Executing Legacy Applications on a Java Operating System

Andreas Gal, Michael Yang, Christian Probst, and Michael Franz
University of California, Irvine
{gal,mlyang,probst,franz}@uci.edu

May 30, 2004

Abstract

One of the biggest disadvantages of using type-safe languages and frameworks such as Java for the implementation of commercial operating system (OS) platforms is the lack of backward compatibility with legacy execution environments. Traditional Java operating systems are designed to execute Java programs. They struggle to support established application programming interfaces such as POSIX or Win32 as these are founded on the idea of direct execution of native machine code. Instead of using the host processor to execute legacy applications – which would eliminate most of the security and portability benefits of using a type-safe language for the OS implementation in the first place – we propose using a virtual processor implemented on top of the virtual machine infrastructure to execute legacy applications.

1 Rationale

The recent surge in security exploits for the Microsoft Windows platform [5] demonstrates the need for foundationally safe operating systems (OSs) in internet-based computing. Language-based security is a promising new approach to rid applications from buffer-overflows, format string attacks and similar vulnerabilities. Currently, type-safe languages such as Java or C# are predominantly used in application code. Operating systems continue to be implemented in unsafe languages such as C or C++. To offer a viable alternative to the established operating system platforms Windows, Linux, etc., Java-based OSs need to learn to support the execution of legacy applications. Existing implementations such as JavaOS [6], JX [3], or JOS [4] implement the majority of OS code in type-safe Java code, but allow to execute Java applications only. Emulating Java-based Application Programming Interfaces (APIs) on top of a Java OS is simple and has been successfully demonstrated by many Java-based OS implementations. However, the vast majority of existing applications are written in type-unsafe languages and many of them are available as target-machine specific executables only. Being able to execute such applications on top of a Java-based OS would allow users to switch to a Java-based platform even before all applications have been ported.

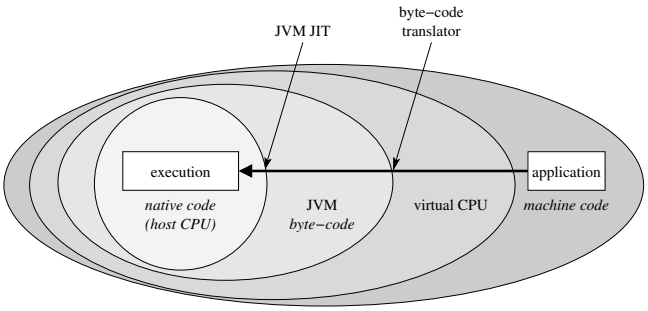

Figure 1: Executing native machine code on top of a Java Virtual Machine (JVM). The machine code is translated to Java byte-code and then forwarded to the JVM for execution. While the JVM will initially interpret the byte-code, the JIT compiler available in most JVMs will select often referenced byte-code 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 byte-code.

2 Performance

Full system virtualization is a challenge and a number of optimizations have to be applied to achieve acceptable performance. Simple interpretation-based processor emulators such as the PowerPC750 Simulator [2] usually suffer from a slowdown from anywhere between 60 to 500 times in comparison to native execution. In contrast to byte-code that has been designed for interpretation, machine code instructions are often expensive to execute in software as they contain optional computations that are essentially free if done in hardware, but costly if performed step-by-step in software. A well-known example for this is the flag register of the Intel 386 CPU. While nearly all arithmetic instructions update the flag registers, less than 10% of the computed flag values are actually evaluated. In contrast to hardware, a software simulator has to pay a hefty price if redundant flag calculations need to be performed. To avoid this overhead, full system virtualizers such as VirtualPC [1] compile the target machine code to host machine code and optimize the host machine code by removing any redundant instructions.

The same principle used by VirtualPC can be applied when executing native code in a Java environment. Instead of trying to build an interpreter in Java – which would be even slower than existing interpreters written in C or C++ –, the machine code is instead translated to Java byte-code (Figure 1) and then forwarded to the Java Virtual Machine (JVM) for execution. While the JVM will initially interpret the byte-code, the JIT compiler available in most JVMs will select often referenced byte-code 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 byte-code. In contrast to existing emulators such as VirtualPC, the code translator depends on the

target system only and can generate efficient host machine code by relying on the JIT compiler as back-end for the code generation.

3 Implementation

We are currently implementing a research prototype for the proposed legacy machine code execution framework. The prototype is able to execute statically linked Linux ELF binaries for PowerPC on top of a standard Java VM.

A significant hurdle we had to overcome was the lack of appropriate reflection and dynamic code generation capabilities in Java. In fact, when dynamically generating Java byte-code on the fly, our framework has to emit Java class-files and load them into the VM using the standard class loader. This process causes a noticeable startup delay, especially for small applications that contain large amounts of run-once startup code. To counter this problem, we decided to add additionally to the code translator a direct interpreter. While the interpreter is many times slower than the execution of translated byte-code, it does not have any startup latency. During interpretation, a monitor detects basic block boundaries by recording all branch targets (basic block entries) and branch instructions (basic block exits) in a hash-table. If the interpreter detects that a basic block is executed repeatedly, the basic block is delegated to the code translator to be compiled to byte-code. The code translator compiles basic blocks of native code to methods of dynamically generated Java classes. Each of these methods takes an object holding the current CPU state (i.e. registers and flags) as parameter and returns the value of the program counter upon completion of the basic block execution.

Using basic-block based code translation, we are able to achieve a speed-up of factor 2 in comparison to pure interpretation by an optimized emulator implemented in C. However, this still represents an overall slowdown of factor 30 in comparison to direct execution of the code by a hardware CPU. The biggest single cost factor, once all relevant hot spots have been translated to byte-code and successively to native code by the JIT, is the interpreter loop that is executed every time a basic block is exited (Figure 2). Each compiled basic block returns the new value of the program counter (PC) as return value. The interpreter loop uses a hash-table to locate the Java method that holds the corresponding byte-code to that location in the native code. Besides the cost for the hash-table lookup, the JIT compiler is also not able to optimize across basic block boundaries, as all the invocation of basic block methods through the hash-table are virtual.

To further boost the performance of the emulator, this overhead has to be eliminated. For this, we are currently implementing a super-block based code translator that is not limited to mapping single basic blocks to Java methods, but tries to fit code located in frequently referenced basic blocks into the same Java method (Figure 2). Using this approach, branch instructions targeting basic blocks that reside in the same Java methods can be implemented using the Java *goto* instruction. Not only is the cost for this instruction much lower than exiting the method and going through the hash-table, but this also allows the JIT to detect loop headers and optimize the code accordingly.

Our prototype system is not yet able to perform super-block based code translation on non-trivial code. However, preliminary benchmarks indicate that we will be able

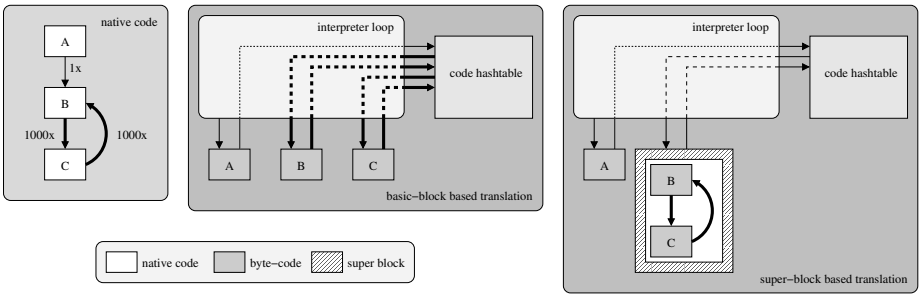

Figure 2: Basic block and super-block based translation of a loop in native machine code to Java byte-code. The basic-block based approach has to go through a hash-table twice per iteration to locate the successor basic block. The super-block based approach fits often referenced basic blocks into the same Java method and branch instructions can be modeled using the Java *goto* operation at much lower cost. Only references to external basic blocks still go through the hash-table.

to reduce the slowdown in comparison to direct hardware execution to a factor of 3 to 15 – depending on the type of application. The best performance can be expected for computational-intensive applications. Memory-intensive applications in contrast suffer significantly from the fact that our system has to emulated untyped memory on top of a safe virtual machine.

4 Conclusions and Future Work

Java-based operating systems have not gained much momentum in the real world. We believe that this is partially caused by the lack of backward-compatibility to legacy execution environments and APIs. We have presented an approach that allows to emulate any target machine on top of a type-safe virtual machine. While we expect a significant slowdown when executing legacy applications, we believe that the proposed approach is still feasible for real-world applications.

As far as future work is concerned, we plan to port our prototype system to the Microsoft .NET virtual machine, which has better capabilities in regards to dynamic code generation. Most importantly, .NET has a standard API to dynamically generate new types and methods whereas Java requires the application to manually assemble class-files that are then loaded into the JVM through the class loader.

We will also explore the feasibility of adding support for untyped memory blocks to the .NET virtual machine to allow a more efficient emulation of machine code instructions. As the same functionality can be achieved through an equivalent implementation using existing VM constructs, we do not consider such a new language constructor to be a threat to the security of the VM.

References

[1] Connectix Corporation. The Technology of Virtual Machines. 2001.

[2] S. Gibral, G. Mouchard, A. Cohen, and O. Temam. PowerPC 750 Simulator, 2004. http://www.microlib.org.

[3] M. Golm, M. Felser, C. Wawersich, and J. Kleinoeder. The JX Operating System. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 45–58, Monterey, CA, June 2002.

[4] G. Herschberger, T. Miller, and R. Fitzsimons. The JOS Java Operating System, 2004. http://sourceforge.net/projects/jos.

[5] Microsoft Cooperation. Microsoft Security Bulletins, 2004. http://www.microsoft.com/technet/security/CurrentDL.aspx.

[6] C. Sasaki. JavaOS Architecture. In *Proceedings of the 1997 JavaOne World Wide Java Developer Conference*, San Fransisco, CA, Apr. 1997.