Article

Addressing architectural differences in your macOS code

Fix problems that stem from architectural differences between Apple silicon and Intel-based Mac computers.

Overview

Assumptions you make for your code on Intel-based Mac computers may not always apply on Apple silicon. Architectural differences between the two systems exist, and may cause your code to run incorrectly or crash. Use the following tips to identify potential problem areas in your code before you begin testing.

Fetch System and Hardware Details Dynamically

Code that depends on specific system details or hardware configurations may crash or yield unexpected behaviors on Apple silicon. Many hardware features are different on Apple silicon and Intel-based Mac computers; some system features may also be different. If you hardcode a particular value for a feature, your code may not work as expected on systems where the value is different.

Instead of hardcoding values related to the underlying system, fetch those values dynamically from system global variables whenever possible. For example, fetch the size of virtual memory pages from the `vm_page_size` global variable. When a global variable isn't available, use the `sysctl` or `sysctlbyname` functions to fetch the information instead.

Some features of Apple silicon are decidedly different than those of Intel-based Mac computers, and may impact your code if you don't fetch them dynamically. These features include:

- Virtual memory page sizes are different. Fetch the value from the `vm_page_size` global variable.

- Cache line sizes are different. Fetch the `hw.cachelinesize` setting using `sysctl`.
- CPU-specific features. Fetch the feature availability using sysctl and a string of the form `hw.optional.`*<feature>*, where *<feature>* is the feature you want. For example, to determine if AVX512 instructions are available, use the `hw.optional.avx512f` string.

To see the list of available hardware features, run the command `sysctl hw` command in Terminal. For other system features, run `sysctl` with a different domain string, such as `kern`, `user`, or `machdep`.

Synchronize Access to Shared Data in Memory

Always protect shared data with locks, memory barriers, and other synchronization primitives present in macOS. A strong memory-ordering model, like the one in Intel-based Mac computers, adds implicit memory barriers to prevent the processor from reordering load and store instructions in a way that might introduce race conditions. A weak memory ordering model, like the one in Apple silicon, gives the processor more flexibility to reorder memory instructions and improve performance, but doesn't add implicit memory barriers. To ensure the correctness of your code on both platforms, add explicit synchronization primitives to your code.

macOS includes numerous synchronization primitives:

- Grand Central Dispatch (GCD) provides serial queues and other ways to synchronize tasks; see Dispatch.
- The `@synchronized` directive creates a mutex lock for Objective-C code.
- The Foundation framework defines standard mutexes, conditions, and other types of locks.
- The os framework provides "unfair" locks for synchronization.
- The pthreads library defines standard mutexes and condition variables.
- The C11/C++11 primitives in `stdatomic.h` support custom memory ordering in atomic operations.

If you use lockless algorithms or custom synchronization techniques, consider replacing them with system-provided primitives. If you're not able to adopt the system primitives, validate the correctness of your custom code on Apple silicon before deploying it in your binary.

> **Note**
>
> Use the thread sanitizer to detect data races and identify places where your code requires synchronization. For more information, see Diagnosing memory, thread, and crash issues early.

Don’t Redeclare a Function to Have Variable Parameters

The `x86_64` and `arm64` architectures have different calling conventions for variadic functions—functions with a variable number of parameters. On `x86_64`, the compiler treats fixed and variadic parameters the same, placing parameters in registers first and only using the stack when no more registers are available. On `arm64`, the compiler always places variadic parameters on the stack, regardless of whether registers are available. If you implement a function with fixed parameters, but redeclare it with variadic parameters, the mismatch causes unexpected behavior at runtime.

To understand the problem, consider the following function, which has only fixed parameters:

```
int foo(const char *mystr, BOOL mybool, char mychar, int myint, long mylong)
{
    NSLog(@"foo(%s, %x, %x, %x, %lx)", mystr, mybool, mychar, myint, mylong);
    return 42;
}
```

It’s possible to redeclare the function elsewhere in your library and execute it successfully with code similar to the following on the `x86_64` architecture:

```
extern int foo(const char *mystr, ...);
void printTestValues() {
   BOOL mybool = YES;
   char mychar = 42;
   int myint = 0xfeedface;
   long mylong = 0x0123456789abcdef;
   foo("hello", mybool, mychar, myint, mylong);
}
```

The same code fails on `arm64` because the caller of the function and the function itself marshal the parameters differently. The function expects all of the parameters to be in registers. However, the caller passes only the first parameter in a register; it passes all remaining parameters on the stack. As a result, the function implementation looks for the parameters in the wrong place, leading to unexpected results.

Even if you don’t redeclare your functions explicitly, functions like `objc_msgSend` redeclare your functions and methods implicitly. For more information, see Enable Strict Type Enforcement for Dynamic Method Dispatching.

Enable Strict Type Enforcement for Dynamic Method Dispatching

Due to calling convention differences between the `x86_64` and `arm64` architectures, update your dynamic-dispatching code to pass parameters correctly on both platforms. A function like `objc_msgSend` calls a method of an object, passing the parameters you supply to that method. Because `objc_msgSend` must support calls to any method, it accepts a variable list of parameters instead of fixed parameters. This usage of variable parameters changes how `objc_msgSend` calls your function, effectively redeclaring your method as a variadic function.

To illustrate the problem, consider an example where you want to call the following method using `objc_msgSend`:

```
- (void)document:(NSDocument*)doc
         didSave:(BOOL)didSave
     contextInfo:(void*)contextInfo;
```

Because `objc_msgSend` declares your method as variadic, the compiler places the method's parameters on the stack, in accordance with the calling conventions for the `arm64` architecture. However, the original method declaration contains fixed parameters, not variable parameters. As a result, the method's implementation looks for its parameters in registers, which is where the compiler places fixed parameters for `arm64`. This mismatch causes the method call to generate undefined results.

To fix dynamic-dispatching issues in your code, define a type-safe function pointer instead of calling `objc_msgSend` directly. You can use type-safe function pointers in both your `arm64` and `x86_64` code. A type-safe function pointer specifies the exact number of parameters, and incorporates the type information for each parameter into the `objc_msgSend` call, allowing the compiler to generate the calling conventions the method expects. For example, a type-safe function pointer for the `document:didSave:contextInfo:` method looks like the following:

```
// Declare a type-safe function pointer.
void (* didSaveDispatcher)(id,SEL,NSDocument *,BOOL,void *) =
       (void(*)(id,SEL,NSDocument *,BOOL,void *))objc_msgSend;
```

To initiate the dynamic dispatch operation, pass the target object, selector, and method parameters to your function pointer, as shown in the following code:

```
// Call the function, dispatching it through objc_msgSend.
didSaveDispatcher(myDelegate, mySelector, myDocument, NO, myPtr);
```

To locate places where you're not calling `objc_msgSend` in a type-safe way, enable the Enable strict Checking of objc_msgSend Calls build setting. When the value of that setting is YES, the

compiler flags your code where you're not using a type-safe function pointer with `objc_msgSend`.

Address Numerical Differences in Specific Frameworks

Some frameworks include minor changes that might impact code when porting to Apple silicon. For example:

- The `NSTextAlignment` enumeration uses different numerical values for some constants on `arm64` and `x86_64` architectures. When referring to constants using numerical values, validate that you use the correct values on each architecture.
- The `NSImage.ResizingMode` and `UIImage.ResizingMode` enumerations uses different numerical values for some constants on `arm64` and `x86_64` architectures. When referring to constants using numerical values, validate that you use the correct values on each architecture.
- Encoder IDs in the Video Toolbox framework may differ on `arm64` and `x86_64` architectures and on different versions of macOS. For example, the value in `kVTVideoEncoderSpecification_EncoderID` may differ between architectures.

For additional information about framework differences, see the specific framework reference.

Replace Raw Assembly Code with Builtin Intrinsics

If your app uses assembly code for specific tasks, or uses processor-specific `__builtin` functions, switch to the compiler's builtin intrinsic functions instead. The compiler's builtin intrinsics give you the same benefits of assembly code, but in a cross-platform package. During compilation, the compiler substitutes the builtin intrinsic function call for the appropriate set of assembly instructions for the current platform.

To illustrate the benefits of builtin intrinsics, consider the implementation of a function that counts the number of leading zeros in a value. The following example shows you how to use the builtin intrinsic for the `CLZ` instruction to count the number of leading zeros in an integer.

```
int GOOD_count_leading_zeroes(int x){
    int count;
    if(x == 0){
        return (sizeof(x) * CHAR_BIT);
    }
#if __has_builtin(__builtin_clz)
    count = __builtin_clz(x); // undefined when operating on 0
#else
    int index = 1;
    for(; x != 1; ++index){
```

```
        x = (unsigned)x >> 1;
    }
    count = ((sizeof(x) * CHAR_BIT) - index);
#endif
```

Implementing the same behavior without the builtin intrinsic requires significantly more code, as shown in the following example. The required code is also more complicated because it provides custom implementations for each processor architecture.

```
int BAD_count_leading_zeroes(int x){
    int count;
    if(x == 0){
        return (sizeof(x) * CHAR_BIT);
    }
#if defined(__x86_64__)
    __asm__ (
        "bsrl %1, %0\n\t" // undefined when operating on 0
        "xorl $0x1f, %1"
        : "=r" (count)
        : "r" (x)
    );
#elif defined(__aarch64__)
    __asm__ (
        "clz %w1, %w0"
        : "=r" (count)
        : "r" (x)
    );
#else
    int index = 1;
    for(; x != 1; ++index){
        x = (unsigned)x >> 1;
    }
    count = ((sizeof(x) * CHAR_BIT) - index);
#endif
    return count;
}
```

For a list of builtin intrinsic functions provided by the clang compiler, see the clang documentation at https://llvm.org.

Update Processor-Specific Vector Instructions

If your code includes instructions for the SSE, AVX, AVX2, or AVX512 units of Intel processors, update that code to support Apple silicon. The best alternative to processor-specific vector code is to use the Accelerate framework, which provides a vast library of vector operations optimized for all Mac computers. Accelerate leverages all available hardware of the current system to perform:

- Vector and matrix computations
- Image manipulation
- Digital signal processing
- Linear algebra computations
- Compression
- Neural network operations

For more information about the Accelerate framework, see Accelerate.

Apply Timebase Information to Mach Absolute Time Values

Always apply timebase information to values you receive from `mach_absolute_time` and never assume that the function returns the number of nanoseconds since boot. The value returned by the `mach_absolute_time` function is different for native and translated processes, and doesn't necessarily correspond to the number of nanoseconds since boot. Applying timebase information ensures that you can distribute time values between processes and between different computers.

The following code shows how to apply the timebase information to the value returned by the `mach_absolute_time` function:

```
uint64_t MachTimeToNanoseconds(uint64_t machTime) {
    uint64_t nanoseconds = 0;
    static mach_timebase_info_data_t sTimebase;
    if (sTimebase.denom == 0)
        (void)mach_timebase_info(&sTimebase);

    nanoseconds = ((machTime * sTimebase.numer) / sTimebase.denom);

    return nanoseconds;
}
```

To retrieve time values in nanoseconds without converting `mach_absolute_time` values, call the `clock_gettime_nsec_np` function instead.

Audit Code that Contains Float-to-Int Conversions

Apple silicon and Intel-based Mac computers handle some float-to-int conversions differently in C-based languages. To illustrate one of the differences, consider the conversion of the floating-point representation of infinity to a `uint32_t` or `int32_t` value:

```
uint32_t a = (uint32_t)INFINITY;
int32_t b = (int32_t)INFINITY;
```

The `arm64` architecture converts this value to the nearest possible integer.

```
a = 0xffffffff = 4294967295  // The largest unsigned integer
b = 0x7fffffff = 2147483647  // The largest signed integer
```

For the unsigned conversion, the `x86_64` architecture wraps the value to `0`. For the signed conversion, it sets the value to an indefinite integer.

```
a = 0x00000000 = 0  // Wrap around to zero.
b = 0x80000000 = -21474836548  // The indefinite integer value.
```

> **Note**
>
> The Swift programming language defines consistent float-to-int conversion behaviors for all CPU architectures.

If your code converts floating-point numbers to integers, audit your code to make sure that you handle boundary conditions correctly. One way to detect invalid conversions is to run the UBSan tool with the `float-cast-overflow` option selected. To detect implicit conversions by the compiler, enable the `-Wconversion` compiler flag when building your code.

For information about how to use the UBSan tool, see Diagnosing memory, thread, and crash issues early.

Treat BOOL Variables as Binary Values

As a rule, the Objective-C `BOOL` type has only two appropriate values: `YES` or `NO`. On Apple silicon, the compiler defines the `BOOL` type to be a native `bool`, but on Intel-based Mac computers, it is a signed `char`. To avoid issues in universal binaries:

- Never perform mathematical operations on `BOOL` variables.

- Never increment or decrement BOOL variables.
- Never assume the numerical value of a BOOL is anything other than 0 or 1.

To illustrate the problem, consider the following example:

```
int nBytes = 1024;
BOOL receivedBytes = nBytes;

if (receivedBytes) {
   printf("Success!\n");
} else {
   printf("Failure...\n");
}
```

On the x86_64 architecture, `receivedBytes` evaluates to `0`, or `false`. On the arm64 architecture, `receivedBytes` evaluates to `true`. One way to fix the preceding code is to include two negation operators (`!!`) in front of the `nBytes` variable before assigning it to the `receivedBytes` variable, as shown in the following example:

```
int nBytes = 1024;
BOOL receivedBytes = !!nBytes;
```

> **Note**
>
> Enable the `-Wobjc-signed-char-bool-implicit-int-conversion` compiler option to generate warnings for implicit BOOL conversions at compile time.

Update Just-In-Time Compilers

Update the workflow of any just-in-time compilers to support Apple silicon, which prevent all memory pages from being simultaneously writable and executable. On Intel-based Mac computers, this same behavior applies only to apps that adopt the Hardened Runtime.

For more information, see Porting just-in-time compilers to Apple silicon.

Update C++ Code

The C++ ABI for Apple silicon matches the ABI for iOS devices, and not the ABI for Intel-based Mac computers. For information about the ABI, see iOS ABI Function Call Guide.

See Also

General porting tips

- Porting your audio code to Apple silicon

  Eliminate issues in your audio-specific code when running on Apple silicon Mac computers.

- Porting just-in-time compilers to Apple silicon

  Update your just-in-time (JIT) compiler to work with the Hardened Runtime capability, and with Apple silicon.