Untyped Memory in the Java Virtual Machine

Andreas Gal and Michael Franz
University of California, Irvine
{gal,franz}@uci.edu

Christian W. Probst
Technical University of Denmark
probst@imm.dtu.dk

July 1, 2005

Abstract

We have implemented a virtual execution environment that executes legacy binary code on top of the type-safe Java Virtual Machine by recompiling native code instructions to type-safe bytecode. As it is essentially impossible to infer static typing into untyped machine code, our system emulates untyped memory on top of Java's type system. While this approach allows to execute native code on any off-the-shelf JVM, the resulting runtime performance is poor. We propose a set of virtual machine extensions that add type-unsafe memory objects to JVM. We contend that these JVM extensions do not relax Java's type system as the same functionality can be achieved in pure Java, albeit much less efficiently.

1 Executing Native Machine Code on top of JVM

Our research prototype VEELS (Virtual Execution Environment for Legacy Software) is able to execute statically compiled Linux [2] executables for PowerPC, StrongARM, and MIPS on top of any standard Java Virtual Machine [4]. All native code and data is stored within a memory object that emulates untyped memory. A virtual CPU fetches instructions from the memory object and interprets them. Frequently executed code blocks are translated to bytecode by a just-in-time compiler to allow more efficient execution Figure 1.

2 Emulating Untyped Memory

The standard implementation of the memory object in VEELS uses a sparse integer array as backing store. Each 4k page of the guest machine is modeled as an array of integers, referenced by a page table. The most common access patterns—reading and writing words from and to memory at an aligned address—are handled efficiently with two memory accesses and a pair of array bounds checks.

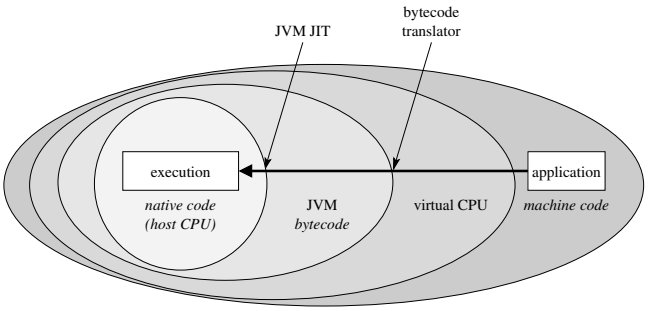

Figure 1: Executing native machine code on top of a Java Virtual Machine (JVM). The machine code is translated to Java byte-code and then forwarded to the JVM for execution. While the JVM will initially interpret the bytecode, the JIT compiler available in most JVMs will select often referenced bytecode fragments and compile them to directly executable machine code for the host CPU. Thus, for relevant hot spots, the proposed architecture is able to translate target machine code to directly executable host machine code via generating portable Java bytecode.

Using a flat integer array to model main memory would be faster, but is unfortunately not feasible because the JVM initializes integer arrays to contain all zeros. This initialization forces the operating system to actually allocate physical memory for the entire array, quickly consuming the entire physical memory of the host machine. It would also not be possible to emulate an entire 32-bit address space on a 32-bit host machine, because the memory object would consume the entire host address space. Using the page-table approach, memory is allocated page by page as needed.

To remove any unnecessary checks we use JVM-internal null-checks and array-bounds check to handle page-faults and alignment-faults (Figure 2). Each page-table (`pt`) entry is an array of 1024 integers. When forming the index into the page itself, an index value greater 1024 is generated for unaligned addresses (two least significant bits of `offs` unequal 0), triggering an `ArrayIndexOutOfBounds` exception, which triggers the slow path implementation that reads individual bytes from memory and assembles them to a word using `to_word()`. Similarly, unallocated pages cause an `NullPointer` exception and `get_bytes()` allocates new pages as needed.

An unexpected side-effect of this approach, which we did not initially consider, concerns garbage collection. The page table for a 4GB address space has 1 million entries and is 4MB in size. Each entry is a potential reference to a page, which is an array of integers at the JVM level. This means that suddenly the garbage collector has to scan a huge array of references during its mark phase, causing a noticeable slowdown.

One possible solution to this problem could be a 2-level page table using a page directory pointing to smaller page tables. However, this would introduce another level of indirection, making memory accesses even slower than they already are. Another possible approach would be to increase the page size, but this would increase the amount

```
public static final int read_word(int addr) {
 int pg = addr >>> 12;
 int offs = addr & 0xfff;
 try {
  return pt[pg][(offs >>> 2) | ((offs&3) << 16)];
 } catch(Exception e) {
  return to_word(get_bytes(addr, 8), offs&3);
 }
}
```

Figure 2: Reading a word from the memory of the guest machine. An array index greater 1024 is formed for unaligned addresses, causing the fast path to fail (alignment fault). An exception is also raised if a page has not yet been allocated (page fault). In the slow path `get_bytes()` is used to read two words worth of bytes from the memory, which are then in turn converted to a word starting at the proper alignment offset. `get_bytes()` also allocates pages when they are accessed for the first time.

of physical memory wasted due to granularity.

A simple microbenchmark shows that the standard memory object is 15 to 5500 times slower than physical memory access.[1] The best access time is achieved for aligned word reads, while the worst relative performance is observed for unaligned double word writes crossing a page boundary.

3 New Virtual Machine Constructs

To improve the performance of our native memory emulation, we have extended IBM's Jikes Research Virtual Machine (RVM) [1] to support untyped memory. For this, we introduced a serious of new *magic methods*, which appear like regular Java-methods at compile-time, but are compiled to specific machine code sequences at runtime.

For the Intel x86 host platform, we use the physical memory management unit (MMU) [3] to translate from the guest address space to the host address space. A dedicated selector is initialized to point to a block of host memory and that selector is loaded into the otherwise unused FS selector register of the CPU. For each magic method we encounter, our modified RVM compiler generates a *fs:mov* instruction that addresses memory relative to the specially prepared FS selector. As a result, all address offset calculation and bounds checking is performed in hardware and memory access is as efficient as in case of native code.

Due to the limited address space of the Intel x86 host platform, however, we cannot emulate an entire 32-bit address with this method. We are currently exploring the applicability of this approach to 64-bit host platforms. As almost no 64-bit CPU offers a segmented memory model as present on the Intel x86 platform, we have to slightly

[1] Measured on a 2.2GHz Pentium 4 Linux workstation with 1 GB of physical RAM. 128 MB of untyped memory was emulated on IBM's JDK 1.4.2. The average time for 500 million access operations was measured and compared to the physical memory bandwidth. All pages were preallocated to remove the page allocation overhead from the benchmark. The garbage collector was disabled during the experiment.

modify our approach to achieve similiar efficiency: Instead of using a selector register, the 64-bit base address of the memory block used as backing store is stored in a general purpose register and register-relative addressing is used to retrieve data from and store into the memory object. Bounds checking becomes obsolete, because the vast address space on 64-bit CPUs allows to allocate an entire 4GB block of memory[2], making any 32-bit address within bounds by definition.

4 Implications for Type-Safety

The Java language is type-safe and it carefully distinguishes not only between references and scalar types, but also between different types of scalars such as integers and floats. Our proposed extension essentially voids most type-safety guarantees for scalar types. Using the untyped memory block, we can write an integer at address X, and then retrieve it as a float. Within the bounds of Java's type-system this would be impossible.

So does our extension make Java any less type-safe?

We contend that the answer is no. Java already allows to arbitrarily transform integers into floats at the bit level by using a native method interface (see floatToRawIntBits [5]). Also, we have implemented the very same memory object in pure Java, without using any language or compiler extensions.

In fact, one could even argue that the mere existance of a pure Java implementation of an untyped memory object proves that pure Java is already type-unsafe as far as scalar types are concerned.[3]

We believe that providing untyped memory, whether trough the slower pure-Java implementation or the more efficient compiler-based approach, also has far reaching implications as far as our general understanding of Java memory-safety guarantees is concerned. So far the general assumption has been that Java programs are immune to buffer overflow problems that have plagued the C/C++ world. Our virtual CPU emulation with its untyped memory area partially invalidates this assumptions, at it suddenly becomes possible for a buffer overflow to happen inside the emulated CPU environment.

5 Conclusions

We have presented an approach to efficiently implement untyped memory in a type-safe virtual machine by extending the just-in-time compiler to recognize certain *magic methods* and to generate specialized machine code instructions at the invocation site instead of actually invoking the magic method itself. We have shown that this approach is partially feasible for 32-bit processors that support a segment memory model, but to emulate a full 32-bit guest address space a 64-bit host CPU is required. We have

[2] As in case of our trivial memory object implementation, only memory pages that are actually referenced at runtime have to be backed by physical memory. The majority of virtual pages in the memory block will never be assigned a physical memory page.

[3] However, neither of our untyped memory object implementations allows to store or retrieve *references* from untyped memory. This is not necessary, because all addresses of the virtual CPU are represented as integer values.

further argued that while at first glimpse adding untyped memory to a type-safe virtual machine seems to compromise type-safety, this very type-safety is not present in Java in the first place, because the same untyped memory object can be implemented in pure Java (albeit much less efficiently).

References

[1] B. Alpern, C. R. Attansio, J. J. Baron, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocci, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russel, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM System Journal*, 39(1), February 2000.

[2] A. Gal, M. Yang, C. W. Probst, and M. Franz. Executing Legacy Applications on a Java Operating System. In *2004 First ECOOP Workshop on Programming Languages and Operating Systems*, Juni 2004.

[3] Intel Coorperation. Intel486 Processor Hardware Reference Manual, 1997.

[4] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[5] Sun Microsystems. Documentation for the `java.lang.Float` class.