Efficient Cross-Platform Method Dispatching for Interpreted Languages

Andrew Weinrich

2007-12-18

1 Introduction

Most work on modern compilers is still targeted towards lower-level languages, such as C and C++. However, a great deal of modern programming effort takes place in very high-level languages like Python, JavaScript, and Ruby that sacrifice execution time for increased flexibility and expressiveness. An interesting point of intersection is scripting languages that are designed to work in tandem with more traditional, compiled languages. For example, there are several languages like F# that are implemented on Microsoft's CLR; programs written in F# can use objects developed in the more conventional C#.

Objective-C is the language of choice for development on Mac OS X. Unlike Java and C#, Objective-C is compiled to actual machine code, but it retains a very flexible runtime that includes the ability to create new classes and methods on the fly. These abilities are exploited by some parts of Apple's frameworks, but they also open the door to creating new scripting languages that use the Objective-C runtime as a platform. Such languages can provide different programming styles while making use of the large body of existing Objective-C code.

F-Script is one such language; it brings Smalltalk syntax and APL-style array processing to Cocoa (the name of Apple's programming environment). The FSClass module allows new classes to be written directly in F-Script; an example is given in Figure 1. Instead of a compiled function, the method implementation is provided by a Block, an object that contains an expression tree and can be evaluated as a closure. Instance variable access is also provided through methods.

The FSClass module must provide implementations of these methods to the Objective-C runtime. Property accessors must be able to find the variable inside the object structure, and method implementations must be able to locate the correct Block, evaluate it, and return the result. This paper describes how FSClass performs these tasks efficiently, while taking into account the multi-platform requirements of Mac OS X.

2 Adding Methods and Classes to the Objective-C Runtime

Objective-C has a low-overhead runtime system implemented in C. Creating new classes and adding methods to those classes is accomplished by creating C structures and using special functions to register them with the runtime. When adding a method, a function pointer of type `IMP` is used. This typedef is variadic to support methods with different numbers of parameters, but method implementations must always take two special implicit parameters. The first is an `id` that holds the receiver of the message, usually called `self` or `this`. (`id` is a pointer to an object of any class). The second is the *selector*: a special `char*` that is registered with the runtime and holds the name of the message. This allows methods to discover their own names, which will be crucial for our naive method implementations. The definitions of these types are reproduced in Figure 2.

Different typedefs and methods are used when methods that return doubles or `structs`, but they are irrelevant to our purposes.

To properly implement new classes, FSClass must provide these function pointers to the Objective-C runtime. We will first present naive versions of these methods, then discuss the shortcomings that lead to a more complicated but efficient implementation.

2.1 Naive Implementation of Method Stubs

The inclusion of the method selector as an implicit parameter suggests a very simple way of implementing methods. There will be a single function, `dispatch_method`, that will be the clearinghouse for all method calls in FSClass. This function will compute the object's class, then look in

```
MyClass := FSClass newClassWithProperties:
  {'foo', 'bar'}.

MyClass onMessage:#doStuff do:[ :self :val |
    ((self foo) + (self bar)) / val
].
```

Figure 1: Sample class created in F-Script.

```
typedef void* id;   // generic object pointer
typedef char* SEL;  // method name

// pointer to a method implementation
typedef id
  (IMP*)(id receiver, SEL selector, ...);
```

Figure 2: Data types used by the Objective-C runtime.

a dictionary to find the Block that implements the given method. If the class contains no Block for that method name, the dispatcher will move up the inheritance hierarchy, until it either finds the appropriate Block or reaches the top and raises an exception.

This approach is very simple and easy to implement, but horribly inefficient. It reproduces a similar search through the inheritance chain that the Objective-C runtime already performs, and does so much more slowly. To reduce the overhead to an acceptable level, we will need to create dedicated stub functions for every method.

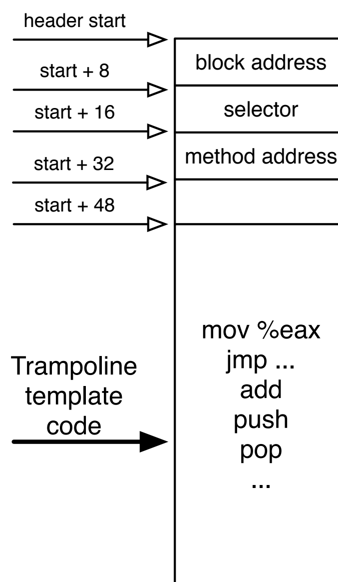

Figure 3: Structure of a method-dispatching trampoline.

3 Trampolines as Alternative

As described above, the one-size-fits-all naive implementations suffer from significant performance problems. Because the runtime requires C function pointers as implementations, the only alternative is to create new blocks of compiled code as methods are created. Such dynamically-created pieces of code to perform redirection are usually called *trampolines*. This leaves us with two choices: perform a full just-in-time compilation from some intermediate form, or use a template that is modified slightly for each new trampoline.

The better choice by far is to stamp out custom trampolines from a template. Each trampoline will have three variable values: a pointer to the implementation Block, the proper selector, and the address of the Block's evaluation method. The most obvious way to write these trampolines is to include these variables as inlined arguments to immediate load instructions. This approach has pitfalls, however, including problems those caused by the necessity of FSClass to run on multiple architectures.

3.1 Multiplatform Concerns

In 2005, Apple announced that it was abandoning the PowerPC platform for Intel-supplied x86. Although Apple has no plans for returning to PowerPC, there is an immense installed base of PowerPC Macintoshes, and it will be quite some time before Apple can drop support for the architecture. The problem is compounded by the introduction of 64-bit support in the most recent release of OS X for both PowerPC and x86. Since we want to build native-code methods on the fly, we will need to have separate code bodies for each of the four architectures, wrapped in `#ifdef` tests to only include the proper code at runtime.

Another problem is that, especially on 64-bit architectures, many immediate values cannot be loaded in one chunk; instead, they must be split into pieces that are moved into a register by separate instructions. 64-bit PowerPC requires 5 instructions to load a 64-bit immediate value. The code to insert all these large words into the appropriate places involves a great deal of bit-masking shifting that is difficult to write. Additionally, if we ever change the source code for the trampoline - say by reorganizing instructions to reduce stack use - we will have to update the corresponding code that inserts those values.

3.2 Ideal Trampoline Builder

The ideal trampoline-generating system would have the *exact same* C code for each system; the only `#ifdef`'d components will be the content of the trampoline itself. In an ideal system like this, the trampoline code could be changed or optimized without affecting the C construction code, and the same construction code could be used on all platforms.

We can realize this ideal by putting all of the variable data in a header in the front of the trampoline. In this system, the trampoline code will not be modified by the template at all, but a new copy will be made each time a trampoline is created. Figure 3 shows the structure of a trampoline with header, including the address that will be given to the runtime system.

This presents the problem of how the trampoline is to access the data in the header. Encoding the address of the header into the trampoline won't work, as that simply reproduces the inlined-constant problem. We can take advantage of the fact that the header always has the same *relative* position to the code that needs it. If the trampoline code could discover its own location at runtime, it could calculate the location of the header, and load the appropriate pointers directly from there.

The remainder of this paper describes the methods for detecting the runtime location of trampolines; extracting information from the headers; and manipulating the stack before calling the implementing Block.

4 32-bit PowerPC Method Trampolines

One interesting feature of the PowerPC instruction set is that it does not distinguish between jumps and calls, as most processors do. Instead, there is a set of "branch" instructions which perform both functions. Branches can be absolute or conditional, and can be to a 16-bit local offset or an absolute address in a register. Most importantly for our purposes, a bit can be set on a branch instruction that will tell the compiler to leave the return address in a special-purpose "link register". Beyond having return addresses deposited in it, the link register can be manipulated by special instructions to move its contents to or from another general-purpose register, or to branch directly to the address it contains. The former are used to save and restore return addresses to the stack, and the latter takes the place of the `return` instruction found on x86.

The following code will discover its own location at runtime:

```
trampoline:
  ; save return address
  mflr r0

  xor.    r16, r16, r16
  bnel    trampoline
  mflr    r16
  subi    r16, r16, 28
  mtlr    r0
```

The instructions in this trampoline header perform the following steps:

1. Temporarily save the contents of the link register. At the beginning of the procedure, it contains the return address of the caller.

2. xor general-purpose register `r16` with itself. The choice of this register is arbitrary. The `.` on the end of the opcode tells the assembler to set a special bit in the instruction. This bit will cause the processor to set special flags on the result of the operation, including whether the result was zero or non-zero/

3. `bnel` stand for "Branch if Not Equal to zero and Link". This branch will execute only if the result of the previous operation was non-zero. Because we XORed a register with itself, we know that the result was zero, and hence that this conditional branch will not execute. However, in either case, the instruction will leave the return address - the address of the next instruction - in the link register.

4. Extract the "return address" from the link register. `r16` now holds the address of the fourth instruction in the trampoline.

5. All PowerPC instructions are 32 bits, so the value of `r16` is 12 bytes from the label `trampoline:`. The trampoline header is 16 bytes, so by subtracting 28, we have the

6. Now that our use of the link register has ended, we restore the caller's return address to the link register.

4.1 Argument Manipulation

Like most other RISC architectures, PowerPC has 32 general-purpose registers, making it feasible to use them for passing parameters. On Mac OS X, the first seven integer parameters are passed in registers 3 through 10; floating point and vector parameters are passed in other registers, but since F-Script uses only object pointers as parameters, we may ignore this complication.

The Block that is implementing our method requires different arguments than those that are passed to the trampoline. In particular, the Block itself must be the receiver, and the selector must be changed from its original value to a message name that will invoke the Block's evaluation. The original receiver is the first parameter to the Block; any original parameters to the method must be moved back one slot. The Block address and selector value are in the trampoline header, and must be extracted and placed in the appropriate headers.

Figure 4 shows the parameter registers on PowerPC at the point of trampoline entrance and exit. This reconfiguration is trivial to accomplish with register-based parameter passing. The following code will move the parameters into the appropriate places, and load the needed data from the trampoline header:

```
; move parameters back one space
mr r8, r7   ; third param
mr r7, r6   ; second param
mr r6, r5   ; first param
```

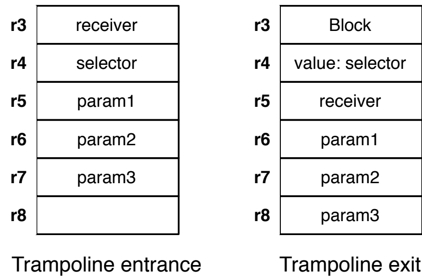

Figure 4: Register contents for PowerPC trampoline.

```
mr r5, r3   ; receiver

lwz r3, 0(r16)  ; load target Block
lwz r4, 8(r16)  ; load selector
```

The selector is located eight bytes from the beginning of the header to allow for 64-bit pointers. On 32-bit architectures, the extra space will be unused.

4.2 Final Trampoline Structure

The final step of the trampoline is to evaluate the Block by invoking the appropriate method (which is different for different Block arities). After rearranging the parameters, we have effectively changed the call stack to look like what the Block expects. We can then load the address of the Trampoline's method implementation and directly jump to it, without setting up a new stack frame. From the Block's perspective, it will have been called directly from the trampoline's caller.

The complete procedure for the trampoline is:

1. Identify its location
2. Shuffle arguments into their proper locations
3. Pull required information from the trampoline
4. Jump to the Block evaluation function

Figure 13 shows the completed PowerPC trampoline code for a 3-argument method. Trampolines for methods of other arities would be identical, save more or fewer parameter-swapping instructions.

5 32-bit x86 Method Trampolines

5.1 Self-location for x86

Unlike PowerPC, x86 does not use an explicit link register, but it does have an explicit stack. The `CALL` instruction leaves the return address on the top of stack, but jumps do not. This makes using the PPC approach impossible, because although x86 does have conditional branches, it does not have a conditional `CALL`.

However, x86 does have a *relative `CALL`* that can be used. The relative `CALL` takes an inlined 4-byte offset from the location of the next instruction after the call itself. This will allow the trampoline to "call" to a location in itself, and then take the "return address" from the stack. With this knowledge, we can use the following code to calculate the location of the trampoline at runtime:

```
trampoline:
    call next_line
next_line:
    pop ecx
    sub ecx, 21
```

The assembler will turn the first instruction into a relative `CALL` with offset 0. When the code executes, the call will have no noticeable effect other than advancing to the next instruction and leaving the return address on the stack. The return address will of course be the address of the next instruction. After popping it from the stack, we now have the location of the second instruction of the trampoline stored in `ecx`. A relative `CALL` is 5 bytes long: one byte for the opcode, and 4 bytes for the offset. Combined with the 16 bytes for the header, this gives an offset of 21. After subtracting that from the call's return address, we have the address of the trampoline header in `ecx`.

5.2 Manipulating the x86 Stack

Due to its paucity of registers, all subroutines in x86 must pass their arguments on the stack. Like most downward-growing stacks, arguments for Darwin function calls are pushed right-to-left. This would appear to make the trampoline simpler: instead of having to shuffle all the parameters, we only have to move up the return address and insert the selector and Block pointer from the trampoline header:

As shown, the arguments do not have to be changed at all. However, this approach would cause catastrophic errors at runtime. We have just increased the size of the stack, with the intention of transferring control to another method. Since we want to jump directly to the implementation without returning, the caller of the trampoline will have to clean up the stack. But the caller is unaware that we have pushed an additional element, and in its stack cleanup code after the call to `objc_msgSend` will not remove it. Due to this problem, the x86 trampoline cannot simply hand off control to the implementing Block as the PowerPC version did; instead it will need to repair the stack after the Block has returned, before itself returning to the caller.

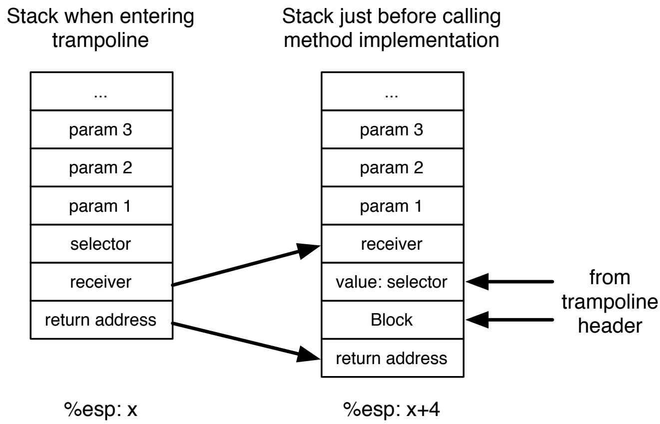

Figure 5: Possible stack for x86 trampoline.

This restriction produces the following general structure for the x86 trampoline:

1. Calculate the current trampoline offset
2. Rearrange the top of the stack, as shown in the (diagram 1)
3. Call the Block implementation
4. Remove the top value from the stack
5. Return to the caller

The stack frame that the caller created for the trampoline may also be used when calling the Block's implementation, so the arguments will not have to duplicated on the stack. Additionally, since the arguments are not being moved, we need only write one version of the trampoline that can be used to implement methods of all arities.

The above procedure appear sound, but in fact a surprising error occurs when it is used: the processor raises a "`MOVNTPS alignment exception`"! As described in the Intel IA32 reference, `MOVNTPS` is an MMX©instructions; it is used to move 128 bits of data in a single step while avoiding cache pollution. As with most SIMD instructions, the address must be aligned to a 128-bit/16-byte boundary.

When reporting the above error, `gdb` claims stack corruption and does not provide a backtrace. This makes it very likely that the error is related to the trampoline, which `gdb` cannot tie back to any source file. However, there are no SIMD instructions in our code at all!

A clue about what is happening comes from Apple's ABI documentation for x86:

The caller ensures that the stack is 16-byte aligned at the point of the function call.

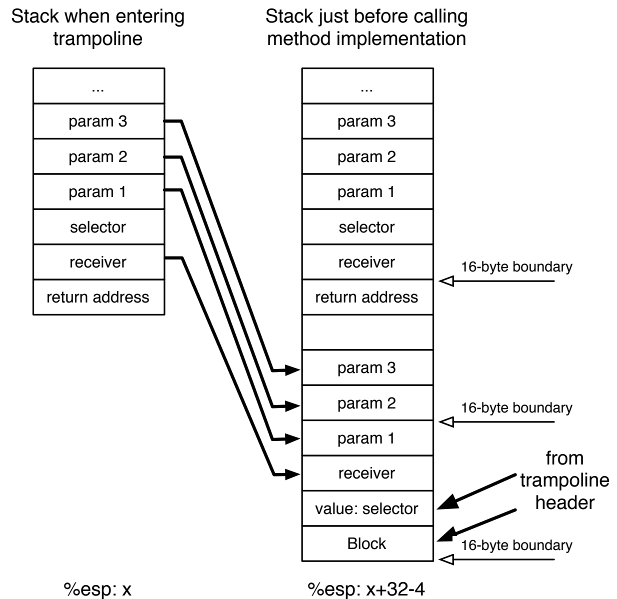

Figure 6: Final stack configuration for x86 trampoline.

"At the point of the function call" means just before the `CALL` instruction is executed, the top of the stack. No further explanation is given, but an inference can be made: the Darwin kernel is using SIMD instructions to quickly move data on the stack, possibly during context switches. We have violated this alignment by pushing an extra word onto the stack, which causes the SIMD instruction to fail at an unpredictable time.

Due to this alignment requirement, we will in fact be forced to create a new stack frame for use with calling the Block; because these size of these frames will be different for different arities, we will need several different trampolines. Figure 4 shows how the new stack frame will be created for the three-argument x86 trampoline. As shown, due to the alignment, some space on the stack will be unused. Figure 14 contains the assembly code for the three-argument x86 trampoline.

5.3 64-bit PowerPC Trampolines

The PowerPC instruction set was designed with future expansion to 64 bits in mind. When running in 64-bit, all instructions are identical, although some care must be taken when using 32-bit words. The function ABI is also identical, and parameters are passed in the same registers as they are for 32-bit mode. The only changes necessary to the 32-bit trampolines are to double the size of the header and the offsets into it. Figure 15 shows the barely-modified code for the three-argument trampoline on 64-bit PowerPC.

5.4 64-bit x86 Trampolines

Although the first x86 processors Apple used were 32-bit, by 2006 they had introduced computers that used Intel processors with AMD's x86-64 extensions. In the fall of 2007, the release of Mac OS X version 10.5 enables 64-bit applications. Currently, F-Script is only available as a 32-bit framework, but it is worth considering how the eventual move to 64-bit will be made.

The x86-64 instruction set fixes many of the problems that had plagued x86 for decades. The most important for compiler engineers is the introduction of 8 more general purpose registers, making it feasible. This makes passing arguments in registers feasible for most functions, including the sorts used for trampolines.

AMD maintains a reference ABI for C and C++ on x86-64, based on System V Unix; Apple completely defers to this model instead of designing their own. On this ABI, the first six integer parameters to a function are passed in %rdi, %rsi, %rdx, %rcx, %r8 and %r9. Unlike PowerPC, the x86-64 ABI has mixed register-stack usage for parameters; the six registers are always used first, but additional parameters will be passed on the stack.

To keep the assembly from becoming too complicated, FSClass only implements trampolines for methods with three or fewer parameters. Figure 7 shows the configuration of the registers when entering and exiting the trampoline, and Figure 16 shows the assembly code for the 3-parameter trampoline.

Register	Trampoline entrance
%rdi	receiver
%rsi	selector
%rdx	param1
%rcx	param2
%r8	param3
%r9	

Register	Trampoline exit
%rdi	Block
%rsi	value: selector
%rdx	receiver
%rcx	param1
%r8	param2
%r9	param3

Figure 7: Register use for x86-64 trampoline.

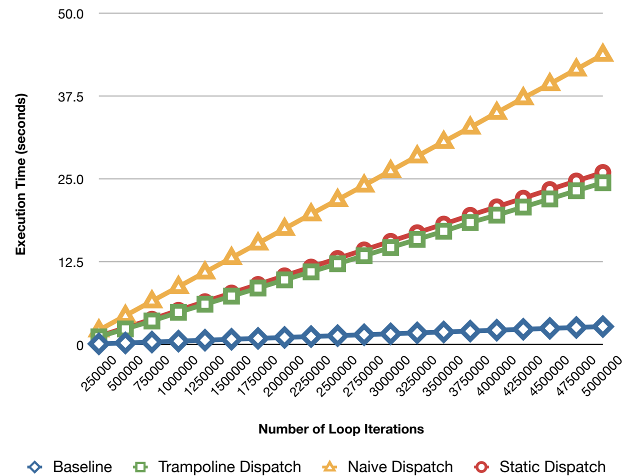

Figure 8: Performance.

6 Analysis and Conclusions

6.1 Trampoline Performance

To evaluate the performance of trampoline as a method-dispatching system, the program in Figure 9 was run with versions of FSClass that had trampoline dispatching and naive dispatching. For comparison, run times for an empty loop (Figure 11), and for a program that uses an equivalent of static dispatch (Figure 10) are included. These programs create a class `MyClass` with a single method that returns nil , then create an instance `m` of that class. The programs then execute a loop for a certain number of iterations (`upperBound`, passed as a command-line argument).

Figure 8 shows the performance of the naive, trampoline, and static dispatch methods. It is apparent that using trampolines enormously reduces the overhead for method calls; the savings is close to 100%.

Interesting, although static dispatch was assumed to be the fastest possible performance for executing the body of the method, using trampolines is actually slightly faster. This is likely due to implementation details of the F-Script interpreter; looking up the function referred to in the loop body appears to be slightly slower than sending a message directly to the object.

6.2 Impact of Using Trampolines

Although the overhead reduction of trampolines is considerable, the overall benefits to an F-Script program are much more modest. Most F-Script objects are implemented in Objective-C, which already has an extremely efficient method-dispatch system based on hashing. These calls to compiled code make up the large bulk of method calls in an F-Script program. Additionally, while removing the overhead of calling a method is important, the large bulk of method execution time is going to be in evaluating the Block's syntax tree. It is possible to compile F-Script to bytecode - its parent language, Smalltalk, had one of the earliest virtual machines - but at the moment, the interpretation costs far outweigh the dispatching overhead.

Currently, there is not a sufficiently large body of code that uses classes written in F-Script to determine the real-world impact of using trampolines in this manner. If the language and tools become more popular, it may become possible to measure the actual performance benefits.

6.3 Applicability to Other Languages

The trampolines described in this paper benefit from the language design of F-Script: every object has exactly one class, and that class exactly determines the capabilities of that object. This model is the same shared by Java and C++ (to the extent that multiple inheritance is not used). Many scripting languages, however, have a more flexible object model, largely derived from Self. Both Ruby and JavaScript treat individual objects as dictionaries, whose methods are closures attached to individual properties of the object. Any object can add or remove methods to itself, with the class being used as a fallback.

This makes using trampolines at the class level impossible, as when an object is sent a message, the interpreter must inspect the object's dictionary to see if it individually overrides the class's method definition. Depending on the design of the interpreter, trampolines may not be useful or may not offer enough performance benefits to justify its additional complexity. More research is needed to determine if the approaches described above are profitable in other scripting languages.

7 References

1. *Intel Architecture Software Developers Manual.* Intel, 2000
2. *Mac OS X ABI Function Call Guide.* Apple Computer, 2007
3. *PowerPC Microprocessor Family.* IBM, 2000
4. *Mach-O Programming Topics.* Apple Computer, 2007
5. *Objective-C Runtime Reference.* Apple Computer, 2007
6. *PC Assembly Language.* Paul A Carter, 2006
7. *PowerPC / OS X (Darwin) Shellcode Assembly.* B-r00t, 2003
8. *System V Application Binary Interface.* AMD, 2006

8 Evaluation Code Listings

```
upperBound := (args at:0) intValue.

MyClass := FSClass newClass:'MyClass'.

MyClass onMessage:#doStuff:withThing:
do:[ :self :stuff :thing |
    nil.
].

m := MyClass alloc init autorelease.

1 to:upperBound do:[
    m doStuff:5 withThing:3.
].
```

Figure 9: Test program for method dispatching

```
upperBound := (args at:0) intValue.

MyClass := FSClass newClass:'MyClass'.

method := [ :self :stuff :thing |
    nil.
].

m := MyClass alloc init autorelease.

1 to:upperBound do:[
    method value:m value:5 value:3.
].
```

Figure 10: Static dispatch test program

```
upperBound := (args at:0) intValue.

MyClass := FSClass newClass:'MyClass'.

MyClass onMessage:#doStuff:withThing:
do:[ :self :stuff :thing |
    nil.
].

m := MyClass alloc init autorelease.

1 to:upperBound do:[
    nil.
].
```

Figure 11: Baseline test program

9 Trampoline Code Listings

```
// gets the property that has the same name as the selector
id _getProperty(id thisId, SEL selector) {
    NSMutableDictionary* properties;
    object_getInstanceVariable(thisId, IVAR_PROP_DICTIONARY, (void**)(&properties));
    NSString* propName = NSStringFromSelector(selector);

    id object = [properties objectForKey:propName];

    // if this object doesn't exist in the properties dictionary, get the class's default value
    if (object==nil) {
        FSClass* currentClass;
        object_getInstanceVariable(thisId, IVAR_FS_CLASS, (void**)(&currentClass));
        id defaultObject = [currentClass defaultValueForProperty:propName];

        // set the default value and return it
        [properties setObject:(defaultObject==nil ? [NSNull null] : defaultObject) forKey:propName];
        return defaultObject;
    }
    else if (object == [NSNull null]) {
        return nil; // substitute out an NSNull for an actual nil
    }
    else {
        return object;
    }
}
```

Figure 12: Naive property-access method.

```
trampoline3:
    mflr r0   ; save return address

    xor.    r16, r16, r16   ; r16 = 0
    bnel    _main           ; branch to _main if not equal
    mflr    r16             ; r16 = main + 12
    subi    r16, r16, 28    ; r16 = main - 16 = beginning of trampoline

    ; move parameters back one space
    mr r8, r7   ; third param
    mr r7, r6   ; second param
    mr r6, r5   ; first param
    mr r5, r3   ; receiver

    lwz r3, 0(r16)  ; load target Block into r3
    lwz r4, 4(r16)  ; load selector to r4

    ; extract objc_msgSend from thunkoline and call it
    mtlr    r0          ; return address
    lwz     r15, 8(r16)  ; load objc_msgSend to r15
    xor     r16,r16,r16
    mtctr   r15         ; copy objc_msgSend to count register
    bctr                ; jump to objc_msgSend, do not update link register
```

Figure 13: Three-argument general purpose trampoline for ppc.

```
;;; Three-argument trampoline
trampoline3:
    call next_line3  ; (5) actual function pointer start
next_line3:
    pop ecx     ; put address of current line into ecx
    sub ecx, 21 ; [header length (16) + offset to line 'pop ecx' (5)]

    ; create new stack frame - 16-byte aligned, so more space than required
    add esp, (32-4)     ; the stack currently has the return address on the other
                        ; side of the boundary, so leave space for it

    mov edx, [esp+48]   ; move parameter 3
    mov [esp+20], edx

    mov edx, [esp+44]   ; move parameter 2
    mov [esp+16], edx

    mov edx, [esp+40]   ; move parameter 1
    mov [esp+12], edx

    mov edx, [esp+28]   ; move receiver
    mov [esp+8], edx

    mov edx, [ecx]      ; load impl block to first stack argument
    mov [esp+4], edx
    mov edx, [ecx+4]    ; load value: selector to second stack argument
    mov [esp], edx

    mov edx, [ecx+8]    ; load address of objc_msgSend and call the Block
    call edx            ; return value is now in eax

    sub esp, 28         ; clean up stack frame

    return
;;; end Three-argument trampoline
```

Figure 14: Three-argument general purpose trampoline for x86.

```
trampoline3:
    mflr r0   ; save return address

    xor.    r16, r16, r16   ; r16 = 0
    bnel    _main           ; branch to _main if not equal
    mflr    r16             ; r16 = main + 12
    subi    r16, r16, 44    ; r16 = main - 32 = beginning of trampoline

    ; move parameters back one space
    mr r8, r7   ; third param
    mr r7, r6   ; second param
    mr r6, r5   ; first param
    mr r5, r3   ; receiver

    ld r3, 0(r16)  ; load target Block into r3
    ld r4, 8(r16)  ; load selector to r4

    ; extract objc_msgSend from thunkoline and call it
    mtlr    r0          ; return address
    ld      r15, 16(r16)  ; load objc_msgSend to r15
    xor     r16,r16,r16
    mtctr   r15         ; copy objc_msgSend to count register
    bctr                ; jump to objc_msgSend, do not update link register
```

Figure 15: Three-argument general purpose trampoline for ppc64.

```
trampoline3:
    call next_line3  ; (5) actual function pointer start
next_line3:
    pop r10     ; put address of current line into r10
    sub r10, 21 ; [header length (16) + offset to line 'pop ecx' (5)]

    mov r8, r9          ; move parameter 3
    mov rcx, r8         ; move parameter 2
    mov rdx, rcx        ; move parameter 1
    mov rdi, rdx        ; mov receiver
    mov [r10], rdi      ; load impl block to stack argument
    mov [r10+8], rsi    ; load value: selector to stack argument

    mov r11, [r10+8]    ; load address of objc_msgSend and jump
    jmp r11
```

Figure 16: Three-argument general purpose trampoline for x86-64.