];
};

typedef struct uint32_t_256 *uint32_t_256_ptr;

struct e1000_adapter { …
  struct e1000_rx_ring test_rx_ring;
  uint32_t_256_ptr config_space;
  int msg_enable;
… };
```

Automatically-generated XDR Definition

Phase 2: Continued

```
typedef unsigned int uint32_t;
struct uint32_256_t {
   uint32_t array_256[256];
};
typedef struct uint32_t_256 *uint32_t_256_ptr;
struct e1000_adapter { …
  struct e1000_rx_ring test_rx_ring;
  uint32_t_256_ptr config_space;
  int msg_enable;
… };
```

Automatically-generated XDR Definition

```
public class e1000_adapter … { …
  public e1000_rx_ring test_rx_ring;
  public uint32_t_256_ptr config_space;
  public int msg_enable;
…
  public e1000_adapter () { … }
  public e1000_adapter(XdrDecStream xdr) { … }
  public void xdrEncode(XdrEncStream xdr) { … }
  public void xdrDecode(XdrDecStream xdr) { … }
}
```

Automatically-generated Java

Driver Evolution

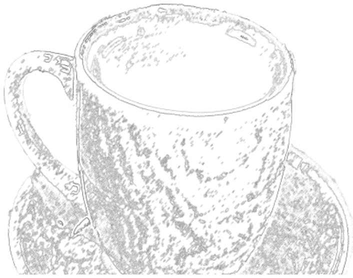
- Example: E1000 network driver 2.6.18.1 to 2.6.27
  - e1000_adapter structure needs additional members

➢ XPC does not transfer new fields automatically

- Solution: the driver is the specification
  1) Add new member definitions to original e1000.h
  2) Re-run DriverSlicer
  3) Use variables in Driver Nucleus or Decaf Driver

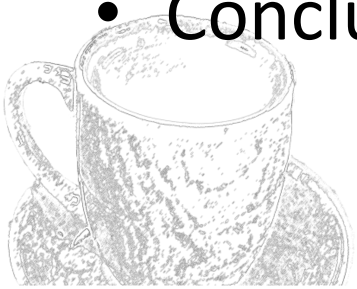
Design Summary

- Decaf meets its goals
- Decaf supports
  - Compatibility with existing drivers
  - A migration path from C to Java
  - Evolution of kernels and drivers



Outline

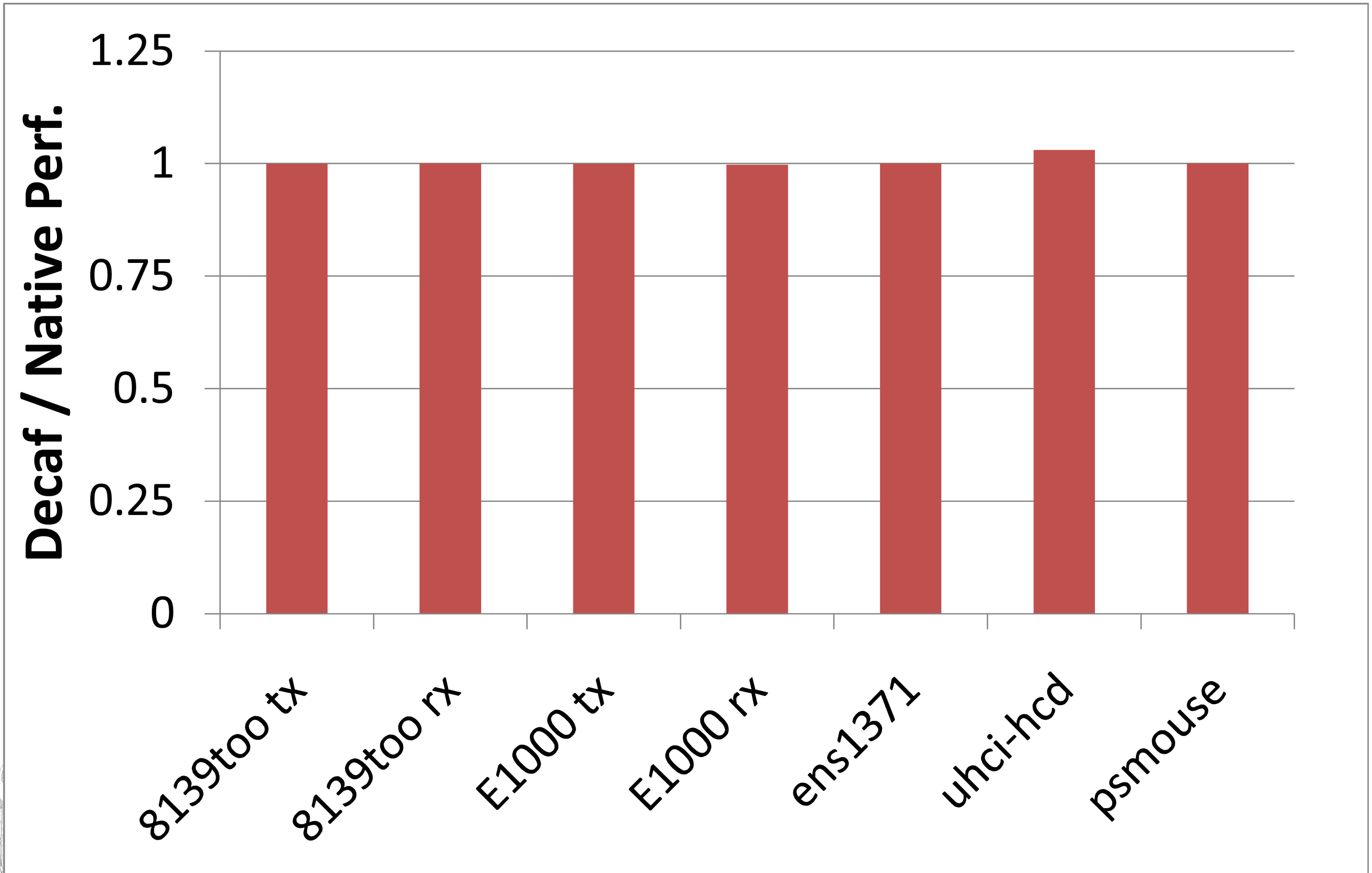
- Introduction
- Overview
- Design and Implementation
- Evaluation
  - Conversion effort
  - Performance analysis
  - Benefits of Decaf Drivers
    - Case study of E1000 gigabit network driver
- Conclusion



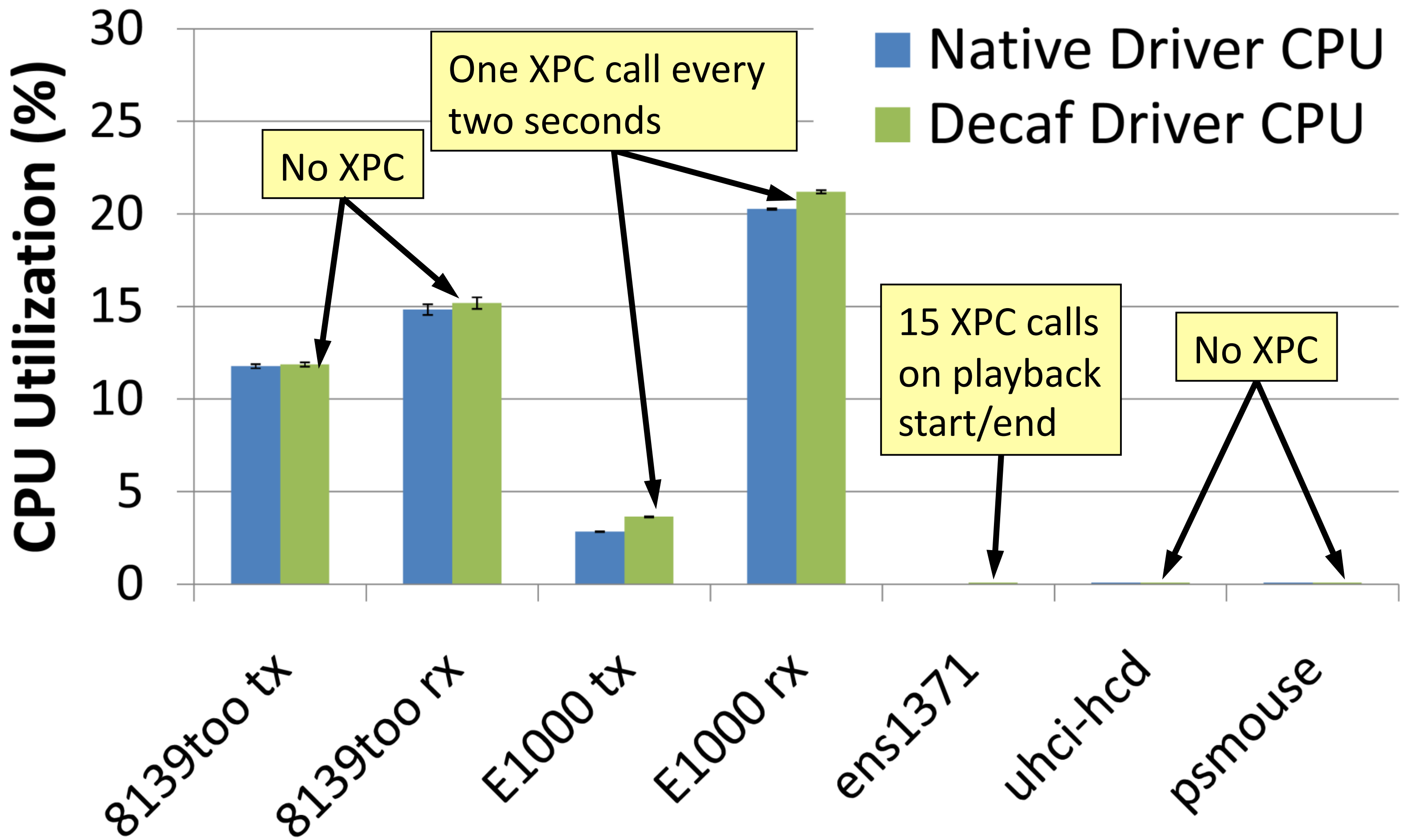
Conversion Effort

Driver	Original Lines of Code	Anno-tations	Functions		
			Driver Nucleus	Decaf Driver	Driver Library
e1000	14,204	64	46	236	0
8139too	1,916	17	12	25	16
ens1371	2,165	18	6	59	0
psmouse	2,448	17	15	14	74
uhci-hcd	2,339	94	68	3	12

Results: Relative Performance



Results: CPU Utilization



E1000: Core 2 Quad 2.4Ghz, 4GB RAM

All others: Pentium D 3.0Ghz, 1GB RAM

Experience Rewriting Drivers

- Step one: initial conversion
  - Largely mechanical: syntax is similar
  - Leaf functions first, then remainder
- Step two: use Java language features
  - Example benefit: E1000 exception handling

Java Error Handling

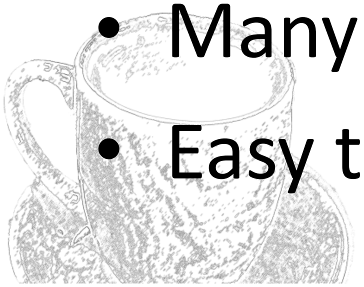
Original C, e1000_hw.c

```
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;
```

- Many extra conditionals
- Easy to miss an error condition



Java Error Handling

Java, e1000_hw.java

```
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);
```

- E1000 Decaf Driver: using exceptions
  - Uncovered at least 28 cases of ignored error conditions
  - Resulting code 8% shorter *overall*

Conclusions

- Decaf Drivers simplify driver programming
  - Provide a migration path from C to Java
  - Allow driver code to run in user mode
  - Support continued driver and kernel evolution
  - Offer excellent performance

Questions?

For more information:

mjr@cs.wisc.edu

swift@cs.wisc.edu

http://pages.cs.wisc.edu/~swift/drivers