On Objectifying Untyped Memory in Java Operating Systems

Christian Wawersich, Meik Felser, Jürgen Kleinöder
University of Erlangen-Nuremberg
Dept. of Computer Science 4 (Distributed Systems and Operating Systems)
Martensstr. 1, 91058 Erlangen, Germany
{wawersich, felser, kleinoeder}@informatik.uni-erlangen.de

Abstract

Accessing memory is essential in hardware programming, because the configuration and controlling of the hardware is often done via memory-mapped registers or control structures in memory. In a programming language like C, the structure of a register or the memory is mapped to a `struct`*. In the JX operating system [GFW+02] the device drivers are written in Java, where the type safety forbids a mapping from objects to untyped memory.*

In this paper we present a way to map an untyped memory to a Java object so that the structure of the memory is represented by the objects fields. Our bytecode to nativecode translator is responsible to redirect accesses and to ensure the type safety of the code.

1 Introduction

When an operating system is intended for a large application domain, the number of supported devices is, among other things, criteria for serviceability. Therefore it is important to encourage developers to create new drivers. One point in driver development is the communication and control of the hardware.

Hardware is programmed by the use of registers. Similar information is often accessed through one register. Therefore registers are often structured in bit fields and are used to access several logical values with one read or write cycle. For easy access to registers they are often mapped into the memory address space.

Beside registers, modern hardware uses configuration structures in main memory that is accessed by the hardware via DMA (direct memory access).

In order to avoid errors when accessing the data stored in a command buffer or in a register, a structured data type is expedient. Most operating systems, including their device drivers, are written in C. For a convenient access to the fields of a command buffer or to an individual value in a register, a structure can be mapped to the respective memory area. The benefit of this operation is that each field or bit of the control structure can be accessed via an individual name. When the mapping is correct, access to a specific field is also correct, mistakes due to false shift or mask operations are eliminated.

Java does not provide such a mapping operation, because Java does not know anything about addresses. Nevertheless, driver development in Java is possible by introducing additional classes or types that where specially treated by the compiler or the run time system to handle untyped memory areas. In consequence a mapping operation is needed to simplify the driver development. It must be similar to C to reduce the barrier to port drivers to Java operating systems. In this paper we present a mapping operation that redirects accesses to Java objects to an associated memory area.

The structure of this paper is as follows: In the next section we give a very short overview of the JX operating system and a detailed description of our memory objects, our abstraction to handle untyped memory in Java. Section 3 describes a simple approach of mapping objects with primitive data types to memory objects. In section 4 this approach is extended to be more flexible by mapping separated object fields. The results are presented in section 5.

2 Overview of the JX Operating System

2.1 Domains

The JX operating system [GFW+02] is based on a small microkernel which is responsible for system initialization, CPU context switching, and low-level protection-domain management. The Java code is organized in components, which are loaded into domains, verified, and translated to nativecode. Domains can be seen as completely isolated JVMs, communicating via lightweight remote method invocation. The microkernel is also represented by a separated domain which we call DomainZero.

2.2 Portals

The lightweight remote method invocation mechanism in JX is done via so called portals. A portal can be seen as a

proxy for an object that resides in another domain. An entity that may be accessed from another domain is called *service*. A service consists of a normal object, which must implement a portal interface, and an associated service thread that performs the service. When a thread invokes a method at a portal, the calling thread is blocked and execution is continued in the service thread.

Portals can be exported by user implemented domains but are also used when communicating with DomainZero. All services provided by DomainZero are accessed using a portal to the appropriate service.

Several portals which are exported by DomainZero are implemented in an optimized way, we call them *fast portals* [WFG+02]. A fast portal invocation looks like a normal portal invocation but is executed in the caller context (the caller thread) by using a function call or even by inlining the code.

2.3 Memory Objects

An operating system needs an abstraction to represent and control access to untyped memory. We use so called *memory objects* to represent memory ranges. Memory objects are a service of DomainZero and are accessed via portal invocation. The base address and the size of the memory area are stored in the corresponding proxy object (see figure 1). On access the service performs range checks and returns or modifies the data at the desired position.

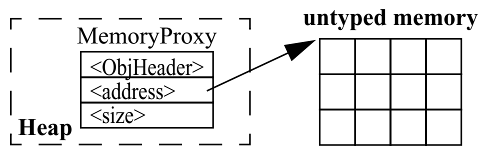

Figure 1: MemoryObject

To eliminate the overhead of portal invocation we extend our bytecode to nativecode translator. All calls to `get` and `set` methods of the memory portal are treated specially: they are replaced by the machine instructions needed for memory range checks and the load or store instruction to access the specified memory position. This makes memory accesses as fast as accesses to Java arrays.

The memory object has still two major disadvantages. It is slower than a memory access in C, because of the range check and fault-prone in comparison with typed data structures.

3 Object Mapping with primitive Types

In JX we use memory objects to represent an area of memory. Memory objects look like data containers and can be used to access individual memory locations using `set` and `get` methods, but there is no possibility to see the structure of the underlying memory. However, the memory that is represented by a memory object is virtually always structured. For example a memory object holding a command buffer that is part of a linked list of buffers can be structured in a command part and the link pointer to the next command buffer. A network packet can be structured into a header and the payload.

In programming languages like C the structure can be represented by mapping a `struct` on the untyped memory. A similar approach is desirable for Java; map an object to an area of memory. In a elementary approach mapping primitive data fields of objects to a continuous memory area would be adequate. Accesses to the object fields are converted to accesses to the underlying memory (see figure 2). A mapping function is responsible to map the object layout to the underlying memory.

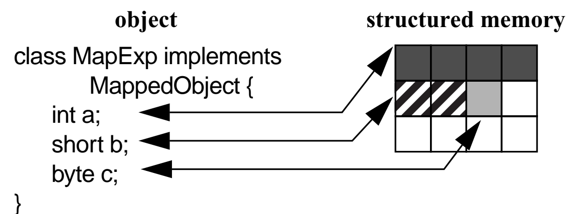

Figure 2: Simple Object Mapping

To distinguish mapped objects from normal ones, the interface `MappedObject` must be implemented by each class whose instances should be mapped to a memory object. Accesses to these objects are identified by the bytecode to nativecode translator and whenever an access to a field occurs the translator inserts code to redirects the access to the associated memory area.

Since data accesses were redirected, there is no need to reserve space for the object's fields in the object layout. Instead of this a reference to the corresponding memory object is stored in the object's data space, so that the compiler generated code finds the correct memory area.

To ensure the type safety of Java we have to introduce additional restrictions to the mapped objects. Our bytecode verifier ensure that mapped objects are only created by a special method at memory portal. Mapping of arbitrary values to reference types can violate the type system, thus mapped objects can only contain primitive integer types (byte, short, char, int, long). This is checked when creating a mapped object.

Beside the improved readability, accessing memory with mapped objects has an additional advantage to directly using the memory object's `get` and `set` methods. The `get` and `set` methods have to check whether the access is in the range of the associated memory area. This check can be accomplished when mapping an object to a memory area. Accesses to the object fields can then be directly translated to memory accesses without range checks.

4 Object Mapping with Classes

4.1 Classes for Mapped Object Fields

The approach of object mapping with primitive types has a few shortcomings.

- only primitive types specified by the JVM can be used, unsigned types or different byte order are not supported
- it is not possible to store references or additional values in a mapped object
- it has to be ensured, that only primitive types are used

```
import jx.zero.memory.*;

public class MapExp implements MappedObject {
    public  MT_Static_32LE a;
    public  MT_Static_16LE b;
    public  MT_Static_8 c;
}
```

Figure 3: Example Class

The problems can be solved by using Java classes instead of primitive types to describe the structure of a memory area. Figure 3 shows the example from chapter 3 implemented with special classes for different types of mapped fields. The object field `a` is of the type `MT_Static_32LE`. This type is used to describe a 32 bit little endian object field in the memory area which is the equal to the primitive type `int` in the first example. For every data size and different byte order an extra class is introduced. All this classes are derived from the class `jx.zero.memory.MT` where `MT` stands for memory type.

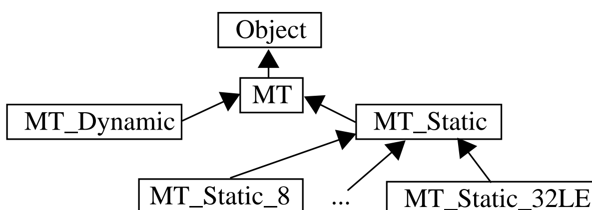

Figure 4: The Hirachie of Memory Types

All classes for fields with a fixed offset inside of the memory area are derived from the subclass `MT_Static.` The field offset is given by the position and size of the fields in the memory-mapped class. The field `a` in the example (figure 3) has an offset of 0, the field `b` has an offset of 4 and the field `c` has an offset of 6. All object fields with a type not derived from `MT_Static` are ignored by the offset calculation. We will call object fields with a type derived form `MT_Static` statically mapped object fields or simply statically mapped fields.

Besides static calculated offsets, it is also possible to map fields with a free adjustable offset. The offset is defined at run time during the map operation. Classes for adjustable object fields are derived from the class `MT_Dynamic`.

4.2 Object Layout

The object layout of a mapped object is changed to store extra information about the memory area (figure 5).

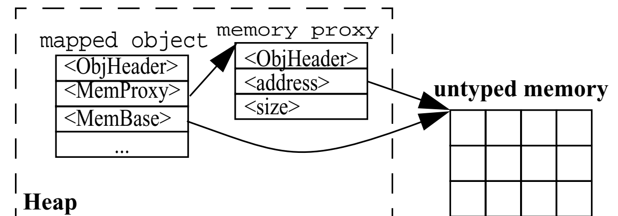

Figure 5: Object Layout of Mapped Object

A field with the name `<MemProxy>` is added to store an object reference to the memory proxy. This reference is needed to protect the memory object from garbage collection. A second field called `<MemBase>` is added to contain the base address for all statically mapped fields. Furthermore, all object fields with types derived from `MT_Static` are removed. Primitive types and normal references are handled as usual. Therefore it is possible to include references to other objects or store additional values in a mapped object.

4.3 Object Creation

The modifications of the object's layout were similar to the modifications mentioned in section 3 but instead of creating an object with special layout, the translator modifies the class description. As consequence memory-mapped objects are created like normal objects.

But there is still a problem, when the object is created it is not yet mapped to a memory area. Therefore accesses to `MT` typed fields can not be allowed. To omit extra checks on field accesses, we extend the constructor so that the extra fields were initialized with reasonable data. The memory proxy reference of a newly created object is set to zero and the base address for static mapped fields are set to an extra memory space allocated in the static area of the class. This kind of initialization makes the object full functional from the beginning and no extra checks are needed.

4.4 Memory Mapping

The objects are mapped by the `mapObject` method of a memory object. The memory object is a service of the DomainZero and has therefore the privilege rights.

The expected parameter type for the `mapObject` method is checked by the Java compiler and verified by the bytecode verifier. Therefore only objects of the type `MappedObject` can be mapped. The `mapObject`

method has to perform an additional test whether the reference is zero or not and can then perform the range check. If the checks are successful, the reference to the memory proxy and the base address of the untyped memory is stored in the object's fields (see figure 5).

4.5 Field Access

Read accesses to fields of a mapped object are treated differently by the JX translator if the field's type is a subclass of `MT`.

Write accesses to this fields are completely removed by the translator, thus, only the `map`Object method of the memory object can adjust the reference to a memory field.

Is a statical mapped field used as a reference to invoke the `get` or `set` method then the translator inserts code to compute the destination address in the memory area associated with the mapped object and the call to the method is replaced by the machine instructions needed to access the memory area. The destination address is computed relative to the base address. The offset of statically mapped object fields is known at compile time, the offset of dynamically mapped fields was determined by the mapping operation.

Is a statical mapped field used for other purposes where a real object reference is needed, for example when the reference is stored in another object field, the compiler inserts code to create a wrapper object.

4.6 Automatic Wrapping of Memory References

For wrapping a memory reference there are real class implementations for all types derived form `jx.zero.memory.MT`. Figure 6 shows the implementation for the class `MT_Static_32LE`. All these classes have one private field `self`, for accessing the represented memory. Furthermore they implement the `MappedObject` interface and therefore they are again mapped objects.

```
package jx.zero.memory;

final public class MT_Static_32LE extends MT_Static {
    private MT_Static_32LE self;

    public  int get() { return self.get(); }
    public  void set(int i) { self.set(i); }
    /* ... */
}
```

Figure 6: The Class `MT_Static_32LE`

The code, generated for wrapping, copies the additional field `<MemProxy>` from the original mapped object and fills the `<MemBase>` field with the address of the wrapped memory reference. Accesses to the `self` field are handled in the way as described in section 4.5. Therefore the `get` and `set` methods of the wrapping objects can delegate the invocation to the `self` reference.

```
Memory mem = memoryManager.alloc(16);
MappedExample m = new MappedExample
m = (MappedExample) mem.mapObject(m);

/* ... start timer ...*/
for (int i=0; i<ntries; i++) {
    mem.set32(0,42);
    mem.set32(4,43);
    mem.set32(8,44);
    mem.set32(12,45);
}
/* ... end timer ... */

/* ... start timer ... */
for(int i=0; i<ntries; i++) {
    m.a.set(42);
    m.b.set(43);
    m.c.set(44);
    m.d.set(45);
}
/* ... end timer ... */
```

Figure 7: Micro Benchmark for Field Access

5 Discussion

Object mapping reduces the number of range checks, this should result in a speedup and reduction of code size. We created a small micro benchmark to measure the time needed to access the memory with mapped objects and compare the result with the time needed for accessing the same memory via the memory object. Figure 7 shows the micro benchmark for write accesses. We executed the benchmarks on a Pentium 4 based PC with 2200 MHz and the time was measured by reading the time stamp counter of the CPU.

Table 1 shows the time needed for 10000 write and read operations. With the mirco benchmarks a relative speedup of 8% for writes and 33% for reads can be shown. The average time needed for a write operation is 3,2 nanoseconds or 7.04 clock ticks at 2200 MHz clock speed.

	mapped object	memory object	absolute speedup	relative speedup
10k writes	32 µs	35 µs	3 µs	8%
10k reads	18 µs	27 µs	9 µs	33%
1 write	3.2 ns	3.5 ns	0.3 ns	
1 read	1.8 ns	2.7 ns	0.9 ns	

Table 1: Results of Field Access Benchmark

The runtime overhead for object mapping is also important to decide whether object mapping is useful for perfor-

mance improvement or not. We measured the time for object mapping without and with object creation. The mapping of 10000 objects needed 286 microseconds or 28.6 nanoseconds per mapping (Table 2). This leads to the conclusion that memory mapping can only improve the performance of memory access when at least 96 write accesses or 32 read accesses per mapping occur.

	mapped object
10k mapping	286 µs
10k create & map	1770 µs

Table 2: Result of Mapping Benchmark

The mirco benchmarks confirm that the reduction of range checks considerably improve the performance of a single memory access. But the absolute time saved is rather small and may therefore be irrelevant for the overall performance. The performance of a driver is stronger affected by the overall driver design. The powerful technique of object mapping can improve the design of device drivers in a way the micro benchmarks cannot show. Driver developers often wrap the memory objects into task-oriented classes that provides methods for structured access to the memory. These methods just delegate the calls to a memory object. This kind of abstraction layer can be replaced by memory-mapped objects and therefore improve the performance.

6 Related Work

Driver development and access to untyped memory in Java is not a width topic of research. The Plurix operating system uses the Java language for operating system development. But they have no support for Java bytecode and, to our knowledge, they do not have a technique to structure untyped memory. JaOS [Rea03] has a layered driver design where the memory access is done in a different language.

The auto wrapping mechanism in section 4.6 is comparable to auto boxing of primitive types, which is most recently introduced to Java 1.5 on the language level. On the bytecode level an extension for boxing is proposed by Shivers [Shi96] and implemented by Yutaka [YKY00]. Both of them where motivated by compiling dynamic languages like Scheme, Lisp, Dylan or Smalltalk to the JVM.

The modified object layout for mapped objects is similar to object inlining. The most well-known work on object inlining is done by Dolby and Chien [DC00] and an improved and Java related work is done by Laud [Lau01] or Lhoták and Hendren [LH02]. They analyse class definitions to detect, whether some child object could be stored together with their parent. In this case the reference from the parent object to the child object will be replaced with the actual data of the child object. We inline all reference to classes of the type `MT` in memory-mapped objects but we are changing the semantic of store operations.

7 Conclusion

In this paper we describe a flexible solution to map an object to an untyped and unstructured memory inside of a Java operating system. Our intention is to improve the driver development with regard to robustness and performance.

A few mirco benchmarks show that object mapping can improve the performance of single memory accesses. But only a real driver implementation can show if object mapping leads to an improved driver performance and a more robust driver development.

8 References

DC00 J. Dolby and A. A. Chien. An automatic object inlining optimization and its evaluation. In: *Proc. of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, p. 345-357, 2000.

GFW+02 M. Golm, M. Felser, C. Wawersich, and J. Kleinöder. The JX Operating System. In: *Proc. of the Usenix Annual Technical Conference*, p. 45-58, Monterey, June 2002.

Int03 Intel Corporation. Intel 825xx 10/100 Mbps Ethernet Controller Family Open Source Software Development Manual. January 2003.

Lau01 P. Laus. Analysis for object inlining in Java. In: *Proc. of the JOSES workshop,* 2001. http://rw4.cs.uni-sb.de/bib2html/old.html

LH02 O. Lhoták and L. Hendren. Run-time Evaluation of Opportunities for Object Inlining in Java. In: *Proc. of the Joint ACM Java Grande - ISCOPE 2002*, p. 175-184, Seattle, WA, USA, November 2002

Rea03 Patrik Reali; Using Oberon's Active Objects for Language Interoperability and Compilation; ETH Dissertation 15022, 2003

Shi96 O. Shivers. Supporting dynamic languages on the Java virtual machine.AIM-1576, 1996. http://citeseer.ist.psu.edu/shivers96supporting.html

WFG+02 C. Wawersich, M. Felser, M. Golm, J. Kleinöder. The Role of IPC in the Component-Based Operating System JX. In: *Proc. of the 5th ECOOP Workshop on Object-Orientation and Operating Systems*, Malaga, June 11, 2002.

YKY00 O. Yutaka, K.Taura and A. Yonezawa. Extending Java virtual machine with integer-reference conversion. In: *Concurrency: Practice and Experience*, volume 1, number 6, p. 407-422, 2000