Marcel Weiher

iOS and macOS™ Performance Tuning

Cocoa®, Cocoa Touch®, Objective-C®, and Swift™

Opened in November 2010, the **Sheikh Zayed Bridge** offers a stunning entryway to the city of Abu Dhabi, United Arab Emirates. Designed by the visionary architect Zaha Hadid, the bridge appears to onlookers as a series of rising and falling concrete waves, reminiscent of the region's nearby sand dunes. Hadid's "waves," reaching 64 meters at their peak, appear to propel themselves toward the city. Her design's extraordinary energy is reinforced by dynamic nighttime lighting, making the bridge an unforgettable local landmark.

iOS and macOS™ Performance Tuning

This page intentionally left blank

iOS and macOS™ Performance Tuning

Cocoa®, Cocoa Touch®, Objective-C®, and Swift™

Marcel Weiher

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Visit us on the Web: informit.com/aw

Library of Congress Number: 2016961010

ISBN-13: 978-0-321-84284-8
ISBN-10: 0-321-84284-7

1 17

Editor-in-Chief
Greg Wiegand

Senior Acquisitions Editor
Trina MacDonald

Development Editor
Songlin Qiu

Managing Editor
Sandra Schroeder

Full-Service Production Manager
Julie B. Nahil

Project Manager
Melissa Panagos

Copy Editor
Stephanie Geels

Indexer
Jack Lewis

Proofreader
Melissa Panagos

Technical Reviewers
Christian Brunschen
BJ Miller
Christian Neuss
Dominik Wagner

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
Lori Hughes

Contents at a Glance

This page intentionally left blank

Contents

About the Author

Marcel Weiher is a software engineer and researcher with more than 25 years of experience with Cocoa-related technologies. Marcel's work has always been performance-focused, ranging from solving impossible pre-press problems on the machines of the day via optimizing one of the world's busiest Web properties at the BBC to helping other Apple engineers improve the performance of their code on Apple's Mac OS X performance team.

In addition to helping established companies and start-ups create award-winning software and turn around development teams, Marcel also teaches, blogs, speaks at conferences, contributes to open source, and invents new techniques such as Higher Order Messaging. He also works on programming languages, starting with an Objective-C implementation in 1987 and culminating in the Objective-Smalltalk architecture research language. Marcel currently works as a principal software engineer at Microsoft Berlin and maintains his own software and consulting company, metaobject ltd.

This page intentionally left blank

Introduction

Performance is one of the most important qualities of software programs. You can't have world-beating software without world-beating performance. For a long time, hardware improvements meant that worrying about software performance seemed a waste of time, but with Moore's Law no longer automatically providing significant automatic performance improvements, performance optimization is coming back to the forefront of both computer science and engineering.

In addition, performance for end users seems to have gotten only marginally better, whereas the performance of the underlying hardware has improved by many orders of magnitude. Bill Gates quipped that "the speed of software halves every 18 months," whereas Wirth's law in *A Plea for Lean Software* states, "Software is getting slower more rapidly than hardware becomes faster."[1]

We are so used to this sorry state of affairs that industry veterans were surprised at the original iPad's fluid UI, despite having a CPU with "only" 1 GHz. That's more than 1,000 times faster than my Apple][, and 40 times faster than my NeXT cube that had a larger screen to deal with. If anything, the surprise should have been that it wasn't faster, especially when considering that it also had a GPU to handle the screen.

This book will try to give insights into the underlying reasons for these developments in the context of Objective-C, Cocoa, and CocoaTouch, and attempt to provide techniques for taking full advantage of the raw power of our amazing computing machines—power that we tend to squander with reckless abandon. It will also try to show when it is actually OK to squander that power, and when it is necessary to pay careful attention. Programmer attention is also a scarce resource, too often squandered attempting to optimize parts of the program that do not matter.

General themes will include latency versus bandwidth, and transactions costs (overhead) versus actual work done, themes that are universal and manifest themselves in different forms at every level of the hardware and software stack.

What you will notice is that due to the speed of our machines, any single operation is, in fact, always more than fast enough, so the crucial equation is $items * cost$. Most optimization is about reducing one or both of the parts of that formula, usually by breaking it up first.

---

1. Niklaus Wirth, *A Plea for Lean Software* (Los Alamitos, CA: IEEE Computer Society Press, 1995), pp. 64–68. http://dx.doi.org/10.1109/2.348001

One frequent method for reducing cost is to realize that *cost* is actually composed of two separate costs, $cost_1$ and $cost_2$, and only one of these needs to be applied to all items: $items \times (cost_1 + cost_2) \rightarrow cost_1 + items \times cost_2$. I would probably call this the fundamental optimization equation; a large part of the optimization techniques fall into this category, and it is also fundamental to the organization of most of the hardware/software stack we deal with every day.

This book has a very regular structure, with four basic areas of performance discussed in turn:

1. CPU performance
2. Memory
3. I/O
4. Graphics and responsiveness

Although an effort has been made to keep the treatment of each subject area independent, there is a logical progression, so at least a passing familiarity with earlier topics helps with later topics.

Within each of these four broad topics, there are again four specific areas of interest:

1. Principles
2. Measurement and tools
3. Pitfalls and techniques
4. Larger real-world examples of applying the techniques

Again, there is a logical structure: You need to have some idea about the principles and know how to measure before you can meaningfully think about actual performance optimization techniques, but again, you should also be able to dip into specific areas of interest if you have a passing familiarity with earlier topics.

This structure yields a total of $4 \times 4 = 16$ chapters, with a special chapter on Swift tucked between memory and I/O for a total of 17. Swift is also used throughout the book where appropriate, but it deserves a chapter of its own due to its unique performance characteristics.

For me, software performance is a passion and a calling that has been a common thread throughout my career. I have learned that performance is something you can't automate, nor can you leave it until the last minute. On the other hand, there are many times when you shouldn't worry about performance in order to have the capacity to concentrate on performance where it is really needed. If that weren't paradoxical enough, having excellent base performance levels is often what makes it possible to get to that state of not having to worry about performance most of the time.

In short, this book is about making software that performs beautifully.

3

CPU: Pitfalls and Techniques

Having had a look at the parameters driving performance and techniques for identifying slow code, let's now turn to actual techniques for making code run fast. We will look at efficient object representations and ways for those objects to communicate and access data. We will also examine streamlining computation. In all this, the objective will typically be to effectively combine the "Objective" and the "C" parts of Objective-C to achieve the desired balance between performance and encapsulation.

In general, the basic idea is for objects to have C on the inside and messages on the outside, and for the objects themselves to be fairly coarse-grained, mostly static entities. When following these principles, it is possible to start with a fully object-oriented implementation without worries, but with the knowledge that it will be possible to later optimize away any inefficiencies. It has been my experience that it is quite possible to achieve the performance of plain C, and sometimes even beyond.

However, there are pitfalls that not only make an Objective-C program slow (slower than so-called scripting languages), but even worse can be major obstacles to later optimization efforts. These pitfalls usually lie in library constructs that are easy to use but have hidden performance costs, costs that are not localized within a single object where they could be eliminated, but present in interfaces and therefore spread throughout the system and much harder to expunge.

The following will show different options for data representation, communication, and computation, along with their respective trade-offs in terms of coupling, cohesion, and performance.

Representation

One of the primary tasks of a program, especially an object-oriented program, is to represent data. Due to the hybrid nature of the language, an Objective-C programmer has many options available for this task.

Without any claims of completeness, structured data can be represented using a C `struct`, Objective-C object, or various forms of key-value stores, most prominently

Foundation's NSDictionary and CoreFoundation's CFDictionary, which are both getting more and more use. Simple scalars can be represented as C float, double, or int and their multitude of variations, Foundation NSInteger and CoreGraphics CGFloat typedefs, and finally Foundation NSNumber and CoreFoundation CFNumber objects. Note that the naming conventions are a bit confusing here: The names NSInteger and NSNumber strongly suggest that these two types are related—for example, with NSInteger being a specific subclass of NSNumber—but in fact they are completely unrelated. NSInteger is a typedef that resolves to a 32-bit int on 32-bit architectures and to a 64-bit long on 64-bit architectures, whereas int is 32 bits in both cases. Similar with CGFloat, which turns into a 32-bit float on 32-bit architectures and a 64-bit double on 64-bit architectures. Example 3.1 shows a few of the possible number representations.

Example 3.1 Numbers as primitives and objects

```
#import <Foundation/Foundation.h>

int main()
{
  int a=1;
  float b=2.0;
  NSNumber *c=[NSNumber numberWithInt:3];
  CFNumberRef d=CFNumberCreate(kCFAllocatorDefault,
                    kCFNumberFloatType, (const void*)&b );
  NSNumber *e=@(5);
  NSLog(@"a=%d b=%g c=%@ d=%@ e=%@",a,b,c,d,e);
  return 0;
}
```

In order to come to a good solution, the programmer must weigh trade-offs between decoupling and encapsulation on one hand and performance on the other hand, ideally getting as much decoupling and encapsulation without compromising performance, or conversely maximizing performance while minimizing coupling.

Primitive Types

Possibly the easiest call to make is in the representation of simple scalar types like characters/bytes, integers, and floating point numbers: use the built-in C primitive types whenever possible, and avoid object wrappers whenever possible.

With the language supporting them natively, scalars are convenient to use and perform anywhere from 10 to more than 100 times better than their corresponding Foundation object NSNumber or its CoreFoundation equivalent CFNumber. Table 3.1 gives the details: the first three columns are times for different arithmetic operations on scalar types. The differences in timings for 32- and 64-bit addition and

Table 3.1 Primitive operations in 32- and 64-bit architectures

Operation	add	multiply	divide	-intVal	NS(int)	CF(float)	NS(float)
64-bit (ns)	0.67	0.79	14	15	44	169	190
32-bit (ns)	0.72	0.76	7.8	22	232	182	211

multiplication are probably measuring artifacts, though they were stable when preparing these measurements and it is important to report actual results as measured, not what we *think* the results should be.

Division is slower than the other arithmetic operations because dividers in CPUs usually only handle a few bits at a time, rather than a full word, which also explains why 64-bit division is significantly slower than 32-bit division.

Compared to entities that can usually be stored in registers and manipulated in a single clock cycle (or less on superscalar designs), any object representation has excessive overhead, and Objective-C's fairly heavyweight objects are doubly so. Foundation and CoreFoundation make this overhead even worse by providing only immutable number objects, meaning any manipulation must create new objects. Finally, scalars like numbers and characters tend to be at the leaves of any object graph and therefore are the most numerous entities in a program, with every object containing at least one but more likely many instances of them.

On the flip side, there is little variation or private data that would benefit from the encapsulation and polymorphism that are made possible by an object representation, and number objects are in many ways even less capable than primitive types, for example, by not providing any arithmetic capabilities. This could change in the future if Foundation or another framework provided a number and magnitudes hierarchy similar to that of Smalltalk or LISP, where small integers automatically morph into infinite precision integers, fractions, floating point, or even complex numbers as needed. Alas, Foundation provides none of these capabilities, though the introduction of tagged integers in the 64-bit runtime on OS X 10.7 along with the addition of number literals in 10.8 could be a sign of improvements in the future.

Of course, there are times when an object is required by some other interface, for example, when adding content to an `NSArray` or `NSDictionary`. In this case, you must either use `NSNumber` or an equivalent or provide alternatives to those interfaces—an option we will explore more later in the chapter.

One wrinkle of Table 3.1 is that although most times are similar between 32 and 64 bits, two numbers are different. The division result is about twice as slow on 64 bit, whereas the creation of integer NSNumber objects is six times faster. The division result is easily explained by the fact that the integer division hardware on the particular CPU used processes a fixed number of bits per cycle, and 64-bit operands simply have twice as many bits. The multiply and add circuits, on the other hand, operate on full 64-bit words at once.

Table 3.2 Tagged and regular pointers

Bits	8–32/64	4–7	3	2	1	0
Regular pointer	upper address bits				0	
Tagged pointer	value	subtype	tag id			1

The difference in allocation speeds for integer objects on the other hand has nothing to do with the CPU differences and everything with the fact that Apple introduced tagged integers in OS X, but only in the modern runtime, and only for the 64-bit version of that runtime. Tagged integers are a technique taken from old LISP and Smalltalk systems where the value of an integer object is encoded not in an allocated structure pointed to by the object, as usual, but rather in the object pointer itself. This saves the pointer indirection when accessing and especially the memory allocation when creating or destroying the data (integers in this case). This representation takes advantage of the fact that object pointers are at least word aligned, so the lower 2 or 3 bits of a valid object pointer are always 0 on 32-bit and 64-bit systems, respectively. Table 3.2 shows how the tagged pointer representation puts a "1" in the low bit to distinguish tagged pointers from regular pointers, another 7 bits for typing the value, and the remaining 24 or 56 bits to store a value.

In fact, it is puzzling that the performance for integer `NSNumber` creation isn't much better than it is, since all it takes is the bit-shift and arithmetic OR shown in the `makeInt()` function of Example 3.2, possibly with some tests depending on the source and target number type—operations that should be in the 1 to 2 ns total range.

Example 3.2 Summing manually created tagged NSNumber objects

```
#import <Foundation/Foundation.h>

#define kCFTaggedObjectID_Integer ((3 << 1) + 1)
#define kCFNumberSInt32Type 3
#define kCFTaggedIntTypeOffset 6
#define kCFTaggedOffset 2
#define kCFTaggedIntValueOffset (kCFTaggedIntTypeOffset+kCFTaggedOffset)
#define MASK (kCFNumberSInt32Type<<kCFTaggedIntTypeOffset)
#define kCFTaggedIntMask (kCFTaggedObjectID_Integer | MASK)

static inline int getInt( NSNumber *o ) {
  long long n=(long long)o;
  if ( n & 1 ) {
    return  n >> kCFTaggedIntValueOffset;
  } else {
    return [o intValue];
  }
}

static inline NSNumber *makeInt( long long o ) {
```

```
  return (NSNumber*)((o << kCFTaggedIntValueOffset) | kCFTaggedIntMask);
}

int main( int argc , char *argv[] )
{
  NSNumber* sum = nil;
  for (int k=0;k<1000000; k++ ) {
    sum =makeInt(0);
    for (int i=1;i<=1000;i++) {
      sum =makeInt(getInt(sum)+i);
    }
  }
  NSLog(@"%@/%@ -> '%@'",sum,[sum class],[sum stringValue]);
  return 0;
}
```

The reason of course is that Apple has so far hidden this change behind the existing messaging and function call application programming interfaces (APIs) going through CoreFoundation. We are also advised that the representation, including the actual tags, is private and subject to change. What we are leaving on the table is significant: The code in Example 3.2 runs in 1.4 s, compared to 11.4 s for the Foundation/CoreFoundation-based code from Chapter 1.

Hopefully this will change in the future, and the compiler will become aware of these optimizations and be able to generate tagged pointers for integer objects and some of the other tagged types that have been added in the meantime. But as of OS X 10.11 and Xcode 7.3, it hasn't happened.

Strings

A data type that almost qualifies as a primitive in use is the string, even though it is actually variable in length and doesn't fit in a processor register. In fact, Objective-C strings were the first and for a long time the only object that had compiler support for directly specifying literal objects.

There are actually several distinct major uses for strings:

1. Human readable text
2. Bulk storage of serialized data as raw bytes or characters
3. Tokens or keys for use in programming

While these cases were traditionally all handled uniformly in C using `char*` pointers, with some `NUL` terminated and others with a length parameter handled out of band, conflating the separate cases is no longer possible now that text goes beyond 7-bit ASCII.

Cocoa has the `NSString` class for dealing with human readable text. It handles the subtleties of the Unicode standard, delegating most of the details to `iconv` library. This sophistication comes at a cost: roughly one order of magnitude slower performance than raw C strings. Table 3.3 shows the cost of comparing 10- and 32-byte `C-Strings` with 10- and 32-character `NSString` objects.

Table 3.3 NSString and C-String operations

Operation	1 ns	!strcmp(10)	strcmp(32)	!nscmp(10)	ns append	nscmp(32)
1 ns	1	3.3	10	76	77	82
!strcmp(10)		1	3	23	23	25
strcmp(32)			1	7.5	7.6	8.2
!nscmp(10)				1	1	1.1
ns append					1	1.1
nscmp(32)						1

Although `NSStrings` are expensive, this is an expense well spent when the subject matter really is human-readable text. Implementing correct Unicode handling is complex, error prone, and inherently expensive. In addition, the option of having multiple representations with a common interface is valuable, allowing string representations optimized for different usage scenarios to be used interchangeably. For example, literal `NSStrings` are represented by the `NSConstantString` class that stores 8-bit characters, whereas the standard `NSCFString` class (backed by `CFString CoreFoundation` objects) stores 16-bit unichars internally. Subclasses could also interchangeably provide more sophisticated implementations such as ropes, which store the string as a binary tree of smaller strings and can efficiently insert/delete text into large strings.

Starting with OS X 10.10, string objects on the 64-bit runtime also got the tagged pointer treatment that we previously saw for integers. This may seem odd, as strings are variable-length data structures, arrays of characters. However, 64 is quite a lot of bits, enough to store seven 8-bit characters and some additional identifying information such as the length. In fact, when I myself proposed tagged pointer strings back in 2007, I also had variants with eight 7-bit ASCII strings, or an even tighter packing that ignores most of the control and special characters to use only 6 bits and thus have room for 9 characters. I don't know if any of those variants are implemented.

Example 3.3 illustrates the different `NSString` implementation types: a literal is an instance of `__NSCFConstantString`, a CF variant of `NSConstantString`. Creating a mutable copy creates a new string object, whereas creating a copy of that mutable copy creates a tagged pointer string because the string is only 5 characters long. All of this is implementation dependent, but the differences are relevant when looking at `NSDictionary` lookup performance.

Example 3.3 Show differences between normal, constant, and tagged strings

```
#import <Foundation/Foundation.h>

void printString( NSString *a ) {
```

```
  NSLog(@"string=%@ %p class: %@",a,a,[a class]);
}

int main()
{
  NSString *cs=@"Const";

  printString(cs);
  printString([cs mutableCopy]);
  printString([[cs mutableCopy] copy]);
}
cc -Wall -o taggedstring taggedstring.m -framework Foundation
./taggedstring
string=Const 0x108fe2040 class: __NSCFConstantString
string=Const 0x7fb359c0d630 class: __NSCFString
string=Const 0x74736e6f4355 class: NSTaggedPointerString
```

While great for human readable text, `NSString` objects are somewhat heavyweight to be used for serialized data, which is handled more safely and efficiently by the `NSData` class. Unlike `NSString`, which requires an encoding to be known for text data and can therefore not be safely used on arbitrary incoming data (it will raise an exception if the data does not conform to the encoding), `NSData` can be used with arbitrary, potentially binary data read from the network or a disk. For performance, it is possible to get a pointer to the `NSData`'s contents via the `-byte` or `-mutableBytes` methods for processing using straight memory access, whereas `NSString` (rightfully) protects its internal data representation, with processing only possible by sending high-level messages or by copying the data out of the `NSString` as 16-bit `unichar` character data or encoded 8-bit bytes.

When parsing or generating serialized data formats, even textual ones, it is significantly more efficient to treat the serialized representation such as the JSON in Example 3.4 as raw bytes in an `NSData`, parse any structure delimiters, numbers, and other non-textual entities using C character processing, and create `NSString` objects exclusively for actual textual content, rather than reading the serialized representation into an `NSString` and using `NSScanner` or other high-level string processing routines.

Example 3.4 Textual content of JSON file is shown in bold

```
[ { "name": "AAPL",  "price": 650.1,     "change": 20.41  },
 {  "name": "MSFT",  "price": 62.79,     "change": -0.9   },
 {  "name": "GOOG",  "price": 340.79,     "change": -5.2   }, ]
```

Even the strings that appear in such a file tend to be structural rather than actual content, such as the dictionary keys in Example 3.4. These types of structural strings are also represented as `NSString` objects in Cocoa, just like human-readable text.

While convenient due to the literal `NSString` syntax (`@"This is a constant string"`), this conflating of human-readable text and functional strings can at times be unfortunate in terms of performance. Fortunately, many types of keys that are more optimized exist—for example, basic C strings, message names, and instance variable names.

Objects

Since you're programming in Objective-C, it is likely that objects are going to be your major data-structuring mechanism.

Use C inside the objects. The messaging interface hides the representation, and users are none the wiser. Try to avoid using fine-grain, semantic-free objects to implement the coarse-grain, semantics-bearing objects.

Accessors

Accessors are methods that just read or write an object's internal data, corresponding roughly to memory read and write instructions. According to good object-oriented style, attributes of an object should not be accessed directly, certainly not from outside the object, but preferably also from within. Objective-C 2.0 properties handle the burden of creating accessors.

However, accessors should also at least be minimized and ideally should be eliminated altogether, because they turn objects from intelligent agents that respond to high-level requests for service to simple data-bearing structures with a higher cost of access. Apart from a cleaner design, passing high-level requests into an object also makes sense from a performance point of view because this means the transaction costs of a message send is paid only once, at which point the method in question has access to all parameters of the message and the object's instance variables, instead of using multiple message sends to gather one piece of data from the object at a time.

Of course, in reality, accessors or property definitions are a common feature of Objective-C programs, partly because program architecture deviates from object-oriented ideals and partly because accessors for object references in Objective-C are also needed to help with reference counting, as shown in Example 3.5.

Example 3.5 Object accessors need to maintain reference counts

```
-(void)setInteger:(int)newInteger {
      _integer=newInteger;
}
-(void)setObject:(id)newObject {
     [newObject retain];
     [_object release];
     _object=newObject;
}
```

As with other repetitive boilerplate, it makes sense to automate accessor generation, for example, by using Xcode macros, preprocessor macros that generate the accessor code. Alternately, the language can take over: Since Objective-C 2.0 properties can automatically synthesize accessors and with Automatic Reference Counting (ARC), the actual reference counting code was moved from the accessors to the code-generation of all variable access.

A caveat with using properties for generating accessors is that the generated code is not under user control, with the default `atomic` read accessors up to five times slower than a straightforward implementation, because they `retain` and `autorelease` the result, place a lock around the read in case of multithreaded access, and finally need to wrap all of that in an exception handler in order to release the lock in case of an exception. An alternative is the accessor macros shown in Example 3.6. These macros generate the correct accessor code just like properties. However, this generation is under user control, meaning not only that you get to decide what code gets run, but also that you can (a) change your mind and (b) extend the idea further without having to modify the compiler, as I will show later.

Example 3.6 Accessor macros

```
#if !__has_feature(objc_arc)
#define ASSIGN_ID(var,value)\
  {\
    id tempValue=(value);\
      if ( tempValue!=var) {  \
        if ( tempValue!=(id)self ) \
          [tempValue retain]; \
        if ( var && var!=(id)self) \
          [var release]; \
        var = tempValue; \
      } \
  }
#else
#define ASSIGN_ID(var,value)    var=value
#endif

#ifndef AUTORELEASE
#if !__has_feature(objc_arc)
#define AUTORELEASE(x)  ([(x) autorelease])
#else
#define AUTORELEASE(x)  (x)
#endif
#endif

#define setAccessor( type, var,setVar ) \
-(void)setVar:(type)newVar { \
    ASSIGN_ID(var,newVar);\
} \
```

```
#define readAccessorName( type, var , name )\
-(type)name { return var; }

#define readAccessor( type, var ) readAccessorName( type, var, var )

#define objectAccessor( objectType, var, setVar ) \
    readAccessor( objectType*, var )\
    setAccessor( objectType*, var,setVar )
```

In OS X 10.11, the slowdown has apparently been reduced to around 35%, with or without ARC enabled.

Due to the pervasiveness of accessors, this overhead is serious enough that teams at Apple sped up whole programs by more than 10% just by switching properties from `atomic` to `nonatomic`. An improvement of 10% may not seem much when we are frequently talking about improvements of 10 to 100 times, but it is actually huge when we are talking about the whole program, where significant engineering effort is often expended for single-digit percentage improvements. And here we get double digits with a single change that had no other effect. So why does `atomic` exist? And why is it the default?

The idea was to protect against code such as that shown in Example 3.7. This code has a stale reference to an object instance variable that was actually released when the pointer went stale, similar to some early Unix `malloc()` implementations having a `free()` function that delayed freeing its memory until the next call to `malloc()`, in essence avoiding a potential crash in buggy code such as that in Example 3.7.

Example 3.7 Stale pointer reference

```
 ...
  id myWindowTitle=[window title];
  [window setTitle:@"New Window title"];  // windowTitle goes stale
  [self reportTitle:myWindowTitle];         // crashes pre-ARC
....
```

The crash will occur if `title` is held onto by the window, and only by the window, because in that case `setTitle:` will release the title and the reference to this object in `myWindowTitle` will not only be stale, that is, no longer pointing to the window's title—but also invalid. Having auto-releasing accessors such as the ones provided by the `atomic` keyword will prevent a crash in this case, but at the cost of hiding the fact that the reference has, in fact, gone stale. I can see two potential reasons for writing this code. The first is that of a simple but slightly premature optimization if the title is used several times and we don't want to go fetch it from the window every time. In this case the code is simply wrong, because you'd actually want to get the new value from the window after it was set, and `atomic` in this case

just masks the incorrect code. A crash would alert the programmer to the fact that the logic is amiss. The second case is that in which the programmer actually intended to stash away the old value. In this case, the code is also plain buggy, because the programmer is well aware that the new value will make the old value invalid—that's why they are stashing it! The corrected code in Example 3.8 not only doesn't need `atomic`, it also makes the intention of the code clear.

Example 3.8 Non-stale pointer reference

```
 ...
  id oldWindowTitle=[[window title] retain] autorelease];
  [window setTitle:@"New Window title"];
  [oldWindowTitle doSomething];   // clear that we want old title
....
```

Note that ARC also prevents the crash, and therefore also hides the staleness of the pointer, just like `atomic` did—by aggressively retaining objects even when they are stored into local variables. The advantage is that you don't have to think as much about the lifetime of your objects. The disadvantage is that you don't have to think as much about the lifetime of your objects, and you get significantly more reference-counting traffic, which impacts performance.

So while it is unclear whether `atomic` would be beneficial at all even if there were no performance penalty, the significant slowdown in a very common operation makes it highly questionable at best. The fact that the collection classes do not support this pattern (for performance reasons) and iOS's UIKit explicitly sets `nonatomic` for over 99% of its property declarations shows that Apple itself is not of one mind in this case.

Even slower than `atomic` accessors is access via key-value coding (KVC): A call such as `[aTester valueForKey:@"attribute"]` is not only more verbose than the equivalent direct message send `[aTester attribute]`, and not only more error prone because the compiler cannot check the validity of the string passed to `valueForKey:`, it is also 20 times slower. If runtime parameterization of the value to get is required, using `[aTester performSelector:@selector (attribute)];` is only twice as slow as a straight message send and 10 times faster than `valueForKey:`.

You might expect from these basic performance parameters that technologies built on top of KVC such as key-value observing (KVO) and Cocoa Bindings can't be too speedy, and you'd be right: Adding a single KVO observer adds a factor of 100 to the time of a basic set accessor (600 ns vs. 6 ns) and a single binding a factor of 150 (900 ns).

KVO and bindings also do not protect against cascading update notifications, which can lead to at least quadratic performance if there are transitive dependencies (`b` depending on `a` and `c` depending on both `a` and `b` will result in `c` being evaluated twice), and can lead to infinite recursion and crashes if there are dependency loops. So

for larger data sets or complex dependencies, it is probably a good idea to investigate using a proper constraint solver in the tradition of the 1978 Xerox PARC *ThingLab* or later developments such as *DeltaBlue*, *Amulet*, or *Cassowary*. In fact, it appears that *Cassowary* was adopted by Apple for Mountain Lion's auto-layout mechanism.

Public Access

When sending messages to access instance variables is too slow, those instance variables can be made `@public`. In this case, access time is essentially the same as for a C `struct`, (non-fragile instance variables mean that the offset is looked up in the class instead of being hard-coded at compile-time, slightly affecting the result) but then again so is safety and encapsulation: none of either. The case can therefore be made that if `@public` access is required, one should use a `struct` instead. In fact, there are some additional benefits to a struct, mainly that it can be allocated on the stack in an `auto` variable, passed by value to a function, or directly embedded into another object or `struct` or array, whereas an Objective-C object must be expensively allocated on the heap and can only be accessed indirectly via pointer.

However, there are also some benefits to keeping such an open object a true Objective-C object—namely, it can have additional functionality attached to it, access can be granted or denied on a per-field basis, and it may be used compatibly with other objects that are not aware of its publicly accessible instance variables. As an example, the PostScript interpreter mentioned in Chapter 1 uses a string object that has all its instance variables public, shown in Example 3.9, but at the same time can be used largely interchangeably with Cocoa `NSString` objects.

Example 3.9 Full public string object definition

```
@interface MPWPSString : MPWPSCompoundObject
{
    @public
    unsigned char *bytes;
    unsigned   length,capacity;
}
```

Of course, breaking encapsulation this way makes evolution of the software harder and should be considered as a last resort when all other techniques have been tried, performance is still not adequate, and careful measurement has determined that the access in question is the bottleneck.

Object Creation and Caching

As we have seen so far, object creation is expensive in Objective-C, so it is best to use objects as fairly static and coarse-grained entities that exchange information via messages, preferably with mostly scalar/primitive arguments. If complex arguments

cannot be avoided and high rates of creation/exchange need to be maintained, both Objective-C and Swift can resort to `structs` instead of objects, which like primitives can be stack allocated, allocated in groups with a single `malloc()`, and passed by value as well as by reference. However, this often means either switching between objects and structs when modeling the problem domain, or even foregoing object modeling altogether.

Another option to lessen or even eliminate the performance impact of object creation when high rates of (temporary) object creation cannot be avoided, is object caching: reusing objects that have already been allocated. The advantage of object caching over using structs is that performance considerations do not interfere with the modeling of the problem domain and all the code involved. Instead, a pure performance fix can be applied if and when performance turns out to be a problem.

Table 1.2 shows that reusing just one object instead of allocating a new one, we have not only saved some memory, but also CPU time equivalent to approximately 50 message sends, allowing us to use objects where otherwise we might have had to revert to C for performance reasons. Object reuse was common in object-oriented languages until generation-scavenging copying garbage collectors with "bump pointer" allocation came online that made temporary objects extremely cheap. Alas, C's memory model with explicit pointers makes such collectors that need to move objects nigh impossible, so object reuse it is!

In order to reuse an object, we have to keep a reference to it in addition to the reference we hand out, for example, in an instance variable or a local collection. We can either do this when we would have otherwise deallocated the object, or we can keep a permanent reference. Then, when it comes time to create another object of the desired class, we check whether we already have a copy of it and use that already allocated copy instead.

Mutability and Caching

When is it safe to reuse an object? Immutable value objects, for example, can be reused as often as desired, because different copies of the same value object are supposed to be indistinguishable. Foundation uses this strategy in a number of places for some global uniquing. Small number objects are kept in a cache once allocated, and constant string objects are merged by the compiler and linker and shared.

In order to cache objects behind the client's back, these objects must be immutable, because sharing between unwitting clients becomes impossible if changes made by one client become visible to another. However, immutability forces creating a new object on every change, and creating a new (uncached) number object every time a new value is needed is around 30 to 40 times more expensive than just setting a new value, even if done safely via an accessor. So how can we reuse mutable objects?

One way, chosen by the UIKit for table cells, is to have a documented API contract that guarantees reusability. Another is to take advantage of the Foundation reference counting mechanism, which we use to track if the only reference left to the object is the one from the cache, in which case the object can be reused. Instead of

Table 3.4 Reference counts for object caching

	in-use	unused	unused action
retain/release	$RC > 0$	$RC = 0$	deallocate
object caching	$RC > 1$	$RC = 1$	reuse

using the $1 \rightarrow 0$ transition to see whether the object needs to be deallocated, we use the $RC = 1$ state to see whether the object can be reused, because the cache is keeping a single reference. Table 3.4 summarizes this information.

Example 3.10 shows how this reference-count-aware[1] cache can be implemented, though the actual implementation that's part of a generic object-cache class is much more heavily optimized. The instance variables referenced here are defined in Example 3.18 and discussed in detail in the "IMP Caching" section in this chapter.

Example 3.10 Circular object cache implementation

```
-getObject
{
    id obj;
    objIndex++;
    if ( objIndex >= cacheSize ) {
        objIndex=0;
    }
    obj=objs[objIndex];
    if ( obj == nil || [obj retainCount] > 1 ) {
        if ( obj != nil ) {
            [obj release];      //--- removeFromCache
        }
        obj=[[objClass alloc] init];
        objs[objIndex]=obj;
    } else {
        obj=[obj reinit];
    }
    return obj;
}
```

The `MPWObjectCache` keeps a circular buffer of objects in its cache that's a C array of `id`s. When `getObject`[2] method is called to create or fetch an object, it

1. Though Apple generally recommends against calling `retainCount`, this use is not of the problematic kind.
2. `id` is the default return type and elided.

looks at the current location and determines whether it can reuse the object or needs to allocate a new one and then bump the location. It assumes objects know how to reinitialize themselves from an already initialized but no longer valid state. The circular buffer structure gives objects that were vended by the cache some time before we try to reuse them, similar to the young space in a generation-scavenging collector. At around 9.5 ns per pop, allocating from the (optimized) object cache is around 15 times faster than straight object allocation, so this is a very worthwhile optimization.

Wraparound of the index is handled via an if-check rather than a modulo operation, because a modulo is a division, and as we saw earlier in this chapter, division is one of the few arithmetic operations that is still fairly slow even on modern CPUs. A different way of implementing a modulo would be by and-ing the low bits of the index, but that would restrict the cache size to powers of 2. Finally, there are many variations of probing and retirement policies that will have different performance characteristics, for example, attempting at least *n* consecutive slots or using random probing. So far, this very simple algorithm has proved to be the best balance for a wide variety of use-cases.

Another potential way of using reference counts is to stick to the 1→0 transition the way traditional reference counting does and then override `dealloc` to enqueue the object in a global cache instead of deallocating regularly. However, that sort of approach, unlike the object cache presented here, couples the target class tightly to the caching behavior and requires use of a global cache. I therefore recommend against that type of global cache, despite the fact that it is quite popular. Not requiring locking, scoping the cache to the lifetime of another object and the specific circumstance of that object's use patterns are a large part of what makes object caching via a cache object powerful and fast.

Lazy Evaluation

Another use of caching is lazy evaluation of properties. When a message requests a property of an object that is expensive to compute and may not even be always needed, the object can delay that computation until the property is actually requested instead of computing the property during object initialization. Alternately, the result of an expensive computation can be cached if it is likely that the computation will be used in the future and if it is known that the parameters of the computation haven't changed.

Lazy accessors have become common enough in my code that they warrant a specialized accessor macro, shown in Example 3.11.

Example 3.11 Lazy accessor macro

```
#define lazyAccessor( type, var ,setVar, computeVar )   \
        readAccessorName( type,var, _##var ) \
        setAccessor( type, var, setVar ) \
-(type)var { \
        if ( ![self _##var] )  { \
```

```
                [self setVar:[self computeVar]]; \
        }  \
        return [self _##var]; \
} \
```

The accessor builds on the macros from Example 3.6 but also has a parameter `computeVar` that defines the message to be sent to compute the result. When the getter is called, it checks whether it has a result. If it has a result, it just returns it; if not, it calls the `computeVar` method and then stores the result before returning it. Another less frequent accessor macro is the relay accessor that simply forwards the request to an instance variable.

Caching Caveats

> **There are only two hard things in Computer Science: cache invalidation and naming things.**
>
> **Phil Karlton**

With all this caching going on, it is important to remember that caching isn't without pitfalls of its own. In fact, a friend who became professor for computer science likes to ask the following question in his exams: "What is a cache and how does it slow down a computer?"

In the worst case of a *thrashing* cache with a hit rate of 0%, the cache simply adds the cost of maintaining the cache to the cost of doing the non-cached computation, and an easy way of reaching a 0% hit rate with the very simple cache policy used so far is invalidating a cache item just before it is needed again, for example, by having a linear or circular access pattern and a cache size that is smaller than the working set size, even by just a single item.

Additionally, caches use up memory by extending the lifetime of objects, and therefore increase the working set size, making it more likely to either push working-set items to a slower memory class (L1 cache to L2 cache, L2 cache to main memory, main memory to disk...) or even run out of memory completely on iOS devices, resulting in the process being killed. Global, transparent caches like CoreFoundation's `CFNumber` cache fare worst in this regard, because they have effectively no application-specific information helping them determine an appropriate size, leading to caches with arbitrary fixed sizes, like 12.

In addition, they can have puzzling bugs and side effects that, because of their transparent nature, are hard for clients to work around. Example 3.12 demonstrates how different constant strings and number objects allocated by completely different means but with the same numeric value turn out to be the same actual object, as shown by logging the object pointer with the "%p" conversion directive.

Example 3.12 Globally uniqued Foundation string and number objects in 32 bit

```
#import <Foundation/Foundation.h>

NSString *b=@"hello world";
int main( int argc, char *argv[] ) {
    NSString *a=@"hello world";
    printf("NSStrings a=%p=b=%p\n",a,b);
    for ( int i=1; i<15; i++) {
        NSNumber *c=[NSNumber numberWithLongLong:i];
        CFNumberRef d=CFNumberCreate(NULL,  kCFNumberIntType,&i);
        printf("%d NSNumber: %p type: %s \
                CFNumberCreate: %p type: %s\n",
                i,c,[c objCType],d,[(id)d objCType]);
    }
    return 0;
}
cc -Wall -m32 -o uniqueobjs uniqueobjs.m -framework Foundation
./uniqueobjs
NSStrings a=0x6b014=b=0x6b014
11 NSNumber: 0x78e7ac30 type: i CFNumberCreate 0x78e7ac30 type: i
12 NSNumber: 0x78e7ac40 type: i CFNumberCreate 0x78e7ac40 type: i
13 NSNumber: 0x78e7ac60 type: q CFNumberCreate 0x78e7ab90 type: i
14 NSNumber: 0x78e7aba0 type: q CFNumberCreate 0x78e7abb0 type: i
```

At the time this test was run, the cutoff for the cache was 12, requests up to that value get a globally unique, cached object, whereas values larger than that result in an allocation. Also note that the `objCType` of all cached values is "i," a 32-bit integer, despite the fact that we specifically asked for a `long long`, type code "q". Once outside the cacheable area, the requested type is honored.

The reason for this odd behavior is that the cache used to always cache the first object created for a specific value, regardless of the type requested. So if the first request for the integer 5 was for a `long long`, then all subsequent requests for a "5" would return that `long long` NSNumber. However, this could and did break code that was not expecting a "q" (`long long`) type code in its `NSNumber` objects, for example, object serializers that used the type code and did not handle the "q" code! This bug was fixed by ignoring the requested type-code for the cached numbers and using "i" instead, which is in fact just as incorrect as the other case, but in practice appears to cause fewer problems. On the 64-bit runtimes, the cache is disabled because all these small integers are implemented as tagged pointers.

Another pitfall is the use of `NSDictionary` or `NSSet` instances to cache de-duplicate string objects. While they may reduce peak memory usage in some cases, they can also increase memory usage by unnecessarily and arbitrarily extending the lifetime of the stored strings. Furthermore, the fact that `NSString` objects have to be created before they can be tested for membership means that the CPU cost has

already been paid before the cache is tested, so the cache actually *increases* CPU use. The way to improve this situation is to create a cache that can be queried using C-strings, either with a custom cache or with a custom set of callbacks for `CFDictionary`.

Pitfall: Generic (Intermediate) Representations

One of the fun features of the NeXTStep system that is the ancestor of OS X and iOS was its programmable graphics and window system based on DisplayPostscript. Just as the transition to OPENSTEP brought us Foundation with the basic object model of `NSString`, `NSNumber`, `NSDictionary`, and `NSArray`, I happened to be working on a kind of "PostScript virtual machine" that redefined PostScript operators to return graphical objects in a structured format rather than paint them on the screen, similar to the "distillery" code that to the best of my knowledge still powers Adobe's Acrobat Distiller PostScript to PDF converter to this day.

As I looked at my fresh install of OPENSTEP, I noticed that the binary object sequence (BOS) format created by the interpreter's `printobject` command included numbers, dictionary, arrays, and strings, mapping perfectly onto the data types provided by the brand new Foundation framework! So all I had to do was create a generic mapper to convert BOS format to Foundation, access the information encoded in those Foundation objects, and use that information to populate my domain objects, which included paths, images, text, and various graphics state parameters such as colors, transformation matrices, font names, and sizes.

While this approach allowed me to construct a prototype graphical object reader reasonably quickly, the performance was "majestic." In a complete surprise, the limiting factor was neither the PostScript procedures that had to emulate the drawing commands and produce output, nor the serialization operator in the PostScript interpreter or the deserialization code, or even the somewhat pokey byte-oriented communications channel. No, the major limiting factor was the creation of Foundation objects, a factor I never would have thought of. After the shock of my disbelief wore off, I replaced the parts that had converted the BOS to Foundation objects with a simple cover object that kept the original data "as-is" but was able to access parts using a generic messaging interface. The parser then accessed this messaging interface instead of converted objects, and performance improved threefold.

This was the first time I learned the lesson that generic intermediate object representations, also known as data transfer objects, are just a Bad Idea™, at least if you care about performance and are using Objective-C. While the general principle holds true in other languages, Objective-C drives that message home with a particular vengeance because of the 1:5:200 performance ratio between basic machine operations, messaging, and object allocation.

Of course, I had to relearn that lesson a couple of times before it finally managed to stick, but the reason why it is true is actually pretty simple: a generic representation will usually have significantly more objects than a final object representation because

it needs to use dictionaries (object header + key and value storage) instead of plain old Objective-C objects (somehow the "POOO" acronym as analogous to Java's Plain Old Java Objects [POJO] never caught on), object keys where objects can use instance variable offsets, and object values where objects can use simple scalar primitive types. So not only will you be creating objects that are significantly more expensive individually, but you will also need to create many more of these objects. Multiplying out these two factors makes generic intermediate object representations pretty deadly for performance in Objective-C and Swift.

Alas, Apple also makes this anti-pattern extremely convenient, so it has become pretty much the default for accessing any sort of serialized representation. A typical example is JSON parsing, with the only method directly supported by the frameworks being converting the JSON to and from in-memory property lists, that is, Foundation collections, `NSNumber`, and `NSString` objects. Even the plethora of Swift JSON "parsing" examples that have sprung up on the Internet essentially all first call `NSJSONSerialization` to do the actual parsing and generation.

Arrays and Bulk Processing

When dealing with a collection of entities accessed by integer indexes, Foundation `NSArray` improves on plain C arrays (`array[index]`) with a number of convenient services: automatic handling of memory management (`retain` / `release`), automatic growth, sorting, searching, subarray generation, and a memory model that allows efficient addition and removal of objects at both ends of the array.

What's missing is a set of arrays of primitives types such as `float`, `double`, or `int` with similar sets of services, as it should be clear by now that wrapping the scalar values in `NSNumber` objects and sticking those in an `NSArray` will not perform particularly well. Such a wrapper is easy to write, and a number of them exist; for example, the author's `MPWRealArray`, the arrays in `FScript`, or `SMUGRealVector`.

As Figure 3.1 shows, the performance benefits of having a homogenous collection of scalars are overwhelming: Summing the values in an array filled with 10,000 numbers is 5 times faster than summing `NSNumbers` in an `NSArray` even if the individual numbers are accessed via a messages send, and 17 times faster when the array is asked to perform the operation in bulk.

The differences are even more pronounced for creating such an array and filling it with the values from 1 to 10,000: The homogenous array is 20 times faster than creating the same values as an `NSArray` of `NSNumbers`, even when every real value is added inefficiently using a separate message send. Moving to bulk operations, where the loop is executed inside the real array, takes the difference to a factor of 270.

Better yet, representing the numbers as a contiguous C array of floats allows us to use the vector processing tools built into OS X such as vDSP library. Using vDSP functions, summing using the `MPWRealArray` code in Example 3.13 becomes yet another 10 times faster than even the bulk processing scalar code, bringing the performance relative to the `NSArray` + `NSNumber` combination to 1.3 μs.

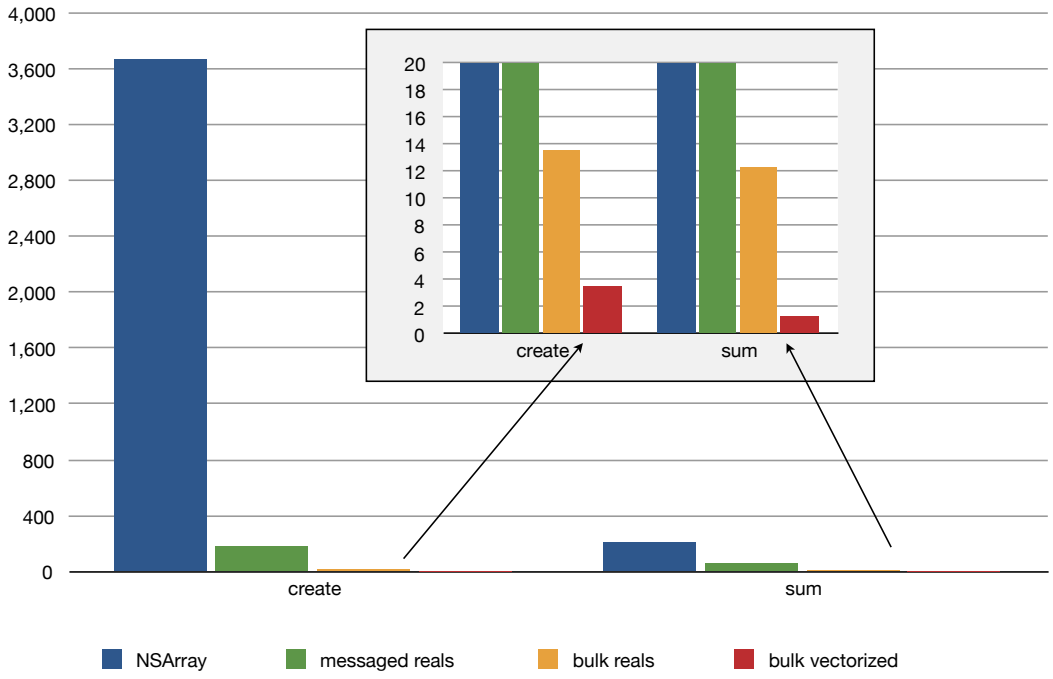

Figure 3.1 Time to create and sum a numeric 10,000-element array (microseconds)

Example 3.13 Summing using vDSP

```
@interface MPWRealArray : NSObject
{
    int capacity;
    NSUInteger count;
    float *floatStart;
}

-(float)vec_reduce_plus
{
    float theSum=0;
    vDSP_sve ( floatStart, 1, &theSum, count );
    return theSum;
}
```

This takes the time for a single addition to only 0.13 ns, showing off the true power of our computing buzz saws. Creation is also another 4 times faster when adding vector functions bulk real processing; at 3.5 μs for the entire array, this is now 1,000 times faster than creating an equivalent `NSArray` of `NSNumbers`.

To bring this into perspective, a factor of 1,000 is close to the difference between the clock speed of a Mac Pro and that of the author's Apple][+ with its 1-MHz 6502 processor!

Of course, you don't have to use an array object. If performance is critical, you can also always use a plain C array, which can store any type of primitive, struct, or object. However, this method is without the conveniences of growability, taking care of reference counting, safe access, or convenience methods.

Swift crosses the `NSArray` of Foundation with plain C arrays to get the Swift array type: It provides most or all of the conveniences of an array object like `NSArray` or `MPWRealArray`, while at the same time using generics to be applicable to all types like a plain C array. The big advantage is that the temptation to use an array of `NSNumber` objects or similar when you just wanted to store some integers or reals has lessened dramatically. You just write `[Int]` when you want an array of integers, or with type inference provide a literal array of integers. Access times are reasonable and you still get growability and bounds checking.

The downside is that while it is harder to get unreasonably bad performance, it is also currently hard to get the best possible performance. In my tests, the Swift summation code was around 4 to 5 times slower than the equivalent Objective-C code, though that is probably going to change as the optimizer is improved. Speaking of the optimizer, unoptimized Swift array code was a shocking 1,000 times slower than reasonably optimized Objective-C code, on par with the object-based code provided here as a counterexample despite using primitives.

It is not exactly clear how Swift manages to be this slow without the optimizer, the typical factor for C code being in the 3 to 4 range, and Objective-C often not affected much at all. What *is* clear is that with this type of performance difference, unoptimized debug builds are probably out of the question for any code that is even remotely performance sensitive: when a task taking 100 ms optimized would take almost 2 minutes in a debug build, you can't really debug.

Dictionaries

The use of strings as keys mentioned in the "Strings" section of this chapter is usually in conjunction with some sort of key-value store, and in Cocoa this is usually an `NSDictionary`. An `NSDictionary` is a hash table mapping from object keys to object values, so features average case constant $O(k)$ read access time.[3]

However, the generic nature of the keys means that, as Table 3.5 and Table 3.6 show, k is a relatively large number in this case, 23 to 100 ns or 10 to 50 times slower than a message-send. Furthermore, `NSDictionary` requires primitive types to be wrapped in objects, with the performance consequences that were discussed in the "Primitive Types" section in this chapter.

3. The average constant access time isn't guaranteed by the documentation, but it has always been true, and the `CFLite` source code available at `http://opensource.apple.com` confirms it.

Table 3.5 Cost of dictionary lookups by type of stored and lookup key, large key

	Time to lookup by lookup key type (ns)			
Stored keys	**Constant string**	**Regular string**	**Mutable string**	**CoreFoundation**
Constant string	35	78	78	83
Regular string	78	80	80	85

Table 3.5 shows the more general cost of dictionary access, which is around 80 ns per read if you don't have hash collisions (when two or more keys map onto to the same slot in the hash table). A single collision adds another 20 ns or so. The only time that deviates from the roughly 80 ns standard is when you have constant strings both as the keys of the dictionary and the key to look up. In this case, the lookup can be more than twice as fast, probably due to the fact that constant strings can be compared for equality using pointer equality due to being uniqued.

For small keys up to 7 characters, the tagged pointer optimization introduced in OS X 10.10 also helps. As with constant strings, pointer comparison is sufficient here because the value is stored in the pointer, but only if both strings are of the same type, either both tagged pointers or both constant strings. Table 3.6 shows this effect: When the key classes match, both constant strings and tagged pointer strings take around 22 ns for a single lookup, but there is no benefit if the classes do not match.

So in order to get optimized dictionary performance, you need to make sure that the class of the key used to store the value into the dictionary and the class of the key used to retrieve the value match. If a string literal (`@"key"`) was used to store the value, it is best if a string literal is used to retrieve it.

If you cannot use a string literal on retrieval, your keys are short enough to fit in a tagged pointer string. And if you retrieve values more often than you store them, it may be helpful to convert the keys you use to store the values to tagged pointer strings as was shown in Example 3.3: first do a mutable copy of the original key and then a copy of the mutable copy. This will make your retrievals 2 to 4 times faster, depending on the circumstances. (The detour via a mutable copy is necessary because

Table 3.6 Cost of dictionary lookups by type of stored and lookup key, small key

	Time to lookup by lookup key type (ns)			
Stored keys	**Constant string**	**Tagged string**	**Mutable string**	**CoreFoundation**
Constant string	23	64	52	74
Tagged string	51	21	44	47

all immutable strings, including constant strings, will just return `self` when asked to `copy` themselves.)

Even with these optimizations for small and constant strings, it is therefore best to look for alternatives to `NSDictionary` when there is a chance that performance may be relevant, unless what is needed exactly matches the capabilities of `NSDictionary`. The vast majority of dictionary uses are much more specialized, for example, using only fixed strings as keys, and often having only a bounded and small set of relevant or possible keys. The XML parser in Chapter 4 uses two types of specialized dictionaries: `MPWXMLAttributes` for storing XML attributes that supports XML semantics such as ordering and multiple values for a key and is tuned for those use-cases, and the `MPWSmallStringTable` that maps directly from a predefined set of C strings to objects.

`MPWSmallStringTable` does not use a hash-table but operates directly on the byte-by-byte character representation, trying to eliminate nonmatching strings as quickly as possible. While it is also approximately 4 times faster than `NSDictionary` for the small constant string cases that `NSDictionary` is specially optimized for, its main use is in dealing with externally generated string values, and for this use-case it is anywhere from 5 to 15 times faster than `NSDictionary`.

Swift dictionaries, which are and use value types, and which benefit from generics, are obviously faster than heavyweight `NSDictionary` objects that use slow objects, right? Alas, that is currently not the case: In all my tests, Swift `Dictionary` access was significantly slower than even `NSDictionary`. For example, a `[String:Int]` map, which maps two value types and is therefore unencumbered by any legacy Objective-C objects and "slow" dynamic dispatch, took anywhere from 140 ns to 280 ns per lookup, depending mostly on whether the key was a string literal or was provided externally, respectively. This slowdown of 3 to 7 times, compared to `NSDictionary` (and 17 to 25 times compared to `MPWSmallStringTable`) was largely independent of compiler flags, though as typical of Swift, compiling without any optimization causes a significant slowdown.

The easiest alternative to `NSDictionary` is to just define objects and use plain messaging to access their contents, especially when the dictionary in question has a reasonably small and mostly fixed set of keys. Not only is the first line of Example 3.14 anywhere from 10 to 100 times faster, it is also a cleaner design because message names are scoped by their class, whereas dictionary keys pollute the global namespace and must therefore use unwieldy long names.

Example 3.14 One of these is 100 times faster

```
[myParagraph setLeading: 10.0];
[myParagraph setAttribute:[NSNumber numberWithFloat:10.0]
                   forKey:kMPWParaStyleLeading];
```

Why might one prefer to use a dictionary instead of an object? With the nonfragile instance variables of the Objective-C 2.0 64-bit runtime and associated

storage, future-proofing against additional required instance variable is no longer an issue. Potentially sparsely populated objects can be handled by partitioning into one or more subobjects to which the corresponding message are delegated and that are allocated as needed.

As long as clients are provided with a messaging interface, the implementation can be varied and optimized to fit. While it is tempting to provide a key-value based interface instead, the flexibility it appears to offer is an illusion. Once an `NSDictionary`-like key-value interface is provided to clients, the performance characteristics are pretty much locked in, because mapping from `NSString` external keys to messages or instance variable offsets internally is just about as costly in terms of CPU usage as an `NSDictionary` proper. So instead, if an `NSDictionary`-based internal representation is desired, it can and probably should be wrapped in an object that maps its accessor messages to the dictionary.

The Macro in Example 3.15 allows you to add a messaging interface to a key in a dictionary either statically by writing `dictAccessor( var, setVar , [self _myDict] )` in your implementation, where `var` is the key and `[self _myDict]` is an expression that returns the dict to be used, or dynamically at runtime, using the `imp_implementationWithBlock()` function to turn a block into a method implementation.

Messaging

> **I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea. The big idea is "messaging."**
>
> **Alan Kay**

Whereas objects in Objective-C are little more than slightly specialized C structures, the efficient and highly flexible message dispatch system is at the heart of Objective-C. It combines true object encapsulation and the dynamicism of languages such as Ruby or Smalltalk. Not only are Objective-C messages powerful, they are also relatively cheap, only around twice the cost of a C function call and within an order of magnitude of basic machine operations. Even an unoptimized message send is around 10 times faster than keyed access via `NSString`, and 50 times faster than object-creation, despite the fact that in the current Objective-C runtime, an Objective-C selector, is really just a C string.

The reason that messaging via string selectors is so quick is that the compiler, linker, and runtime conspire to guarantee that every C string representing an Objective-C selector has a unique address, and therefore the Objective-C messenger function `objc_msgSend()` does not have to concern itself with the string that the selectors point at, but just uses the pointer itself as an uninterpreted unique integer

Example 3.15 Generate and test dictionary-backed accessor method (statically or dynamically)

```
#import <Foundation/Foundation.h>
#import <objc/runtime.h>
#define dictAccessor( objectType, var, setVar, someDict ) \
    -(objectType*)var { return someDict[@""#var]; } \
    -(void)setVar:(objectType*)newValue { \
        someDict[@""#var]=newValue;\
     }\
@interface MyObject : NSObject
@property (retain) NSMutableDictionary *dict;
@end
@interface MyObject(notimplemented)
@property (retain) NSString *a;
@property (retain) NSString *b;
@end
@implementation MyObject
-(instancetype)init {
  self=[super init];
  self.dict=[NSMutableDictionary new];
  return self;
}
-(void)addDictAccessorForKey:(NSString*)key
{
  SEL selector=NSSelectorFromString( key );
  id (^block)()=^{
    return self.dict[key];
  };
  imp=imp_implementationWithBlock( block );
   class_addMethod([self class], selector, imp , "@:");
}
dictAccessor( NSString, b, setB , self.dict )
@end
int main()
{
        MyObject *m=[MyObject new];

        [m addDictAccessorForKey:@"a"];
        m.dict[@"a"]=@"Hello";
        m.b=@"World!";
        NSLog(@"m.a: %@ m.b: %@",m.a,m.b);
        return(0);
}
```

value. In fact, as Brad Cox writes in *Object-Oriented Programming: An Evolutionary Approach*, this selector-uniquing process was the main driver for converting Objective-C from a set of C macros to an actual preprocessor, which then made it possible to create a distinct syntax.

On Mac OS X 10.11 with Xcode 7.3.1, the code in Example 3.16 prints `selector: 'hasPrefix:'`, but the compiler already warns that `cast of type 'SEL' to 'char *' is deprecated; use sel_getName instead`. In the GNU runtime, selectors are structure that reference both the message name and its type encoding.

Example 3.16 Printing a selector as a C string using Apple's runtime

```
#import <Foundation/Foundation.h>

int main()
{
  SEL a=@selector(hasPrefix:);
  printf("selector: %s\n",(char*)a);
  return 0;
}
```

IMP Caching

Although developers new to Objective-C tend to worry most about message sending, for example, compared to C++ virtual function invocation, the Objective-C messenger function objc_msgSend() (or objc_msg_lookup() in GNU-objc) has been highly optimized and is usually not a bottleneck.

In the rare cases that it does become a factor, it is possible to retrieve the function pointer from the runtime and call that instead. The technique is known as IMP caching because the type definition of an Objective-C method pointer is called an IMP (implementation method pointer, or just IMPlementation). IMP caching can be useful in a tight loop with a fixed receiver when the method itself is trivial and therefore message dispatch is a major contributor. Example 3.17 shows a greater than 2.5-times improvement in runtime from 2.8 ns to 1.08 ns after subtracting loop overhead.

Example 3.17 Replacing a plain message send with an IMP-cached message send

```
#import <MPWFoundation/MPWFoundation.h>
@interface MyInteger : NSObject
@property (assign) int intValue;
@end

@implementation MyInteger
@end
```

```
int main()
{
  MyInteger *myObject=[MyInteger new];
  int a=0;
  myObject.intValue=42;
  for ( int i=0; i<1000; i++) {
    a+=[myObject intValue];
  }

  IMP intValueFun=[myObject methodForSelector:@selector(intValue)];
  for ( int i=0; i<1000; i++) {
    a+=(int)intValueFun( myObject, @selector(intValue) );
  }
}
```

Due to the dynamic nature of Objective-C, there is no automatic way of determining at compile time whether this optimization is safe, which is one reason the Objective-C compiler doesn't do it for you. Fortunately, it is usually very easy for a developer to make that determination. While there are numerous ways for the IMP to change during execution (for example, loading a bundle that includes a category, and using runtime functions to add, remove, or change method implementations or even change the class of the object in question), all of these are rare events that happen fairly predictably.

It is the developer's job to ensure that either none of these events happen, or alternately, that they do not have an impact on the computation.

A special case that needs to be considered when doing `IMP` caching is the `nil` receiver. The Objective-C messenger quietly ignores messages to `nil`, simply returning zero instead of dispatching the message. This short-circuiting protects receivers from having to worry about a `nil self` pointer, and sender from having to special case `nil`-receivers. `IMP` caching breaks this protection on several counts: If the receiver is `nil` when requesting the `IMP`, a `NULL` function pointer will be returned, and invoking such a `NULL` function pointer will crash the program. On the other hand, if a correct function pointer was obtained from an earlier, non-nil object pointer, calling that function pointer will call a method with a `nil self` pointer. Any instance variable access from within that method will also crash the program.

So you will need to ensure both that you are not getting a NULL IMP and that you don't call an IMP with a nil receiver.

IMP caching can be particularly useful when sending messages to "known" objects such as delegates or even `self`. Example 3.18 shows part of the actual header of the object cache discussed in the "Mutability and Caching" section of this chapter. In addition to the cache itself (`objs`, `cacheSize`) and the current pointer into the cache `objIndex`, it also maintains `IMP` pointers for all the message sent in the `-getObject` method from Example 3.10, allowing the actual `-getObject` to run

without once invoking the messenger. In addition, it makes the `IMP` for the `-getObject` method itself available in a `@public` instance variable, along with a `GETOBJECT()` C-preprocessor macro to invoke it. The `GETOBJECT` macro is actually slightly less code to write than a normal `alloc-init-autorelease`, is 8% faster even with a cache miss, is 15 times faster with a cache hit, and last but not least decouples the user of the cache from the specific class used.

Example 3.18 Definition and use of an object cache for integer objects

```
@interface MPWObjectCache : MPWObject
{
    id          *objs;
    int         cacheSize,objIndex;
    Class       objClass;
    SEL         allocSel,initSel,reInitSelector;
    IMP         allocImp,initImp,reInitImp,releaseImp;
    IMP         retainImp,autoreleaseImp;
    IMP         retainCountImp,removeFromCacheImp;
    @public
    IMP         getObject;
}

+(instancetype)cacheWithCapacity:(int)newCap class:(Class)newClass;
-(instancetype)initWithCapacity:(int)newCap class:(Class)newClass;
-getObject;
#define GETOBJECT( cache )
    ((cache)->getObject( (cache), @selector(getObject)))
...
@end
integerCache=[[MPWObjectCache alloc] initWithCapacity:20
          class:[MPWInteger class]];
MPWInteger *integer=GETOBJECT( integerCache );
[integer setIntValue:2];
```

If IMP caching is insufficient and you have the source code of the method you need to call available, you can always turn it into a C function, an inline function, or even a preprocessor Macro.

Considering how little of a problem dynamic dispatch is in practice, and how easy it is to remove the problem in the rare cases it does come up, it is a little surprising how much emphasis the Swift team has placed on de-emphasizing and removing dynamic dispatch from Swift for performance reasons.

Forwarding

While close to C function call speeds on one end, Objective-C messages are flexible enough to take the place of reified messaging and control structures on the other end. For example, Cocoa does not have to use the *Command* pattern because messages carry enough runtime information to be reified, stored, and introspected about so something like the `NSUndoManager` can be built using the fast built-in messaging system.

For your own projects, I would always recommend mapping any requirements for dynamic runtime behavior onto the messaging infrastructure if at all possible, and with a full reflective capabilities what is possible is very broad. The code in Example 3.19 will execute the message to the object in question as a Unix shell command, so `[object ls]` will execute the `ls` command, and `[object date]` the `date` command. A more elaborate example would translate message arguments to script arguments.

Example 3.19 Mapping sent messages to shell commands

```
#import <Foundation/Foundation.h>
@interface Shell:NSObject
@end
@interface Shell(notimplemented)
-(void)ls;
@end
@implementation Shell

-(void)forwardInvocation:(NSInvocation*)invocation {
  system( [NSStringFromSelector( [invocation selector])
           fileSystemRepresentation] );
}
-(void)dummy {}
-methodSignatureForSelector:(SEL)sel
{
   NSMethodSignature *sig=[super methodSignatureForSelector:sel];
   if (!sig) {
      sig=[super methodSignatureForSelector:@selector(dummy)];
   }
   return sig;
}
@end

int main()
{
    Shell *sh=[Shell new];
    [sh ls];
    return 0;
}
```

Example 3.20 reads the file that is named by the sent message instead of executing it, and perhaps somewhat more realistically, Example 3.21 looks up the selector in a local dictionary.

Example 3.20 Mapping sent messages to file contents

```
#import <Foundation/Foundation.h>
@interface Filer:NSObject
@end
@interface Filer(notimplemented)
-(NSString*)hello;
@end
@implementation Filer

-(void)forwardInvocation:(NSInvocation*)invocation {
  NSString *filename=NSStringFromSelector( [invocation selector]);
  NSString *contents=[[NSString alloc]
                              initWithContentsOfFile:filename
                              encoding:NSISOLatin1StringEncoding
                              error:nil];
  [invocation setReturnValue:&contents];
}
-(NSString*)dummy { return @""; }
-methodSignatureForSelector:(SEL)sel
{
   NSMethodSignature *sig=[super methodSignatureForSelector:sel];
   if (!sig) {
      sig=[super methodSignatureForSelector:@selector(dummy)];
   }
   return sig;
}
@end

int main()
{
    Filer *filer=[Filer new];
    NSLog(@"filer: %@",[filer hello]);
    return 0;
}
```

Example 3.21 Mapping sent messages to dictionary keys

```
-(void)forwardInvocation:(NSIvocation*)invocation {
     id result=[[self dictionary] objectForKey:
               NSStringFromSelectr([invocation selector])];
     [invocation setReturnValue:&result];
}
```

Uniformity and Optimization

Although there is no actual performance benefit for the *implementations* of Examples 3.19 to 3.21, the benefit comes from using the fastest plausible *interface*, an interface that can be kept the same all the way from reading files (3.20) via using runtime introspection to look up keys (3.21), generating accessors to a keyed store at runtime or compile time (3.15) or switching to an accessor for an actual instance variable, and finally IMP caching that message send. You don't have to start out fast, but you have to use interfaces that allow you to become fast should the need arise.

The more I have followed Alan's advice to focus on the messages, the better my programs have become, and the easier it has been to make them go fast.

Methods

Objective-C methods generally fall into two rough categories: lean and mean C data manipulation on one hand and high-level coordination using message sends on the other.

For the data-manipulation methods, all the usual tricks in the C repertoire apply: moving expensive operations out of loops (if there is no loop, how is the method taking time?), strength reduction, use of optimized primitives such as the built-in memory byte copy functions or libraries such as vDSP, and finding semantically equivalent but cheaper replacements. Fortunately, the compiler will help with most of this if optimization is turned on. In fact, instead computing the end-results of the loops, LLVM/clang managed to optimize away most of the simple loops from our benchmark programs unless we specifically stopped it.

In order to keep data manipulation methods lean and mean, it is important to design the messaging interface appropriately, for example, passing all the data required into the method in question, rather than having the method pull the data in from other sources.

High-level coordination methods should generally not be executed very often and therefore do not require much if any optimization. In fact, I've had excellent performance results even implementing such methods in interpreted scripting languages. A method triggering an animation lasting half a second, for example, will take less than 0.2% of available running time even if it takes a full millisecond to execute, which simply won't be worth worrying about.

Pitfall: CoreFoundation

One of the recurring themes in this chapter has been leveraging C for speed and making careful tradeoffs between the "C" and the "Objective" parts of the language in order to get a balance between ease of use, performance, and decoupling and dynamicism that works for the project at hand.

However, it is possible to get this terribly wrong, as in the case of CoreFoundation. CoreFoundation actually throws out the fast and powerful bits of Objective-C (messaging, polymorphism, namespace handling) and manages to

provide a cumbersome monomorphic interface to the slow bits (heap allocated objects). It then encourages the use of dictionaries, which are an order of magnitude slower still. The way CoreFoundation provides largely monomorphic interfaces to CoreFoundation objects that actually have varying internal implementations means that each of those functions, with few exceptions, has to check dynamically what representation is active and then run the appropriate code for that representation. You can see this in the OpenSource version of CoreFoundation available at http://opensource.apple.com/source/CF.

An Objective-C implementation leaves that task to the message dispatcher, meaning that both method implementations can be clean because they will only be called with their specific representation, also making it easier to provide a greater number of optimized representations.

While I've often heard words to the effect that "our code is fast because it just uses C and CoreFoundation and is therefore faster than it would be if it were to use Objective-C," this appears to be a myth. I've never actually found this claim to be true in actual testing. In fact, in my testing, pure Objective-C equivalents to CoreFoundation objects are invariably faster than their CoreFoundation counterparts, and often markedly so. Sending the `-intValue` message shown in Example 3.17 is already 30% faster than calling the CoreFoundation `CFGetIntValue()` function, despite the message-passing overhead. Dropping down to C using IMP caching makes it over 3 times faster than the CoreFoundation equivalent.

The same observations were made and documented when CoreFoundation was first introduced, with users noticing significant slowdowns compared to the non-CoreFoundation OPENSTEP Foundation (apps twice as slow on machines that were supposed to be faster[4]). This obviously does not apply to the NSCF★ classes that Apple's Foundation currently uses; these cannot currently be faster than their CoreFoundation counterparts because they call down to CoreFoundation.

Multicore

As we saw in Chapter 1, Moore's Law is still providing more transistors but no longer significant increases in clock frequency or performance per clock cycle. This shift in capabilities means that our single-threaded programs are no longer getting faster just by running them on newer hardware. Instead, we now have to turn to multithreading in order to take advantage of the added capabilities, which come in the form of additional cores. Getting multithreading right is a hard problem, not just due to the potential for race conditions and deadlocks, but also because the addition of thread management and synchronization actually adds significant overhead that can be difficult to break even on, despite the additional CPU resources that are unlocked with multithreading.

4. http://www.cocoabuilder.com/archive/cocoa/20773-does-ppc-suck-or-has-apple-crippled-cocoa.html#20773

Due to the pretty amazing single-core performance of today's CPUs, it turns out that the vast majority of CPU performance problems are not, in fact, due to limits of the CPU, but rather due to suboptimal program organization.[5] I hope the factors 3 to 4, 10 to 20, and 100 to 1,000 of often easily attainable performance improvements I have presented so far will convince you to at least give the code-tuning option serious consideration before jumping into multithreading, which at best can achieve a speedup to the number of cores in the system—and this is only for perfectly parallelizable, so-called "embarrassingly parallel" problems.

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}} \tag{3.1}$$

Amdahl's Law (Equation 3.1), relating the potential speedup (S) due to parallelization with N cores ($S(N)$) to the fraction of the program that can be parallelized (P) shows that the benefit of newer cores peters off very quickly when there are even small parts of the program that cannot be parallelized. So even with a very good 90% parallelizable program, going from 2 to 4 cores gives a 70% speedup, but going from 8 to 12 cores only another 21%. And the maximum speedup even with an infinite number of cores is factor 10. For a program that is 50% parallelizable, the speedup with 2 cores is 33%, 4 cores 60% and 12 cores 80%, so approaching the limit of 2.

While I can't possibly do this topic justice here, it being worthy of at least a whole book by itself, I can give some pointers on the specifics of the various multithreading mechanisms that have become available over the years, from `pthreads` via `NSThread` and `NSOperationQueue` all the way to the most recent addition, Grand Central Dispatch (GCD).

Threads

Threading on OS X is essentially built on a kernel-thread implementation of POSIX threads (`pthreads`). These kernel threads are relatively expensive entities to manage, somewhat similar to Objective-C objects, only much more so. Running a function `my_computation( arg )` on a new POSIX thread using `pthread_create`, as in Example 3.22, takes around 7 μs to of threading overhead on my machine in addition to the cost of running `my_computation()` by itself, so your computation needs to take at least those 7 μs to break even, and at least 70 μs to have a chance of getting to the 90% parallelization (assuming we have a perfect distribution of tasks for all cores).

Creating a new thread using Cocoa's `NSThread` class method `+detachNewThreadSelector:…` adds more than an order of magnitude of overhead to the tune of 120 μs to the task at hand, as does the `NSObject` convenience method `-performSelectorInBackground:…` (also Example 3.22).

5. James R. Larus. "Spending Moore's dividend," *Communications of the ACM* No. 5 (2009).

Taking into account Amdahl's Law, your task should probably take at least around 1 ms before you consider parallelizing, and you should probably consider other optimization options first.

Example 3.22 Creating new threads using pthreads, Cocoa NSThread, or convenience messages

```
pthread_create( &pthread, attrs, my_computation, arg );
[NSThread detachNewThreadSelector:@selector(myComputation:)
          toTarget:self
          withObject:arg];
[self performSelectorInBackground:@selector(myComputation:)
      withObject:arg];
```

So, similar to the balancing of OOP vs. C, getting good thread performance means finding independent tasks that are sufficiently coarse-grained to be worth off-loading into a thread, but at the same time either sufficiently fine-grained or uniformly sized that there are sufficient tasks to keep all cores busy.

In addition to the overhead of thread creation, there is also the overhead of synchronizing access to shared mutable state, or of ensuring that state is not shared—at least, if you get it right. If you get it wrong, you will have crashes, silently inconsistent and corrupted data, or deadlocks. One of the cheapest ways to ensure thread-safe access is actually `pthread` thread-local variables, accessing to such a variable via `pthread_getspecific()` is slightly cheaper than a message send. But this is obviously only an option if you actually want to have multiple separate values, instead of sharing a single value between threads.

In case data needs to be shared, access to that data generally needs to be protected with `pthread_mutex_lock()` (43 ns) or more conveniently and safely with an Objective-C `@synchronized` section, which also protects against dangling locks and thus deadlocks by handling exceptions thrown inside the `@synchronized` section. Atomic functions can be used to relatively cheaply (at 8 ns, around 10 times slower than a simple addition in the uncontended case) increment simple integer variables or build more complex lock-free or wait-free structures.

Work Queues

Just like the problem of thread creation overhead is similar to the problem of object-allocation overhead, so work queues are similar to object caches as a solution to the problem: They reuse the expensive threads to work on multiple work items, which are inserted into and later fetched from work queues.

Whereas Cocoa's `NSOperations` actually take slightly longer to create and execute than a `pthread` (8 μs vs. 7 μs), dispatching a work item using GCD introduced in Snow Leopard really is 10 times faster than a `pthread`, at 700 ns per item for a simple static block, and around 1.8 μs for a slightly more complex block with arguments like the one in Example 3.23.

Example 3.23 Enqueuing GCD work using straight blocks

```
dispatch_async(
     dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
      ^{ [self myComputation:arg];} );
```

I personally prefer convenience messages such as the `-async` Higher Order Message (HOM),[6] which simplifies this code to the one shown in Example 3.24 at a cost of an extra microsecond.

Example 3.24 Enqueuing GCD work using HOM convenience messages

```
[[self async] myComputation:arg];
```

In the end, I've rarely had to use multithreading for speeding up a CPU-bound task in anger, and chances are good that I would have made my code slower rather than faster. The advice to never optimize without measuring as you go along goes double for multithreading. On the flip side, I frequently use concurrency for overlapping and hiding I/O latencies (Chapter 12) or keeping the main thread responsive when there is a long running task, be it I/O or CPU bound (Chapter 16). I've also used libraries that use threading internally, for example, the vDSP routines mentioned earlier or various image-processing libraries.

Mature Optimization

> **We should forget about small efficiencies, say about 97% of the time; premature optimization is the root of all evil.**
>
> **D.E. Knuth**

Optimizing Objective-C programs is, in the end, not necessarily hard. In fact, this very amenability to optimization in general and late-in-the-game optimization in particular is a large part of what makes this language popular with expert programmers: you really can leave the "small efficiencies," a few of which we've shown, for later.

Although Knuth's quote above is well-known, what is *less* well-known is that it is just an introduction to extolling the importance and virtues of optimization. It continues as follows:

> Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified.

6. Implementation can be found at https://github.com/mpw/HOM.

And the section before the one in question couldn't be more different:

> The conventional wisdom shared by many of today's software engineers calls for ignoring efficiency in the small; but I believe this is simply an overreaction to the abuses they see being practiced by penny-wise-and-pound-foolish programmers, who can't debug or maintain their "optimized" programs. In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal; and I believe the same viewpoint should prevail in software engineering. Of course I wouldn't bother making such optimizations on a one-shot job, but when it's a question of preparing quality programs, I don't want to restrict myself to tools that deny me such efficiencies.
>
> "Structured Programming with Go To Statements," Knuth, 1974.

The quote is embedded in the paper "Structured Programming with Go To Statements" from 1974, which is largely about achieving better performance via the use of go to statements. It is in fact, in large part, an advocacy piece *for* program optimization, not against it, containing such gems as the idea that engineers in other disciplines would be excluded from practicing their profession if they gave up performance as readily as programmers.

What makes Objective-C so powerful is that once you have the information as to what needs optimization, you can really pounce, smash-bits, and exploit all the hardware has to give. Both until that point and for the parts that don't need it, you can enjoy the remarkable productivity of a highly dynamic object-oriented language.

Swift takes a different approach: make everything much more static up-front and then let the compiler figure it out. While superficially sound, this approach inverts Knuth's dictum by making microperformance a deciding factor in not just application modeling, but language design. In addition to the approach being questionable in principle, it currently just doesn't work: Swift is not just slower than optimized Objective-C, it is often significantly slower than non-optimized Objective-C, without any further recourse than waiting for the compiler to get better or rewriting your code in C. So that questionable premature optimization doesn't even pay off.

That said, a little bit of structural forethought and planning is extremely helpful in order to enjoy the benefits of late optimization: You should have an idea of the order of magnitude of data you will be dealing with (one, a thousand, a million?), what operations you need to support, and whether the machine you are targeting can handle this amount of data, at least in principle.

As you are designing the system, keep in mind the asymmetric 1:5:50:200 relationship for primitive operations : messaging : key-value access : object creation that we have illuminated throughout this chapter. With that in mind, see if your most numerous pieces of data can be mapped to primitives, and try to keep your interfaces as message-centric as possible. The messaging system has a nice sweet spot in the relationship between cost and expressiveness.

The arguments of those messages should be as simple (primitive types preferred) and expressive as possible. Large-volume data should be contained in bulk objects and

hidden behind bulk interfaces. Key-value stores, if needed, should be hidden behind messaging interfaces and temporary objects should be avoided, especially as a requirement for an interface. If temporary objects can't be avoided, try to keep your APIs defined in such a way that you will be able to "cheat" with object caches or other techniques for reusing those objects when the time comes.

Fortunately, these measures tend to simplify code, rather than make it more complicated. Simpler, smaller, well-factored code is not only often faster than complicated code, because code that isn't there doesn't take any time to run, it also makes a much better basis for future optimization efforts because modifying a few spots will have a much greater impact.

This page intentionally left blank

Index

Numbers

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z