Complete Translation of Unsafe Native Code to Safe Bytecode

Brian Alliet
Rochester Institute of Technology
bja8464@cs.rit.edu

Adam Megacz
University of California, Berkeley
megacz@cs.berkeley.edu

This document was typeset using D. E. Knuth's original TEX, which was both compiled and executed entirely within a Java Virtual Machine without the use of native code.

Abstract

Existing techniques for using code written in an unsafe language within a safe virtual machine generally involve transformations from one source code language (such as C, Pascal, or Fortran) to another (such as Java) which is then compiled into virtual machine bytecodes.

We present an alternative approach which translate MIPS binaries produced by any compiler into safe virtual machine bytecodes. This approach offers four key advantages over existing techniques: it is language agnostic, it offers bug-for-bug compiler compatibility, requires no post-translation human intervention, and introduces no build process modifications.

We also present NestedVM, an implementation of this technique, and discuss its application to six software packages: LINPACK (Fortran), which was used as one of our performance tests, TEX (Pascal), which was used to typeset this document, `libjpeg`, `libmspack`, and FreeType (all C source), which are currently in production use as part of the Ibex Project [**?**], and `gcc`, which was used to compile all of the aforementioned.

Performance measurements indicate a best case performance within 3x of native code and worst case typically within 10x, making it an attractive solution for code which is not performance-critical.

1 Introduction

Unsafe languages such as C and C++ have been in use much longer than any of today's widely accepted safe languages such as Java and C# Consequently, there is a huge library of software written in these languages. Although safe languages offer substantial benefits, their comparatively young age often puts them at a disadvantage when breadth of existing support code is an important criterion.

The typical solution to this dilemma is to use a native interface such as JNI [**?**] or CNI [**?**] to invoke unsafe code from within a virtual machine or otherwise safe environment. Unfortunately, there are a number of situations in which this is not an acceptable solution. These situations can be broadly classified into two categories: *security concerns* and *portability concerns*.

Security is often a major concern when integrating native code. Using Java as an example, JNI and CNI are prohibited in a number of contexts, including applet environments and servlet containers with a `SecurityManager`. Additionally, even in the context of trusted code, `native` methods invoked via JNI are susceptible to buffer overflow and heap corruption attacks which are not a concern for verified, bounds-checked bytecode.

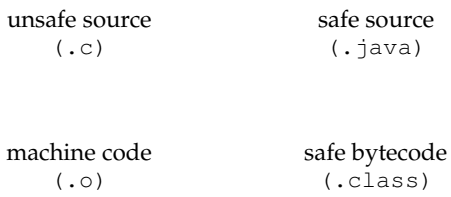
The second class of disadvantages revolves around portability concerns; native interfaces require the native library to be compiled ahead of time for every architecture on which it will be deployed. This is unacceptable for scenarios in which the full set of target architectures is not known at deployment time. Additionally, some JVM platform variants such as J2ME [**?**] simply do not offer support for native code.

The technique we present here uses typical compiler to compile unsafe code into a MIPS binary, which is then translated on an instruction-by-instruction basis into Java bytecode. The technique presented here is general; we anticipate that it can be applied to other secure virtual machines such as Microsoft's .NET [**?**], Perl Parrot [**?**], or Python bytecode [**?**].

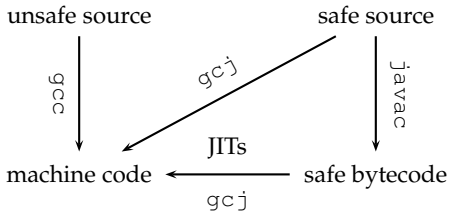
The remainder of this paper is divided as follows: in the next section we review the relevant set of program representations (safe source, unsafe source, binary, and bytecode) and survey existing work for performing transformations between them. In the third section we introduce NestedVM and cover its two primary translation modes in detail. Section four describes the NestedVM runtime, which plays the role of the OS kernel. Section five addresses the optimizations we employ and quantifies NestedVM's performance. Section six reviews our experiences in applying NestedVM to various popular software packages. We conclude with an analysis of NestedVM's weaknesses and potential for future improvements.

2 Existing Work

The four program representations of interest in this context, along with their specific types in the C-to-JVM instantiation of the problem are shown in the following diagram:



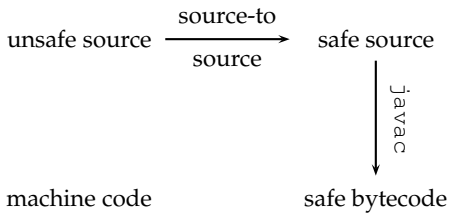
To illustrate the context of this diagram, the following arcs show the translations performed by a few familiar tools:



Existing techniques for translating unsafe code into VM bytecode generally fall into two categories, which we expand upon in the remainder of this section: source-to-source translation and source-to-binary translation.

2.1 Source-to-Source Translation

The most common technique employed is partial translation from unsafe source code to safe source code:



A number of existing systems employ this technique; they can be divided into two categories: those which perform a partial translation which is completed by a human, and those which perform a total translation but fail (yield an error) on a large class of input programs.

2.1.1 Human-Assisted Translation

Jazillian [**?**] is a commercial solution which produces extremely readable Java source code from C source code, but only translates a small portion of the C language. Jazillian is unique in that in addition to *language migration*, it also performs *API migration*; for example, Jazillian is intelligent enough to translate "`char* s1 = strcpy(s2)`" into "`String s1 = s2`". Unfortunately such deep analysis is intractable for most of the C language and standard library; indeed, Jazillian's documentation notes that *"This is not your father's language translator... Jazillian does not always produce code that works correctly."*

MoHCA-Java [**?**] is the other major tool in this category, and steps beyond Jazillian by providing tools for analysis of the source C++ abstract syntax tree. Additionally, MoHCA-Java's analysis engine is extensible, making it a platform for constructing application-specific translators rather than a single translation tool. However, MoHCA-Java does not always generate complete Java code for all of the C++ programs which it accepts.

2.1.2 Partial-Domain Translation

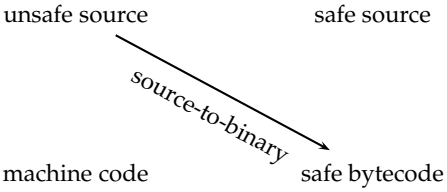
The `c2j` [**?**], `c2j++`, Cappucinno [**?**], and Ephedra [**?**] systems each provide support for complete translation of a *subset* of the source language (C or C++). Each of the four tools supports a progressively greater subset than the one preceding it; however none covers the entire input language.

Ephedra, the most advanced of the four, supports most of the C++ language, and claims to produce "human readable" Java code as output. Notable omissions from the input domain include support for fully general pointer arithmetic, casting between unrelated types, and reading from a `union` via a different member than the one most recently written.

Unfortunately, when the program being translated is large and complex, it is quite likely that it will use an unsupported feature in at least one place. In the absence of a programmer who understands the source program, a single anomaly is often enough to render the entire translation process useless. As a result, these tools are mainly useful as an *aid* to programmers who could normally perform the conversion themselves, but want to save time by automating most of the process.

2.2 Source-to-Binary Translation

Source-to-binary translation involves a compiler for the unsafe language which has been modified to emit safe bytecode.



An experimental "JVM backend" for the `gcc` compiler, known as `egcs-jvm` [**?**], attempts this approach. Since `gcc` employs a highly modular architecture, it *is* possible to add RTL code generators for nonstandard processors. However, `gcc`'s parsing, RTL generation, and optimization layers make fundamental assumptions (such as the availability of pointer math) which cannot be directly supported; thus the compiler still fails for a substantial class of input programs.

A Java backend for the `lcc` compiler [**?**], known as `lcc-java`, is also available. Although this system is quite clean and elegantly designed, it lacks any form of system library (`libc`), so very few C programs will run without custom modification (which would cause them to diverge from the upstream sources). The memory model employed by `lcc-java` is also somewhat awkward; a separate `int[]` is maintained for the stack and heap, leading to difficulties mingling pointers to these two memory regions. Additionally, the heap is a single `int[]`, which means that it must be copied in order to be expanded, and prevents `lcc-java` from taking advantage of `NullPointerException` checking, which costs nothing in the "common case" since nearly all JVMs use the host CPU's MMU to detect this condition.

3 NestedVM

The principal difference between NestedVM and other approaches is that NestedVM *does not* attempt to deal with source code as an input, instead opting for *binary-to-source* and *binary-to-binary* translation. This offers three immediate advantages:

- **Language Agnosticism**

  Because NestedVM does not attempt to implement the parsing and code generation steps of compilation, it is freed from the extremely complex task of faithfully implementing languages which are often not fully or formally specified (such as C and C++), and is able to support any language for which a MIPS-targeted compiler exists.

- **Bug-for-bug compiler compatibility**

  Since NestedVM uses the compiler's *output* as its own *input*, it ensures that programs which are inadvertently dependent on the vagaries of a particular compiler can still be used.

- **No build process modifications**

  NestedVM does not modify existing build processes, which can be extremely complex and dependent on strange preprocessor usage as well as the complex interplay between compiler switches and header file locations.

NestedVM's approach carries a fourth benefit as well, arising from its totality:

- **No post-translation human intervention**

  NestedVM offers total support for all non-privileged instructions, registers, and resources found on a MIPS `R2000` CPU, including the add/multiply unit and floating point coprocessor. As such, it constitutes a total function mapping from the entire domain of (non-kernel-mode) programs onto (a subset of) the semantics of the Java Virtual Machine. This ensures that the translation process is fully automated and always succeeds for valid input binaries.

This last point has important software engineering implications. If post-translation human intervention is required, then the *human becomes part of the build process*. This means that if a third party library used in the project needs to be upgraded, *a human must intervene* in the rebuild process. In addition to slowing the process and introducing opportunities for error, this task often requires specialized knowledge which becomes tied to the particular individual performing this task, rather than being encoded in build scripts which persist throughout the lifetime of the project.

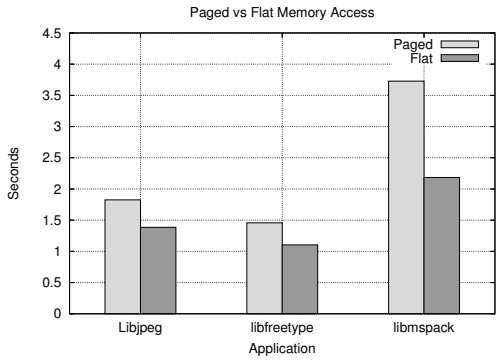
3.1 Mapping the R2000 onto the JVM

We chose MIPS as a source format for three reasons: the availability of tools to compile legacy code into MIPS binaries, the many similarities between the MIPS ISA and the Java Virtual Machine, and the relatively high degree of program structure that can be inferred from ABI-adherent binaries.

The MIPS architecture has been around for quite some time, and is well supported by the GNU Compiler Collection, which is capable of compiling C, C++, Java, Fortran, Pascal, and Objective C into MIPS binaries.

The MIPS R2000 ISA bears many similarities to the Java Virtual Machine. Most of the instructions in the original MIPS ISA operate only on 32-bit aligned memory locations. This allows NestedVM to represent memory as a Java `int[][]` array indexed by page (the top n bits of the address) and offset (the remaining bits) without introducing additional overhead. MIPS's non-aligned memory load instructions are only rarely emitted by most compilers since they carry a performance penalty on physical MIPS implementations.

Our choice of a paged representation for memory carries only a small performance disadvantage:



Additionally, this representation lets us to take advantage of the fact that on most JVM's, checking for

a `NullPointerException` carries no performance penalty unless the exception is thrown (the host CPU's MMU is generally used to detect this condition). This allows us to lazily expand the MIPS memory space as it is used. Additionally, we maintain two page arrays, one which is used for read operations and another for writes. Most of the time these page arrays will have identical entries; however, we can simulate a portion of the MIPS MMU functionality by setting the appropriate entry in the write page table to `null`, thereby write protecting the page while still allowing reads.

Unlike its predecessor, the R2000 supports 32-bit by 32-bit multiply and divide instructions as well as a single and double precision floating point unit. These capabilities map nicely onto Java's arithmetic instructions and `int`, `long`, `float`, and `double` types.

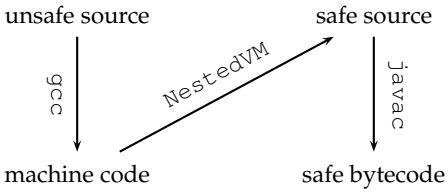
Although MIPS offers unsigned arithmetic and Java does not, few MIPS instructions actually depend on non-two's-complement handling of integer math. In the few situations where these instructions *are* encountered, the `unsigned int` is cast (bitwise) to a Java `long`, the operation is performed, and the result is cast back. On host architectures offering 64-bit arithmetic this operation carries no performance penalty.

In addition to its similarities to the JVM, the MIPS ISA and ABI convey quite a bit of information about program structure. This information can be used for optimization purposes:

- The structure of MIPS branching and jump instructions make it easy to infer the set of likely target instructions.
- The MIPS ABI specifies particular registers as caller-save and callee-save, as well as designating a register for the return address after a function call. This allows NestedVM to optimize many operations for the common case of ABI-adherent binaries.
- All MIPS instructions are exactly 32 bits long.

3.2 Binary-to-Source Mode

The simplest operational mode for NestedVM is binary-to-source translation. In this mode, NestedVM translates MIPS binaries into Java source code, which is then fed to a Java compiler in order to produce bytecode files:

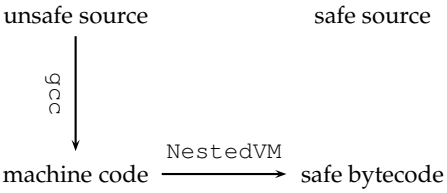


Translating unsafe code for use within a JVM proceeds as follows:

1. The unsafe source code is compiled to a statically linked binary, including any libraries (including `libc`) it needs.
2. `NestedVM` is invoked on the statically linked binary, and emits a `.java` file.
3. The resulting `.java` code is compiled into a `.class` file using `jikes` or `javac`.
4. At runtime, the host Java code invokes the `run()` method on the generated class. This is equivalent to the `main()` entry point.

3.3 Binary-to-Binary Mode

After implementing the binary-to-source compiler, a binary-to-binary translation mode was added.



This mode has several advantages:

- There are quite a few interesting bytecode sequences that cannot be generated as a result of compiling Java source code.
- Directly generating `.class` files Eliminates the time-consuming `javac` step.
- Direct compilation to `.class` files opens up the interesting possibility of dynamically translating MIPS binaries and loading them via `ClassLoader.fromBytes()` *at deployment time*, eliminating the need to compile binaries ahead of time.

4 The NestedVM Runtime

The NestedVM runtime fills the role typically assumed by an OS Kernel. Communication between MIPS code and the runtime is mediated by the `SYSCALL` instruction, just as the `libc`-kernel interface is on other MIPS implementations.

Two implementations of the runtime are offered; a simple runtime with the minimum support required to comply with ANSI C, and a more sophisticated runtime which emulates a large portion of the POSIX API.

4.1 The ANSI C Runtime

The ANSI C runtime offers typical file I/O operations including `open()`, `close()`, `read()`, `write()`, and `seek()`. File descriptors are implemented much as

```
private final static int r0 = 0;
private int r1, r2, r3, /* ... */ r30;
private int r31 = 0xdeadbeef;
private int pc = ENTRY_POINT;

public void run() {
    while (true)
        switch(pc) {
            case 0x10000: r29 = r29 - 32;
            case 0x10004: r1 = r4 + r5;
            case 0x10008: if (r1 == r6) {
                            /* delay slot */
                            r1 = r1 + 1;
                            pc = 0x10018:
                            continue; }
            case 0x1000C: r1 = r1 + 1;
            case 0x10010: r31 = 0x10018;
                          pc = 0x10210;
                          continue;
            case 0x10014: /* nop */
            case 0x10018: pc = r31; continue;
            ...
            case 0xdeadbeef:  System.exit(1);
...
```

```
public void run_0x10000() {
    while (true) switch(pc) {
        case 0x10000: ...
        case 0x10004: ...
        case 0x10010: r31 = 0x10018;
                      pc = 0x10210;
                      continue;
....

pubic void run_0x10200() {
    while (true) switch(pc) {
        case 0x10200: ...
        case 0x10204: ...
....

public void trampoline() {
    while (true) switch(pc&0xfffffe00) {
        case 0x10000:    run_0x10000(); break;
        case 0x10200:    run_0x10200(); break;
        case 0xdeadbe00: ...
....
```

Figure 1: Trampoline transformation necessitated by Java's 64kb method size limit

they are in OS kernels; a table of open files is maintained and descriptors act as an index into that table. Each file descriptor is represented as a Java `RandomAccessFile` in order to match the semantics of `seek()`.

Process-level memory management is done through the `sbrk()` system call, which extends the process heap by adding more pages to the memory page table. Fast memory clear and copy operations can be performed with `memset()` and `memcpy()`, which invoke the Java `System.arraycopy()` method.

The `exit()` call records the process' exit status, marks the VM instance as terminated and halts execution. The `pause()` syscall implements a crude form of Java-MIPS communication by returning control to the Java code which spawned the MIPS process.

4.2 The Unix Runtime

The Unix runtime extends the simple ANSI C file I/O model to include a unified-root filesystem, device nodes, and `fcntl()` APIs. Device nodes are generally simulated by mapping reads, writes, and `fcntl()`s on the device to the appropriate Java API.

MIPS processes can "mount" other filesystems within the virtual filesystem made visible to the MIPS process. Each filesystem is implemented by a Java class, which could, for example, offer access to the host filesystem (including `state()`, `lstat()`, `mkdir`, and `unlink()`, and `getdents()`), the contents of a zip archive, or even a remote HTTP server.

The `fork()` call is implemented in an elegant manner by calling the Java `clone()` method (deep copy) on the VM object itself. The new instance is then added to a static process table to facilitate interprocess communication.

The `exec()` method actually loads a MIPS binary image from the filesystem, feeds it to the MIPS-to-bytecode translator, and then loads the resulting bytecode on the fly using `ClassLoader.loadBytes()`.

The `waitpid()` API allows a parent process to block pending the completion of a child process, which is modeled quite easily with the Java `wait()` method. The `pipe()` system call permits parent-to-child IPC just as on a normal Unix system.

Simple networking support is provided by the `opensocket()`, `listensocket()`, and `accept()` methods, which are not yet fully compatible with the standard Berkeley sockets API.

4.3 Security Concerns

NestedVM processes are completely isolated from the outside world except for the `SYSCALL` instruction. As previously mentioned, the programmer can choose from various runtime implementations which translate these invocations into operations in the outside world. By default, none of these implementations allows file or network I/O; this must be explicitly enabled, typically on a per-file basis.

Wild writes within the MIPS VM have no effect on the larger JVM or the host OS; they can only cause the `SYSCALL` instruction to be invoked. A judicious choice of which system calls to enable offers extremely strong security; for example, the `libjpeg` library does not need any host services whatsoever – its sole task is to decompress one image (which is pre-written into memory before it starts up), write the decompressed image to another part of memory, and exit. With all system calls disabled, `libjpeg` will function correctly, and even if it is compromised (for example by a maliciously-constructed invalid image), the only effect it can have on the host is to generate an incorrect decompressed image.

4.4 Threading

The NestedVM runtime currently does not support threading. Providing robust support for "true threads", whereby each MIPS thread maps to a Java thread is probably not possible as the Java Memory Model [**?**], since all MIPS memory is stored in a set of `int[]`'s and the Java Memory Model does not permit varying treatment or coherency policies at the granularity of a single array element.

While this presents a major barrier for applications that use sophisticated locking schemes such as *hash synchronization* and depend on atomic memory operations, it is probably possible to apply this threading model to "well behaved" multithreaded applications which make no concurrency assumptions other than those explicitly offered by OS-provided semaphores and mutexes.

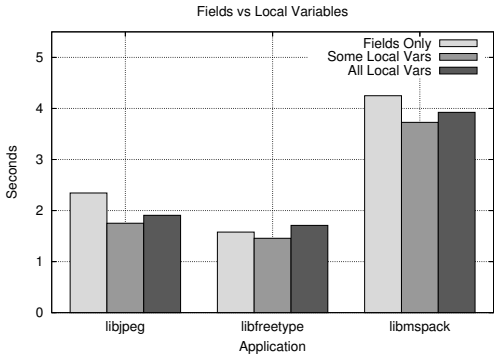
Complex synchronization and incorrectly synchronized applications can be supported by implementing a variant of *user threads* within a single Java thread by providing a timer interrupt (via a Java asynchronous exception). Unfortunately this requires that the compiled binary be able to restart at any arbitrary instruction address, which would require a `case` statement for every instruction (rather than every jump target), which would degrade performance and increase the size of the resulting class file.

5 Optimization and Performance

5.1 Binary-to-Source Mode

Generating Java source code instead of bytecode frees NestedVM from having to perform simple constant propagation optimizations, as most Java compilers already do this. A recurring example is the treatment of the `r0` register, which is fixed as `0` in the MIPS ISA.

Lacking the ability to generate specially optimized bytecode sequences, a straightforward mapping of the general purpose hardware registers to 32 `int` fields turned out to be optimal. Using local variables for registers did not offer much of a performance advantage, presumably since the JVM's JIT is intelligent enough to register-allocate temporaries for fields.



Unfortunately, Java imposes a 64kb limit on the size of the bytecode for a single method. This presents a problem for NestedVM, and necessitates a *trampoline transformation*, as shown in Figure **??**. With this trampoline in place, large binaries can be handled without much difficulty – fortunately, there is no corresponding limit on the size of a classfile as a whole.

One difficulty that arose as a result of using the trampoline transformation was the fact that `javac` and `jikes` are unable to properly optimize its switch statements. For example, the following code is compiled into a comparatively inefficient `LOOKUPSWITCH`:

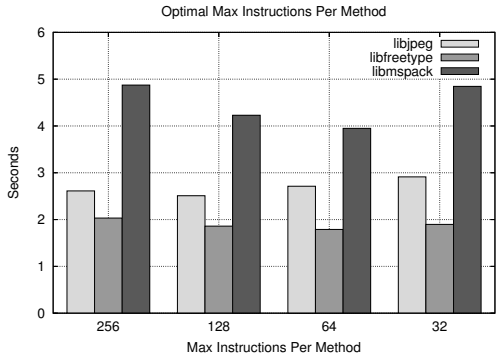
```
switch(pc&0xffffff00) {
    case 0x00000100: run_100(); break;
    case 0x00000200: run_200(); break;
    case 0x00000300: run_300(); break;
}
```

Whereas the next block of code code optimized into a `TABLESWITCH`:

```
switch(pc>>>8) {
    case 0x1: run_100();
    case 0x2: run_200();
    case 0x3: run_300();
}
```

This problem was surmounted by switching on a denser set of `case` values, which is more amenable to the `TABLESWITCH` structure. This change alone nearly doubled the speed of the compiled binary.

The next performance improvement came from tuning the size of the methods invoked from the trampoline. Trial and error led to the conclusion that HotSpot [**?**] – the most widely deployed JVM – performs best when 128 MIPS instructions are mapped to each method.



This phenomenon is due to two factors:

- While the trampoline method's `switch` statement can be coded as a `TABLESWITCH`, the `switch` statement within the individual methods is to sparse to encode this way.
- Hybrid Interpretive-JIT compilers such as HotSpot generally favor smaller methods since they are easier to compile and are better candidates for compilation in "normal" programs (unlike NestedVM programs).

After tuning method sizes, our next performance boost came from eliminating extraneous case branches, which yielded approximately a 10%-25% performance

improvement. Having case statements before each instruction prevents JIT compilers from being able to optimize across instruction boundaries, since control flow can enter the body of a `switch` statement at any of the `case`s. In order to eliminate unnecessary case statements we needed to identify all possible jump targets. Jump targets can come from three sources:

- **The `.text` segment**

  Every instruction in the text segment is scanned, and every branch instruction's destination is added to the list of possible branch targets. In addition, the address[1] of any function that sets the link register is added to the list. Finally, combinations of `LUI` (Load Upper Immediate) and `ADDIU` (Add Immediate Unsigned) are scanned for possible addresses in the `.text` segment since this combination of instructions is often used to load a 32-bit word into a register.

- **The `.data` segment**

  When compiling `switch` statements, compilers often use a jump table stored in the `.data` segment. Unfortunately they typically do not identify these jump tables in any way. Therefore, the entire `.data` segment is conservatively scanned for possible addresses in the `.text` segment.

- **The symbol table**

  The symbol table is used primarily as a backup source of jump targets. Scanning the `.text` and `.data` segments should identify any possible jump targets; however, adding all function symbols in the ELF symbol table also catches functions that are never called directly from the MIPS binary, such as those invoked only via the NestedVM runtime's `call()` method.

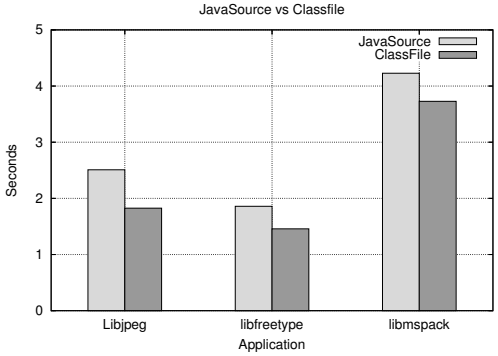
Despite all the above optimizations, one insurmountable obstacle remained: the Java `.class` file format limits the constant pool to 65535 entries. Every integer literal greater than `32767` requires an entry in this pool, and each branch instruction generates one of these.

One suboptimal solution was to express constants as offsets from a few central values; for example "`pc = N_0x00010000 + 0x10`" (where `N_0x000100000` is a non-final field to prevent `javac` from inlining it). This was sufficient to get reasonably large binaries to compile, and caused only a small (approximately 5%) performance degradation and a similarly small increase in the size of the `.class` file. However, as we will see in the next section, compiling directly to `.class` files (without the intermediate `.java` file) eliminates this problem entirely.

5.2 Binary-to-Binary Mode

Compiling directly to bytecode offers a substantial performance gain:

[1] actually `addr+8`



Most of this improvement comes from the handling of branch instructions and from taking advantage of the JVM stack to eliminate unnecessary `LOAD`s and `STORE`s to local variables.

The first optimization gained by direct bytecode generation came from the use of the JVM `GOTO` instruction. Despite the fact that the Java *language* does not have a `goto` keyword, the VM does in fact have a corresponding instruction which is used quite heavily by `javac`. NestedVM's binary-to-binary mode exploits this instruction to avoid emitting inefficient `switch..case` structures.

Related to the `GOTO` instruction is branch statement optimization. When emitting source code, NestedVM translates branches into Java source code like this:

```
if (condition) {
    pc = TARGET;
    continue;
}
```

This requires a branch in the JVM *regardless* of whether the MIPS branch is actually taken. If `condition` is false the JVM has to jump over the code to set `pc` and go back to the `switch` statement; if `condition` is true the JVM has to jump to the `switch` block. By generating bytecode directly, NestedVM is able to emit a JVM bytecode branching directly to the address corresponding to the target of the MIPS branch. In the case where the branch is not taken the JVM doesn't branch at all.

A side effect of the previous two optimizations is a solution to the excess constant pool entries problem. When jumps are implemented as `GOTO`s and branches are taken directly, the `pc` field does not need to be set. This eliminates a huge number of constant pool entries. The `.class` file constant pool size limit is still present, but it is less likely to be encountered.

Implementation of the MIPS delay slot offers another opportunity for bytecode-level optimization. In order to take advantage of instructions already in the pipeline, the MIPS ISA specifies that the instruction after a jump or branch is always executed, even if the jump/branch is taken. This instruction is referred to as the "delay slot[2]." The instruction in the delay slot is actually executed *before* the branch is taken. To fur-

[2] Newer MIPS CPUs have pipelines that are much larger than early MIPS CPUs, so they have to discard instructions anyways

ther complicate matters, values from the register file are loaded *before* the delay slot is executed.

Fortunately there is a very elegant solution to this problem when directly emitting bytecode. When a branch instruction is encountered, the registers needed for the comparison are pushed onto the stack to prepare for the JVM branch instruction. Then, *after* the values are on the stack the delay slot instruction is emitted, followed by the actual JVM branch instruction. Because the values were pushed to the stack before the delay slot was executed, any changes the delay slot instruction makes to the registers are not visible to the branch bytecode.

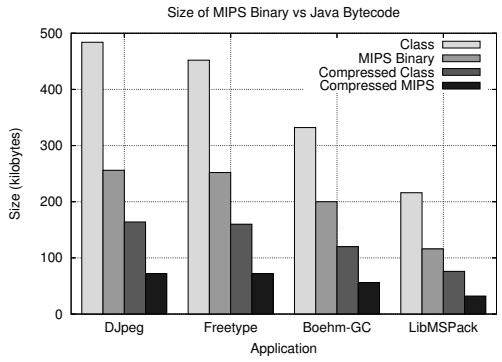
One final advantage of emitting bytecode is a reduction in the size of the ultimate `.class` file. All the optimizations above lead to more compact bytecode as a beneficial side effect in addition, NestedVM performs a few additional size optimizations.

When encountering the following `switch` block, both `javac` and `jikes` generate redundant bytecode.

```
switch(pc>>>8) {
    case 0x1: run_1(); break;
    case 0x2: run_2(); break
    ...
    case 0x100: run_100(); break;
}
```

The first bytecode in each case arm in the switch statement is `ALOAD_0` to prepare for a `INVOKESPECIAL` call. By simply lifting this bytecode outside of the `switch` statement, each `case` arm shrinks by one instruction.

The net result is quite reasonably sized `.class` files:



5.3 Compiler Flags

Although NestedVM perfectly emulates a MIPS R2000 CPU, its performance profile is nothing like that of actual silicon. In particular, `gcc` makes several optimizations that increase performance on an actually MIPS CPU but actually decrease the performance of NestedVM-generated bytecode. We found the following compiler options generally improve performance when using `gcc` as the source-to-MIPS compiler:

- `-falign-functions`

  Normally a function's location in memory has no effect on its execution speed. However, in the NestedVM binary translator, the `.text` segment is split on power-of-two boundaries due to the trampoline. If a function starts near the end of one of these boundaries, a performance critical part of the function winds up spanning two Java methods. Telling `gcc` to align all functions along these boundaries decreases the chance of this sort of splitting.

- `-fno-rename-registers`

  On an actual silicon chip, using additional registers carries no performance penalty (as long as none are spilled to the stack). However, when generating bytecode, using *fewer* "registers" helps the JVM optimize the machine code it generates by simplifying the constraints it needs to deal with. Disabling register renaming has this effect.

- `-fno-schedule-insns`

  Results of MIPS load operations are not available until *two* instructions after the load. Without the `-fno-schedule-insns` instruction, `gcc` will attempt to reorder instructions to do other useful work during this period of unavailability. NestedVM is under no such constraint, and removing this reordering typically results in simpler bytecode.

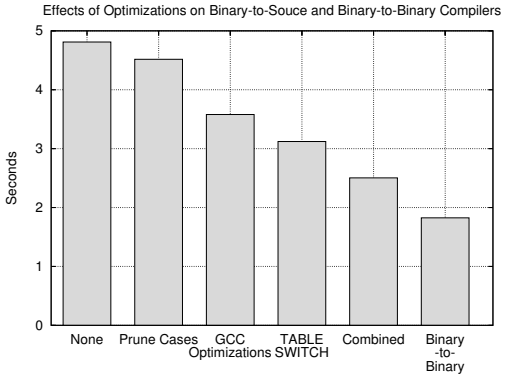
- `-mmemcpy`

  Enabling this instruction causes `gcc` to use the system `memcpy()` routine instead of generating loads and stores. As explained in the previous section, the NestedVM runtime implements `memcpy()` using `System.arraycopy()`, which is substantially more efficient.

- `-ffunction-sections -fdata-sections`

  These two options are used in conjunction with the `--gc-sections` linker option, prompting the linker to more aggressively prune dead code.

The following chart quantifies the performance gain conferred by each of the major optimizations outlined in this section:

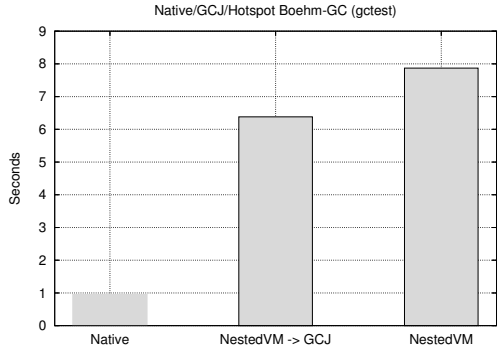


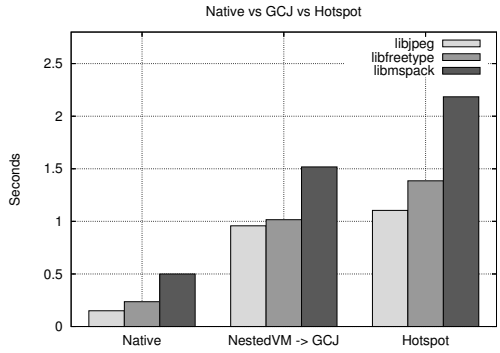
5.4 Overall Performance

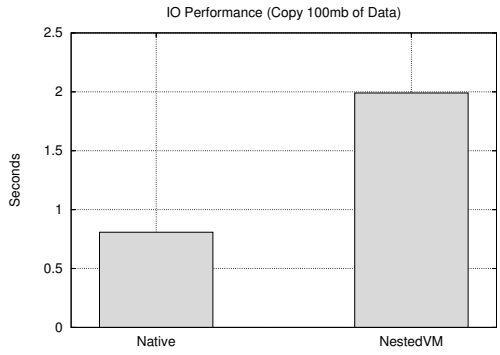
All times are measured in seconds. All tests were performed on a dual 1Ghz Macintosh G4 running Apple's

latest JVM (Sun HotSpot JDK 1.4.1). Each test was run 8 times within a single VM. The highest and lowest times were removed and the remaining 6 were averaged. In each case only the first run differed significantly from the rest.

The `libjpeg` test consisted of decoding a 1280x1024 jpeg and writing a tga. The `mspack` test consisted of extracting all members from `arial32.exe`, `comic32.exe`, `times32.exe`, and `verdan32.exe`. The `libfreetype` test consisted of rendering ASCII characters 32-127 of `Comic.TTF` at sizes from 8 to 48 incrementing by 4 for a total of 950 glyphs.







6 Sample Applications

6.1 FreeType, `libmspack`, and `libjpeg`

The Ibex Project utilizes three libraries for which no Java-only equivalent exists. The first is the FreeType font library, which parses, hints, and rasterizes TrueType and Postscript fonts with exceptional quality. The project also needed an open source JPEG decompresser; surprisingly, none exist for Java. While encoders are plentiful, decoders are rare, since Sun's J2SE VM includes a native method to invoke `libjpeg`.

These three libraries make minimal use of the standard library and OS services, and are all written in very portable ANSI C code, which made them easy targets for initial development.

6.2 The GNU Compiler Collection

Our next target, `gcc`, was initially chosen in order to relieve developers from the time-consuming and complex task of building a compiler themselves; The Ibex Project requires a specially configured and patched version of `gcc` and its ahead-of-time Java compiler (`gcj`) which is cumbersome to build.

`Gcc` was the first "major" application NestedVM was used on, and drove the development of most of the system library interface development; particularly support for `fork()` and `exec()`, which require the NestedVM Runtime to perform binary-to-bytecode translation on the fly.

`Gcc` also makes extensive use of 64-bit integers (`long long`), which – for performance reasons – are typically manipulated using nonobvious instruction sequences on the 32-bit MIPS architecture. Dealing with these operations uncovered a number of bugs in the translator.

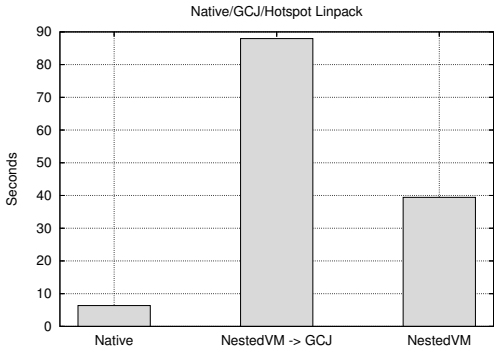
Despite our original goal, we have not yet been able to translate the `C++` or Java front-ends, as the resulting binary produces a trampoline which exceeds the maximum size of a single method. Future work will explore a multi-level trampoline to address this issue.

6.3 TEXand LINPACK

In order to distinguish NestedVM from other single-language translators for the JVM, we undertook the task of translating TEX (written in Pascal) and the Fortran source code for LINPACK into Java bytecodes.

Although actually producing the initial MIPS binaries from the TEX source code turned out to be an exceptionally tedious and frustrating task, the resulting binary translated and executed perfectly on the first run, as did LINPACK. Our reward for this effort was typesetting our presentation of NestedVM using NestedVM itself. We have also had initial successes running TEX in a Java Applet, and intend to produce a `jar` for embedding TEX code ("TEXlets") in web pages without the use of a post-processing tool.

The LINPACK benchmark called our attention to Java's lack of an API for checking the "cpu time" of a process. Unfortunately we had to substitute wall-clock time on an otherwise-quiescent machine as an approximation.



7 Conclusion

We have presented a novel technique for using libraries written in unsafe languages within a safe virtual machine without resorting to native interfaces. We have implemented this technique in NestedVM and demonstrated its utility by translating six popular software applications.

7.1 Future Directions

Although we have only implemented it for the Java Virtual Machine, our technique generalizes to other safe bytecode architectures. In particular we would like to demonstrate this generality by re-targeting the translator to the Microsoft Intermediate Language [**?**].

Additionally, we would like to explore other uses for dynamic loading of translated MIPS binaries by combining NestedVM (which itself is written in Java) and the `ClassLoader.defineClass()` mechanism.

7.2 Availability

NestedVM is available under an open source license, and can be obtained from

`http://nestedvm.ibex.org`