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

Abstract

Most existing techniques for using code written in an unsafe language within a safe virtual machine involve transformations from one source code language (such as C) to another (such as Java) and then to virtual machine bytecodes. We present an alternative approach which uses a standard compiler to turn unsafe source code into unsafe MIPS binaries, which are then translated into safe virtual machine bytecodes. This approach offers four key advantages over existing techniques:

- Total coverage of all language features
- No post-translation human intervention
- No build process modifications
- Bug-for-bug compiler compatability

We have implemented this technique in NestedVM, a binary-to-source and binary-to-binary translator targeting the Java Virtual Machine. NestedVM-translated versions of the `libfreetype`, `libjpeg`, and `libmspack` libraries are currently in production use. 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*.

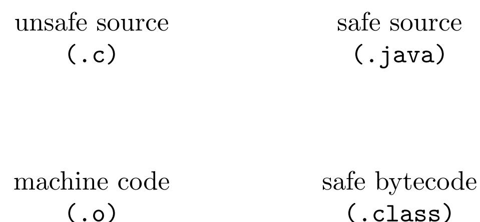
Using Java as an example, JNI and CNI are prohibited in a number of contexts, including applets environments and servlet containers with a `SecurityManager`. Additionally, even in the context of trusted code, `native` methods invoked via JNI are susceptible to buffer overflow and heap corruption attacks which are not a concern for verified 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 they will be deployed. This is unworkable for situations 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.

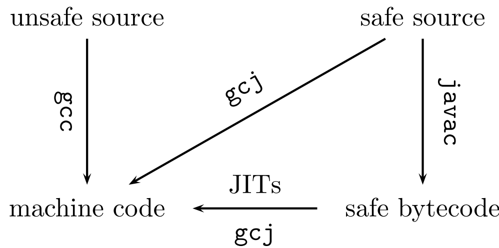
The technique we present here uses typical compiler to compile unsafe code into a MIPS binary, which is then translated on an instruction-by-instruction basis into Java bytecode. The technique presented here is general; we anticipate that it can be applied to other secure virtual machines such as Microsoft's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

2 Approaches to Translation

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:



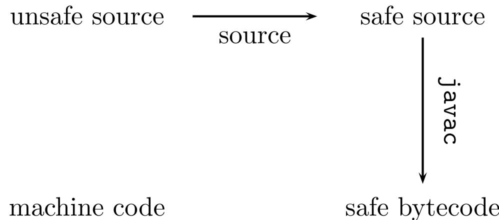
Techniques for translating unsafe code into VM bytecode generally fall into four categories, which we expand upon in the next two sections:

- source-to-source translation
- source-to-binary translation
- binary-to-source translation
- binary-to-binary translation

3 Existing Work

3.1 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.

3.1.1 Incomplete Translation

Jazillian [?] is a commercial solution which produces extremely readable Java source code from C source code, but ony 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 intractible for most of the C language and standard library; Jazillian's documentation notes that *"This is not your father's language translator. It's not generating ugly code that's guaranteed to work out of the box... 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.

3.1.2 Partial Domain Translation

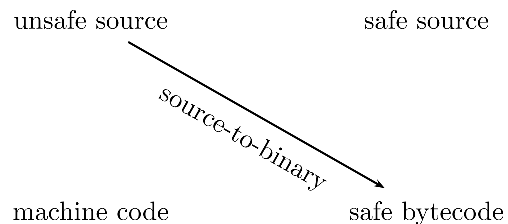
The c2j [?], c2j++ [?], Cappucinno [?], and Ephedra [?] systems each provide support for complete translation of a *subset* of the source language (C or C++). Each of the four tools supports a progressively greater subset than the one preceding it; however none covers the entire input language.

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 anomoly 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.

3.2 Source-to-Binary Translation

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



The primary occupant of this category is `egcs-jvm` [?], an experimental "JVM backend" for the GNU Compiler Collection (`gcc`) [?]. Since `gcc` employs a highlym odular 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.

4 NestedVM

The principal difference between NestedVM and other approaches is that NestedVM *does not* attempt to deal with source code as an input. This leads immediately to three advantages:

- **Total coverage of all language features**

  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++).

- **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.

- **Bug-for-bug compiler compatability**

  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.

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 is a much more important factor than is obvious at first glance. 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.

4.1 Why MIPS?

We chose MIPS as a source format for three reasons: the availability of tools to compile legacy code into MIPS binaries, the close similarity 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 a striking similarity to the Java Virtual Machine:

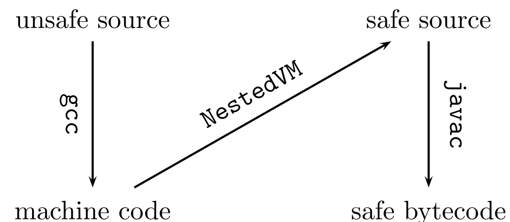
- Most of the instructions in the original MIPS ISA operate only on 32-bit aligned memory locations. This allows NestedVM to represent memory as a Java `int[]` array without introducing additional overhead. The remaining non-aligned memory load instructions are only rarely emitted by most compilers since they carry a performance penalty on physical MIPS implementations.
- 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.

Finally, 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.

4.2 Binary-to-Source

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:



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

public void run() {
    for(;;) {
        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.err.println(``Exited.'');
                System.exit(1);
        }
    }
}
```

```
public void run_0x10000() {
    for(;;) {
    switch(pc) {
        case 0x10000:
            ...
        case 0x10004:
            ...
        ...
        case 0x10010:
            r31 = 0x10018;
            pc = 0x10210;
            return;
        ...
    }
    }
}

pubic void run_0x10200() {
    for(;;) {
    switch(pc) {
        case 0x10200:
            ...
        case 0x10204:
            ...
    }
    }
}

public void trampoline() {
    for(;;) {
    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

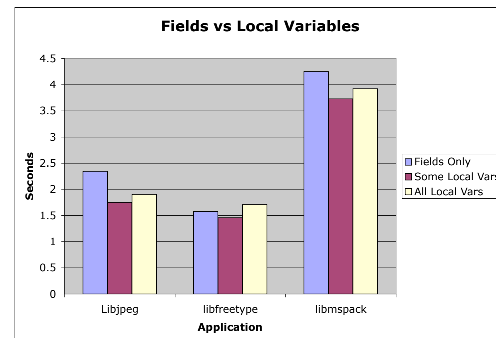
Translating unsafe code for use within a JVM proceeds as follows:

1. Compile the source code to a statically linked binary, targeting the MIPS R2000 ISA. Typically this will involve linking against `libc`, which translates system requests (such as `open()`, `read()`, or `write()`) into appropriate invocations of the MIPS `SYSCALL` instruction.
2. Invoke `NestedVM` on the statically linked binary.
3. Compile the resulting `.java` code using `jikes` [?] or `javac`.
4. From java code, invoke the `run()` method on the generated class. This is equivalent to the `main()` entry point.

4.2.1 Optimizations

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.



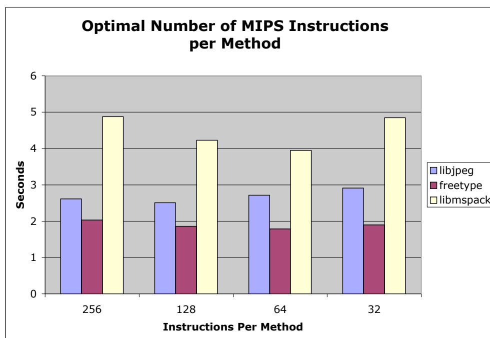
Unfortunately, Java imposes a 64kb limit on the size of the bytecode for a single method. This presents a problem for NestedVM, and necessitates a *trampoline transformation*, as shown in Figure 1. With this trampoline in place, large binaries can be handled without much difficulty – fortunately, there is no corresponding limit on the size of a classfile as a whole.

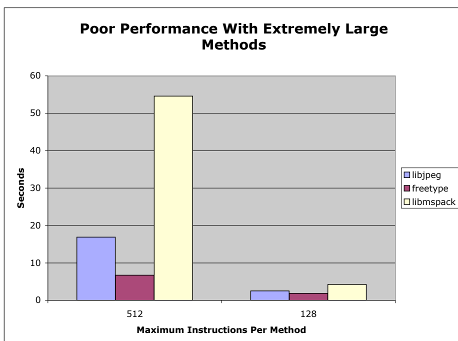
One difficulty that arose as a result of using the trampoline transformation was the fact that `javac` and `jikes` are unable to properly optimize its switch statements. For example, the following code is compiled into a comparatively inefficient `LOOKUPSWITCH`:

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

Putting more than 256 instructions in each method lead to a severe performance penalty. Apparently Hotspot does not handle very large methods well. In some tests the simple moving from 256 to 512 instructions per method decreased performance by a factor of 10.

Put chart here

The next big optimization was eliminating unnecessary case statements. Having case statements before each instruction prevents JIT compilers from being able to optimize across instruction boundaries. In order to eliminate unnecessary case statements every possible address that could be jumped to directly needed to be identified. The sources for possible jump targets come from 3 places.

- **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, any function that sets the link register is added to the list [1]. 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**

  This is mainly used as a backup. 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.

Eliminating unnecessary `case` statements provided a 10-25% speed increase.

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

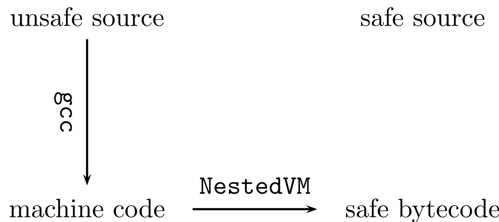
[1] actually `addr+8`

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 degredation 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.

4.3 Binary-to-Binary

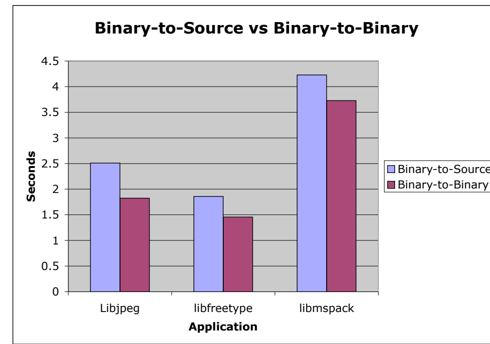
After implementing the binary-to-source compiler, a binary-to-binary translation mode was added.



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.

Most of the performance improvemen where made where in the handling of branch instructions and in 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` statemenmt; 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 further complicate matters, values from the register file are loaded *before* the delay slot is executed.

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

Fortunately there is a very elegent solution to this problem which can be expressed in JVM 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 made to the registers are not visible to the branch bytecode.

One final advantage that generating bytecode directly allows 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 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.

4.3.1 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 could be used to improve performance:

- `-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. 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, so removing this reordering typically generates simpler machine code.

- `-mmemcpy`

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

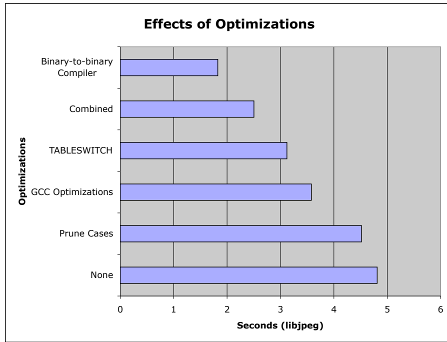
  NestedVM has two primary ways of executing code, the interpreter, and the binary translators. Both the interpreter and the output from the binary translators sit on top of a Runtime class. This class provides the public interface to both the interpreter and the translated binaries.

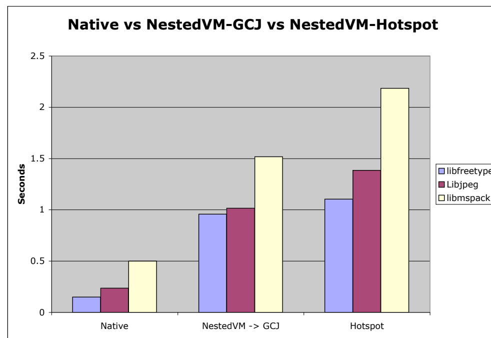
  The Runtime class does the work that the operating system usually does. Conceptually the Runtime class can be thought of as the operating system and its subclasses (translated binaries and the interpreter) the CPU. The Runtime fulfills 5 primary goals:

- `-fno-rename-registers` Some processors can better schedule code when registers aren't reused for two different purposes. By default GCC will try to use as many registers as possibly when it can. This excess use of registers just confuses JIT's trying to compile the output from the binary translator. All the JIT compilers we tested do much better with a few frequently used registers.

- `-fno-delayed-branch` The MIPS CPU has a delay slot (see above). Earlier versions of NestedVM didn't efficiently emulate delay slots. This option causes GCC to avoid using delay slots for anything (a NOP is simply placed in the delay slot). This had a small performance benefit. However, recent versions of NestedVM emulate delay slots with no performance overhead so this options has little effect. Nonetheless, these delay slots provide no benefit under NestedVM either so they are avoided with this option.

The effects of the various optimizations presented in this chapter are summarized in the table below.





5 The NestedVM Runtime

In addition to binary-to-source and binary-to-binary translation, NestedVM also includes a MIPS binary interpreter. All three translation approaches expose the same API to both the translated binary and the surrounding VM (including peer Java code).

5.1 The Runtime Class

The runtime fulfills four roles:

- It provides a simple, consistent external interface. The method of actually executing the code (currently only translated binaries and the interpreter) can be changed without any code changes to the caller because only runtime exposes a public interface. This includes methods to pass arguments to the binary's `main()` function, read and write from memory, and call individual functions in the binary.
- It manages the process's memory. The runtime class contains large `int[]` arrays that represent the process's entire memory space. Subclasses read and write to these arrays as required by the instructions they are executing, and can expand their memory space using the `sbrk` system call.
- The runtime provides access to the host file system and standard I/O streams. Subclasses of `runtime` can access the file system through standard UNIX syscalls (`read()`, `write()`, `open()`, etc). The runtime manages the file descriptor table that maps UNIX file descriptors to Java `RandomAccessFiles`, `InputStreams`, `OutputStreams`, and `Sockets`.
- It provides general OS services, including `sleep()`, `gettimeofday()`, `getpagesize()`, `sysconf()`, `fcntl()`, and so on.

6 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 retargeting 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.fromBytes()` mechanism.

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, which is currently used by the Ibex project[3] to perform font rasterization (via `libfreetype`), JPEG decoding (via `libjpeg`), and CAB archive extraction (via `libmspack`), three libraries for which no equivalent Java classes exist.

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

`http://nestedvm.ibex.org`

8 Appendix: Testing Methodology

The MIPS binaries can also invoke a special method of Runtime called callJava().When the MIPS binary invokes the `CALL_JAVA` syscall (usually done through the `_call_java()` function provided by the NestedVM support library) the callJava() method in Runtime is invoked with the arguments passes to the syscall.

```
// Java
private Runtime rt = new MyBinary() {
    pubilc int callJava(int a, int b, int c, int d) {
        System.err.println("Got " + a + " " + b);
    }
};
public void foo() { rt.run(); }

/* C */
```

[3] `http://www.ibex.org`

```
void main(int argc, char **argv) {
    _call_java(1,2);
}
```

These two methods can even be combined. MIPS can call Java through the CALL_JAVA syscall, which can in turn invoke a MIPS function in the binary with the call() method.

Users preferring a simpler communication mechanism can also use Java Streams and file descriptors. Runtime provides a simple interface for mapping a Java Input or OutputStream to a File Descriptor.

Users preferring a simpler communication mechanism can also use Java Stream's and file descriptors. Runtime provides a simple interface for mapping a Java Input or OutputStream to a File Descriptor.

9 Future Directions

10 Conclusion

11 Appendix A: Testing Environment

All times are measured in seconds. These were all run 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 (thebride_1280.jpg) and writing a tga. The mspack test consisted of extracting all members from arial32.exe, comic32.exe, times32.exe, and verdan32.exe. The freetype test consisted of rendering characters 32-127 of Comic.TTF at sizes from 8 to 48 incrementing by 4. (That is about 950 individual glyphs).

12 References