High Performance Programming for OS X

Timothy Ritchey

High Performance Programming for OS X
by Timothy Ritchey

Table of Contents

List of Tables

List of Figures

Chapter 1. High Performance Programming and OSX

It took Apple a long time, and the purchase of an ex-founder's company to do it, but they finally have a cutting-edge operating system running on their hardware. At the core of OS X is a very old technology--Unix. OS X's combination of advanced features and Unix underpinning has brought many developers to the Apple platform. In fact, it is very easy for a programmer to bring their existing Unix bag of tricks to the OS X table and be very productive.

what does this do for them though? They would be missing out on some great technology like: obj-c, DO, Cocoa, Quartz, Mach. so, that is what this book aims to cover.

We are taught to code first, then profile, then optimize. This leaves out the critical step of program design, which can have just as much, or more impact on the *ultimate* speed our program is able to achieve. By not thinking about performance at all, we risk creating a program that is unable to achieve its goals.

It is important to strike the right balance.

1.1. Design

foo

1.2. Profiling

foo

1.3. Optimization

foo

Chapter 2. Algorithms and Design Patterns

2.1. Basic Principles

todo

2.1.1. Introduction to Algorithms

todo

2.1.2. Algorithms Versus Design Patterns

todo

2.2. Algorithms

todo

2.3. Patterns

todo

2.3.1. Class Clusters

show how to use class clusters as a design pattern for overcoming the overloading limitation.

2.3.2. Surrogate Objects

show how to use the \cmdline{forwardInvocation:} as a proxy for real objects. This can be used when you want to implement lazy optimization, and have a stand-in until the real items of interest are loaded from the system. (see p151 in the Objective-C manual).

2.4. Anti-patterns

todo

Chapter 3. Profiling

3.1. Why Profile?

todo

3.2. Basic Profiling Tools

Here’s a handy function to get millisecond accurate timestamps using the kernel UpTime() function, which avoids the overhead of NSDate.

```
#import <DriverServices.h>
unsigned int UpTimeInMilliseconds()
{
 unsigned int result;
 Nanoseconds n = AbsoluteToNanoseconds(UpTime());
 unsigned long long int i = n.hi;
 i <<= 32;
 i += n.lo;
 i /= 1000000;
 result = i;
 return result;
}
```

See also the documentation at MP.97.html (search for "UpTime" documentation in Project Builder) for more code to create very accurate timestamps.

3.2.1. top

If you use "top -d" (or "top -ud"), Darwin’s top won’t consume that many cycles either (it’s still slightly more than under the other OS’es though). It’s the gathering of memory statistics that consumes so much time. On darwin-development, Apple engineers also always say this is because traversing the mach memory maps is a lot of work. Maybe the memory mapping under FreeBSD (and Linux) is in fact much simpler than under mach... Anyway, the source of all things involved is available, so nothing prevents you from finding out what the real cause is (except for time, probably :)

3.2.2. Sampler

todo

3.2.3. QuartzDebug

todo

3.2.4. Looking Forward

We will look at programs such as ThreadViewer, MallocDebug and ObjecAlloc, as Well as OmniGroup's ObjectMeter in later chapters as we discuss the programming areas that they touch upon.

3.3. Advanced Profiling Tools

todo

3.3.1. CHUD Kernel Extensions

todo

3.3.2. Using Sharkiri

todo

3.3.3. MONster

todo

3.4. Sample Profiling Cases

todo

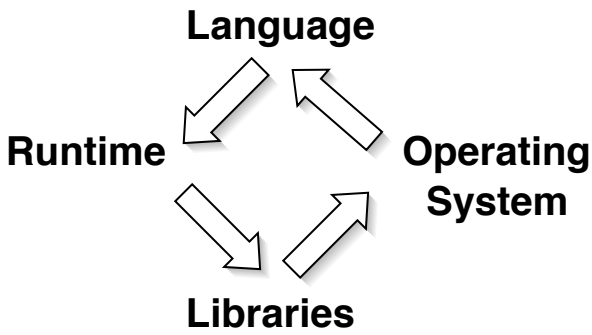
3.5. Detecting Anti-patterns

todo

Chapter 4. Under the Hood with Objective-C

Just like knowing where the gas and brake pedal are located doesn't automatically make you a formula-one driver, knowing how to program in any particular language doesn't necessarily give you the wherewithal to create high performance programs. Sometimes you have to get under the hood to get a good idea of what it takes to go fast.

Superficially, Objective-C is a very easy language to learn and use. Of course, nothing comes for free. The simplicity of programming for OS X using Objective-C masks the various elements that must come together to make it all work.



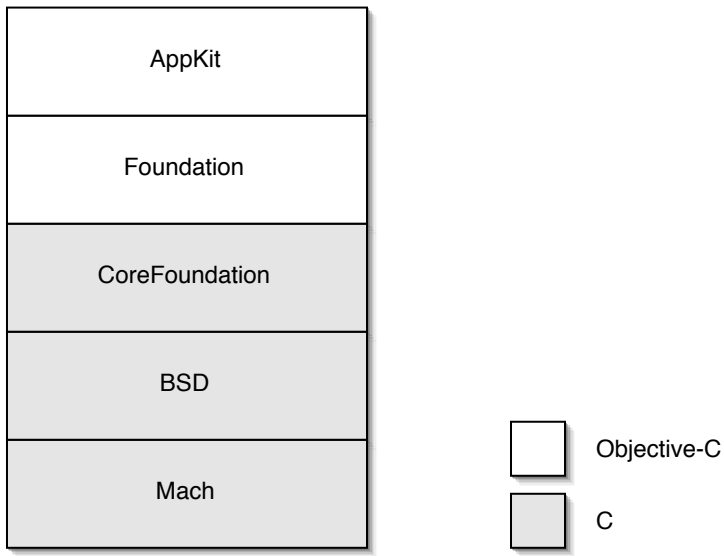
In the above diagram, the component we are most cognizant of is the language itself—it is what confronts us continuously in our editor windows. Hopefully you are already familiar with the Objective-C language. If you have come to this book from a non-Objective-C background, Section 4.6, at the end of the chapter provides several good sources for learning Objective-C. Languages, no matter how good, do not stand on their own. Depending upon its complexity, certain features of a language may require a runtime system, as is the case with Objective-C. Many of Objective-C's features that make it stand out from other languages are made possible by its runtime.

Just as critical as a language's syntax and semantics are the available libraries. Ultimately, libraries are how real work gets done in any language. Otherwise, programmers would have to create their own code for interacting with system resources such as networks, storage, input devices and displays. With Objective-C and OS X we have a plethora of choices when it comes to using libraries. This variety comes from Objective-C's ability to seamlessly use C, Objective-C, and even C++ (See Section 4.5). This variety of language options means that Objective-C programmers can interact with OS X at many different levels. Figure 4-1 shows some of the many faces of OS X available to Objective-C programmers.

Of course, the OS is the environment within which all of this is possible. You have undoubtedly been forced to use, at one time or another, operating systems that were less than robust[1]—that perhaps felt more like an unsteady deck of cards, than a fine tuned piece of software. Thankfully, most operating systems these days do not fall into this category. Understanding how these modern operating systems work is important to understanding how your applications will perform when running on top of them.

In the next few chapters, we will descend through these different layers of programming in Objective-C on OS X, focusing on the implementation details that will give you an intimate understanding of what is going on under the hood. This chapter focuses on the underlying mechanisms of the Objective-C language and runtime. Chapter 5,

Figure 4-1. Interacting With OS X



covers the use of system and third-party libraries, as well as issues involved in creating your own. Chapter 6 will take a more detailed look at the OS X operating system, including both its benefits and quirks.

4.1. Objective-C's Family Tree

Every Thanksgiving my family would take a 500 mile trip south to my grandparents in Tennessee. When we finally stumbled out of the car, dazed and sickened from exhaust fumes, roller-coaster hills and eight hours of eight-track music, everything always seemed just a little weird. Of course, being from Illinois, we were considered the "Yanks." Everyone spoke with an unintelligible accent, and there were always a few strange, toothless old men sitting around on the porch, drinking iced tea from mason jars, spitting tobacco, and otherwise having a high old time.

Coming from C++, this about describes the feeling I had upon looking at Objective-C code for the first time. It was disconcerting at first, but very quickly you come to realize you are among family. Of particular consternation was the use of brackets to perform, what I thought at the time were, C++'s version of object method calls:

```
NSMutableArray *cutList = [[NSMutableArray alloc] init];
```

Why go to all the trouble? I thought to myself. It wasn't until I came to understand the difference in the object systems for C++ and Objective-C that the syntax began to make sense.

Both Objective-C and C++ were originally conceived as object oriented extensions to C. C++ was written by Bjarne Stroustrup and comes from the Simula 67 school, which places more import on compile-time type safety over run-time dynamism. The syntax of C++ sticks closer to that of C proper, and is built to make user-defined classes act as much as possible like the built-in types of C. Features like operator overloading allow programmers to write classes that intuitively feel like their keyword counterparts. For numerical programming, this allows for great elegance. One need only look at the plethora of matrix classes that allow operations such as:

```
A += B * C;
```

to see why C++ has so many proponents. In addition, Generic Programming and the Standard Template Library (STL) in particular, have pushed forward the idea of what can be encapsulated in code. Libraries such as Andrei Alexandrescu's pattern templates, which provide the ability to concretely specify and implement design patterns, and the Graph Boost Library, which provides generic graph storage and algorithm templates, are excellent examples of the power of generic programming. Without a doubt, for hard core number crunchers, C++ has some great advantages.

Objective-C was written by Brad Cox in order to add smalltalk-80 extensions to C, and had not experienced widespread acceptance in either the operating system or application developer community. The one place Objective-C *was* picked up happened to be Steve Jobs post-Apple computer company NeXT. When Apple purchased NeXT in order to acquire what would later become OS X, it also inherited Objective-C.

Objective-C is fundamentally a dynamic language. That is, almost all decisions regarding object type and method calls are made at run-time, as opposed to compile-time. Many aspects of Objective-C and Cocoa that make OS X such a pleasant platform to program are derived from this dynamism. Tools such as Interface Builder and technologies like Distributed Objects get much of their power from the ability to handle completely new objects without undue overhead. While techniques exist in many forms on different platforms (such as .NET or CORBA), Objective-C has a twofold advantage: First, Objective-C is still C, retaining the lower-level speed and efficiency traditionally associated with the language. Second, Objective-C's object-oriented extensions are simple--the set of syntactic changes are small compared to the power they offer a programmer.

Unlike C++, which seems like it is trying to hide its object-oriented nature behind traditional C syntax, Objective-C's terminology and syntax are meant to emphasis the difference between objects and C's traditional procedural paradigm. Seeing

```
[cutList addObject:newPiece]
```

immediately shouts to us that `cutList` is an object [2]. When you see the brackets, you know you are doing something radically different from typical C code. In this case, you are interacting with a dynamic run-time system to pass messages between objects.

4.2. Why Objective-C?

If C++ has so many advantages, why would we choose to ever use Objective-C? To start, for an object oriented language, Objective-C is small. Apart from getting used to Objective-C's unusual bracket syntax, most programmers familiar with C and another object-oriented language can pick up Objective-C very quickly. For those programmers new to object-oriented programming altogether, which is becoming an ever smaller group, Objective-C is a very good language with which to learn OOP principles.

At the same time that Objective-C is quick to learn, combined with Cocoa, it allows developers to create complex graphical applications more quickly. Personally, I find myself completing applications or tools that I would have *never* considered tackling by myself on Windows or Linux. By using a simpler language, you will find yourself spending less time fixing typos and double checking syntax peculiarities, and more time getting your projects done.

In fact, this should be the first rule of high performance programming:

> The program that is written, *no matter how slow*, will always finish faster than the program that is never completed.

Many hours of programmer productivity have been lost to premature optimization—which you will hear about time and time again in any discussion about high performance programming. Unfortunately, It is every programmer's right to learn this lesson the hard way. One of the first optimization decisions we make is choosing the language in

which to write our application's code. Choosing a language presents us with the classic trade-off—do we choose a language that makes the program easier to write, or do we choose the language that will result in the fastest run-time? Objective-C is able to provide a RAD-like development environment, while retaining its low-level C functionality. This means you can first concentrate on getting your application built, and then worry about making it fast. In many ways, this follows the development of OS X itself.

As a final point, if you are still struggling with the choice between the ease of Objective-C and the power of C++, worry not. Apple has provided a version of Objective-C that can interact easily with both called Objective-C++. This language version will be presented in more detail in Section 4.5.

4.3. Objective-C Programming in OS X

Apple's primary development for environment for Objective-C is Project Builder. This integrated development environment provides all of the traditional graphical tools programmers have come to expect, including editing, class browsing, compiling and debugging. Being Unix, you can imagine that command line tools are not far away either. In fact, the underlying compiler and debugger will, more than likely, be very familiar to you. Finally, the development tools available from Apple include a number of ancillary utilities for handling resources, packaging applications, and profiling applications.

4.3.1. Project Builder

I would expect most readers to be intimately familiar with Project Builder, and use it in their day-to-day OS X application development. There are other options, of course, including Metrowerk's Code Warrior, or for the real hackers among us: **emacs** or **vi**, with **gcc** and **gdb** from the command line.

For the rest of us mere mortal programmers, IDEs like Project Builder bring all of the most useful tools together in one place, and provide pre-packaged project templates that isolate us from complex build scripts. In most cases, this is a good thing. The less time you spend debugging **make** files, the more time you can spend debugging your own code. At the same time, the number of questions regarding missing libraries on the Cocoa mailing lists indicates that many new programmers are so used to IDEs doing all the work, that they may not fully understand what is going on underneath. The next several chapters should hopefully bring you up to speed.

4.3.2. The Compiler

Apple provides an excellent development environment for OS X in Project Builder, especially considering that it is available for free. Under the hood though, everything is running on open source products. This includes the **gcc** compiler, a debugger **gdb**, and other miscellaneous tools such as **make**. As of writing, the latest tool chain delivered with OS X 10.2 is based on **gcc** 3.1. Project Builder in OS X 10.1 used **gcc** 2.95.2.

> If you are moving from OS X 10.1 to 10.2, with an attendant change in **gcc** from 2.95.2 to 3.1, there are fundamental changes in certain areas that can make libraries incompatible. In particular, the C++ ABI was changed for **gcc** 3.0+, which makes it incompatible with earlier versions. If you have recently upgraded, and are having problems with a project linking, make sure you clean your project by selecting: Build⟶Clean from Project Builder's menu. It is a good idea any time you upgrade your tools to clean and rebuild your projects.

Apple's use of the FSF tools is not a one-way street. Apple has its own developers working on patches and updates to **gcc**, mostly involving changes to better support OS X and the PowerPC processor. These updates are fed back into the **gcc** project. Not all of this code is incorporated into the "official" **gcc** releases right away, so be sure you check to make sure all the features you are using are available across the platforms in which you might be interested. One

example of this lag is in support for Objective-C++. Apple has submitted it for inclusion, but as of version 3.1, the official **gcc** release does not include Objective-C++ support. You will only be able to find Objective-C++ in Apple's distribution of the tools for the time being.

We will leave **gcc** for now; however, we will return to it in Chapter 7, where we will take a closer look at how to use its optimization features.

4.3.3. Objective-C API's

OS X's Objective-C API is called `Cocoa`. `Cocoa` is basically the OS X port of the original NeXTStep API. This is apparent when you see the names assigned to of all the classes, such as `NSString`, `NSWindow` and `NSView`. The NS stands for NeXTStep. NeXT later called this API OpenStep, when it was ported to additional platforms, such as Solaris and Windows NT, but kept the NS designator. OS X continues this tradition.

`Cocoa` supports both Objective-C and Java as languages, although it is more traditionally considered an Objective-C based API. The API itself is separated into two frameworks: `Foundation` and `AppKit`. If you

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

, you are in fact including both frameworks. This is what is known as an *Umbrella Framework*.

4.3.3.1. Foundation Kit

`Foundation Kit` is the framework that provides all of the core, non-graphical classes for `Cocoa`, as well as many structures and stand-alone functions. Examples of typical `Foundation` classes are `NSSet`, `NSArray` all the way up the hierarchy to `NSObject`. If you want to program at any level, graphical or not, in Objective-C, you must include the `Foundation` framework.

4.3.3.2. AppKit

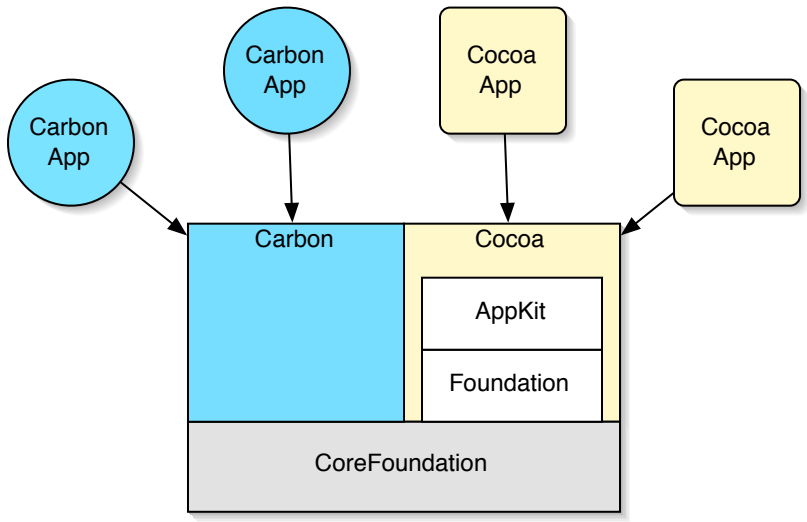
AppKit is the framework that provides all of the elements necessary for creating graphical applications with `Cocoa`. Primarily, this means all of the GUI elements, but also includes classes for supporting different kinds of applications, such as single window or document-based.

If you know you will not be referring to `AppKit` classes in a source file, you can save some compile time by only importing the Foundation framework. Project Builder does this by default when you select File⟶New File... , and choose the Objective-C Class option. These kinds of optimizations are covered in more detail in Chapter 5

4.3.3.3. CoreFoundation

Apple is in the unfortunate position of needing to support two developer bases: old Macintosh programmers, and old NeXT programmers. Dealing with this split personality has consumed an amazing amount of programming effort on Apple's part, and generated very heated debates among Macintosh and NeXT devotees. The end result of all this activity is a three pronged approach in OS X. For existing MacOS 9 and earlier applications, OS X provides a "Classic" environment in which they can run. If you want your application to run natively, you can either use the more Mac-like Carbon C API, or the NeXT-like Cocoa Objective-C API. Before the client version of OS X was released, Apple made a fundamental change to both the Carbon and Cocoa implementations. Instead of two entirely separate APIs and attendant libraries, Apple developed a CoreFoundation framework that implements many of the

Figure 4-2. Carbon, Cocoa and CoreFoundation



common features of both systems. Carbon and Cocoa can now both call into this common library, allowing Apple to more easily maintain and reuse code between the two APIs (See Figure 4-2).

You can recognize CoreFoundation methods and structures by their `CF*` designation. While CoreFoundation is entirely C-based, its implementation and structures are designed in a very object-oriented-like manner. Many Foundation objects can be directly mapped to their implementations in CoreFoundation. CFDictionary and NSDictionary are a good example of this pairing.

While the source code for Carbon and Cocoa themselves are not available, Apple has decided to redistribute CoreFoundation a part of the underlying Darwin Open Source project. This means all of the source code for CoreFoundation is available for us to study. Because so many of the fundamental data structures we use every day in our applications are implemented here, understanding how they work is important when it comes time to figuring out where performance improvements can be made in our code.

4.3.3.4. Cross-platform Objective-C APIs: GNUStep

When talking about cross-platform support for Objective-C we are really talking about the NextStep API. Since **gcc** is available on essentially every platform, the Objective-C language and runtime itself is also available. Of course, this cross-platform availability does us no good if the libraries we use to create our applications are not available as well. GNUStep is a project to deliver an open source implementation of the OpenStep API that can be run on all the major platforms for which **gcc** is available.

GNUStep is a work in progress. Foundation and Distributed Object support exists, and is in very good shape. AppKit support is a little spottier. While it is nearly feature complete on UNIX derivatives—including OS X itself, its Win32 port is somewhat behind the curve. This means, for the time being, cross-platform Objective-C programming does not fully address Win32, to which unfortunately, is what platform most developers are hoping to port. Hopefully, by the time this reaches bookshelves, the situation will be somewhat improved. If you are targeting OS X and Unix platforms only, then GNUStep is able to provide a great cross-platform solution. We will return to GNUStep when we develop a cross platform distributed application in Chapter 14

4.4. The Objective-C Runtime

Objective-C's ease-of-use, flexibility and power come at a price. As you can guess, that price is almost always in performance. Instead of calculating class types and function addresses at compile-time, Objective-C waits to perform these lookups when they are actually called on at run-time. This is unfortunately the same precious time we are trying to conserve in our programs. Object-oriented programming is foremost a tool to speed the *process* of programming, not necessarily a programs processing speed.

There is room for optimism, however. As we have mentioned before, Objective-C is still C. If you stick with the C in Objective-C, many of the traditional performance techniques apply. When you need to put the petal to the metal, you can always fall back on one of the most efficient languages around.

As soon as you put brackets around something in Objective-C, you intuitively know a whole host of new object-oriented issues must be addressed. These issues can be generally broken down into three broad themes: Object Creation, Message Passing and Object Destruction. This section presents these themes, and shows how consideration of object lifecycle is of paramount importance to high performance object oriented programming. As long as you know what the Objective-C runtime is doing, you can learn how to use it to best effect.

4.4.1. Creating Objects

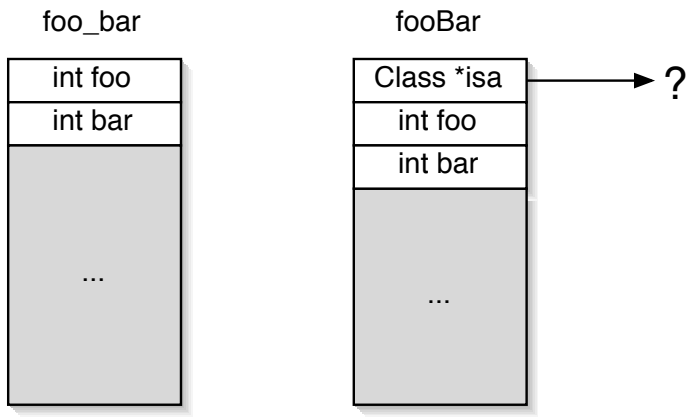
Given the following trivial class interface:

```
@interface FooBar : NSObject {
    int foo;
    int bar;
}
- (id)initWithFoo:(int)f bar:(int)b;
- (int)foo;
- (void)setFoo:(int)f;
- (int)bar;
- (void)setBar:(int)b;
@end
```

Even though you might not know what any particular class or method name of `FooBar` means, you probably won't have any problem reading the following code:

```
FooBar* fooBar = [[FooBar alloc] initWithFoo:21 Bar:12];

NSLog(@"fooBar's Foos:%d Bars:%d", [fooBar foo], [fooBar bar]);

[fooBar setBar:3];

NSLog(@"fooBar's Foos:%d Bars:%d", [fooBar foo], [fooBar bar]);

[fooBar release];
```

Figure 4-3. foo_bar Versus `FooBar` Memory Layout



As Objective-C programmers, we have a good feeling for the syntax of the language. What most Objective-C programmers don't know is what exactly the objects they are throwing around in code actually *are*. If I present you with the following structure:

```
struct _foo_bar {
    int foo;
    int bar;
} foo_bar;
```

you probably have a good feeling for what `foo_bar` looks like in memory. When we create the Objective-C object `fooBar` above, we know we receive a pointer to a memory location, but what does that memory location look like? It turns out that Objective-C objects aren't much different than C's structures. Figure 4-3 shows us how both the C structure and Objective-C object are laid out in memory.

The only difference between the structure and object in memory is the prefix of the `isa` pointer at the beginning of the memory location. It is this pointer that provides the Objective-C runtime all the information it needs to handle the object. The following code explicitly shows how we can create our own Objective-C object by hand:

```
typedef struct _foo_bar {
    Class isa;
    int foo;
    int bar;
} foo_bar;

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    // traditionally created object
    FooBar *fooBar = [[FooBar alloc] initWithFoo:12 bar:21];

    // creating our own
```

```
    foo_bar* fb = malloc(sizeof(foo_bar));
    fb->isa = (Class*)[FooBar class];
    fb->foo = 6;
    fb->bar = 2;


    // use the alloc'ed FooBar object fooBar
    NSLog(@"fooBar's Foos:%d Bars:%d", [fooBar foo], [fooBar bar]);
    [fooBar setBar:3];
    NSLog(@"fooBar's Foos:%d Bars:%d", [fooBar foo], [fooBar bar]);
    [fooBar release];


    // use our handcrafted foo_bar object fb
    NSLog(@"fooBar's Foos:%d Bars:%d", [fb foo], [fb bar]);
    [fb setFoo:62];
    [fb setBar:26];
    NSLog(@"fooBar's Foos:%d Bars:%d", [fb foo], [fb bar]);

    [pool release];
    return 0;
}
```

In this case, we create a structure foo_bar that explicitly holds our Class pointer `isa`, as well as our two `int`s `foo` and `bar`. In order to use our structure, we need to allocate memory for it, which we do using the traditional `malloc()` system call. We then fill in the structure with a pointer to `FooBar`'s class, as well as initialize the two integer variables. Once we go to the trouble, we can use our made-from-scratch foo_bar just like it was a proper Objective-C object[3].

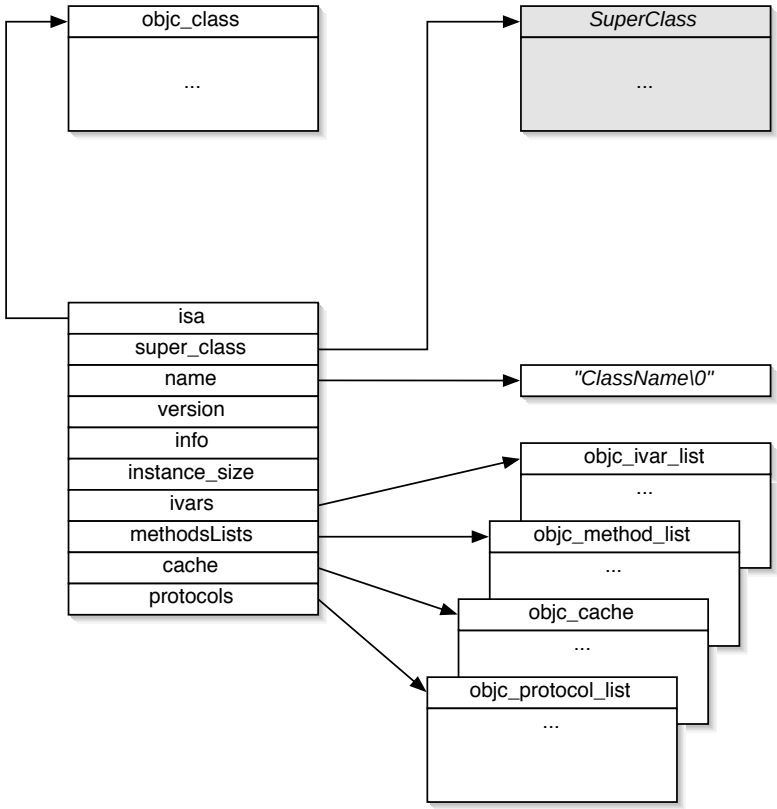
> You might have run across the term "Toll-free Bridging" in reference to structures in the CoreFoundation library that can be freely interchanged with objects in Objective-C code. One example of this kind of object is CFDictionary and NSDictionary. If a CoreFoundation function call requires an object of type CFDictionary, you can pass in an NSDictionary unaltered. You can see why this is easy for the Apple engineers to support—the C structure is layed out identical to the Objective-C object. The C code merely ignores the `isa` pointer at the beginning.

4.4.1.1. What `isa` Class

Up to this point, we have referred to the FooBar class somewhat cavalierly, even calling `[FooBar class]` without explanation. What *is* `isa` pointing to? If you look in `/usr/include/objc/objc.h` you will find that Class is defined as: `typedef struct objc_class *Class;`. So, Class is a pointer to an objc_class structure, which is defined in `/usr/include/objc/objc-class.h` as:

```
struct objc_class {
    struct objc_class *isa;
    struct objc_class *super_class;
    const char *name;
    long version;

    long info;
```

Figure 4-4. The objc_class Structure




```
    long instance_size;
    struct objc_ivar_list *ivars;
    struct objc_method_list **methodLists;
    struct objc_cache *cache;
    struct objc_protocol_list *protocols;
};
```

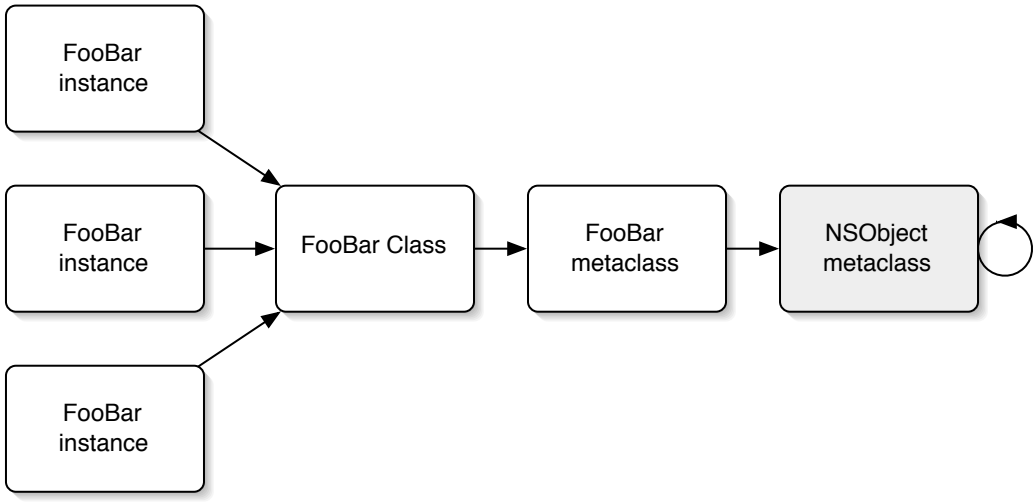
So, for every class, the Objective-C runtime system creates a `Class` object based on the objc_class structure. This structure provides all of the necessary bookkeeping for our instance objects. This bookkeeping includes the class name, a pointer to its superclass, versioning information, as well as pointers to structures holding method and protocol information. Figure 4-4 shows the layout of the objc_class structure.

Note that even the Class structure is a proper Objective-C object, holding its own `isa` pointer. For every class, such as `FooBar`, the Objective-C runtime creates a `Class` object. It is this object which responds to the class (as opposed to instance) methods we define.

When you create a class such as:

```
@interface FooBar : NSObject {
}
```

Figure 4-5. Classes and Objects




```
+ fooBarWithFoo:(int)f bar:(int)b;
@end
```

it is the `FooBar Class` object that receives the message when you call:

```
FooBar *fb = [[FooBar fooBarWithFoo:12 bar:34] retain];
```

Objective-C keeps track of this object just like every other object in the system, with its own objc_class structure pointed to by `isa`. This `Class` object's `Class` is considered the *metaclass* for `FooBar`. Just as our `FooBar Class` keeps track of *instance* methods, the `FooBar` metaclass keeps track of *class* methods. You can see this relationship in Figure 4-5.

It is important to press home an often misunderstood detail at this point. For each class, there is only *one* `Class` object. This single `Class` object is often referred to as a *factory*. That is, it is a factory that knows how to create objects of a particular type. This notation even turns up in **gcc** error messages like:

```
cannot find class (factory) method
```

All objects of a specific class type point to the same `Class` (or factory) object through their `isa` member. This emphasizes the difference we mean when we refer to a *class* versus an *object*. There is only one `FooBar Class`, while there can be any number of `FooBar` objects (See again Figure 4-5).

All of this class-level information is contained behind the single `isa` pointer that sits at the beginning of each class, and is maintained by the Objective-C runtime. For creating objects, it is left to `alloc` and `init` to prepare the instance variables that make each object unique.

4.4.1.2. Class Information

The Objective-C runtime maintains a wealth of information about a class in the objc_class structure. At the most basic level is data regarding the class name, version, etc.. The following are the declarations for these objc_class structure elements:

```
const char *name;
long version;

long info;
long instance_size;
```

The `name` member is what you would expect—a c-string representation of the class name as read by the compiler in the original class `@interface` declaration.

Objective-C supports the concept of versioning at the class-level, which is expressed by the existence of a `version` member of the objc_class structure. `NSObject` provides a couple class methods for setting the version number of a class called: `+setVersion:` and `+version`. `+setVersion:` is called in the class' `+initialize` class method:

```
+ (void)initialize
{
    [Foo setVersion:2];
}
```

Class versioning has typically been used to provide backward compatibility for archiving objects. By using the `+version` method, un-archiving code can detect the original version of the serialized class, and restore it appropriately.

The objc_class `info` variable is used to flag certain class conditions. For the curious, the Flags are defined in `/usr/include/objc/objc-class.h` as:

```
#define CLS_CLASS  0x1L
#define CLS_META  0x2L
#define CLS_INITIALIZED  0x4L
#define CLS_POSING  0x8L
#define CLS_MAPPED  0x10L
#define CLS_FLUSH_CACHE  0x20L
#define CLS_GROW_CACHE  0x40L
#define CLS_NEED_BIND  0x80L
#define CLS_METHOD_ARRAY        0x100L
// the JavaBridge constructs classes with these markers
#define CLS_JAVA_HYBRID  0x200L
#define CLS_JAVA_CLASS  0x400L
// thread-safe +initialize
#define CLS_INITIALIZING 0x800
```

Unless you are hand coding custom classes, or working on the Objective-C runtime, you will probably not have any need for the `info` variable.

The `instance_size` member of the objc_class structure holds the number of bytes an instance of the class requires in memory. This size includes the 4 bytes required for the `isa` pointer at the beginning. Given the following classes:

```
@interface Foo : NSObject {
    int a;
    double b;
}
@end

@interface Bar : Foo {
    NSString *c;
}
@end
```

the `instance_size` for `Foo` will hold the value 16—four bytes for the `Class` pointer `isa`, which is inherited from `NSObject`, four bytes for the `int a` and eight bytes for the `double b`. The `instance_size` for `Bar` is 20. The size of `Bar` includes the 16 bytes required for `Foo`, plus the additional four bytes for the `NSString` pointer. The `instance_size` variable is inclusive of all superclass requirements.

4.4.1.3. Instance Variables

The objc_class structure also keeps track of your class' instance variables in the objc_ivar_list structure `ivars`. This structure is declared in `/usr/include/objc/objc-class.h`.

```
struct objc_ivar_list {
        int ivar_count;
#ifdef __alpha__
        int space;
#endif
        struct objc_ivar ivar_list[1];          /* variable length structure */
};
```

> **Tip:** Note the use of the `#ifdef __alpha__` conditional construct. This makes sure the `ivar_list` array is aligned on an eight-byte boundary, or 64-bits. We will take a closer look at alignment in Chapter 8.

The objc_ivar_list structure holds a variable length array of objc_ivar structures:

```
typedef struct objc_ivar *Ivar;

struct objc_ivar {
        char *ivar_name;
        char *ivar_type;
        int ivar_offset;
#ifdef __alpha__
        int space;
```

Table 4-1. `ivar_type` String Values for Objective-C Types

Type	Label
void	v
char	c
unsigned char	c
short	s
unsigned short	S
int	i
unsigned int	I
long	l
unsigned long	L
float	f
double	d
bit field	b
pointer	^
char*	*
id	@
Class	#
SEL	:
IMP	^
BOOL	c
undefined	?

```
#endif
};
```

The objc_ivar structure holds information on each instance variable declared in a class' `@interface`. This information includes the variable's name, its type, and its offset within an instantiated class' memory layout. The variable type is encoded as a C-string. For Objective-C types, Table 4-1 shows the relevant codes.

Pointers are prefaced with a caret (^), so that `void *` would return a type string of `^v`. Objective-C classes are encoded as their full class names. An`NSString` instance variable would return a type of `@"NSString"`.

When presenting type information for structures or unions, the `ivar_type` variable is a little more verbose. The following are the type output for the NSRect structure, and a union comprised of and `int x` and a `double y`:

```
{_NSRect="origin"{_NSPoint="x"f"y"f}"size"{_NSSize="width"f"height"f}}
(?="x"i"y"d)
```

Class Variables

If you are familiar with C++, you will probably have run across the use of the static keyword to define class variables.

Objective-C does not have the concept of class variables. Instead, Objective-C uses static variables at *file* scope to simulate class variables.

4.4.1.4. class-dump

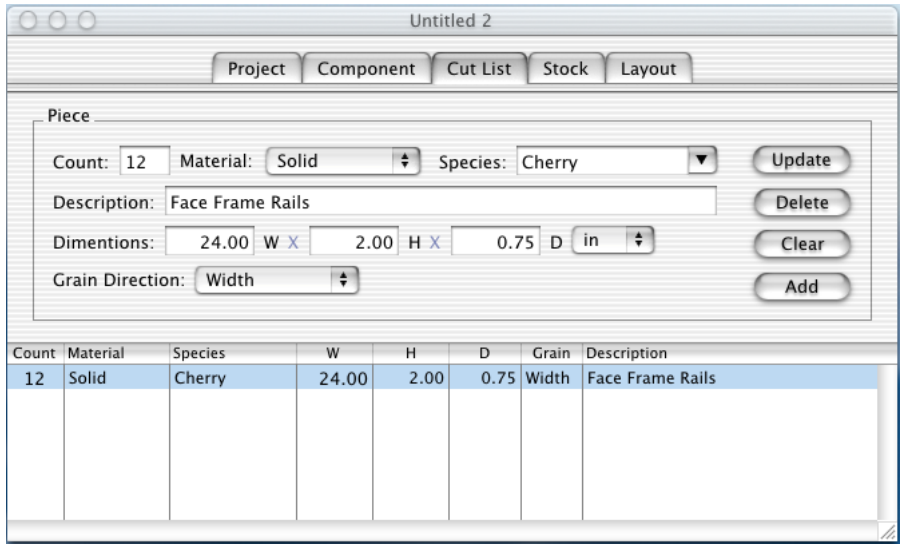
The dynamic nature of Objective-C leads to some interesting side-effects. Because the structures of classes, protocols and categories are included in a program's object files in human readable form, this information is available to anyone wanting to take a look. **class-dump**[4], by Steve Nygard, is a useful utility for examining the Objective-C segment of Mach-O files. See the section entitled "Mach-O" in Chapter 6 for more information on the Mach-O file format.

```
class-dump 2.1.5 Usage:
class-dump [-a] [-A] [-e] [-R] [-C regex] [-r] [-s] [-S] executable-file

-a  show instance variable offsets
-A  show implementation addresses
-e  expand structure (and union) definition whenever possible
-R  recursively expand @protocol <>
-C  only display classes matching regular expression
-r  recursively expand frameworks and fixed VM shared libraries
-s  convert STR to char *
-S  sort protocols, classes, and methods
```

here is a small sample of class-dump's output on Lignarius:

```
@interface CutListController:NSObject {
 NSTextField *countTextField;
 NSTextField *widthTextField;
 NSTextField *heightTextField;
 NSTextField *depthTextField;
 NSPopUpButton *scalePopUpButton;
 NSTextField *descriptionTextField;
 NSPopUpButton *materialPopUpButton;
 NSComboBox *speciesComboBox;
 NSPopUpButton *grainPopUpButton;
 NSButton *updateButton;
 NSButton *deleteButton;
 NSButton *addButton;
 NSTableView *cutListTableView;
 WoodSelectionController *woodSelection;
 NSMutableArray *cutList;
}
- init;
- (void)awakeFromNib;
```

Figure 4-6. Lignarius Cut List Tab




```
- (void)addPiece:fp12;
- (void)woodSelectionMaterialChanged:fp12;
- (void)woodSelectionSpeciesChanged:fp12;
- (void)updatePiece:fp12;
- (void)deletePiece:fp12;
- (void)clearForm:fp12;
- (void)materialSelected:fp12;
- (int)numberOfRowsInTableView:fp12;
- tableView:fp12 objectValueForTableColumn:fp16 row:(int)fp20;
- (void)tableViewSelectionDidChange:fp12;
- (void)dealloc;

@end
```

This object is the controller for the window shown in Figure 4-6

class-dump is a handy tool when you want to take a peek at what objects make up an application, and how those objects operate.

4.4.1.5. `alloc`

When we subclass `NSObject`, we inherit the `alloc` and `allocWithZone:` methods. These are the standard methods that allocate the memory our object needs, and set the `isa` pointer in the object's class structure. For most `Cocoa` programming, `alloc` and `allocWithZone` are all you will need.

While the default `alloc` methods we inherit from `NSObject` do their jobs well, they can't make the kind of optimization assumptions we can—if we have a good idea how our objects will be used. It frequently turns out that object creation (along with object destruction) is one direction we can turn in order to speed up our programs. In

Chapter 8 we will take a more in depth look at object creation and memory usage by Objective-C objects for optimization purposes.

4.4.2. Talking to Objects

Creating objects is not much different than what we have always done in C to manage data structures. The real power of OOP, however, comes from the encapsulation of data behind an interface of well defined methods. While the Objective-C object structure provides the necessary bookkeeping for associating objects with their methods, it is the Objective-C runtime that makes it all possible. Passing a message to an Objective-C object is much different than calling a C function.

Consider the traditional C methodology for calling a function. The following code calls a simple function named `foo()`:

```
#include "stdio.h"

int foo(int bar)
{
    return bar + 12;
}

int main() {
    int bar = foo(12);
    printf("foo:%d\n", bar);
    return 0;
}
```

We can take a look at the compiled code disassembled in **gdb** using the following command:

```
(gdb) disass main
```

> While pouring over assembly is becoming an ever rarer occurence in modern day computer programming, it is still useful in optimization and high performance applications. If you are not familiar with using **gdb** or the PowerPC assembly language, there is an introduction to **gdb** in Chapter 3 and PowerPC assembly in Chapter 7. Later, we will look more closely at AltiVec assembly in Chapter 11 and Chapter 12.

The following assembly is the first portion of **gdb**'s dump of the `main()` routine, right up until the point it calls `foo()`.

```
0x1e74 <main>:  mflr    r0
0x1e78 <main+4>:        stmw    r30,-8(r1)
0x1e7c <main+8>:        stw     r0,8(r1)
0x1e80 <main+12>:       stwu    r1,-80(r1)
0x1e84 <main+16>:       mr      r30,r1
0x1e88 <main+20>:       bcl-    20,4*cr7+so,0x1e8c <main+24>
0x1e8c <main+24>:       mflr    r31
0x1e90 <main+28>:       li      r3,12
0x1e94 <main+32>:       bl      0x1e48 <foo>
0x1e98 ...
```

We are interested in the very last line, where `foo()` is called with `bl 0x1e48`. `bl` is the PowerPC assembly mnemonic for *Branch and Link*. This command tells the processor to jump to location `0x1e48` in the program, after saving the next address after the branch code in the `link` register—in this case `0x1e98`. After the return address is saved, the processor starts executing code at the new address. We can see this in assembler using `(gdb)` **`disass foo`**:

```
0x1e48 <foo>:   stmw    r30,-8(r1)
0x1e4c <foo+4>: stwu    r1,-48(r1)
0x1e50 <foo+8>: mr      r30,r1
0x1e54 <foo+12>:        stw     r3,72(r30)
0x1e58 <foo+16>:        lwz     r9,72(r30)
0x1e5c <foo+20>:        addi    r0,r9,12
0x1e60 <foo+24>:        mr      r3,r0
0x1e64 <foo+28>:        b       0x1e68 <foo+32>
0x1e68 <foo+32>:        lwz     r1,0(r1)
0x1e6c <foo+36>:        lmw     r30,-8(r1)
0x1e70 <foo+40>:        blr
```

These eleven lines comprise the entirety of the `foo()` function. Once `foo` has finished its work, the `blr`, or *Branch to Link Register*, command tells the processor to jump back to the address stored in the `link` register. In our case, this is the address `0x1e98`, as stored by `main()` before calling `foo()`.

Calling a function in C is as simple as telling the processor where in memory it needs to jump next. There is an additional layer of indirection added when dealing with functions dynamically linked into your application, but the principle is the same. Calling a function in Objective-C is an entirely different proposition. So much so, that we use a different language and syntax when we talk about it. Instead of calling functions, we *send messages to objects*, and instead of typing `function(arg, ...)` we use `[receiver selector:arg]`.

Under the hood, our fancy Objective-C syntax is turned into an ordinary C function call. You can see this function declared in `/usr/include/objc/objc-runtime.h`:

```
id objc_msgSend(id self, SEL op, ...);
```

`objc_msgSend()` is the cornerstone to Objective-C's dynamism. It is where much of the visible object oriented behavior we use is implemented. Unlike C, or even C++ for that matter, Objective-C waits until run-time to make many decisions. There are two critical questions for message passing that are decided only at run-time by the call to `objc_msgSend()`:

- Which object should be the receiver of the message?
- Is the method associated with the selector defined in the object class, one of its superclasses, or an entirely separate category?

Deciding which object should be the receiver is the easiest part. We make this explicit in selecting a receiver for our messages, which is the first token after the open bracket (`[receiver ...]`). The compiler takes the receiver, and uses it as the first argument to `objc_msgSend()`, named `self`[5].

Once we know what object is going to receive the message, figuring out which method to invoke seems like it should be relatively simple—but it's not. Let's consider the following class hierarchy:

```
// FooBar.m

@implementation FooBar

...

- (NSString*)description
{
    return [NSString stringWithFormat:@"Foo: %d, Bar: %d", foo, bar];
}

@end

// FooBarToo.h

@interface FooBarToo : FooBar {
}
@end

// FooBarToo.m

@implementation FooBarToo

...

- (NSString*)description
{
    return [NSString stringWithFormat:@"Foo: %d, Bar: %d Too", bar, foo];
}

@end
```

`FooBarToo` inherits from `FooBar`, and both override `NSObject`'s `description` method. Even if we use static typing to reassign a `FooBarToo` object to a `FooBar` pointer, Objective-C's runtime is still smart enough to know we really have a `FooBarToo` on our hands:

```
FooBar *fb = [[FooBar alloc] initWithFoo:12 bar:34];
FooBarToo *fbt = [[FooBarToo alloc] initWithFoo:12 bar:34];

FooBar *fooBar = [fbt retain];

NSLog(@"%@", fb);
NSLog(@"%@", fbt);
NSLog(@"%@", fooBar);
```

results in:

```
2002-08-27 23:13:21.902 FooBarToo[1207] Foo: 12 Bar: 34
2002-08-27 23:13:21.915 FooBarToo[1207] Foo: 34 Bar: 12 Too
2002-08-27 23:13:21.924 FooBarToo[1207] Foo: 34 Bar: 12 Too
```

This paradigm contrasts with C++, where programmers must explicitly indicate their wish for this behavior by using the `virtual` keyword. Under certain conditions in C++, passing objects by value can even result in sliced objects that loose all but their superclass functionality. C++'s `virtual` keyword makes sure that overridden methods in subclasses are always called, but requires a class developer to explicitly declare methods as `virtual`. In Objective-C *all* methods are automatically "virtual".

In order for Objective-C to provide true dynamic messaging, where the right method implementation is called every time, the run-time system must be able to figure out to what selectors an object can respond. Lets take a look at the structure declaration for objc_class that we saw in the last section again:

```
struct objc_class {
    struct objc_class *isa;
    struct objc_class *super_class;
    const char *name;
    long version;

    long info;
    long instance_size;
    struct objc_ivar_list *ivars;
    struct objc_method_list **methodLists;
    struct objc_cache *cache;
    struct objc_protocol_list *protocols;
};
```

In addition to the basic information that tells the system about the class, such as its name, version and size, the objc_class structure also holds several pointers to structures that allow the Objective-C runtime to pass messages to the correct method implementation. The primary structure that makes this possible is the `methodLists` pointer. `methodLists` is in fact a pointer to an array of objc_method_list structures[6]. The Objective-C runtime uses this pointer to an array of structures because it cannot know in advance how many methods a class might have.

Unlike languages such as C++ or Java, Objective-C allows programmers to extend class functionality on-the-fly using *categories*. Catagories allow programmers to add methods to a class without subclassing, so they do not exist in the normal chain of inheritance. Each time a category is loaded—whether at application startup, or dynamically when loading a bundle—the Objective-C runtime will add that category's new methods to the class' `methodLists` variable as an additional objc_method_list structure. That structure is defined as:

```
typedef struct objc_method *Method;

struct objc_method {
        SEL method_name;
        char *method_types;
        IMP method_imp;
};

struct objc_method_list {
```

```
        struct objc_method_list *obsolete;

        int method_count;
#ifdef __alpha__
        int space;
#endif
        struct objc_method method_list[1];      /* variable length structure */
};
```

Every objc_method_list structure holds an array of objc_method structures, each of which contains three elements: `method_name`, `method_types`, and `method_imp`.

The `method_name` entry in objc_method is of type `SEL`, which is defined in `/usr/include/objc/objc.h` as:

```
typedef struct objc_selector  *SEL;
```

Underneath, the selector of a method is defined as a C-string holding the name of the message itself. For example, the selector for `NSObject`'s method `- (BOOL)isMemberOfClass:(Class)aClass` is "isMemberOfClass:". You are undoubtedly familiar with the Objective-C compiler macro `@selector()`, used to generate `SEL`s from literal method names. For example, if you have an `NSArray` of `NSPort`s, and wanted to invalidate all of the receivers, you could use:

```
[portArray makeObjectsPerformSelector:@selector(invalidate)];
```

Even though `NSArray` knows little about the objects it stores, it does know one critical piece of information: each object should be a subclass of `NSObject`[7]. Knowing this, `NSArray` is able to use `NSObject`'s `performSelector:` method to forward the `invalidate` message.

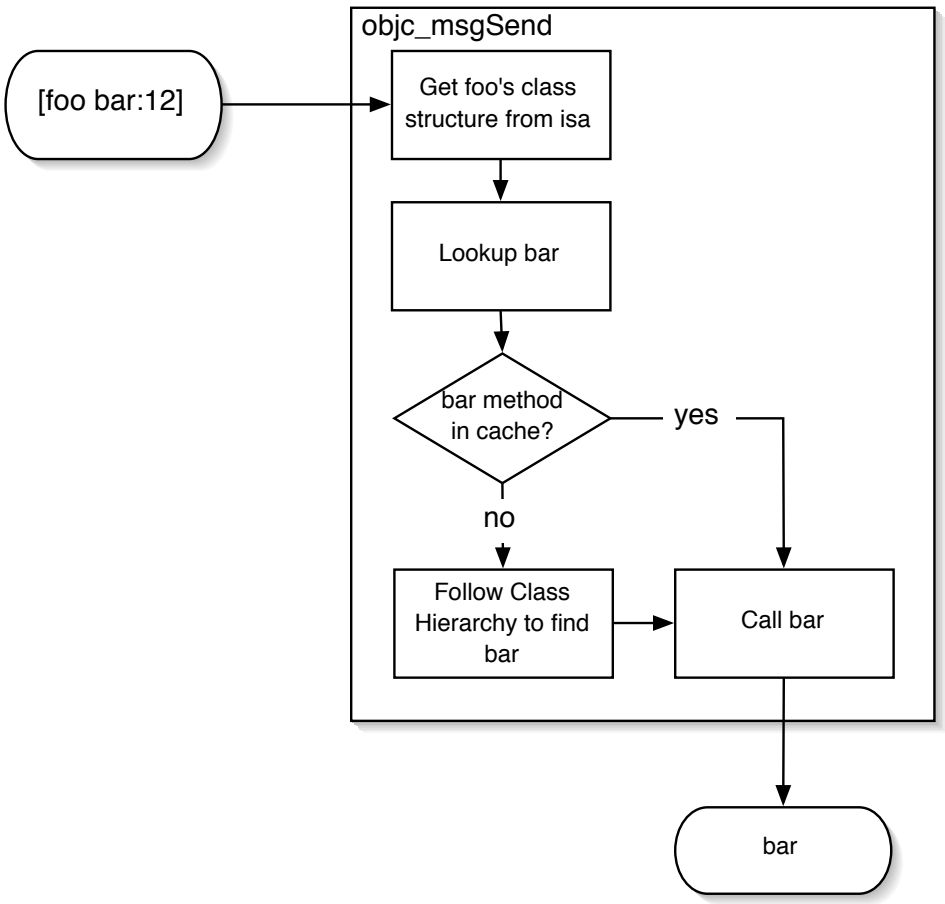
Once the Objective-C runtime has a selector, it can use the objc_method structure to match a method name with its implementation. The `method_imp` entry in objc_method stores the actual implementation location of the method. `method_imp` is of type `IMP`, which is defined in `/usr/include/objc/objc.h` as a function pointer returning an `id`:

```
typedef id (*IMP)(id, SEL, ...);
```

The `method_types` variable, like the `ivar_type` variable in objc_class, stores information about the types used in the method call. The layout of the C-string is identical to the `IMP` declaration above. That is, first the return type is presented, then the receiving object type—which in our case is always an `id`, then the selector type and finally the arguments. Each of these type values is followed by an offset number. An example entry for `NSObject`'s method `- (BOOL)isMemberOfClass:(Class)aClass` would be `c0@4:8#12`. `c` is the `BOOL` return type. `@` represents the type of the receiver, again, always an `id`. `:` is the type of the selector, which is always `SEL`. `#` is the type of the first and only argument: `Class`. `method_types` uses the same character codes presented in Table 4-1 that are used for determining instance variable type.

Whenever you pass a message to a class, the Objective-C runtime must match the message you wish to send with the method implementation by which it is implemented. This means the runtime needs to search through every

Figure 4-7. Sending a Message With `objc_msgSend()`



method_list in an object's `Class` looking for just the right method. `NSObject` alone has 16 method_list structures containing a total of 114 methods! This list is generated by Objective-C from the class itself, plus all of its attendant categories.

Luckily, the Objective-C runtime does not have to search the entire method_list tree every time it needs to send a message. Instead, the runtime keeps track of all messages it has passed to a class in objc_class's objc_cache variable called appropriately: `cache`. The penalty for looking up a method is only incurred the first time it is invoked. Thereafter, `objc_msgSend()` will find the method in the class' method cache. Figure 4-7 provides a diagram of how the `objc_msgSend()` decides which method implementation should receive a message.

Static Typing Versus id

In Objective-C we can refer to objects as either `id`, or as pointers to their respective types (e.g. `NSString*`). Using these pointers in your code is referred to as *Static Typing*. Static typing can be used by the compiler to check whether you are using your objects properly. For example, the compiler can tell you when you are passing the wrong kind of object as an argument in a selector, or trying to use a selector to which an object does not respond.

There has been some debate in newsgroups and message boards about whether static typing in Objective-C has any impact on *run-time* performance. In every case, the consensus was that the purported boost in speed from using static typing was a placebo effect. Using static typing when writing programs only provides compile-time type safety.

Why *couldn't* **gcc** use static typing to shortcut the run-time system's dynamic dispatch by linking straight to the method implementation? Unfortunately, this would break the expected behavior of inheritance. While we might declare an object to be of a static type—say `NSButton*`, what we are really saying is the object is of type `NSButton`, *or any of its subclasses*. It is not until runtime that we may definitively know which actual instance method will get called.

4.4.2.1. KeyValue Coding

Objective-C has been described as a verbose language. It does not shy away from long lines of descriptive code. What is lacks in brevity, it makes up for in flexibility. One area where Objective-C's open object implementation is used to good effect is in the `NSKeyValueCoding` protocol:

```
@interface NSObject (NSKeyValueCoding)

- (id)valueForKey:(NSString *)key;
- (void)takeValue:(id)value forKey:(NSString *)key;
- (id)storedValueForKey:(NSString *)key;
- (void)takeStoredValue:(id)value forKey:(NSString *)key;
+ (BOOL)accessInstanceVariablesDirectly;
+ (BOOL)useStoredAccessor;

@end
```

All classes derived from NSObject automatically inherit an implementation of the `NSKeyValueCoding` protocol, which allows programmers to access the instance variables of a class using a string-based key. You don't have to know in advance what all of a class' instance variable's names might be. the `-valueForKey:` and `-takeValue:forKey:` methods are able to take a string holding the name of a class' instance variable, and get or set that variable's value.

The default implementation actually tries several ways at pulling this off:

1. Given a key named `key`, look for accessor methods named `-setKey:`, `-key` or `-getKey`.
2. If any of those methods are not available, the implementation will look for the above method names with leading underscores:`-_setKey:`, `-_key` or `-_getKey`

3. If there are no accessor methods, the implementation will look directly for instance variables named either `key` or `_key`.
4. If none of the above work, the `NSKeyValueCoding` implementation will give the class a chance to handle the unbound key, or throw an exception.

The `NSKeyValueCoding` implementation is able to accomplish all of this using the information made available through the objc_class structure and supporting functions.

4.4.2.2. Optimizing with `IMPs`

Even with caching, the Objective-C runtime must still perform a lookup each time a message needs to be sent to an object. In most situations, this added level of indirection is negligible. However, there are certain situations in which you might want to bypass Objective-C's dynamic dispatch for a faster, static approach. Consider the following method from Lignarius's `RRGGOptimizer` class:

```
- (void)optimize
{
    BOOL finished = NO;

    // zero out the structures
    [rectangleValues removeAllObjects];
    [verticalCuts removeAllObjects];
    [horizontalCuts removeAllObjects];
    [self setValue:0 forSize:stockSize];

    first.width = wStepping;
    first.height = hStepping;
    second.width = wStepping;
    second.height = hStepping;


    [self findVerticalCut];
    [self findHorizontalCut];


    while(!finished) {
        while(second.width < stockSize.width) {
            second.width += wStepping;
            [self findVerticalCut];
            [self findHorizontalCut];
        }
        if(second.height < stockSize.height) {
            second.height += hStepping;
            second.width = wStepping;
            [self findVerticalCut];
            [self findHorizontalCut];
        } else {
            finished = YES;
        }
    }
}
```

When looking for an optimal layout for a given set of parts and stock material, this is the method that kicks it all off. You may notice that there are a couple Objective-C messages called within the main loop that comprise the bulk of this method's computation.

`-findVerticalCut` and `-findHorizontalCut` are deep in the nested loops, and depending on the size of the stock cutting problem, selectors could be sent thousands of times. Each time, the Objective-C runtime system must follow the steps outlined in Figure 4-7 to find the proper method implementation. An important optimization we can make is to perform this implementation lookup outside of our loop beforehand. By caching the location of the implementation, we can call the implementation directly, thus bypassing the costly lookup every time through the loop.

Recall that the definition of `IMP` is:

```
typedef id (*IMP)(id, SEL, ...);
```

In other words, a pointer to a function returning id. While we could trudge through the objc_class structure in order to find a method's associated `IMP`, `NSObject` provides a handy method for obtaining the `IMP` from a selector. Given an Objective-C class instance method returning id, we would grab a method's implementation by calling:

```
- (IMP)methodForSelector:(SEL)aSelector
```

Once we have a method's implementation pointer, we can call it like any traditional C function pointer. Consider the following method from `NSDictionary`:

```
- (id)objectForKey:(id)key;
```

We could call this method using its `IMP` as follows:

```
IMP objectForKeyIMP =
    [dict methodForSelector:@selector(@"objectForKey:")];

while(...) {
    object = objectForKeyIMP(dict, @selector(@"objectForKey:"), key);
    ...
}
```

Unfortunately, not all methods match the `IMP` typedef. In the case of `findVerticalCut` and `findHorizontalCut` our method returns `void`. In these cases, you must make your own function pointer, and cast the return of `methodForSelector` like:

```
void (*findVerticalCutIMP)(id, SEL);
findVerticalCutIMP = (void (*)(id, SEL))[self
```

```
    methodForSelector:@selector(@"findVerticalCut")];
```

Our `optimize` method, without the use of Objective-C messaging, now looks like:

```
- (void)optimize
{
    // ...
    void (*findVerticalCutIMP)(id, SEL);
    void (*findHorizontalCutIMP)(id, SEL);

    findVerticalCutIMP = (void (*)(id, SEL))[self
                         methodForSelector:@selector(@"findVerticalCut")];

    findHorizontalCutIMP = (void (*)(id, SEL))[self
                         methodForSelector:@selector(@"findVerticalCut")];

    findVerticalCutIMP(self, @selector(@"findVerticalCut"));
    findHorizontallCutIMP(self, @selector(@"findHorizontalCut"));


    while(!finished) {
        while(second.width < stockSize.width) {
            second.width += wStepping;
            findVerticalCutIMP(self, @selector(@"findVerticalCut"));
            findHorizontallCutIMP(self, @selector(@"findHorizontalCut"));
        }
        if(second.height < stockSize.height) {
            second.height += hStepping;
            second.width = wStepping;
            findVerticalCutIMP(self, @selector(@"findVerticalCut"));
            findHorizontallCutIMP(self, @selector(@"findHorizontalCut"));
        } else {
            finished = YES;
        }
    }
}
```

We have now eliminated any Objective-C message passing from the main loop of the `optimize` routine. This does not, however, guarantee that there are no messages being sent from within the `findVerticalCut` and `findHorizontalCut` methods themselves.

This technique is obviously not necessary every time we need to send a message to an Objective-C object. What are the right times to use it then? If the following two conditions are met, your implementation may benefit from this optimization:

1. You are sending many messages to an object in a tight loop.
2. The object itself is not changing in a way that would alter the location to which the `IMP` was pointing.

Given the above, it is important to remember that you need to be sending many hundreds or thousands of messages in repetition to an object for this technique to make a noticeable difference.

This technique will also not work if there is a chance for the `IMP` to change. Imagine a situation where you have a list of `Shape` objects to which you want to send the message `-area`. If those objects are all each actually subclasses of type `Rectangle`, `Circle`, etc., that override the `-area` method, you won't know beforehand which method implementation you will need to call for any particular object. `Rectangle` objects will want to use their `-area` implementation and `Circle` objects will want to use their's. Be wary if you are only holding pointers to superclass objects and wanting to use the `IMP` invocation optimization.

4.4.3. Destroying Objects

We have talked about how objects are creating in Objective-C, as well as how they are used. The final stage in an object's lifecycle is its destruction. The act of destruction itself is not that difficult to understand—when you are done with an object, remove its instance from memory. It is determining when you are done with an object that turns out to be a more than trivial task. Objective-C, through the `NSObject` class solves the problem of knowing when to send objects to the great bit-bucket in the sky using a technique known as reference counting.

In traditional C programming, dynamic memory allocation using `malloc()` must be paired with an equal number of `free()`s. This memory management pattern, while conceptually simple, is in practice not easy to get right. This is especially true when calling system-level routines or third party libraries where you may not have the original source code. If you are handed a block of memory through a pointer, is it your responsibility to free it, or will the system? What if you then pass it on to a third library, is that library going to release it out from under you? Ultimately, the malloc/free pattern does not address the issue of memory lifetime and ownership itself. It is only through careful coordination between software developers that leaks can be avoided.

This confusion over memory "ownership" is especially confusing when dealing with encapsulated objects. Different object oriented languages have tried dealing with this problem in many ways. The two most popular methods for managing memory are reference counting (RC) and garbage collection (GC). Of the two methods, RC is conceptually the simplest. When you are given an object that you want to hold beyond the current scope, you increment a *reference count* variable associated with the object. When you are finished with the object you decrement the reference count. If the count falls to zero, the system knows no one else is interested in the object, and releases it.

GC has been around for a long time, but was popularized by the Java programming language. GC starts with the same principle as RC, but automates the process. Reference counts are automatically incremented and decremented by the runtime as objects come into and go out of scope. When an object's reference count falls to zero, the object is automatically destroyed. GC allows programmers to by and large avoid issues of memory management. Java's GC implementation is a significant part of what makes Java such an easy language for new programmers. The biggest complaint leveled against GC is that it can be slow. The overhead of keeping track of these references in the runtime system imposes a penalty on the entire application.

4.4.3.1. Objective-C Reference Counting

The Objective-C language itself doesn't specify a memory management technique inasmuch as it leaves this detail to library developers. The root class for all objects in the Foundation and AppKit libraries, `NSObject`, uses reference counting. You are probably already familiar with the rules for memory management in Cocoa. You are only responsible for releasing an object if you have done one of the following:

1. allocated and initialized an object
2. retained an object
3. copied an object

You are not responsible for releasing any other objects, no matter how you might have come to hold a reference to that object.

4.4.3.2. Autorelease Pools

While reference counting alone works well, autorelease pools provide an additional level of automation in memory management by automatically releasing objects when their reference count falls to zero. Autorelease pools can be viewed as an attempt at giving reference counting some of the ease-of-use of garbage collecting.

By sending `-autorelease` to an object, you place it in the current autorelease pool. At the end of the current event loop, or when the autorelease pool itself is released, it goes through the list of its objects and checks to see if any object's reference count is zero. If it finds an object with no references, it sends the `-release` message to that object. Just as with garbage collection, there is an additional overhead when using autorelease pools in your code. judicious use of autorelease pools can be an important optimization technique.

Reference counting, autorelease pools and general memory management in Objective-C will be covered in much more detail in Chapter 8.

4.5. Back Together Again: Objective-C++

Mixing C and Objective-C code has never been a problem on OS X. In fact, some of the functionality provided by `Cocoa` is simply a wrapper around C-based Carbon libraries. This means you can mix and match C code as appropriate in your applications. There are plenty of reasons you might want to fall back on C. Most likely, it will be because you either have legacy code with which you need to inter-operate, or find the need to implement highly optimized routines. Whatever the reason, for all practical purposes, Objective-C *is* C.

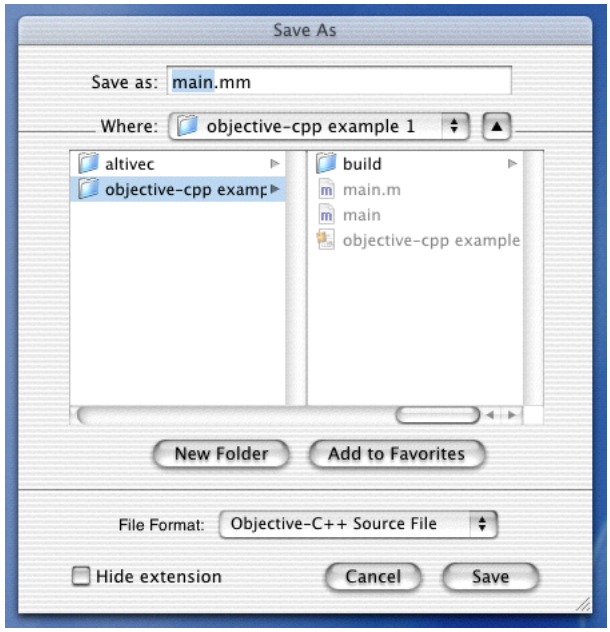
The situation is not as simple with C++. Previously on OS X, if you wanted to mix Objective-C and C++ you could only use their common denominator, C. This typically meant programmers had to develop compatibility layers that necessarily strip the object oriented nature of either the Objective-C or C++ libraries, thus rendering much of their attraction moot. It turns out that NeXT had a version of Objective-C that could inter-operate with C++ code called (not surprisingly) Objective-C++, that at first did not make NeXTStep's transition to the Macintosh platform. Apple soon rectified the situation, and later OS X developer tools include support for the Objective-C++ language. Objective-C++ is not widely advertised by Apple, but it exists, and has been updated to support the current C++ standard.

Support for Objective-C++ is built into **gcc** and ProjectBuilder. ProjectBuilder uses the file extension to determine in which language a file has been written. For Objective-C++, the expected extension is `*.mm`. If you choose File⟶Save As... in ProjectBuilder you have the option of selecting an Objective-C++ file type (See Figure 4-8).

To get an idea of what Objective-C++ can do, let's look at an example. The following code is the header file for a very simple C++ class called `Widget`:

```
#include <string>

class Widget {
 friend ostream& operator<<(ostream &os, const Widget &w);
```

Figure 4-8. ProjectBuilder File Type for Objective-C++




```
public:
 string getName();
 void setName(string new_name);

protected:
 string name;
};
```

Notice that the `Widget` class declaration uses the standard

```
#include <string>
```

syntax for include files, as well as the other traditional fare you would expect for a C++ class. Objective-C++'s compliance with the C++ standard follows from **gcc**'s support of the standard. Earlier versions of **gcc** (pre 3.x) had some limitations in this regard. The current C++ support in **gcc** is much more complete. `Widget` is obviously a pretty simple beast, so we probably won't be stressing the C++ implementation too any great degree. It has one member variable: `name`, two methods: `getName` and `setName` as well as a single friend function: the overloaded left-shift operator (<<).

The implementation file `widget.cpp` for `Widget` is:

```
#include "widget.h"

string Widget::getName()
```

```
{
  return name;
}


void Widget::setName(string new_name)
{
 name = new_name;
}

ostream& operator<<(ostream &os, const Widget &w)
{
 return os << w.name;
}
```

Again, the `Widget` class implementation follows the standard conventions for C++ programming. `Widget` is a full fledged C++ class in all respects.

Using this C++ class in our Objective-C++ code turns out to be very easy. The following listing is the `main.mm` file for a very simple application showing how Objective-C++ can mix Objective-C and C++ code.

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

int main (int argc, const char * argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 Widget w;
 NSString* s = @"Widget Number 1";

 w.setName([s cString]);

 NSLog([NSString stringWithCString:w.getName().c_str()]);
 cout << w << endl;

 [pool release]; return 0;
}
```

Like all Objective-C programs, we start by creating an autorelease pool to take care of any Objective-C objects we might create along the way. Our application then creates two objects—a `Widget` `w` and an `NSString` `s`. Next, we set `w`'s name using `NSString`'s `cString` selector, which returns a traditional null-terminated char*. Since `Widget`'s `setName()` method expects a `string`, `string`'s C++ style constructor is implicitly called. We then get `w` to output its name using both the Foundation framework's `NSLog()` function, and C++'s `cout` object. In the case of `NSLog()`, we have to get a C-style string from `w`, while `cout` uses our overloaded friend `<<`. Finally, we follow proper Objective-C convention and release our pool, and exit out of main.

Despite this example's small amount of code, there is a lot going on under the hood. If you take a second to consider what we have just done, it is actually pretty amazing. Without very much fuss, we have integrated objects from two different languages. There are a couple features of Objective-C++ that make this transition easy: First,

Objective-C++ allows us to program with the full range of both Objective-C and C++ syntax. We are not limited to a poor-man's subset of commonalty. If you want to use the full expressiveness of C++'s operator overloading, feel free to do so. If you want to incorporate Distributed Objects or Objective-C's dynamism, do so as well. Second, while our use of the `Widget` class shows how we can incorporate custom C++ objects, this code also shows how we have all of the standard C++ libraries at our disposal as well. We were able to implicitly and explicitly use the `string` class and `cout` object.

If you need to work with existing C++ libraries, ProjectBuilder's Objective-C++ implementation makes your work fairly painless. There is no need to create a bridging layer of code between your Objective-C and C++ work. In addition, there are a few side benefits to using ProjectBuilder's Objective-C++ mode. In addition to using C++'s object-oriented features, you can also take advantage of the more mundane changes C++ makes to C syntax. Take the following Objective-C implementation as an example:

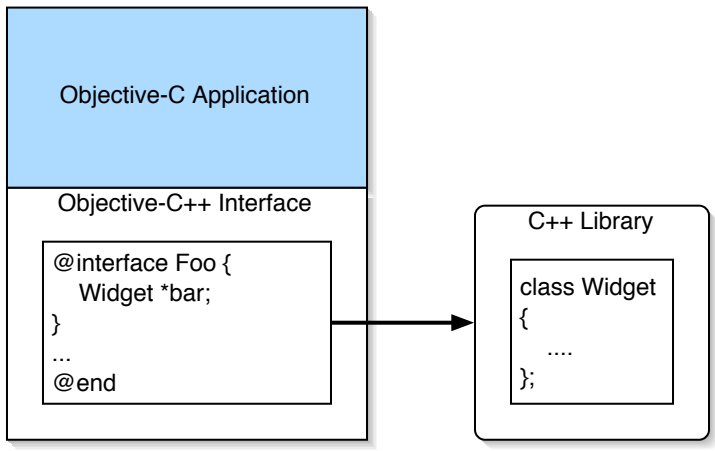
```
#import "widget.h"

@implementation Widget

- (void)doSomething {
 NSString *s = @"foo";
 NSLog(s);

 for(int i = 0; i < 10; ++i) {
  // do something clever
 }

}

@end
```

Save it as `widget.m`, and **gcc** will give you: `widget.m: In function '-[Widget doSomething]': widget.m:10: parse error before 'int'`

However, save the exact same file as `widget.mm`—an Objective-C++ file type, and **gcc** will happily compile it. In this case, Objective-C is using the old C rules about where we can declare variables in a block of code, which means that all of our variables must be declared at the *beginning* of the block. As Objective-C++ code, we get to declare variables anywhere we wish, including within the `for` loop expression. Some programmers have even been known to automatically use the `*.mm` file extension in order to simply avoid the more restrictive syntax of C![8]

Apart from avoiding C limitations, there are actually more productive reasons for using Objective-C++. Typically, this is to integrate C++ and Objective-C code into a single application. There are many ways that a programmer might want to combine Objective-C and C++; however, for the sake of simplicity, I would recommend keeping a safe distance between the Objective-C and C++ portions of your code. It is possible to turn otherwise perfectly well behaved Objective-C classes in hybrid monsters that devour your productivity. Just because Objective-C++ lets you freely intermingle C++ and Objective-C code is no reason to abandon good design choices. Good application design will isolate libraries behind well documented interfaces, so that as implementations change, an entire application does not have to be retooled. Objective-C++ provides the bridge with which to create this interface.

Figure 4-9. Calling C++ from Objective-C



4.5.1. Using C++ Classes from Objective-C

The most common use of Objective-C++ is probably when developers need to integrate existing C++ libraries into their code. Figure 4-9 depicts this relationship.

Take another look at the `main.mm` file from the `Widget` example:

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

int main (int argc, const char * argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

 Widget w;
 NSString* s = @"Widget Number 1";

 w.setName([s cString]);

 NSLog([NSString stringWithCString:w.getName().c_str()]);
 cout << w << endl;

 [pool release];
 return 0;
}
```

Now, what would you consider this application—an Objective-C program using C++ objects, or a C++ program using Objective-C objects? In fact, you could pick either position and be correct. This example is a little *too* simple, in that no Objective-C or C++ object directly interacts with the other language's constructs. If you look closely, we are simply passing around pointers to characters between the objects. In any real Objective-C application, `main()`

Figure 4-10. A Simple Table View

will be a distant memory up the calling stack, and we will need to interact with our C++ libraries from within proper Objective-C objects.

In this example, we will create a very simple application that combines the use of an Objective-C GUI with a C++ back-end that loads a list of wood stock from a file. You can imagine the C++ backend might be code that accesses a legacy database or proprietary binary file formats. In our case, the files are simple text files holding a series of wood names and counts.

First, we need to properly isolate the portions of our code that will interact with C++ from the portions of our code that will be pure Objective-C. The best way to do this is through the use of *protocols*. Here is our simple protocol definition:

```
@protocol DataSourceProtocol <NSObject>
- (void)loadFilename:(NSString*)filename;
@end
```

It is this protocol by which the Objective-C GUI will tell the C++ portions of the code to load a file. The basic strategy is this: encapsulate the C++ portions of the code in Objective-C++ objects. Define a protocol to which the Objective-C++ object responds. Connect the Objective-C portions of the code to this hybrid object through an `id` pointer using InterfaceBuilder and let the Objective-C runtime pass messages back and forth between these objects.

Here is the header file for our Objective-C++ object. In this case, we are interested in using C++'s collection classes from the STL. The DataSource object is going to hold a pointer to a vector storing the information our program is interested in.

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

```
#include <vector>
#include <string>

#import "DataSourceProtocol.h"

@interface DataSource : NSObject <DataSourceProtocol> {
 vector< pair<string, float> > *data;
}
@end
```

The datasource class needs to implement a couple of protocols, both formal and informal, that are not apparent in its header file. The first of these is the tableview data source *informal* protocol. We need to accept two messages: **numberOfRowsInTableView:** and **tableView:objectValueForTableColumn:row:**. In addition, datasource implements the formal protocol we declared in **DataSourceProtocol.h**, which is the **loadFilename:** message.

```
#import "DataSource.h"
#include <iostream>
#include <fstream>

@implementation DataSource

- (id)init
{
 if(self = [super init]) {
  data = new vector< pair< string, float> >;
 }
 return self;
}

- (void)loadFilename:(NSString*)filename
{
 ifstream infile([filename cString]);
 string wood;
 float feet;

 if(!infile) {
  NSRunAlertPanel(@"Error",
   [NSString stringWithFormat:@"Error Opening File %@", filename],
   nil, nil, nil);
  return;
 }

 while( infile >> wood ) {
  infile >> feet;
  data->push_back( make_pair< string, float > (wood, feet));
 }
}

- (int)numberOfRowsInTableView:(NSTableView *)aTableView {
 return data->size();
```

```
}

- (id)tableView:(NSTableView *)aTableView
objectValueForTableColumn:(NSTableColumn *)aTableColumn row:(int)rowIndex
{

 if([[aTableColumn identifier] isEqualTo:@"index"])
  return [NSString stringWithFormat:@"%d", rowIndex];

 if([[aTableColumn identifier] isEqualTo:@"wood"])
  return [NSString stringWithCString:((*data)[rowIndex]).first.c_str()];

 if([[aTableColumn identifier] isEqualTo:@"feet"])
  return [NSString stringWithFormat:@"%.2f", ((*data)[rowIndex]).second];

 return @"";
}


- (void)dealloc {
 delete data;
 [super dealloc];
}

@end
```

Notice that in our `-init` method we dynamically create the C++ vector, and properly dispose of it in our `-dealloc` method. This is a very important point:

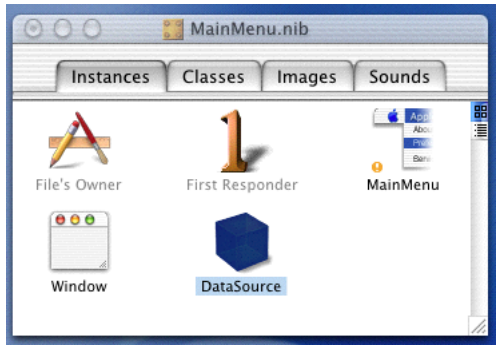
> *Hold all objects as pointers*. Just like your statically typed Objective-C objects, make sure you use pointers to C++ objects as instance variables, instead of the objects themselves. Since the Objective-C run-time isn't familiar with the C++'s object model, it will not be able to properly instantiate the variables. It will not automatically call the C++ constructors and destructors as you are familiar with. By using pointers, you can properly construct the instance variables in your `-init` method, as well as properly destroy the variable in your `-dealloc` method. The same goes for Objective-C classes as members of C++ objects.

We now have a completed datasource object that bridges the gap between the C++ `vector` and `pair` classes and Objective-C. It does so without undue exposure to C++ in the rest of the application.

At this point, we can instantiate it in our nib, and connect it to the table view as the table view's datasource (See Figure 4-11).

There are a few things to remember. First, the table view datasource is the Objective-C++ object. Notice that the table view doesn't care. It happily sends messages to the Objective-C++ datasource. This is the advantage of the dynamic nature of Objective-C. Even though tableview is a pure Objective-C object, we can set its datasource outlet to our Objective-C++ object, and it happily trundles away. However, if you were to try to directly set the table view's datasource to the Objective-C++ object programmatically, you would have difficulty getting things to compile. The use of the protocol helps to better define our intentions to the compiler. Also, we cannot include the datasource header file directly in our GUI code since the header has C++ portions that will fail to compile in combination with the Objective-C-only portions of the code as we will see next.

Figure 4-11. Create an Instance of our Objective-C++ Object `DataSource`



There is an additional class we need to complete our application. That is a controller to load the `NSOpenPanel`, and tell the datasource to load the new data. It uses the formal protocol we defined earlier to do this. Here is the header file:

```
#import <Cocoa/Cocoa.h>
#import "DataSourceProtocol.h"

@interface WoodCountController : NSObject {
 IBOutlet NSTableView* table;
 IBOutlet id<DataSourceProtocol> datasource;
}
- (IBAction)open:(id)sender;
@end
```

```
#import "WoodCountController.h"

@implementation WoodCountController

- (IBAction)open:(id)sender
{
 NSOpenPanel *panel = [NSOpenPanel openPanel];
 [panel setDelegate:self];
 [panel setPrompt:@"Open"];

 [panel beginSheetForDirectory:nil file:nil types:nil
       modalForWindow:[table window] modalDelegate:self
              didEndSelector:@selector(openPanelDidEnd:returnCode:contextInfo:)
          contextInfo:[NSString stringWithString:@"open"]];
}

- (void)openPanelDidEnd:(NSOpenPanel*)openPanel
returnCode:(int)returnCode contextInfo:(void*)x
{
 if(returnCode == NSOKButton) {
```

Figure 4-12. Create an Instance of the `WoodCountController` Class

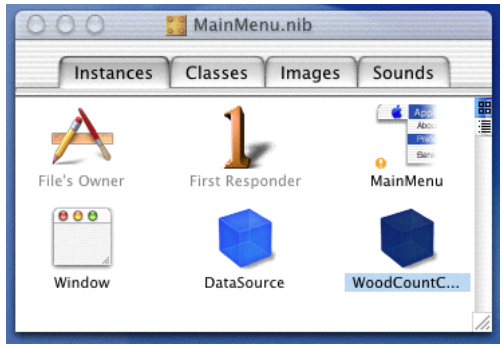
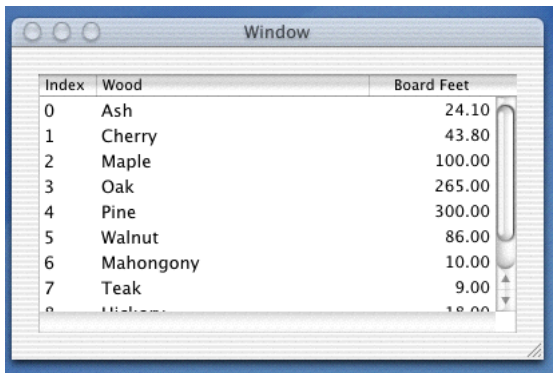


Figure 4-13. Our New Mixed Objective-C and C++ Application Running




```
  [datasource loadFilename:[openPanel filename]];
  [table reloadData];
 }

}

@end
```

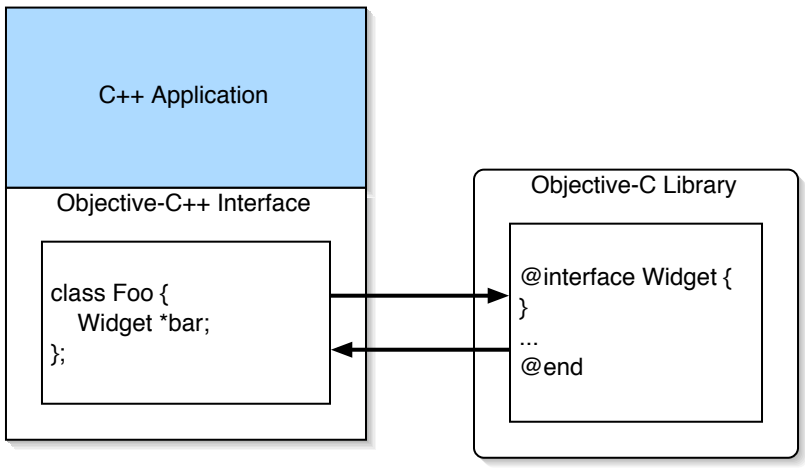
as with the datasource class, create an instance of the `WoodCountController` class in Interface Builder (See Figure 4-12), make the outlet connections to `table` and `datasource` and compile.

Figure 4-13 shows our new mixed Objective-C and C++ application running. In this case, the graphical front end is entirely Objective-C, while the back-end pulling data uses C++ objects and data structures. Objective-C++ has been used to effectively bridge between these to languages.

4.5.2. Using Objective-C Classes from C++

Most programmers used Objective-C in the past in order to program for NeXTStep, and the same holds true for OS

Figure 4-14. Integrating Objective-C into a C++ Program



X[9]. When you find yourself needing Objective-C++, 99.9% of the time you will probably be writing a graphical application for OS X, and needing to integrate your favorite C++ library. A good architecture would dictate proper separation between the two languages by a strong interface. Because of modern GUIs' use of event-driven design, control typically moves through an application from the visible elements, written in Objective-C, to the back-end, where the majority of the C++ code exists. This means that Objective-C code will call the C++ code more often than the other way around.

In what cases might the situation be reversed?

1. Your application might in fact be written using a mix of C and C++ and the Carbon libraries, and you want to take advantage of something in `Cocoa`.
2. Algorithms or routines in your C++ code need access to data structures best kept in the Objective-C realm.

In these cases, it might make more sense to flip our point of view (see Figure 4-14. When we were integrating C++ into our Objective-C application, we created a bridge by using the C++ structures encapsulated in Objective-C++ classes. For integrating Objective-C code into a C++ program, we will do the reverse. Our bridging code will be written as a C++ class holding pointers to the Objective-C objects we wish to incorporate.

From C++, Objective-C objects look like exactly what they are: pointers. In other words, you don't change your normal Objective-C object sytax when refering to Objective-C objects in C++.

We cheated with the previous example by using Interface Builder to side-step the creation of the Objective-C++ object. In this case, we are again avoiding intermingling our C++ code with Objective-C code, except we are using a factory method with which to accomplish this as you will see later on.

The following code example shows how a hypothetical C++ stock cutting optimizer might be able to use Objective-C's Distributed Object (DO) support to communicate with a master server.

The following is the Objective-C class `LClient` that communicates with the server using DO. Its job is act as a proxy for requests up to the server from the C++ optimizer. If you are not familiar with Distributed Objects. they are covered in several chapters, including Chapter 10 and Chapter 14.

```
#import <Foundation/Foundation.h>
#import "LServerProtocol.h"
#import "LClientProtocol.h"


@interface LClient : NSObject <LClientProtocol> {
 id server;
 id delegate;
}
- (id)delegate;
- (void)setDelegate:(id)del;
- (void)registerWithServerName:(NSString*)serverName hostName:(NSString*)hostName;
- (NSDictionary*)getCutList;
- (void)completedWork:(NSDictionary*)cutList;
@end

@interface LClientDelegateProtocol : NSObject {
}
- (void)serverTerminatingForClient:(LClient*)lClient;
@end




#import "LClient.h"

static int runs = 0;

@implementation LClient

- (id)delegate
{
 return delegate;
}

- (void)setDelegate:(id)del
{
 [delegate autorelease];
 delegate = del;
}

- (void)registerWithServerName:(NSString*)serverName
hostName:(NSString*)hostName
{
 server = [NSConnection rootProxyForConnectionWithRegisteredName:serverName
                  host:hostName];
 [server setProtocolForProxy:@protocol(LServerProtocol)];

 [server registerClient:self];

}
```

```
// ClientProtocol Implementation

- (void)serverTerminating
{
 NSLog(@"terminating");

 if(delegate)
  [delegate serverTerminatingForClient:self];
}

// methods encapsulating the server object

- (NSDictionary*)getCutList
{
 NSLog(@"getting cut list");

 if(++runs > 5)
  [self serverTerminating];

 return [server getCutList];
}

- (void)completedWork:(NSDictionary*)cutList
{
 NSLog(@"sending");
 [server completedWork:cutList];
}

@end
```

The Following code is the abstract C++ class that defines a C++ compatible interface into the `LClient` proxy.

```
#include <utility>
#include <string>
#include <vector>

typedef pair<float, float> rectangle;

class LignariusClientProtocol {
public:
 virtual ~LignariusClientProtocol() {}
 virtual void registerWithServer(const string &serverName,
         const string &hostName) = 0;
 virtual vector<rectangle>& getCutList() = 0;
 virtual void completedWork(vector<rectangle>& cl) = 0;

 bool terminated;
};

class LignariusClientFactory {
```

```
public:
 static LignariusClientProtocol* newLignariusClient();
};
```

Earlier we mentioned that we would be using a factory to create our Objective-C++ object. Our C++ code can include the above header, which doesn't include any Objective-C code, and grab a new `LignariusClient` using the following factory method implementation:

```
#include "LignariusClient.h"

LignariusClientProtocol* LignariusClientFactory::newLignariusClient()
{
 return (new LignariusClient);
}
```

Notice in Figure 4-14 that interaction between the C++ and Objective-C objects goes both directions: as with many Objective-C objects, when making DO connections there is a delegate. In our case, we want to feed this delegate information back to C++ objects. We need to feed an Objective-C object to `LClient` when it makes its DO connection, but have that object call back into a C++ object. So, our Objective-C++ interface has two classes declared. One in C++, and one in Objective-C.

```
#include <Foundation/Foundation.h>
#include "LignariusClientProtocol.h"
#import "LClient.h"

class LignariusClient;

@interface LignariusClientCallback : NSObject {
 LignariusClient *callback;
 LClient *client;
}
- (id)initWithCallback:(LignariusClient*)cb lClient:(LClient*)lc;
@end

class LignariusClient : public LignariusClientProtocol {
public:
 LignariusClient();
 virtual ~LignariusClient();
 void registerWithServer(const string &serverName, const string &hostName);
 vector<rectangle>& getCutList();
 void completedWork(vector<rectangle> &cl);
 void serverTerminated(bool term);

private:
 LClient *client;
 LignariusClientCallback *cbProxy;
 vector<rectangle> rectangles;
};
```

The Objective-C object is our DO callback, and the C++ object implements the protocl we defined earlier. Here is the implementation files for both the Objective-C and C++ classes:

```
#import <Foundation/Foundation.h>
#include "LignariusClient.h"

@implementation LignariusClientCallback

- (id)initWithCallback:(LignariusClient*)cb lClient:(LClient*)lc
{
 if(self = [super init]) {
  callback = cb;
  client = lc;
  [client setDelegate:self];
 }
 return self;
}

- (void)serverTerminatingForClient:(LClient*)lClient
{
 if(client = lClient)
  callback->serverTerminated(true);
}

@end

LignariusClient::LignariusClient()
{
 client = [[LClient alloc] init];
 cbProxy = [[LignariusClientCallback alloc] initWithCallback:this
              lClient:client];
}

LignariusClient::~LignariusClient()
{
 [client release];
 [cbProxy release];
}

void LignariusClient::registerWithServer(const string &serverName,
           const string &hostName)
{
 [client registerWithServerName:[NSString
     stringWithCString:serverName.c_str()]
              hostName:[NSString
        stringWithCString:hostName.c_str()]];
}

vector<rectangle>& LignariusClient::getCutList()
{
```

```
 // ...
 [client getCutList];
 return rectangles;
}

void LignariusClient::completedWork(vector<rectangle> &cl)
{
 // ...
 NSDictionary* dict = [[NSDictionary alloc] init];
 [client completedWork:dict];
}

void  LignariusClient::serverTerminated(bool term)
{
 terminated = term;
}
```

As with handling C++ objects from Objective-C, the C++ object model does not automatically take into account the reference counting and autorelease pools of Objective-C. The C++ runtime will not automatically call Objective-C constructors or destructors. When you are finished with an Objective-C object, make sure you properly release it if necessary.

Finally, we can see the pure C++ `main()` implementation.

```
#include <iostream>
#include "LignariusClientProtocol.h"

int main (int argc, const char * argv[])
{
 // insert code here...
 cout << "Starting Client...\n";

 LignariusClientProtocol* c = LignariusClientFactory::newLignariusClient();

 while(!c->terminated) {
  vector<rectangle> cl = c->getCutList();

  // do our work // ...

  c->completedWork(cl);
 }
 cout << "Completed\n";
 return 0;
}
```

4.5.3. Things to Watch Out For

There are several limitations to keep in mind when using Objective-C++. These limitations arise from the fact that when a program incorporating Objective-C and C++ code is run, the runtime systems that support each language are entirely separate, and therefore do not know how to handle each other's objects. For example, you cannot use Objective-C messaging syntax on C++ objects, or intermingle C++ and Objective-C class hierarchies. Check the release notes for a full list of specific language limitations (see Section 4.6). In most cases, if you follow the design suggestion of properly insulating your Objective-C and C++ code from each other, you can avoid many of the problems; however, there are a still a couple of issues with using Objective-C++ you may want to consider.

Using the different exception systems from C++ and Objective-C can be a little tricky. Objective-C's NSException class uses the `setjmp`, `longjmp` calls, while C++'s exception system is build into the language and run-time support. Mixing Objective-C and C++ code that might throw exceptions takes a little extra effort.

Like almost all compilers on the market, support for the full C++ standard in **gcc** is not always what one would hope for. The C++ standard, including the STL, is enormous, and it takes time to implement every feature, and even when a feature is implemented, it may not be stable. The latest ProjectBuilder release includes **gcc** version 3.1, which has much better support for the C++ standard Particularly troublesome for the compiler and library vendors has been full support for generic programming and templates. Many advanced C++ computational libraries make use of more subtle template features, and test compiler implementations to their breaking point. As a rule of thumb, if the C++ code you are interested in using works with **gcc**, you should be okay.

Both Objective-C and C++ impose an overhead on applications over straight C. When you combine the two, you get the overhead of both. What does this mean in practical terms? The first hit you will take is in compile speed. **gcc** is not fast at the best of times, so you can expect its speed on the Objective-C++ portions of your code to be slower than C or Objective-C alone. **gcc** is always getting better in this regard, and there are ways to speed up your compile times.

> NOTE: Currently, ProjectBuilder does not by default precompile headers for Objective-C++ code. This can lead to long compile times when including C++ code in your projects. In order to turn on pre-compiled headers, use the `-cpp-precomp` option. Newer versions of ProjectBuilder include **GCC** 3.1, or newer, and support an entirely new pre-compiled header system. Chapter 5 covers how to use pre-compiled headers in the latest version of ProjectBuilder.

The second hit you will take with Objective-C++ is in application size. However, this is a relatively small penalty. For a minimal program, compiling the same code as Objective-C and again as Objective-C++ results in a 4KB difference. If you have **gcc** generate debugging symbols, the Objective-C++ code will be much larger in order to include the extra information. Application file size is only one half of the equation. You will also see an increase in the runtime memory requirements of your application as it will require the C++ libraries in order to run. As long as you do not statically link the libraries, this cost is amortized over all of the C++ applications running the shared library.

Luckily, there is really no impact on the runtime speed of your application after you have compiled your code as Objective-C++ instead of Objective-C. One of C++'s goals has been to avoid imposing overhead for features programmers are not using. This is one reason garbage collection has been avoided in the standard for such a long time. If you are not using a feature of C++, then you should not see an impact on the runtime performance of your application, apart from the pressure loading larger libraries creates on the memory subsystem. In general, this will only be apparent during program loading, and low memory situations, where the system is forced to swap pages to and from disk to support your application.

With many high performance libraries already in existence for C and C++, the ability to seamlessly integrate this code into `Cocoa` applications can save you the effort of reinventing the wheel. If you need to use C++, Objective-C++ is a great tool to have around. Apple's developer tools let you incorporate mixed C, C++ and Objective-C code all in one application. For this reason, don't think of Objective-C++ as a different language, but more of a C++ compatibility switch you can throw in Objective-C when needed.

4.6. Resources

Resources

class-dump

http://www.omnigroup.com/$\sim$nygard/Projects/

The Objective-C Programming Language

file:///Developer/Documentation/Cocoa/ObjectiveC/ObjC.pdf

Objective-C++ Release Notes

file:///Developer/Documentation/ReleaseNotes/Objective-C++.html

Notes

1. I don't think there is any cause for naming names—we all know which OSs I am talking about (and they were not *all* from Redmond, either.
2. This is one nit pick I have with Cocoa's naming scheme for traditional C structures. You can't know intuitively that `NSWindow` is a class, but NSRect is not. It is easy enough to teach yourself the difference, but creates undue difficulty for beginners.
3. Ignoring the compiler warnings.
4. See Section 4.6 at the end of this chapter for information on where you can get a copy of **class-dump**
5. `self` in this context seems a little strange, but it makes sense if you think about it. To the class method, the receiver *should* be `self`.
6. The double pointer is C shorthand for a pointer to an array of objects, like `char **argv` versus `char *argv[]`
7. Technically, you could implement your own root class that is not derived from NSObject; however, making any objects that were derived from this root class work with the rest of the Foundation or AppKit frameworks would be time consuming. In addition to `performSelector:`, there are many methods implemented by NSObject that are used by the Objective-C runtime and standard libraries that would require duplication.
8. The latest version of **gcc** does support the declaration of variables other than at the beginning of blocks, although, this doesn't include the `for` loop expression above
9. Not to dismiss GNUStep and other Unix operating systems

Chapter 5. Foundation

5.1. Objective-C Performance Considerations

5.1.1. Object Creation

TODO

separate init and alloc—why?

5.1.2. Memory Usage

TODO

zones

5.1.3. Message Passing

TODO

selectors versus function pointers, C++ virtual function table, etc.

use IMP to speed up

5.1.4. Object Destruction

TODO

5.2. The Foundation Framework

talk about organization, maybe a layout of classes, etc. talk about CoreFoundation

5.2.1. CoreFoundation

todo

5.3. Containers

```
FROM: Chris Kane
DATE: 2001-08-20 19:29

On Monday, August 20, 2001, at 02:49 ÊAM, John C. Randolph wrote:
```

> On Sunday, August 19, 2001, at 06:34 ÊPM, Chris Kane wrote:
>> Mutable collections tend to have "extra slots" allocated to avoid Ê
>> reallocation on every addition, which kills performance.
>
> I don’t suppose you have by any chance, any plans for a fragmentable
> array for those occasions where you might have a *lot* of objects in
> the array? ÊOr, say, NSReallyBigAndNotNecessarilyContiguousData, for
> the future when we have a 64-bit address space?

Well, I suppose we could implement something like that.

<< Chris waves his magic wand, or maybe he wriggles his nose, or maybe he folds his arms and nods his head ...; there’s a pinging sound >>

OK, your copy of Mac OS X 10.0 now does this for large arrays. ÊWith NSArray at least.

NSData is a tougher case, though years ago when we thought about this, that was where we thought to do it. ÊBut the API of NSData promises a contiguous buffer via the -bytes method. ÊSo, you say, we could delay the continuity until that is requested and concat the buffers at that point. ÊAnd, we could, but don’t. ÊWe ran some tests with existing apps at the time and found that that the -bytes (or -mutableBytes) were usually requested fairly quickly after creation, and not usually after all appending was done. ÊSo we decided the extra complexity wasn’t worth it.

NSString does also use the same technique as NSArray, however, and it finds its way into NSTextStorage backing stores as well. ÊThe downside is that accessing a byte at random in such a structure is O(lg N).

Chris Kane
Cocoa Frameworks, Apple

Hi John,

Check out the CoreFoundation source code some time. There are several implementations of mutable CFArrays: for small arrays the implementation is a ring buffer (which is efficient for queues), and for very large arrays the implementation is to use a CFStorage, which looks sort of like an in-memory b-tree. I’m not sure what you mean by "fragmentable", but this probably qualifies.

Since NSMutableArray uses CoreFoundation, you’ve already got what you are looking for, more or less. It’s true that the array isn’t thread-safe, so you do need a couple more lines of code to write a thread-safe FIFO.

--Greg

```
http://www.cocoadev.com/index.pl?ClassClusters

On Monday, September 9, 2002, at 06:36 PM, Bill Bumgarner wrote:

On Monday, Sep 9, 2002, at 19:13 US/Eastern, Mike Shields wrote:
On Monday, September 9, 2002, at 07:25  AM, Bill Bumgarner wrote:
I need to create a set of maps between integer keys and either objects or integer values, depending

I have long used NSMapTable's API in this context, but was curious if CFDictionary's API is now pref

Right. The NULL values for retain/release are exactly what you're looking for. You'd probably also w

But this makes the assumption that sizeof(int) == sizeof(void *) -- i.e. if the code were to move to

There's no problem if, say, pointers go to 64-bits and ints stay at 32 (and you're saying "int" beca

Now, if pointers stay 32-bits and int goes to 64, then there are many int values that won't squeeze

"int" is a very generic type that can change.  If you really don't care too much about the range of

Or use CFNumbers as you said.
```

see this thread: http://cocoa.mamasam.com/MACOSXDEV/2001/08/1/9764.php

priority queue

```
Ah, but here is where you are mistaken. You are making the assumption
that NSMutableArray is implemented internally as a simple C array. In
fact, NSMutableArray is implemented in terms of CFMutableArray, and the
CoreFoundation project is part of Darwin so you can go look at the
source yourself. If you do so you will find that there are, in fact,
three different implementations of mutable array that the code switches
among depending upon the size of the array and whether that size is
fixed.

The most common implementation (for a non-fixed but small size - which
is what NSMutableArray will use unless you add many many objects or give
it a very large capacity on initialization), is actually a circular
buffer!

Thus, [array objectAtIndex:0] is O(1), [array removeObjectAtIndex:0] is
O(1), and [array addObject:newObject] is O(1) - unless the buffer needs
to be expanded. That seems like a pretty decent queue implementation to
me...
```

hash functions in core foundation SUCK. here is a fix:
http://www.mulle-kybernetik.com/artikel/Optimization/opti-7.html

Talk about Umbrella Frameworks (find software on mamasam that shows what frameworks an umbrella framework provides)

5.4. Project Organization

TODO: reorg

5.4.1. Speeding Compile Times

Up to this point

5.4.1.1. Precompiled Headers

TODO:pfe

5.4.1.2. Using `#import`

TODO: describe `#import`

5.4.1.3. Have a little `@class`

Using the `#import` functionality of the Objective-C language provides protection against importing a header file more than once; however, it does not protect you from importing header files that you don't need in the first place. Often in the Model-View-Controller (MVC) pattern, you will find yourself with `controller` objects that refer to, and by, both the `model`, and the `view`, side of the application. Traditionally, the `view` objects use `AppKit` and `Foundation`, while `model` objects only need to refer to `Foundation` objects. If you create your headers just right, you can over-complicate the build process by including `AppKit` headers in your `Foundation`-only classes. The following simple headers are a good example of this pathological behavior, from which `#import` isn't able to help us:

```
// MyController.h

#import <Foundation/Foundation.h>
#import "MyModel.h"
#import "MyView.h"

@interface MyController : NSObject {
    MyView *view;
    MyModel *model;
    ...
}
...
@end

// MyView.h

#import <Cocoa/Cocoa.h>
#import "MyController.h"

@interface MyView : NSView {
    IBOutlet NSButton* button;
```

```
    MyController *controller;
}
...
@end

// MyModel.h

#import <Foundation/Foundation.h>
#import "MyController.h"

@interface MyModel : NSObject {
    NSMutableArray *data;
    MyController *controller;
}
...
@end
```

Even though `MyModel` doesn't need to know anything about `AppKit` (or even `MyView` for that matter), in this layout, it ends up including it because it imports the header file for `MyController`, which in turn imports the header file for `MyView`, which *does* import the `AppKit` headers when it calls:

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

While precompiled headers will go a long way toward helping, you could fix this problem manually by rearranging how your classes interact. For example, you could make sure that `MyModel` knows nothing about the controller class. While good application design will prevent such dependencies, it doesn't always make sense to spend too much time worrying about nested imports three or four levels deep.

One easy solution is to use Objective-C's `@class` expression to forward declare classes in header files. We could rearrange the previous header and implementation files as follows:

```
// MyController.h

#import <Foundation/Foundation.h>

@class MyView, MyModel;

@interface MyController : NSObject {
    MyView *view;
    MyModel *model;
    ...
}
...
@end


// MyController.m

#import "MyController.h"
```

```
#import "MyModel.h"
#import "MyView.h"

@implement MyController
...

// MyView.h

#import <Cocoa/Cocoa.h>
@class MyController;

@interface MyView : NSView {
    IBOutlet NSButton* button;

    MyController *controller;
}
...
@end

// MyView.m

#import "MyView.h"
#import "MyController.h"

@implement MyView ...

// MyModel.h

#import <Foundation/Foundation.h>
@class MyController;

@interface MyModel : NSObject {
    NSMutableArray *data;
    MyController *controller;
}
...
@end

// MyModel.m

#import "MyModel.h"
#import "MyController.h"

@implement MyModel
...
```

Note that *none* of the header files include anything except the `Foundation` framework header. Instead, each header file uses `@class` to forward declare the class names specifically mentioned in the header file. We don't worry about the `Foundation` headers, since every Objective-C class must include it. Now, you know that no matter what custom header file you include, you will only get the class declarations in which you are interested.

Now, when `MyModel` needs to include `MyController.h`, it does not drag along the `AppKit` header files to boot. This forward declaration of classes keeps header files free from extraneous baggage, and speeds the compile time for classes that are not interested in the entire `Cocoa` object hierarchy. This works just as well to keep interdependence between classes in your own projects down as well.

> TIP: Consider forward declaring *all* non-foundation classes in your header files, and only `#import` their declarations in the implementation files. This will greatly reduce interdependencies and as a result also reduce build times.

5.4.1.4. Compiling C++ and Objective-C++

```
i had lots of trouble converting our project (to 10.2 with gcc 3.1)
which uses a lot of  .CPP, .MM and .M  files

but now i’m happy. it works better than before. here’s what i’ve done :

I have created GlobalPrecomp.h

In this file, I include every system header  + our own headers (the ones
that do not change often). The trick is to put objc only (.MM and .M) in
#ifdef __OBJC__ blocks, and C++ only headers (.CPP and .MM) in
#ifdef __cplusplus blocks.
(for example, I have Cocoa.h, Carbon.h, QuickTime.h, iostream etc in
this header)

1) I choose this file to be the prefix header
2) I click on ’precompile it’
3) in gcc settings, i choose "use PFE"
4) i’m not using any special flag like -cpp-precomp

And now it is really fast (more than 800% speed increase wrt not using precomps)

Hope this helps

Benjamin
```

5.5. Modularizing Applications

this section on dealing with frameworks, plugins, bundles, etc.

Using Bundles

Plugin Architecture

weak linking

5.6. Resources

todo

Resources

F-Script

http://www.fscript.org/

Chapter 6. Inside OS X

6.1. The organization of OS X

todo

6.2. Mach

todo

6.2.1. Ports

todo

6.2.2. Virtual Memory

todo

6.3. The Mach-O Executable Format

Like PE on Windows and ELF on Linux, Mach-O is the executable file format for native binaries compiled from C, C++ or Objective-C on OS X.

Is Mach-O slow?

6.4. Resources

Resources

Mach-O Runtime Architecture Document

http://developer.apple.com/techpubs/macosx/DeveloperTools/ MachORuntime/MachORuntime.pdf

Chapter 7. Processor

7.1. Looking at CPU Usage

todo

7.2. Compiler Optimizations

todo

1. Don’t use externs or static variables. 2. If you are going to use an extern variable in a tight loop, don’t use a local variable and assign it after the loop. 3. Pass the option -mdyanmic-no-pic to gcc if the source is in the final program because it does not work in a boundle or a dynamic library (or framework). The AIX ABI/PEF ABI uses a register called the TOC for PIC code but it is stored with the function reference so you lose one register if the Darwin ABI goes over to the PEF ABI. You get one more register to play around with if you do not use extern or static variables. Why do people hate Muslims?

Does anyone have a Cunning Strategy for how to work this out, or do you just pick semi-random addresses below 0x3fffffff? It doesn’t matter, just ensure that it has an address on a page boundary. In 10.2, fix_prebinding will adjust things so they "just work".

inline

talk about dangers of using #define (check out stroustrup’s latest on avoiding macros--bad. Show undefined behavior for something like #define cube(x) ((x)*(x)*(x)) when thrown *ptr++.

7.3. Choosing Algorithms

todo

7.4. Traditional Hand Methods

todo

7.5. OS X Scheduling Considerations

todo

7.5.1. The OS X Scheduler

todo

```
Darwin has the equivalent capabilities of both the low latency and preemption patches floating aroun
```

```
There were some studies done with audio latencies that showed Mac OS X/Darwin having the best overal
```

```
I haven’t seen anything published since then, but I think the results would be comparable (and quite

--Jim
```

7.5.2. Real-time Scheduling

todo

Chapter 8. Memory

When talking about an object's lifecycle, we are referring to everything that happens from the time it is created to the time it is destroyed. Just like real life objects, every object in your program consumes environmental resources while being made, during its lifetime, and finally being destroyed. These resources may be processor clock cycles, memory usage, or I/O with the network or a disk. No matter what the case, ultimately it is a consumption of the most scarce resource of all: time.

8.1. The OS X Memory Manager

todo: compare Darwin MM to NT/Linux/Solaris, etc.

8.2. Looking at Memory Usage

todo

explain **top**

this number is nothing special. You should use top -w and look onto VPRVT RPRVT(delta) columns to see the "real" numbers. Check man top for explanations. This is not the amount of memory it uses, first of all vram is video ram, you're probably refering to the VSIZE, vsize is the total size of the allocated addressing space, note this includes any libraries, bundles etc the process loads. As you might know each process runs in it's own addressing space on MacOS X, Which is necessary to implement protected memory, the amount of memory it actually uses are reflected in the RSIZE.

8.3. UNIX Memory Allocation

malloc. Individual calls return aligned memory (what is the step size? 1K, page size, etc.). If you are using MANY small objects, better to use a pool design.

Alignment of structure depending on start element size, unintended padding? 1) The struct starts with a double. This means that sizeof must be a multiple of sizeof(double) to preserve alignment in the case that you have an array of the structs 2) This particular struct ends up leaving 4 bytes at the end of the struct unused due to the padding inserted for #1 3) Since this struct is used as a superclass, the compiler somehow realizes that you *cannot* have an array of the structs and thus the extra four bytes at the end of the superclass is free for use by the subclass. Thus, the first member of my subclass gets packed into the area 'owned' by the superclass.

from BadAndy: MacArsian Bloodhounds on the trail..... Folks, before going to more discussion... I think this thread is an example of MacAch at it's best and also the general value of a significant user group ... a problem gets posted, folks chime in, information comes in quicker and at lower individual cost (and across more systems) than any one person could easily manage... and we are getting somewhere. Dete makes some good general background points ... one of which I needed to be reminded of (once I saw it I said "I knew that" ... but I hadn't been thinking of it earlier): quote: So there's two levels of memory allocation going on here: malloc(), the "high level" (which can allocate buffers of any size) and the low level system allocator (vm_alloc?), which always allocates a page at a time. When you allocate a page at the low-level, nothing really happens except that an extra page is added to the VM table for that process, and the addressing space is reserved. Only when you access that addressing space does something "really" happen: A page of physical memory gets assigned to that process, possibly causing someone else's page to

be swapped to disc. Newly allocated pages are *always* zero’d out in Unix, when you first access them. This is a security feature to keep you from being able to read the data from other processes. (Interesting aside: This is the reason why the C language specification guarantees that global variables will be zero in a newly launched program...) To be clear: it is not malloc() that is writing zeros to these blocks, it is the OS. {snip} What this means is that for small allocations, such as those used in the original example, malloc touches every page and thus the VM system must create real pages for each allocated block, swapping out everyone else’s pages, and eventually its own. It’s this last issue about the zeroing which is important in the context of understanding what this bomb does at runtime ... it is actually creating pages, filling them with zeros (zeroing is very efficient on G4/G4+ because it can be done with dcbz instructions... which zero whole cache-blocks without reading them in first), and then it is creating the local datastructures for the allocated memory. {incidentally, in the real-time hardware OS domain I play in, the zeroing generally doesn’t occur because security is derogated in the interest of speed ... security is that these are closed systems with no external users, and are generally running a static locked-down complete code-store, and processes "browsing" other processes paged-out space is presumed to be irrelevant) Dete is correct about this in Unix, but it doesn’t change my fundamental characterization of this as a bug. Of course, folks need to understand that I am an absolute hardnosed purist in these regards, coming from a hardware/embedded/realtime high-reliability perspective ... but then the whole point of a modern multi-tasking OS is to deliver the highest reliability and security it can. IMO anything that baldly compromises this is absolutely a bug. quote: About Bad Andy’s assertion that it is a bug: First, as I explained above, since malloc is writing to those pages, as far as the kernel is concerned those pages *are* being used. So the malicious code is allocating AND USING as much memory as it will be allowed. So what’s the bug? That a user-level process can initiate enough swapping so that other user-level processes are unusable? It’s not ideal, but I’m not sure you can call it a bug. It is a bug because it is a trivial mechanism for denial-of-service attacks ... and by definition we expect no single user to be able to halt services for other user processes OR processes at higher priorities. Incidentally, in this context the discussion hasn’t included the issue of user level ... if this problem exhibited itself only if the process was launched from root then I would _still_ find this behavior "far less than ideal," but I might accept abstract notion that it is "not a bug... because root gives you the suicide switch if you are stupid enough in endless other ways too." The problem here is that this is a user-level disaster. To put it concretely... suppose I’m running a "real" big Unix machine with 500 first-year comp. sci. kiddies (BTDT ... as a grad-student TA) and every goddam time one of them makes a stupid endless memory allocation error the whole OS comes down. You going to argue this isn’t a bug? OK, so will somebody post this to Apple’s bug-tracker? (I’ve posted so many, and I’m not the originator here.) There is a reasonable chance this got fixed with Jaguar, but if not it needs to be fixed PRONTO. Folks, lets be honest ... this is an embarrassing bug to have so blatently in our faces. Competent programmers should have tested/thought about it. If this kind of thing were apparent on WinXP there would be howls of castigation from all sides about crapware ... what is sauce for the goose must be for the gander. Yes, Darwin/osX is still green, but this is an _embarrassing_ bug to have. Any decent VM/tasker needs to guard _resources_ for many processes as well as just time-slices. And inadvertent malloc bombs are so common as programming mistakes that it is just plain common sense that a VM hitting disc-storage-space limits (and thus faced with killing/crippling MANY running processes unless it makes a triage decision) should kill low-priority user-tasks as needed starting from the largest memory allocations down ... to keep higher priority tasks running.

The POSIX API is the supported and recommended way of doing shared memory (shm_*).

8.4. Objective-C Memory Allocation

easy tool: use init and dealloc to store increment/decrement a global counter to see how many objects are around at the end of an application

look at alignment:http://www.mulle-kybernetik.com/artikel/Optimization/opti-2.html use class-dump to look at offsets of instance variables

8.4.1. Autorelease Pools

Autorelease Pools: SIGSEGV/SIGBUS at pool release means autoreleasing a released object with a *very high* probability

8.4.2. Zones

todo

8.5. Strategies for Improving Memory Performance

todo

8.6. Shared Libraries

todo

Chapter 9. Disk

9.1. Looking at Disk Usage

todo

9.2. UNIX File Support

todo

9.3. Cocoa File Handling

todo

9.4. Coders and Archiving

todo

9.5. Implications for Object Design

todo

9.6. Database Access

todo

Chapter 10. Network

10.1. Looking at Network Usage

todo

10.2. Traditional Socket Programming

todo

talk about different socket implementations: SimpleSocket EDNetworking?, OmniNetworking, CFSocket.

Hi, I'm having a trouble with efficiency in SmallSockets. My problem occurs with a latency in when text arrives and the speed in which I can process it. Currently I'm developing a text-based game client which connects to a game server via a simple TCP/IP connection. My problem is that when I go to read from the socket, I first have to put the data from the socket into a NSData, read from the NSData into a NSString, then insert the NSString into a NSTextView. Under heavy load (10-20 lines/second) this seems to become very bogged down, taking a few seconds to 'catch up' to where the current status is. Here's my current working code, the function is run on a seperate Thread:

```
while([mySocket isConnected])
{
    if([mySocket isReadable])
    {
        [mySocket readData: myData];
        while([myData containsString: @"\n"])
        {
            line = [[NSString alloc] initWithData: [myData getBytesUpToString: @"\n"] encoding:
            [mainText setEditable: YES];
            [mainText insertText: line];
            [mainText setEditable: NO];
            [line release];
            [mainText moveToEndOfDocument:mainText];
            [mainText scrollRangeToVisible:[mainText selectedRange]];
        }
    }
    else
    {
        [NSThread sleepUntilDate:[NSDate dateWithTimeIntervalSinceNow:0.05]];
    }
}
```

A couple observations: (1) Don't sleep after the read, just use blocking mode (which is the default) to wait for data to arrive rather than calling isReadable. Use an NS_DURING/NS_HANDLER block to catch disconnection and other exceptions. If you're running your network code in a separate thread as you indicate (and as you should), this is much more efficient; the sleepUntilDate call is probably contributing substantially to your performance problems. (2) Make sure to reserve enough space in myData (using dataWithCapacity: or initWithCapacity:) to avoid it having to resize during readData. (3) You may have problems modifying the NSTextView in a thread other than the main UI

thread. You should consider using something like Toby Paterson's InterThreadMessaging or other code to pass the NSData to the main thread. (4) You should insert text into the view by manipulating the NSTextView's textStorage object directly, rather than using insertText. See NSTextStorage for more info (and the key point to remember is that NSTextStorage is a subclass of NSMutableString). (5) If your data is already text, you can probably just stuff the whole thing into the text storage at once, unless there's some reason you need to first split it into separate lines as you show in your while() loop--but I'm not sure what you may be doing to the incoming text. Hope this helps,

There are many different interprocess communication mechanisms available to you on Mac OS X. Here are a few of them, roughly in increasing order of layering: Mach ports, sockets, CFMachPort, CFSocket, CFMessagePort, distributed notifications, AppleEvents, NSMachPort, NSSocketPort, NSMessagePort, distributed objects. For broadcast local-machine communications, the natural choice would be distributed notifications. For connection-oriented communications between Cocoa apps, the natural choice would be distributed objects. You can find both of these documented on developer.apple.com, but here is a page that is particularly on point for system preferences:
http://developer.apple.com/techpubs/macosx/AdditionalTechnologies/PreferencePanes/Tasks/Communication.html
Douglas Davidson

10.3. Security: SSL

todo

10.4. Distributed Objects

todo

```
  Thanks very much.  Given this example, I was able to find an earlier
message from Chris Kane at apple (2/24/01) showing the same approach.  I
tried it this way, and it works.  The only downside is it appears you
have to know the host where the server is located ahead of time
(i.e. "*" does not seem to work for hostname).  But that's much better
than it not working at all.

To sum up for anyone else looking through the archives in the future, DO
and NSConnection work on the network (OSX), but currently you have to
set up the port manually as this example shows.

Mike Giddings

On Thursday, June 21, 2001, at 08:35 PM, Larry Campbell wrote:

> I believe you have to construct a port from an NSSocket. Here's a
> couple of snippets of code I wrote a while back that I know worked.
>
> Server:
>
>     NSSocketPort *port = [[NSSocketPort alloc]
> initWithTCPPort:SERVER_PORT];
>     NSConnection *theConnection = [NSConnection
```

```
> connectionWithReceivePort:port sendPort:nil];
>
> Client:
>
>     NSSocketPort *port = [[NSSocketPort alloc]
> initRemoteWithTCPPort:SERVER_PORT host:@"172.24.81.250"];
>     NSConnection *connection = [NSConnection
> connectionWithReceivePort:nil sendPort:port];
>
>
```

10.5. The Distributed Object Lifecycle

todo

10.6. Protocol Design Consideration

todo

Chapter 11. Basic Vector Programming

11.1. TODO

```
<ulink url="http://arstechnica.infopop.net/OpenTopic/page?a=tpc&s=50009562&f=8300945231&m=8790959504
```

11.2. Introduction to Vectorization

Modern processors have come a long way in closing the gap historically present between custom supercomputers and desktop PCs. Pipelining, super-scalar architectures, reduced-instruction-set-computing, and other advances once the sole purview of extremely expensive hardware, are now integrated into almost every microprocessor in one form or another. Most of these advances have been introduced in order to speed the execution of general-purpose applications, and do not require explicit support from either the programmer, or in many cases, even the compiler. In a sense, we benefit from these faster processors for free--simply run your existing code on the new processors, perhaps with only a simple re-compile, and watch the seconds, minutes, or even hours fall off your execution time.

Unfortunately, these techniques, while impressive, can only take us so far. Personal computers have moved from devices almost solely used in business environments for word-processing and spreadsheets to machines used as media, gaming and, more importantly for our discussion here, scientific and engineering workstation platforms. These new applications all benefit from a different kind of processor optimization. What all of these applications have in common is their dependence on tight loops of code that perform identical operations repeatedly on a large set of data. It doesn't matter if these data are pixels destined for a 3D scene in Quake III, video streams in an MPEG movie, or vector fields in a fluid dynamics simulation of a nuclear explosion.

One operation typical of these kinds of applications is computing the dot product of two vectors `a` and `b`:

$$
 c = \sum_{i=0}^{n-1} a_i b_i
$$

If for the sake of argument we ignore some of the techniques presented in previous chapters (e.g. loop unrolling), we might naively implement the equation above for a fixed, four-element vector in C as follows:

```
float dot_product(float *a, float *b)
{
    float c = 0.0;
    int i = 0;

    for(i = 0; i < 4; ++i) {
        c += a[i]*b[i];
    }

    return c;
}
```

Figure 11-1. Order of Instruction Execution for Scalar Implementation of Dot Product

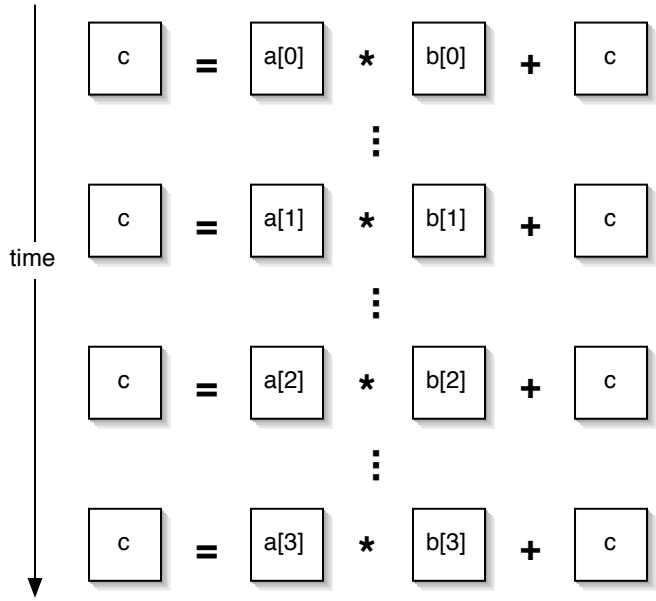


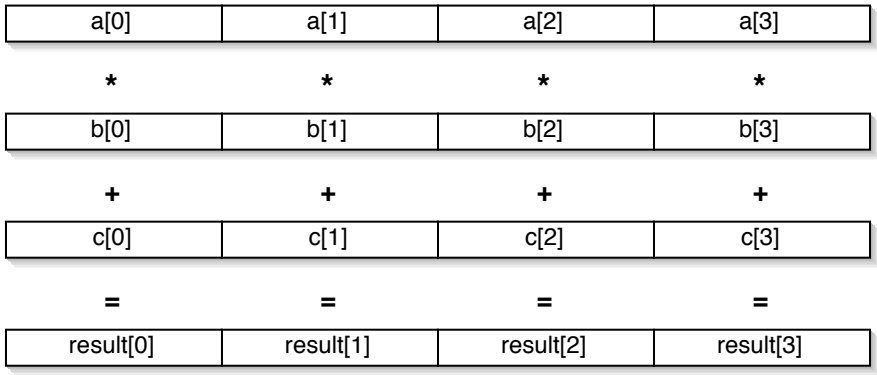
Figure 11-1 shows graphically how the `dot_product` function works through one element of the vector at a time, multiplying each element from vector `a` with its corresponding element from vector `b`, and then adding this value to `c`. In this figure, time moves vertically down the page, and all operations together on a horizontal line occur at the same time.

I compiled the `dot_product` function using `-O3` optimization, and then looked at the assembly code in **gdb** using the `(gdb)` **`disass dot_product`** command. (See Chapter 7 for information on this, and other useful commands). The five most important lines from the output have been presented below. These instructions implement the `for` loop that performs the actual dot product evaluation:

```
0x1dfc <dot_product+40>:        lfsx    f13,r9,r3
0x1e00 <dot_product+44>:        lfsx    f0,r9,r4
0x1e04 <dot_product+48>:        addi    r9,r9,4
0x1e08 <dot_product+52>:        fmadds  f1,f13,f0,f1
0x1e0c <dot_product+56>:        bdnz    0x1dfc <dot_product+40>
```

This dot product implementation requires five instructions for every element in the vector. The miscellaneous housekeeping instructions before and after the loop are not shown here. Notice the `fmadds` instruction. This opcode performs both a multiply and an add at once. This operation is represented in Figure Figure 11-1 in that the add, multiply and assignment are all shown horizontally together. The other four instructions fill the registers with the vector elements needed by the `fmadds` instruction (`lfsx`), as well as check for when it is time to exit out of the loop (`bdnz`). Computing the dot product for a four element vector one hundred million times using this routine takes approximately 3.87 seconds on my 500MHz G4.

Figure 11-2. The vec_madd AltiVec Instruction



11.2.1. An AltiVec Dot Product

Below is an implementation of the same dot product operation, this time using the G4's AltiVec engine.

```
float altivec_dot_product(vector float a, vector float b)
{
    vector float zero = (vector float) vec_splat_s8(0);
    vector float tmp;
    float c;
    int i;

    tmp = vec_madd(a, b, zero);
    tmp = vec_add( tmp, vec_sld( tmp, tmp, 4));
    tmp = vec_add( tmp, vec_sld( tmp, tmp, 8));

    vec_ste(tmp, 0, &c);
    return c;
}
```

As you can immediately see, implementing an algorithm in AltiVec is not a straightforward affair and often does not follow the conventions programmers are familiar with when programming for the scalar arithmetic units of most processors. First, notice that instead of taking pointers to floats as its arguments, the AltiVec version expects a vector float. The vector engine, not surprisingly, works on *vectors*. The vector float you see here is one example of an AltiVec vector. In this case, a vector float holds four single-precision floating-point values. Second, notice there is *no loop*. The fused multiply-add operation `vec_madd` takes in the entire vectors `a` and `b`, as well as (in this case) a `0` vector. The multiply-add instruction operates on every element of each vector at once. Figure 11-2 shows how the individual components of the vectors are used to compute the final result.

This idea of operating on multiple data points--a vector of four floats--with a single instruction--the `vec_madd` call--is the core concept behind the AltiVec engine, and vector machines in general. This computing model has its own acronym called SIMD, or Single Instruction, Multiple Data. Contrast this approach to our first dot product implementation, where every element in the first vector had to be individually multiplied with its corresponding element in the second vector. This more traditional approach taken by the arithmetic units of most processors is

known as SISD: Single Instruction, Single Data. We also refer to the traditional arithmetic unit in this text as the *scalar* unit to distinguish it from the *vector* AltiVec unit.

If the `vec_madd` instruction is a good example of the benefits of vectorized programming, the next lines in the AltiVec dot product are a good example of one of the down-sides. The two lines in the AltiVec implementation after `vec_madd` add up the four resulting floats held in `tmp` for the final dot product solution. Unfortunately, the AltiVec implementation does not provide a single instruction for adding up the individual components of a float vector together--that is $c = a_0 + a_1 + a_2 + a_3$. There is one for integers, but not for floating-point numbers. The two `vec_add` with embedded `vec_sld` calls are one way of producing the sum. This is an example of the kinds of hacks that are typical with AltiVec optimizations. In this chapter and the next we will take a closer look at this, and other interesting ways of getting things done with AltiVec.

As with the scalar implementation of the dot product, let's take a look at the assembly generated by the compiler for the AltiVec dot product. The equivalent assembly in `altivec_dot_product` for the `for` loop in the scalar `dot_product` function is:

```
0x2d84 <altivec_dot_product+24>:        vmaddfp v3,v2,v3,v1
0x2d88 <altivec_dot_product+28>:        vsldoi  v0,v3,v3,4
0x2d8c <altivec_dot_product+32>:        vaddfp  v3,v3,v0
0x2d90 <altivec_dot_product+36>:        vsldoi  v1,v3,v3,8
0x2d94 <altivec_dot_product+40>:        vaddfp  v3,v3,v1
```

Like the scalar implementation, the assembly amounts to five instructions, with one important difference: where the scalar dot product implementation had to go through the loop four times--once for every element in the array, the AltiVec implementation only requires five instructions for the *entire* computation. You can see in the assembly output that the opcodes match very closely with the function calls in the C source code. In fact, while programming AltiVec from C parses as function calls, the programming model is significantly different. We will look more closely at how AltiVec programming in C is implemented in the next section.

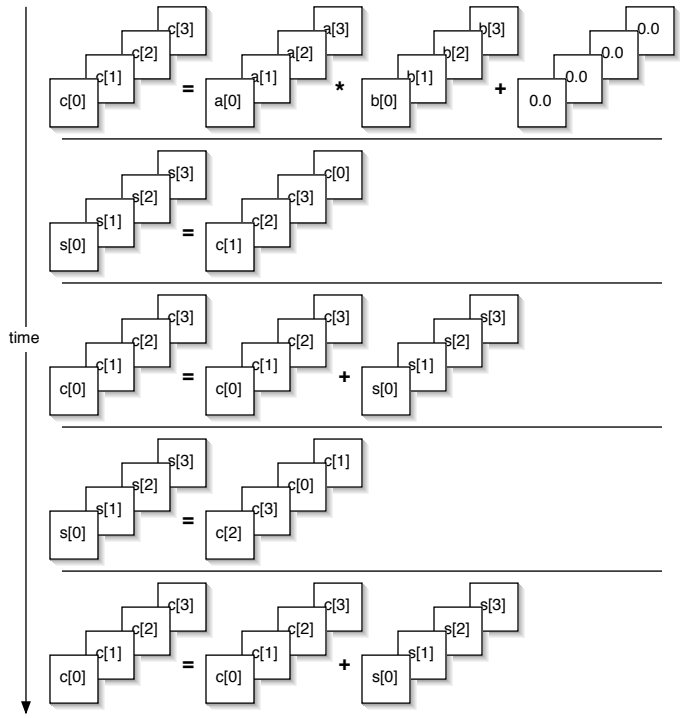
> **Tip:** One issue with looking at the assembly for altivec code is that **gdb** does not always give you the opcode mnemonic you are expecting. The first operation: `vmaddfp v3,v2,v3,v1` actually prints out in **gdb** as `.long 0x106208ee`. Appendix B of Motorola's "AltiVec Technology Programming Environments Manual" provides the binary representation of all the AltiVec instructions, so you can figure out what instruction is being executed, as well as what registers are being used. See Section Section 11.7 at the end of the chapter for references to the Motorola documentation

Figure 11-3 shows how the vectorized dot product implementation moves through the AltiVec engine. This implementation of the dot product takes approximately 2.24 seconds to complete one hundred million times on a 500MHz G4. This is a 42% improvement over the time required for the traditional scalar implementation of the dot product. We have almost cut our time to execute a dot product in half!

11.2.2. Now for the Real World

Given these numbers, I'm sure you are ready to start rewriting all of your code to take advantage of the AltiVec engine. In fact, instead of preparing for a massive hacking session, you might instead be asking yourself: if both implementations have the same number of assembly instructions, but the scalar version has to execute its version

Figure 11-3. Order of Instruction Execution for AltiVec Implementation of Dot Product



four times to complete the algorithm, why isn't the AltiVec version four times faster instead of less than two? It turns out that while the two dot product implementations make for relatively good examples of scalar versus vector programming, they are not really very good at getting the most out the PowerPC hardware. Unfortunately, the dot product example eschews a little practical reality for the sake of pedagogy.

In the case of the scalar dot product, placing a `for` loop in the code when we know we are working with a fixed four-element vector isn't really that smart. Instead of reprogramming for AltiVec, we could have simply unrolled the loop:

```
float dot_product(float *a, float *b)
{
    float c = 0.0;

    c += a[0] * b[0];
    c += a[1] * b[1];
    c += a[2] * b[2];
    c += a[3] * b[3];

    return c;
}
```

If we fix this glaring issue, things suddenly take a turn for the worse. The scalar dot product implementation completes in 2.45 seconds - a 37\% improvement over its looping brethren. The AltiVec version has gone from

dropping 42\% off of the scalar time, to just nudging it out by 9\%! Suddenly, that overnight hacking fest to wrangle your code into using AltiVec doesn't sound like such a good use of your limited resources. Surely, these aren't the heart-pounding numbers we were expecting from AltiVec. What are we doing wrong?

It turns out that while the AltiVec engine can dispatch operations on entire vectors at once, it doesn't actually *complete* them in once cycle. Depending on the instruction, it can take from one to five clock cycles for the AltiVec engine to complete an operation. To make sure we are not spinning our wheels waiting for the first instruction to complete, the AltiVec engine can start additional operations, as long as they do not depend upon the result of any pending operations. In the case of our dot product code, every call to the AltiVec engine depended upon the previous call. This meant our function was stalling between instructions. `vec_madd` and `vec_add` both require 4 cycles to complete. By stacking one operation on top of another with interdependencies, we were causing the AltiVec engine to stall. The following code shows how we might overcome this problem:

```
vfloat altivec_dot_product(vector float *a, vector float *b)
{
    vfloat result;
    vector float zero = (vector float) vec_splat_s8(0);
    vector float tmp1, tmp2, tmp3, tmp4;

    tmp1 = vec_madd(a[0], b[0], zero);
    tmp2 = vec_madd(a[1], b[1], zero);
    tmp3 = vec_madd(a[2], b[2], zero);
    tmp4 = vec_madd(a[3], b[3], zero);
    tmp1 = vec_add( tmp1, vec_sld( tmp1, tmp1, 4));
    tmp2 = vec_add( tmp2, vec_sld( tmp2, tmp2, 4));
    tmp3 = vec_add( tmp3, vec_sld( tmp3, tmp3, 4));
    tmp4 = vec_add( tmp4, vec_sld( tmp4, tmp4, 4));
    tmp1 = vec_add( tmp1, vec_sld( tmp1, tmp1, 8));
    tmp2 = vec_add( tmp2, vec_sld( tmp2, tmp2, 8));
    tmp3 = vec_add( tmp3, vec_sld( tmp3, tmp3, 8));
    tmp4 = vec_add( tmp4, vec_sld( tmp4, tmp4, 8));

    vec_ste(tmp1, 3, &(result.e[0]));
    vec_ste(tmp2, 3, &(result.e[0]));
    vec_ste(tmp3, 3, &(result.e[0]));
    vec_ste(tmp4, 3, &(result.e[0]));

    return result;
}
```

Instead of passing in two vectors `a` and `b`, we are passing in an array of vectors. In this case, we are assuming that `a` and `b` both hold four vector float values. Notice how we have interleaved the operations for computing the dot products for each vector. First, we send off all four pairs of vectors to be multiplied, then we perform both shift/add combinations. By interleaving the operations, we ensure that the AltiVec engine has plenty of work to do. Table Table 11-1 shows the new times for both our naive and optimized implementations.

By making sure the AltiVec pipeline is full, we are able to get a four to one reduction in computation time over the naive AltiVec version, and 4.4 times faster than the unrolled scalar dot product. This is almost exactly what we would expect from an execution unit that works on four floating-point values at one time. As with any benchmark, your mileage may vary. These numbers do not take into account function call overhead, and they exercise their respective

Table 11-1. Naive vs. Optimized Dot Product Implementations

	Naive	Optimized
Scalar:	3.87s	2.45s
AltiVec:	2.24s	0.56s

execution units using only a very small amount of data, which means that hardware limitations such as memory bandwidth do not impact these results. Working with AltiVec can be difficult, but for the right kinds of problems, the rewards can be well worth it.

Hopefully this introduction to AltiVec has given you an idea of both the upside potential, and downside issues related to optimizing your code for AltiVec. The rest of this chapter will focus on familiarizing you with the AltiVec programming model, keywords and functions so that you can starting thinking about how you might reimplement algorithms to take advantage of AltiVec. The next chapter will start looking at more advanced AltiVec issues, such as optimization issues and available libraries.

11.3. The AltiVec Engine

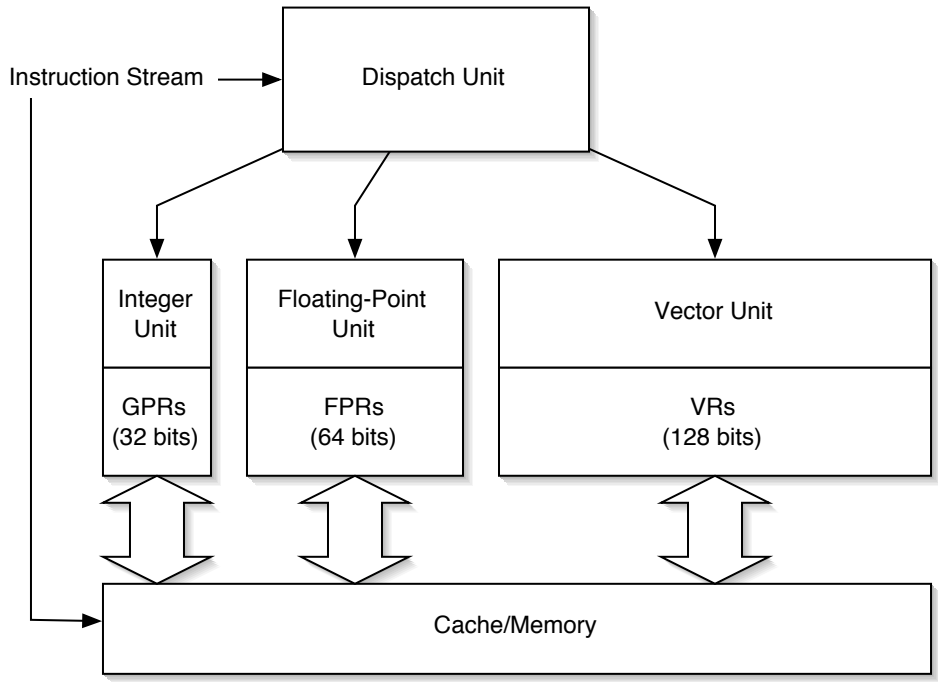
Up to this point, we have ignored many of the implementation specific aspects of programming for AltiVec in order to stress the overall issues related to programming for vector engines. It is now time to start looking at the specifics of programming for AltiVec and OSX.

The AltiVec engine, or *Velocity Engine* in Apple terms, is a third Arithmetic Logic Unit (ALU) added to the PowerPC architecture, in addition to the traditional integer and floating-point units, that operates on 128-bit wide vectors (see Figure Figure 11-4). The vector unit (VU) itself is comprised of several different sub-units that are able to operate concurrently such as the vector permute unit (VPERM) and the vector arithmetic logical unit (VALU). We will discuss these different units when we talk about the different AltiVec operations in this chapter, as well as in the next chapter when we discuss optimization techniques. In addition to the vector unit, the AltiVec engine provides an additional set of registers called the Vector Register File (VRF). This VRF provides 32 128-bit registers for use by the Vector Unit.

Motorola has added a host of new instructions for the PowerPC architecture to support the AltiVec unit. The “AltiVec Technology Programming Environments Manual” provides a detailed description of this low-level method of programming for AltiVec (see Section Section 11.7). In order to facilitate higher-level language adoption, Motorola has also specified a C/C++ language interface for the AltiVec engine that has been spelled out in the “AltiVec Technology Programming Interface Manual.” Both of these manuals are important references that you will want to have handy. They both come as bookmarked PDF, so are easy to keep open and search while you are working on a project without taking up real-world deskspace.

While optimizing functions with hand-tuned assembly is almost considered a rite-of-passage to some, we are going to focus on the C interface. The high-level language interface creates a number of new language extensions for supporting the wider types, as well as built-in function calls to support the new AltiVec operations. As you saw in the examples we have already shown, the new altivec instructions fit in very nicely with more familiar C programming, saving you from having to intersperse your code with `asm()` calls.

Figure 11-4. AltiVec \texttt{vector



11.3.1. Using AltiVec in ProjectBuilder

Apple's developer tools have built-in support for the C AltiVec programming interface. By default, this feature is not turned on when you create a new project using ProjectBuilder}. To turn on AltiVec, under the Targets tab, choose the target, which is typically the name of your project, then, choose the Build Settings tab. Down this page is the Compiler Settings section, select the Other Compiler Flags field, and add `-faltivec`. If you are working from the command line, simply add the `-faltivec` option when you run **cc**. Figure Figure 11-5 shows where to set the `-faltivec` flag in ProjectBuilder.

> **Tip:** `illegal statement, missing ';' after 'vector'`. If you find that ProjectBuilder is giving you this error, followed by: `'vector' undeclared` at the point in your code where you are using the vector keyword then you have forgotten to turn on AltiVec in the compiler settings.

Since the AltiVec extensions to C and C++ are defined at the compiler level, once you give **cc** the right flags, you are ready to start using AltiVec. You don't need to link with any special libraries or frameworks.

11.3.2. AltiVec Types

AltiVec defines a number of new vector types in C. These ar bonafide types, and not simply typedefs or macros to structures. Table 11-2 enumerates the different possible type combinations available with AltiVec.

All AltiVec vectors are 128bits wide, which allows for a varying number of component elements in each vector depending upon their type. Figure Figure 11-6 shows how different sized vector components are organized in memory. Table Table 11-2's *Count* column indicates how many elements of each type a vector can store.

Just like normal types, you can use pointers to vectors:


```
vector float vf;
vector float *pvf = &vf;
```

Figure 11-5. Setting the `-faltivec`

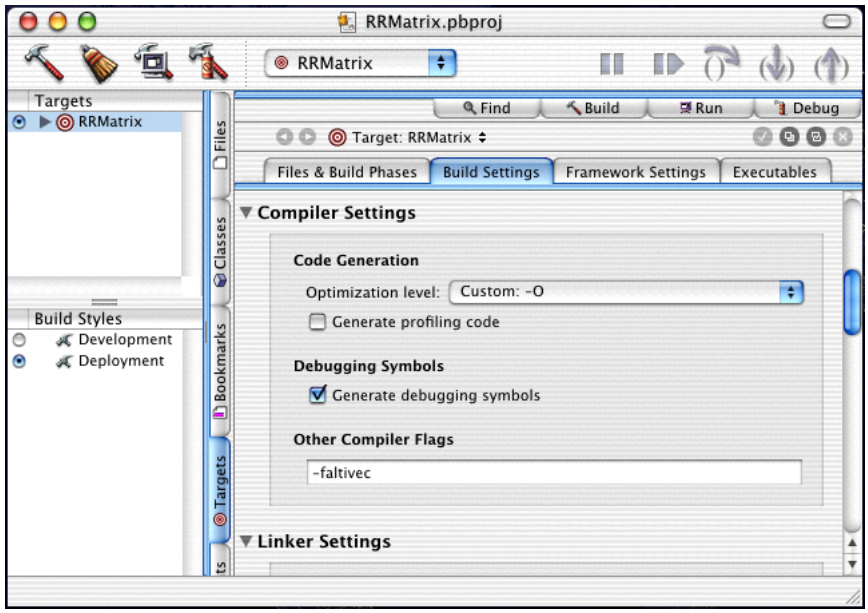


Table 11-2. C AltiVec Types

Type	Count	Range
vector unsigned char	16	0...255
vector signed char	16	-128...127
vector bool char	16	0 (false), 255 (true)
vector unsigned short	8	0...65536$
vector unsigned short int		
vector signed short	8	-32768...32767
vector signed short int		
vector bool short	8	0 (false), 65535 (true)
vector bool short int		
vector unsigned int	4	0...2^{32} - 1
vector unsigned long		
vector unsigned long int		
vector signed int	4	-2^{31}...2^{31} - 1
vector signed long		
vector signed long int		
vector bool int	4	0 (false), 2^{32} - 1 (true)
vector bool long		
vector bool long int		
vector float	4	IEEE-754 Values
vector pixels	4	1/5/5/5 pixel

Figure 11-6. AltiVec \texttt{vector

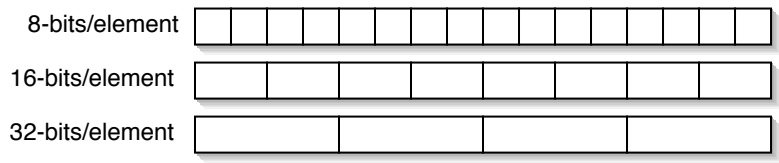


Table 11-3. AltiVec Literals

Vector Literal Format
`(vector unsigned char) (unsigned int)`
`(vector unsigned char) (unsigned int, ... , unsigned int)`
`(vector signed char) (int)`
`(vector signed char) (int, ... , int)`
`(vector unsigned short) (unsigned int)`
`(vector unsigned short) (unsigned int, ..., unsigned int)`
`(vector signed short) (int)`
`(vector signed short) (int, ... , int)`
`(vector unsigned int) (unsigned int)`
`(vector unsigned int) (unsigned int, ..., unsigned int)`
`(vector signed int) (int)`
`(vector signed int) (int, ..., int)`
`(vector float) (float)`
`(vector float) (float, ... , float)`

```
                                                  '!', ' ', ' ', ' ');
vector signed short c = (vector signed short) (-10);
vector signed short d = (vector signed short) (0, 1, 2, 3,
                                               4, 5, 6, 7);
vector float e = (vector float) (3.14159276);
vector float f = (vector float) ( 0.0, 1.0, 2.0, 3.0 );
```

Note that the vector unsigned char literal spelling out “hello, world!” does not fill up all sixteen elements of the vector, so must be padded with the remaining three elements. In this case, we just use spaces. It should also be noted at this point that vector literals are not always the fastest method of obtaining constant vectors for your operations. In the next section, we will look at built-in AltiVec operations that can fill vectors with a combination of values very quickly. In the next chapter we will discuss why you might want to use these operations instead of the simpler literals.

> **Tip:** *Error:*`too few initializers`. If you receive this error, it means that you have not provided enough values in your vector literal statement. For example, a vector unsigned char expects either 1 or 16 values. A vector float expects 1 or 4.

11.3.2.2. Casts

The AltiVec programming interface supports C-like casts between vector types. Table 11-4 shows the new cast types supported. The cast themselves do not change the bit pattern of the vector. You should be careful when casting between types of different element sizes such as char to int or float. Numbers in one vector type typically do not match one-to-one in another.

```
// this is okay - an all-zeros bit pattern is zero in IEEE-754
vector unsigned int a = (vector unsigned int) (0);
vector float b = (vector float) a;

// trouble - you are not going to end up with
// a (4.0, 4.0, 4.0, 4.0) vector here.
vector unsigned int a = (vector unsigned int) (4);
vector float b = (vector float) a;
```

If you need to make conversions between vector types, you are safer to use the explicit AltiVec conversion operations (See Section 11.3.3). You cannot cast between vector and scalar types, which should make sense since no traditional scalar type has enough storage for a 128-bit vector. You can cast a vector to a pointer to a scalar type such as:

```
vector unsigned int a = (vector unsigned int) ( 3, 2, 1, 0 );
unsigned int *pui = (unsigned int*)(&a);
```

Table 11-4. AltiVec Casts

(vector unsigned char)
(vector signed char)
(vector bool char)
(vector unsigned short)
(vector signed short)
(vector bool short)
(vector unsigned int)
(vector signed int)
(vector bool int)
(vector float)
(vector pixel)

11.3.2.3. Use Unions

You may have noticed a bit of code in the introductory examples that doesn't match up with the AltiVec types. Specifically, the optimized AltiVec dot product function was defined as:

```
vfloat altivec_dot_product(vector float *a, vector float *b)
{
    vfloat result;
    ...
    vec_ste(tmp1, 3, &(result.e[0]));
    ...
    return result;
}
```

The vfloat type used in this code is actually a `union`:

```
typedef union {
    float e[4];
    vector float v;
} vfloat;
```

By using a `union`, you can make it much easier to access the individual elements of the vector. You will find many of the examples in this chapter and the next make use of these kinds of unions, as well as many of the AltiVec tutorials and examples on the web. To be sure, your unions will be much easier to read than trying to dereference the vector directly to get at individual elements:

```
vector float a = (vector float) ( 0.0, 1.0, 2.0, 3.0 );
float second_element = *(((float*) &a)+1);
```

Here are some sample unions for different sized data types. Note that the element array `e` in each union is properly sized for the number of elements in the vector:

```
typedef union {
    unsigned char e[16];
    vector unsigned char v;
} vuchar;

typedef union {
    unsigned short int e[8];
    vector unsigned short int v;
} vushort;

typedef union {
    signed int e[4];
    vector signed int v;
} vsint;
```

The following code shows everything we have talked about up to this point: AltiVec types, using unions, casts and vector literals. It is AltiVec's version of "Hello, World!"

```
#include <stdio.h>

typedef union {
    unsigned char e[16];
    vector unsigned char v;
} vuchar;

int main (int argc, const char * argv[]) {
    int i;
    vuchar hello;
    hello.v = (vector unsigned char) ('h', 'e', 'l', 'l',
                                      'o', ',', ' ', 'w',
                                      'o', 'r', 'l', 'd',
                                      '!', ' ', ' ', ' ');

    for(i = 0; i < 16; ++i)
        printf("%c", hello.e[i]);
    printf("\n");

    return 0;
}
```

If you fire up ProjectBuilder, create a new Standard Tool project, set the compiler to use `-faltivec`, type the above code into `main.c` and compile you should see the output `hello, world!` in the Run tab.

> **Tip:** *Note:* In addition to the compiler-specific language extensions, the Motorola AltiVec documentation specifies extensions to certain library functions such as `printf` to support vector types. Instead of looping over the individual characters in the vector, we should have been able to call: `printf("\%vc", hello.v);`, using the

`\%vc` formatting string to indicate a vector of characters. The current version of Apple's libraries (OSX v10.1.3) do not support these extensions, and you will receive `warning: unknown conversion type character 'v' in format` if you try to use them.

11.3.2.4. Alignment

AltiVec expects vectors to be 16-byte aligned. Fortunately, the compiler provides this alignment automatically for any vector type, as well as any structures or unions that have vector components. Dealing with unaligned memory locations can be bothersome, but is possible. The next section details operations available for dealing with unaligned memory. In practice, you should not run into this problem as OSX's implementation of `malloc` returns 16-byte aligned memory. You *could* get yourself into trouble with something like:

```
int main (int argc, const char * argv[]) {

    int m = 5;
    int n = 5;

    float *matrix = (float*) malloc(sizeof(float)*m*n);

    // assign some values to the matrix elements

    vector float *vf1 = (vector float*)(matrix);
    vector float *vf2 = (vector float*)(matrix + n);
    vector float *vf3 = (vector float*)(matrix + n*2);
    vector float *vf4 = (vector float*)(matrix + n*3);

    *vf4 = vec_madd(*vf1, *vf2, *vf3);

    free(matrix);
    return 0;
}
```

In the above code, a matrix is allocated using `malloc`, but the matrix's dimensions are not divisible by four. This means that each row after the first will not align on the 16-byte boundary. We can then imagine coming along later wanting to send this data to the AltiVec engine for some kind of computation. Because of the misalignment, we wouldn't be able to do that by just assigning pointers as we have done in this example. Worst of all, the AltiVec engine silently accepts all of these memory locations without throwing an exception. On a load/store of a misaligned vector, is simply ignores the low-order bits, effectively pushing the pointer down to the lower aligned memory location. The AltiVec engine then silently proceeds to read and write its vectors using this incorrect pointer! Needless to say, this can lead to some very difficult to track bugs that might only show up as incorrect results.

You will be more likely to run into this issue when dealing with legacy code. Often in scientific and engineering applications we will be dealing with enormous amounts of information. Because memory bandwidth is always a limiting factor, making sure your objects are as small as possible is important--making sure rows align on 16-byte boundaries usually doesn't enter the picture. While we typically view optimization as a last step in program development, when writing new programs that you are almost sure will be using AltiVec, it is a good idea to keep these kinds of memory organizational details in mind.

Table 11-5. `vec_add` Assembly Mapping

d	a	b	maps to
	vector unsigned char	vector unsigned char	
vector unsigned char	vector unsigned char	vector bool char	`vaddubm d,a,b`
	vector bool char	vector signed char	
	vector signed char	vector signed char	
vector signed char	vector signed char	vector bool char	`vaddubm d,a,b`
	vector bool char	vector signed char	
	vector unsigned short	vector unsigned short	
vector unsigned short	vector unsigned short	vector bool short	`vadduhm d,a,b`
	vector bool short	vector signed short	
	vector signed short	vector signed short	
vector signed short	vector signed short	vector bool short	`vadduhm d,a,b`
	vector bool short	vector signed short	
	vector unsigned int	vector unsigned int	
vector unsigned int	vector unsigned int	vector bool int	`vadduwm d,a,b`
	vector bool int	vector signed int	
	vector signed int	vector signed int	
vector signed int	vector signed int	vector bool int	`vadduhm d,a,b`
	vector bool int	vector signed int	
vector float	vector float	vector float	`vaddfp d,a,b`

11.3.3. AltiVec Operations

Motorola's AltiVec C/C++ programming interface uses built-in functions to provide a more C-like interface to the underlying AltiVec operations. If you recall from the dot product example, these operations to not relate to actual function calls. Instead the assembly instructions are inserted directly at the point of use—much like using inlining to optimize away the function call and return overhead. We have already seen a couple of these functions, such as `vec_madd`, `vec_add` and `vec_sld`. Each of these functions overloads to one or more underlying assembly instructions that operate on specific vector types. This reduces the overall number of operation definitions you must remember, as well as simplifies the calling of some operations that would take multiple assembly instructions to complete. Lets look at a couple of concrete examples:

The `vec_add` operation, which takes two matched vectors of any kind except for vector pixel and adds their component elements, maps to the assembly `vaddubm`, `vadduhm`, `vadduwm`, or `vaddfp` depending upon whether the arguments passed to it are char, short, int, or float respectively. Table 11-5 shows how the arguments for `vaddubm d,a,b` are mapped to the different underlying assembly instructions. Figure 11-7 graphically depicts how the different AltiVec instructions operate on their target vectors.

With the `vec_add` operation, vector types which have the same size elements, such as vector unsigned char and vector bool char can be mixed on the input so that you can make calls such as:

Figure 11-7. AltiVec \texttt{vec_add

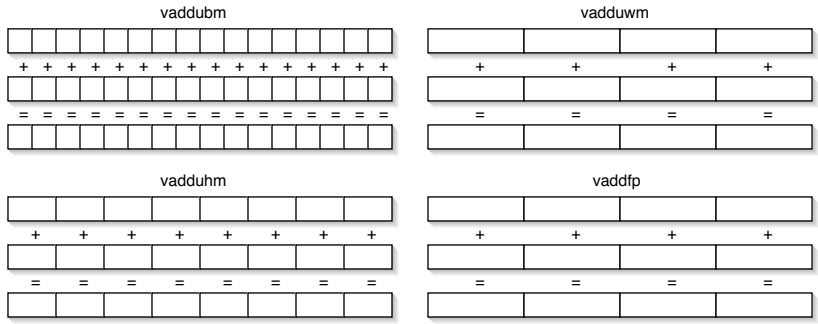


Table 11-6. `vec_abs` Assembly Mapping

d	a	maps to	
		`vspltisb z,0`	
vector signed char	vector signed char	`vsububm t,z,a`	
		`vmaxsb d,a,t`	
		`vspltisb z,0`	
vector signed short	vector signed short	`vsubuhm t,z,a`	
		`vmaxsh d,a,t`	
		`vspltisb z,0`	
vector signed int	vector signed int	`vsubuwm t,z,a`	
		`vmaxsw d,a,t`	
		`vspltisw m,-1`	
vector float	vector float	`vslw t,m,m`	
		`vandc d,a,t`	

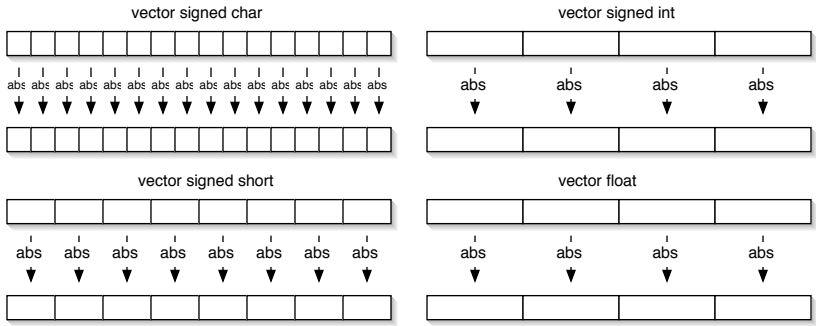
```
vector unsigned char a = (vector unsigned char) (1);
vector bool char b = (vector bool char) (0);

vector unsigned char c = vec_add(a, b);
```

The `vec_add` operation is a good example of an operation that overloads onto several different assembly instructions depending upon the argument types; however, in each case, the operation is mapped onto a single instruction. The AltiVec programming interface also provides operations that map onto multiple underlying assembly instructions. The `vec_abs` operation, which can take in any of the signed vector types (including floats), gets mapped to three assembly operations that result in the return of a vector holding the absolute values of the elements in the original vector. Given the operation `d = vec_abs(a)`, Table 11-6 shows the different assembly instructions it is mapped to. Figure 11-8 graphically shows how the operation acts on its target vectors.

Every AltiVec operation is spelled out in detail in Motorola's "AltiVec Technology Programming Interface Manual." See Section 11.7 for where you can download a copy. It provides an alphabetical list of all operations, and shows what assembly instructions they are mapped to. In the next chapter we will take a look at how this assembly information can be critical to wringing the last drop of speed out of your code.

> **Tip:** *Error:* `no instance of overloaded built-in function '[function name]' matches the parameter list`. This error can occur in several ways. First, you could provide the wrong number of arguments for a built-in vector function. Second, you could give the function a set of parameters it is unable to match with an underlying AltiVec operation. Some AltiVec operations are limited to certain vector types, and passing in a vector type it cannot handle will generate this error.

Figure 11-8. AltiVec `vec_abs` Operation



will look at the range of operators available, and show examples of some important ones in use. It would be difficult to go into much detail on every single instruction here, so this chapter will necessarily touch on a few of the more commonly used operations. Chapter 15 provides a full listing and description of all 113 operations, and is a handy Quick-Reference guide when you are looking for the right operation for a particular task.

11.3.3.1. Conversion and Creation Operations

When vector casts were introduced, it was mentioned that the cast does not change the bit pattern of the actual vector, it simply comforts the compiler into believing there is a measure of type safety. To properly convert between different vectors you should use the `vec_ctf(a,b)`, `vec_cts(a,b)` and `vec_ctu(a,b)` operations. `vec_ctf` is somewhat confusingly called “convert from fixed-point word” by the Motorola documentation, when it seems it should stand for “convert to float,” which is in fact what it does—convert a vector unsigned int or vector signed int to a vector float. `vec_cts` and `vec_ctu` provide conversions in the opposite direction for signed and unsigned integers respectively.

```
vector unsigned int a = (vector unsigned int) ( 0, 1, 2, 3 );
vector signed int b = (vector unsigned int) ( 0, -1, -2, -3 );

vector float c = vec_ctf(b, 0);
a = vec_ctu(c, 0);
b = vec_cts(c, 0);
```

The first argument for the conversion operations is the vector to be converted. The second argument determines the conversion’s multiplier or divisor. The `vec_ctf` operation performs the integer to float conversion, and then divides the elements of the vectors by 2^{b} before returning the vector. The `vec_ctu` and `vec_cts` operations multiply their converted vector by 2^b before returning it. In the case of the examples above, a zero (2^0) gives us a divisor (or multiplier) of one, which is the straightforward conversion we are looking for.

All three of these conversion operations work on vectors holding 4 32-bit elements, and not with the smaller 8 or 16-bit element vectors. To move between vectors of varying element size you need to use the pack and unpack operations: `vec_pack(a,b)`, `vec_packs(a,b)`, `vec_packsu(a,b)`, `vec_unpackh(a)`, and `vec_unpackl(a)`. The pack commands take two vectors and pack the wider values into a single vector. For example, two vector unsigned int vectors each holding four 32-bit elements would be packed into a single vector unsigned short holding eight 16-bit elements. The operations shrink each element of the two argument vectors by either truncating (See

Figure 11-9. AltiVec `vec_pack` Operation

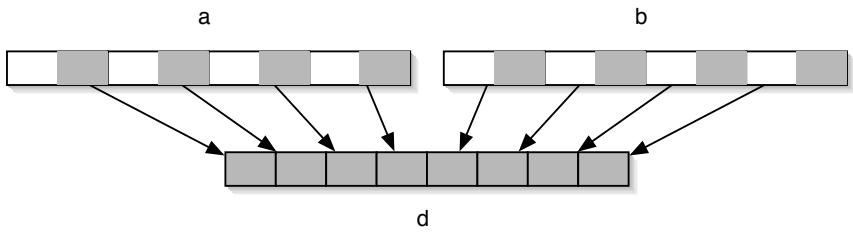


Figure 11-10. AltiVec `vec_splat` (\textsf{a

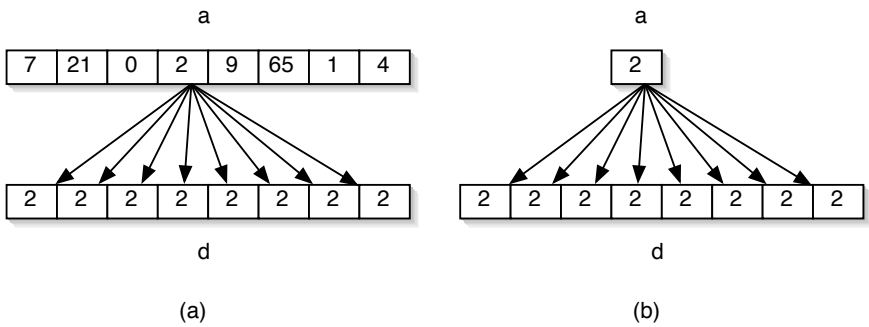


Figure 11-9), or using a signed or unsigned saturated value. The `vec_packpx(a,b)` operation is used for packing vectors of pixel type.

AltiVec provides a number of splat operations to quickly fill a vector's elements with a value. `vec_splat(a,b)` fills the return vector's elements with the value of the element at the index given by `b` in the argument vector `a`. The following code fills vector `d` with all 2's (See Figure 11-10 (a)):

```
vector unsigned short a = (vector unsigned short) ( 7, 21, 0, 2,
                                                    9, 65, 1, 4 );
vector unsigned short d = vec_splat(a,3);
```

The other six splat operations provide signed or unsigned vectors of either 8, 16 or 32-bit sized elements: `vec_splat_s8(a)`, `vec_splat_s16(a)`, `vec_splat_s32(a)`, `vec_splat_u8(a)`, `vec_splat_u16(a)`, and `vec_splat_u32(a)`. Each of these operations takes a single integer that is used to fill in the result vector. Figure 11-10 (b) shows the `vec_splat_u16` in action.

The splat operations turn out to be more than merely convenience methods. It turns out that, if possible, you will almost always want to use the splat operations to fill your vectors instead of using vector literals. The reason for this is the fact that constants are stored in memory, and loaded into the vector register file as needed. If you are unlucky enough, the constant will not be in cache, and your code will stall while waiting for it to load. By programmatically filling your vectors, you do not risk a cache miss and save precious cache real-estate to boot.

```
vector float a = (vector float) ( 10.0, 10.0, 10.0, 10.0 );
vector float b = vec_ctf( vec_splat_u32( 10 ), 0 );
```

Both of these assignments will give you a vector filled with 10.0, but the second is much faster. You will see examples of this usage in the code examples throughout this chapter and the next.

The following inline function is used to provide a vector float full of negative zeros. A vector of -0.0 is often preferred in calculations over a vector full of positive zeros, which could more easily be created with the simple cast `a = (vector float) vec_splat_u32(0);` since all-zeros is the same binary representation in an integer vector as it is in a float vector.

```
inline vector float neg_zero(void)
{
    vector unsigned int unz = vec_splat_u32(-1);
    return (vector float) vec_sl(unz, unz);
}
```

11.3.3.2. Mathematical Operations

AltiVec provides a number of mathematical operations that support floating-point, integer and mixed-type vector arguments. Many of the operations, like `vec_madd` perform multiple operations in one call.

AltiVec supplies `vec_add(a,b)` and `vec_sub(a,b)` operations that support basic addition and subtraction for float and integer vectors. There is also a `vec_abs(a)` operation that supports both signed integer and floating-point vectors.

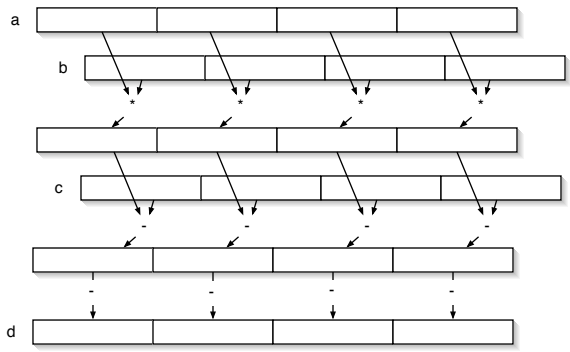
> **Tip:** *Note:* For the sake of brevity, when we refer to integer vectors in reference to argument types for AltiVec operations, we are generally speaking about any of the vector types excluding vector float and vector pixel. vector pixel is generally not referenced at all in these discussions, and will be mentioned expressly when needed. We may also refer to either unsigned or signed vectors, which refers to any char, short or int vector of appropriate signability. Most operations that support integers are overloaded to support the different sized integer elements.

Taking the prize as the most re-used operation is the multiply. AltiVec provides nine different multiplication operations, and ironically enough, none of them is just a plain multiplication of two vectors. The basic fused multiply-add `vec_madd` we have already seen in action. Of the additional multiplication operators, only `vec_nmsub(a,b,c)` supports floating point values. We will come back to the other multiplication operations later when we talk about integer-only operations. Figure 11-11 shows `vec_nmsub` in action.

Like `vec_nmsub`, a number of the mathematical operations provided by AltiVec only apply to vectors of floating-point values. The `vec_ceil(a)`, `vec_floor(a)`, `vec_trunc(a)` and `vec_round(a)` unary operations carry out the same operations as their scalar counterparts by either rounding up, rounding down, truncating or rounding to the nearest floating-point integer every element of `vector float a`.

There is no direct support for division in the AltiVec programming interface as there is with scalar floating-point units. Instead, to compute a_i/b_i for every element in `a` and `b`, we must first compute $1/b_i$ and then compute

Figure 11-11. AltiVec `vec_nmsub` Operation



$a_i\frac{1}{b_i}$. The first step uses the reciprocal estimate operation `vec_re(a)`. To complicate matters, the `vec_re` operation only returns a half-precision estimate of the reciprocal. We must then perform a single *Newton-Raphson* refinement step ($u_{i+1} = u_i(2-vu_i)$) in order to get a full single-precision reciprocal value. The final step is to use the `vec_madd` operation to complete the division by multiplying the numerator `a` by the reciprocal:

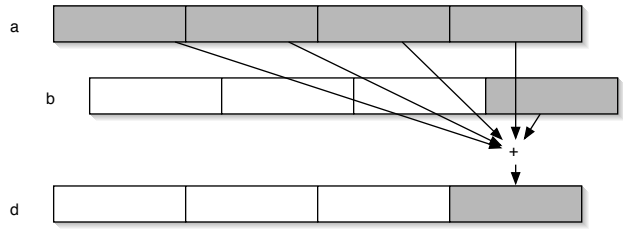
```
inline vector float vec_div(vector float a, vector float b)
{
    vector float vone = vec_ctf( vec_splat_u32(1), 0) );
    vector float re = vec_re(b);
    re = vec_madd( re,
                   vec_nmsub (re, b, vone,
                   re);
    return vec_madd(a, re, (vector float) vec_splat_u32(0));
}
```

Note the use of the `vec_nmsub` operation presented earlier.

Obtaining the square roots of vector float elements is very similar to the method of acquiring a full-precision division. First, we must use the `vec_rsqrte(a)` operation to obtain an estimate of the reciprocal of the square root ($1/a_i^{1/2}$), then using a similar *Newton-Raphson* step ($u_{i+1} = \frac{1}{2}u_i(3-vu_i^2)$), we can obtain the full single-precision reciprocal square root. Finally, we obtain the square root by multiplying our original value by the reciprocal square root ($\sqrt{a_i} = a_i\frac{1}{a_i^{1/2}}$).

```
inline vector float vec_sqrt(vector float a)
{
    vector float vone_half = vec_ctf( vec_splat_u32(1), 1 );
    vector float vone = vec_ctf( vec_splat_u32(1), 0 );
    vector float rsqrte = vec_rsqrte(a);
    vector float re = vec_madd(rsqrte, rsqrte, neg_zero() );
    vector float half_rsqrte = vec_madd(rsqrte,
                                    vone_half,
                                    neg_zero() );
    vector float tmp = vec_nmsub( a, re, vone );
```

Figure 11-12. AltiVec `vec_sums` Operation




```
    rsqrte = vec_madd( tmp, hrsqrte, rsqrte );
    return vec_madd( a, rsqrte, neg_zero() );
}
```

While the division and square root operations are obviously costly, there are two things to keep in mind: First, In almost all FPU implementations division is more costly than any other operation, even on the PowerPC's FPU, where almost all floating-point calculations incur the same overhead. The AltiVec ALU is no different. However, most algorithms these days take this performance hit into account in their design already and avoid any gratuitous use of these operations. Second, we are working on vector floats, so each function call is simultaneously calculating the value for four floats. A powerful example of the performance benefits of AltiVec's computation model.

In addition to the reciprocal and reciprocal square root estimators, AltiVec also provides estimators for 2^v and $log_2 v$: `vec_expte(a)` and `vec_loge(a)` respectively.

Many of AltiVec's mathematical operations support integer-only vectors. `vec_abss(a)`, `vec_adds(a,b)` and `vec_subs(a,b)` are the saturated versions of the more general absolute, add and subtract operations `vec_abs`, `vec_add` and `vec_sub`. `vec_addc(a,b)` and `vec_subc(a,b)` instead of returning a result vector `d` filled with the addition or subtraction result, return vectors indicating whether there was a carry resulting from the addition or subtraction of each element. `vec_avg(a,b)` returns the average of the integer values in the two argument vectors `a` and `b`.

The `vec_sums(a,b)`, `vec_sum4s(a,b)` and `vec_sum2s(a,b)` operations are some of the few *intra-vector* operations supported by AltiVec within the family of mathematical operations. Opposed to the traditional `vec_add*` operations, the `vec_sum*` operations sum across the vector elements (intra-vector), instead of between elements of different vectors (inter-vector). In the case of `vec_sums`, the result is a vector with its last element the sum of $a_0+a_1+a_2+a_3+b_3$. Figure 11-12 shows the `vec_sums` operation in action.

The `vec_sum4s(a,b)` and `vec_sum2s(a,b)` operations are shown in Figure 11-13 (a) and (b).

The summation functions have `vec_msum(a,b,c)` and `vec_msums(a,b,c)` versions that multiply the `a` and `b` vectors before performing a `vec_sum4s`-like operation with `c`. (See Figure 11-14).

There is a saturated version of `vec_madd` for integer vectors `vec_madds`, as well as a number of additional integer multiplication operations: `vec_mladd`, multiply low and add unsigned half word; `vec_mradds`, multiply round and add saturated; `vec_mule` multiply even; and, `vec_mulo`, multiply odd. See Chapter 15 for more detailed definitions of these, and all operations we have mentioned here.

Figure 11-13. AltiVec `vec_sum4s` (a) and `vec_sum2s` (b) Operations

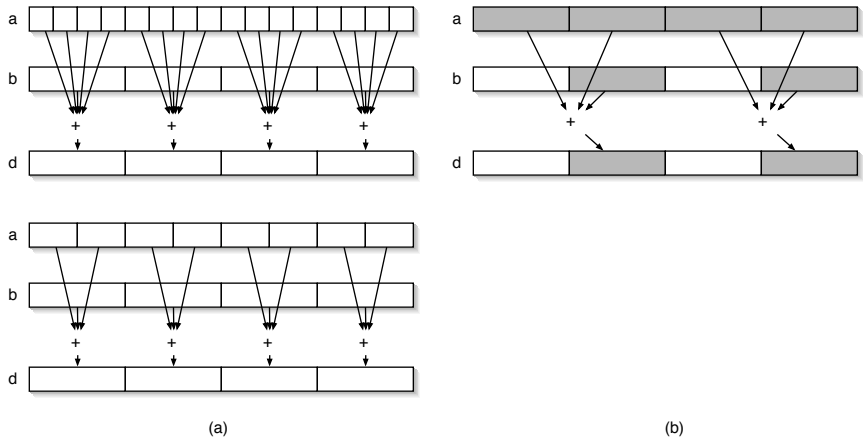


Figure 11-14. AltiVec `vec_msum` Operation

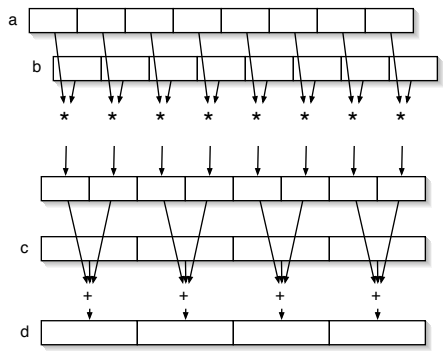
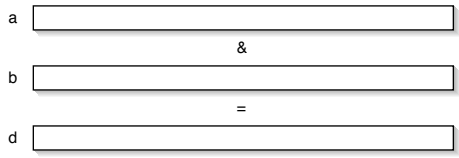


Figure 11-15. AltiVec vec_and Operation



11.3.3.3. Logical, Comparator and Predicate Operations

AltiVec provides the following *logical* operations:

`vec_and(a,b)`	AND
`vec_andc(a,b)`	AND with complement
`vec_nor(a,b)`	NOR
`vec_or(a,b)`	OR
`vec_xor(a,b)`	XOR

Each of these operations performs a bit-wise AND, NOR, XOR, etc. on the two vectors `a` and `b`. These operations work on all vector types, as they do not care about vector's element structure--that is whether the vector's elements are laid out as 8, 16 or 32-bit quantities. See Figure 11-15.

The *comparison* operations look at each element in the argument vectors, and make the relevant comparison, returning either *true* or *false* for each element in the result vector. The size of the result vector vector bool char, vector bool short or vector bool int depends on the size of the argument vectors. For example, if the input vectors `a` and `b` are vector float, vector unsigned int or vector signed int, the comparison operators will return a vector bool int. Likewise for the 8 and 16-bit element vectors.

```
<function>vec_cmpeq(a,b)</function> & Equal \\ \hline
<function>vec_cmpge(a,b)</function> & Greater Than or Equal \\ \hline
<function>vec_cmpgt(a,b)</function> & Greater Than \\ \hline
<function>vec_cmple(a,b)</function> & Less Than or Equal \\ \hline
<function>vec_cmplt(a,b)</function> & Less Than \\ \hline
\end{tabular}
\end{center}
```

There is one comparison operation that does not return a bool vector, and that is `vec_cmpb(a,b)`: compare bounds. This operations returns a vector signed int that indicates where a_i falls in relation to the range $(-b_i, b_i)$. The following code shows how to use the results to determine the bound relationship:

```
typedef union {
    signed int e[4];
    vector signed int v;
```

Table 11-7. AltiVec Predicate Operations


```
} vsint;

void bounds_print(vsint a) {
    int i;
    for(i = 0; i < 4; ++i) {
        if(!a.e[i])
            printf("a[%d] in bounds:\n", i);
        else if(a.e[i] < 0)
            printf("a[%d] above bounds:\n", i);
        else if(a.e[i] > 0)
            printf("a[%d] below bounds:\n", i);
    }
}

int main (int argc, const char * argv[]) {
    vsint result;
    vector float a = (vector float) (0.5, 5.0, -0.5, -4.0);
    vector float b = (vector float) (1.0, 1.0, 1.0, 1.0);
    result.v = vec_cmpb(a, b);

    bounds_print(result);
    return 0;
}
```

Note that if a_i is above bounds, `result.e[i]` holds a very large negative number, and if a_i is below bounds, it holds a very large positive number. The operation works by setting the two high-order bits in each element to indicate whether a_i is out-of-bounds in either direction.

Two additional comparisons are `vec_max(a,b)` and `vec_min(a,b)`. Each of these operations returns a vector holding either the maximum or minimum element from each pair of corresponding vector elements. In the following code, `d = (3.14, 0.42, -0.5, 1.0);`

```
vector float a = (vector float) (3.14, 0.23, -0.5, -4.0);
vector float b = (vector float) (1.0, 0.42, -1.0, 1.0);
vector float d = vec_max(a,b);
```

The AltiVec Programming interface includes a number of *predicate* operations that return a single int indicating whether *all* elements in the vectors meet a certain criteria, or whether *any* individual element or element pair meets the criteria. The criteria itself is dependent upon the operation. Table 11-7 lists the different predicate operations. Note that most operations take two argument vectors, and that the predicate criteria is based in an inter-element comparison of the two vectors' elements. The `vec_*_nan` and `vec_*_numeric` predicates only take a single vector.

```
<function>vec_all_eq(a,b)</function> & all elements equal \\ \hline
<function>vec_all_ge(a,b)</function> & all elements greater than or equal \\ \hline
```

```
<function>vec_all_gt(a,b)</function> & all elements greater than \\ \hline
<function>vec_all_in(a,b)</function> & all elements in bounds \\ \hline
<function>vec_all_le(a,b)</function> & all elements less than or equal \\ \hline
<function>vec_all_lt(a,b)</function> & all elements less than \\ \hline
<function>vec_all_nan(a)</function> & all elements not a number \\ \hline
<function>vec_all_ne(a,b)</function> & all elements not equal \\ \hline
<function>vec_all_nge(a,b)</function> & all elements not greater than or equal \\ \hline
<function>vec_all_ngt(a,b)</function> & all elements not greater than \\ \hline
<function>vec_all_nle(a,b)</function> & all elements not less than or equal \\ \hline
<function>vec_all_nlt(a,b)</function> & all elements not less than \\ \hline
<function>vec_all_numeric(a)</function> & all elements numeric \\ \hline
<function>vec_any_eq(a,b)</function> & any element equal \\ \hline
<function>vec_any_ge(a,b)</function> & any element greater than or equal \\ \hline
<function>vec_any_gt(a,b)</function> & any element greater than \\ \hline
<function>vec_any_le(a,b)</function> & any element less than or equal \\ \hline
<function>vec_any_lt(a,b)</function> & any element less than \\ \hline
<function>vec_any_nan(a)</function> & any element not a number \\ \hline
<function>vec_any_ne(a,b)</function> & any element not equal \\ \hline
<function>vec_any_nge(a,b)</function> & any element not greater than or equal \\ \hline
<function>vec_any_ngt(a,b)</function> & any element not greater than \\ \hline
<function>vec_any_nle(a,b)</function> & any element not less than or equal \\ \hline
<function>vec_any_nlt(a,b)</function> & any element not less than \\ \hline
<function>vec_any_numeric(a)</function> & any element numeric \\ \hline
<function>vec_any_out(a,b)</function> & any element out of bounds \\ \hline
```

The following code shows using a `vec_splat_s32` and `vec_any_ge` operation to quickly detect whether any element of two vectors is greater than 10. In the next chapter we will see an example of this kind of predicate in action when we use it to detect algorithms that are getting close to the limits of our single-precision floating point accuracy.

```
void print_status(int s)
{
    if(s)
        printf("Warning!\n");
    else
        printf("all systems nominal\n");
}

int main (int argc, const char * argv[]) {

    vector signed int a = (vector signed int) (5, 2, -3, -4);
    vector signed int b = (vector signed int) (8, 12, 3, 2);

    print_status(vec_any_ge(a, vec_splat_s32(10)));
    print_status(vec_any_ge(b, vec_splat_s32(10)));

    return 0;
}
```

Figure 11-16. AltiVec `vec_perm` Operation

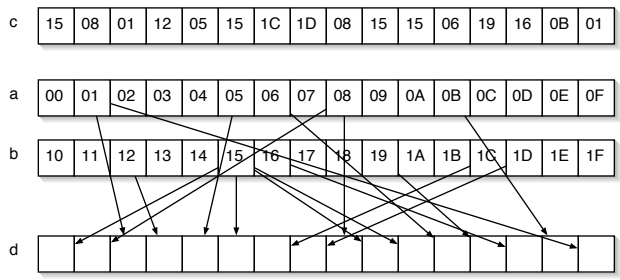
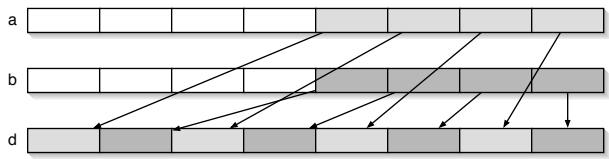


Figure 11-17. AltiVec `vec_mergel` Operation



11.3.3.4. Permutation Operations

The AltiVec permutation operations allow us to move vector elements around within and between vectors in a very efficient manner. Because there is a separate permute unit (VPU) that operates in parallel with the vector ALU, permutations operations can be carried out as the same time we are performing calculations on another set of vectors.

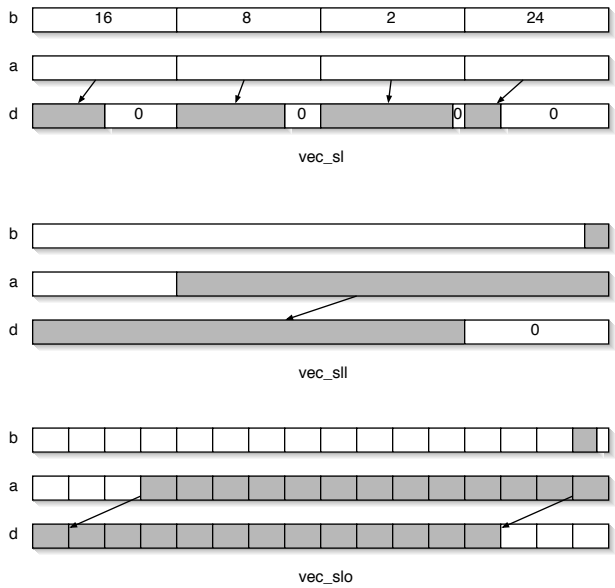
The basic permutation operation is `vec_perm(a,b,c)`. This operation uses the values in the `vector unsigned char c` to arbitrarily move the elements of the two vectors `a` and `b` into the return vector. The permute operation assigns an index to each element running in order from (`00--0F`) for vector `a` to (`10--1F`) for vector `b`. Figure 11-16 depicts how the permute operation uses these values in the `c` vector to pick which elements to place into the return vector `d`.

The merge operations `vec_mergeh(a,b)` and `vec_mergel(a,b)` take two vectors and merge either the lower or upper half of each vector into the result vector by interleaving elements from each vector. Figure 11-17 shows the `vec_mergel` operation in action for two vector unsigned ints.

It turns out the merge operations work very well at transposing a matrix. If we have a 4×4 matrix stored in four vector floats, we can use the following function to transpose the vector in place:

```
inline void vec_transpose(vector float *a0, vector float *a1,
                      vector float *a2, vector float *a3)
{
    vector float t0, t1, t2, t3;
    t0 = vec_mergeh(*a0,*a2);
    t1 = vec_mergeh(*a1,*a3);
    t2 = vec_mergel(*a0,*a2);
    t3 = vec_mergel(*a1,*a3);
    *a0 = vec_mergeh(t0,t1);
    *a1 = vec_mergel(t0,t1);
```

Figure 11-18. AltiVec Shift Operations




```
    *a2 = vec_mergeh(t2,t3);
    *a3 = vec_mergel(t2,t3);
}
```

We will use this routine later when we are performing matrix-matrix multiplies to transform our data into the proper format for the AltiVec engine.

The `vec_sel(a,b,c)` selection operation works on vectors at the bit level, so does not concern itself with element size or vector type. The `vec_sel` operation's `c` vector is used to determine which vector `a` or `b` each bit in the result vector will come from. For every bit `d[i]` in the result vector `d`:

```
d[i] = (c[i] == 0 ? a[i] : b[i]);
```

The various shift operations available offer a range of whole-vector, intra-vector and inter-vector versions. The `vec_sl(a,b)` (shift left) and `vec_sr(a,b)` (shift right) operations shift each element of `a` left or right by the number of bits specified in vector `b`, zero-filling behind. Note that each vector element can be shifted by a different amount in these routines. `vec_sll(a,b)` shift left long and `vec_srl(a,b)` (shift right long) shift the entire vector left or right by the amount specified in the last three bits of vector `b`'s last element, zero-filling behind. `vec_slo(a,b)` (shift left octet) and `vec_sro(a,b)` (shift right octet) shift vector `a` in byte-sized chunks, specified by bits 1:4 in vector `b`'s last element. `vec_sra(a,b)` (shift right algebraic) operates like `vec_sr`, except instead of zero-filling, fills with the sign bit of each element. Figure 11-18 shows examples of these shift operations.

Finally, there is `vec_sld(a,b,c)` (shift left double), which we have seen before in our dot product implementation. This operation shifts vector `a` left by `c` bytes, filling in with the same number of bytes from the left side of vector `b`.

Figure 11-19. AltiVec `vec_sld` Operation

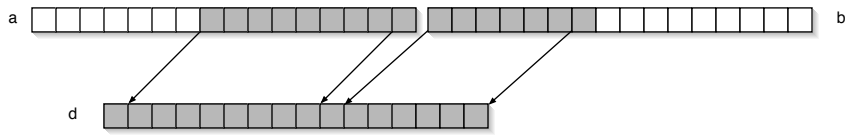


Figure 11-20. `vec_sld` Used to Sum Across a Vector

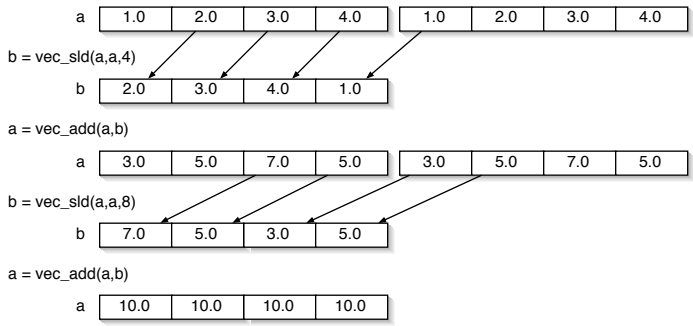


Figure 11-19 shows the `vec_sld` operation in action. Figure 11-20 shows how the shift in combination with the `vec_add` instruction was used to sum across a vector of floats.

The rotate operation `vec_rl(a,b)` rotates the each element of `a` left by the number of bits specified in `b`. Figure Figure 11-21 shows this rotation for a `vector unsigned short a`.

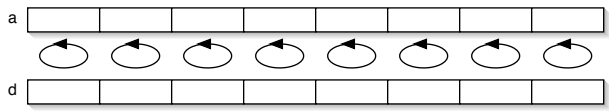
11.3.3.5. Memory Operations

The AltiVec programming environment includes a number of memory operations that aid in loading aligned and unaligned vectors from memory, loading and storing individual vector elements, and optimizing memory access for vector data. Memory operations are very important to the design of optimized altivec routines. The biggest bottleneck for the AltiVec engine is data, which in turn is limited by memory bandwidth. For now, we will save the advanced optimization techniques for the next chapter, and focus on the memory operations that support more mundane AltiVec algorithm issues.

Of the `vec_lde(a,b,c)` and `vec_ste(a,b,c)` operations, we have already seen `vec_ste` in action in our dot product example:

```
float altivec_dot_product(vector float a, vector float b)
{
    vector float zero = (vector float) vec_splat_s8(0);
    vector float tmp;
    float c;
    int i;

    tmp = vec_madd(a, b, zero);
```

Figure 11-21. AltiVec `vec_rl` Operation




```
    tmp = vec_add( tmp, vec_sld( tmp, tmp, 4));
    tmp = vec_add( tmp, vec_sld( tmp, tmp, 8));

    vec_ste(tmp, 0, &c);
    return c;
}
```

The next-to-last line in this function stored the element at index 0 of the `tmp` vector register to the scalar variable `c`. The `vec_lde` operation works in the same way, but instead of moving an element from a vector register to a scalar memory location, it moves a scalar value pointed to by `c` into an element location `b` in a vector register `a`.

To load entire vectors, you can use the `vec_ld(a,b)` and `vec_st(a,b,c)` operations. for `vec_ld`, `b` is a pointer to a memory location, and `a` is an index that can be used for stepping through memory. For `vec_st`, `a` is the vector to be stored, `b` an index, and `c` the memory location where `a` is to be stored. Both of these operations assume that the pointer to memory plus index is 16-byte aligned. The AltiVec engine will silently truncate the address to the lower 16-byte aligned address.

If you *must* load or store misaligned vectors you will need to use the `vec_lvsl(a,b)` and `vec_lvsr(a,b)` these two functions return vector unsigned chars that make perfect permutation vectors for loading unaligned data. The following function takes a pointer to a vector at an arbitrary memory location, and returns the proper vector.

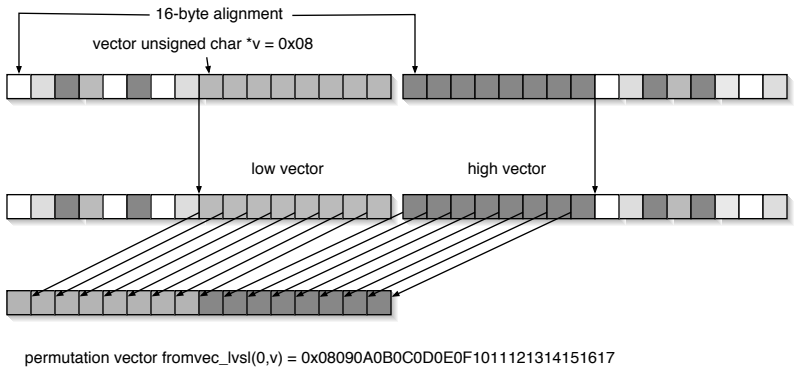
```
vector unsigned char vec_ldua( vector unsigned char *v )
{
    vector unsigned char perm = vec_lvsl( 0, (int*) v);
    vector unsigned char low = vec_ld( 0, v );
    vector unsigned char high = vec_ld( 16, v);
    return vec_perm( low, high, perm );
}
```

Since this function loads and permutes the vectors on a byte-by-byte basis, you can cast the argument and return to any vector type you wish. Figure 11-22 shows how this function would work with an vector with a memory location that ended in 0x08.

Storing a vector back to an unaligned memory location is a little more difficult:

```
void vec_stua( vector unsigned char a, vector unsigned char *v )
{
    // load the aligned vectors around our unaligned vector
    vector unsigned char low = vec_ld( 0, v );
    vector unsigned char high = vec_ld( 16, v);
```

Figure 11-22. Loading an Unaligned Vector




```
    // create a mask that will allow us to select which areas
    // of the low and high vectors we want to write our vector
    vector unsigned char perm = vec_lvsr( 0, (int*) v);
    vector unsigned char vcones = vec_splat_u8( -1 );
    vector unsigned char vczero = vec_splat_u8( 0 );
    vector unsigned char mask = vec_perm( vczero, vcones, perm );

    // rotate the source data vector
    a = vec_perm( a, a, perm);

    // create the low and high aligned vectors
    // by using the mask to determine
    // which part of the new vector is from the
    // memory location, and which part is from
    // the source vector
    low = vec_sel( a, low, mask );
    high = vec_sel( high, a, mask );

    // store the vectors back into their aligned positions
    vec_st( low, 0, v );
    vec_st( high, 16, v );
}
```

Storing a vector to an unaligned location is difficult because we risk overwriting the areas around our vector's memory location. As in Figure 11-22, our vector stradles a 16-byte boundary. In order to preserve any data that might be in the locations around our vector, we must first load the aligned areas from memory, create new vectors that are a combination of the memory locations and our vector source, and then store these vectors back to the aligned locations. Figure 11-23 shows this process.

In either case—loading or storing unaligned vectors--you can see why you are urged to avoid it. The drum was pounded loud and often in the last chapter to avoid optimization until you know where your code is slow. Because memory organization can be very deeply rooted in programs, even if you are not *sure* you are going to use AltiVec, this is one area where it is best to do a little pre-planning. Try to make sure your data is 16-byte aligned. When it

Figure 11-23. Storing an Unaligned Vector

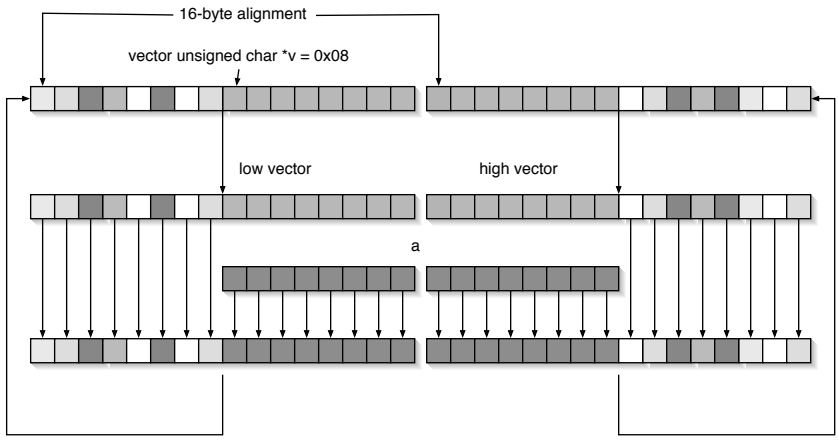


Table 11-8. AltiVec Data Stream Operations

turns out that the 80-20 rule strikes a procedure that *is* vectorizable, you won't have to completely rewrite large areas of your code to account for memory alignment.

The memory operations shown in Table 11-8 are used to optimize memory access by AltiVec code. In brief, they allow you to manage the streaming of data from memory so that it will be ready when you code calls for it, and can provide a serious speed boost when working with large amounts of data.

```
[ht]
\begin{center}
\begin{tabular}{|l|l|}
\hline
<function>vec_dss</function> & data stream stop \\ \hline
<function>vec_dssall</function> & data stream stop all \\ \hline
<function>vec_dst</function> & data stream touch \\ \hline
<function>vec_dstst</function> & data stream touch for store \\ \hline
<function>vec_dststt</function> & data stream touch for store transient \\ \hline
<function>vec_dstt</function> & data stream touch transient \\ \hline
\end{tabular}
\end{center}
\caption{}
```

The last four memory operations are `vec_ldl` and `vec_stl`, which are cache conscious load and store operations, and `vec_mfvscr` and `vec_mtvscr`, which allow you to get at the vector unit's status control register. In the next chapter we will see why you might need to use these four operations.

And with that, you have seen all one hundred and thirteen operations provided by the AltiVec programming interface. To be sure, we have left quite a lot on the table. For a breakdown and description of every operation you can use the quick reference guide in Chapter 15. For the most detailed information, I urge you to have the Motorola documentation mentioned in Section 11.7 at hand. Hopefully by now you have a good idea of what tools are in the garage. In the next section we will take a look at putting some of them to good use.

11.4. Vectorization in Action

Now that we have had a chance to look at all of AltiVec's features, let's try putting it to good use in our ever handy

```
@interface RRAVMatrix : RRMatrix {
}

@end
```

In this case, our goal is not to add any functionality, but to reimplement some of it, so our header file is pretty sparse.

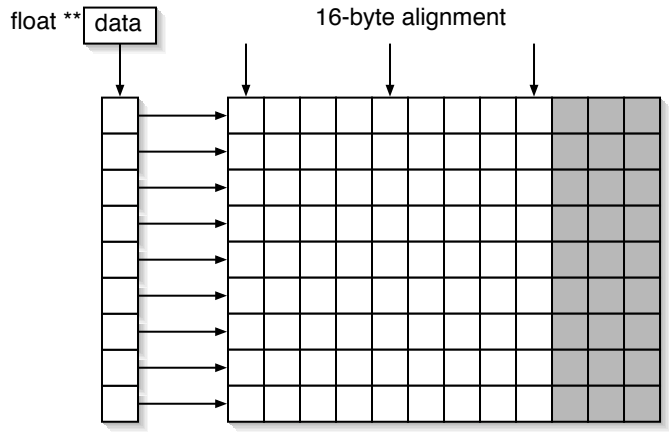
For our implementation we are going to be overriding two of the methods of our `RRMatrix` class: `initWithRows:columns:` and `multiply:in:`. For our initialization method, we are going to take the time to make sure that our rows are aligned in memory:

```
- (id)initWithRows:(unsigned int)rowCount columns:(unsigned int)columnCount
{
    int i;
    float *array;

    // get the number of whole vectors that
    // will hold one row of this matrix
    int columnVectorCount = (columnCount + 3)/4;

    if(self = [super init]) {
        m = rowCount;
        n = columnCount;

        // allocate the data for this matrix to fit whole vectors
        array = malloc(sizeof(vector float)*rowCount*columnVectorCount);
        data = (float**)malloc(sizeof(float*)*m);

        // point each row pointer in data to the
        // aligned location
        for(i = 0; i < m; i++)
            data[i] = &(array[i*columnVectorCount*4]);
    }

    return self;
}
```

The code in this routine makes sure that we are allocating our rows in whole-vector multiples, that way, even if the matrix size is not a multiple of four, each row will fall on a 16-byte boundary. Figure 11-24 shows how the `float **data` is allocated for a 9×9 matrix.

This figure clearly shows how this approach to alignment can be wasteful. As we saw with the routines used to load and store unaligned vectors though, working with unaligned data can really slow us down. It is a tough tradeoff. For now, we are going to focus on the simplest solution, and make sure our data is aligned from the outset. In the next chapter we will see why this may not in fact be the fastest solution, and what we might do to further optimize our implementation.

Figure 11-24. \textsf{RRMatrix



Now that we know our matrix rows will always begin on a 16-byte boundary, lets see what our AltiVec optimized matrix-matrix multiply method `multiply:in:` looks like:

```
- (void)multiply:(RRMatrix*)rhs in:(RRMatrix*)place
{

    // keeping with standard terminology,
    // the multiplication is C = A*B, with
    // A an m x n matrix, B an n x p matrix and C an m x p matrix.
    int i, j, k;
    int p = [rhs columns];
    float **C = [place data];
    float **A = data;
    float **B = [rhs data];

    // hold 4x4 blocks of A, B, C
    vector float ar0, ar1, ar2, ar3;
    vector float bc0, bc1, bc2, bc3;
    vector float cr0, cr1, cr2, cr3;

    // these vectors will hold our temporary
    // values that we accumulate while
    // moving through the matrix
    vector float t00, t01, t02, t03;
    vector float t10, t11, t12, t13;
    vector float t20, t21, t22, t23;
    vector float t30, t31, t32, t33;

    // loop over the columns of B - 4 columns at a time
    for (j=0; j < p; j+=4) {

        // loop over the rows of A - 4 rows at a time
        for (i=0; i < m; i+=4)  {
```

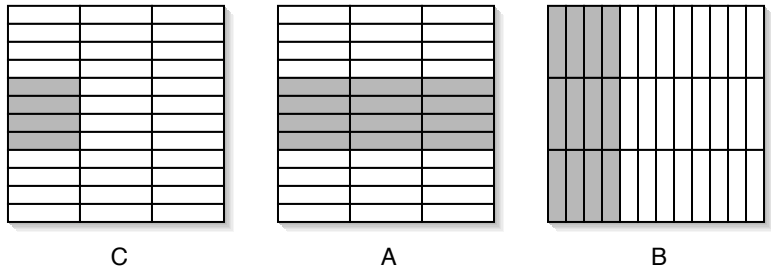
```
// zero out our temporary vectors
t00 = t01 = t02 = t03 =
    t10 = t11 = t12 = t13 =
    t20 = t21 = t22 = t23 =
    t30 = t31 = t32 = t33 = vfneg_zero();

// loop over columns of A - again, 4 rows at a time
for (k=0; k < n; k+=4) {

    // load four rows of A at a time
    // this gives us a 4x4 block of A
    ar0 = vec_ld(0, &A[i][k]);
    ar1 = vec_ld(0, &A[i+1][k]);
    ar2 = vec_ld(0, &A[i+2][k]);
    ar3 = vec_ld(0, &A[i+3][k]);

    // load a 4x4 block like we did for A
    // notice that as we move across the
    // columns of A, we are moving down
    // the rows of B
    bc0 = vec_ld(0, &B[k][j]);
    bc1 = vec_ld(0, &B[k+1][j]);
    bc2 = vec_ld(0, &B[k+1][j]);
    bc3 = vec_ld(0, &B[k+1][j]);

    // the matrix-matrix multiply is actually
    // a series of dot-products on
    // matrix A's rows, and B's columns
    // transpose our block of B so that our
    // vectors are actually the columns of B
    vec_transpose(&bc0, &bc1, &bc2, &bc3);

    // perform a mmmul of the two 4X4
    // submatrices we now have
    t00 = vec_madd(ar0, bc0, t00);
    t01 = vec_madd(ar0, bc1, t01);
    t02 = vec_madd(ar0, bc2, t02);
    t03 = vec_madd(ar0, bc3, t03);

    t10 = vec_madd(ar1, bc0, t10);
    t11 = vec_madd(ar1, bc1, t11);
    t12 = vec_madd(ar1, bc2, t12);
    t13 = vec_madd(ar1, bc3, t13);

    t20 = vec_madd(ar2, bc0, t20);
    t21 = vec_madd(ar2, bc1, t21);
    t22 = vec_madd(ar2, bc2, t22);
    t23 = vec_madd(ar2, bc3, t23);

    t30 = vec_madd(ar3, bc0, t30);
    t31 = vec_madd(ar3, bc1, t31);
    t32 = vec_madd(ar3, bc2, t32);
    t33 = vec_madd(ar3, bc3, t33);
```

```
            }

            // here we sum across the products
            // we have accumulated to produce
            // the final values in the 4x4 block
            // of C we are currently computing
            t00 = vec_add(t00, t01);
            t02 = vec_add(t02, t03);

            t10 = vec_add(t10, t11);
            t12 = vec_add(t12, t13);

            t20 = vec_add(t20, t21);
            t22 = vec_add(t22, t23);

            t30 = vec_add(t30, t31);
            t32 = vec_add(t32, t33);

            cr0 = vec_add(t00, t02);
            cr1 = vec_add(t10, t12);
            cr2 = vec_add(t20, t22);
            cr3 = vec_add(t30, t32);

            // store this block back into C
            vec_st(cr0, 0, &(C[i][j]));
            vec_st(cr1, 0, &(C[i+1][j]));
            vec_st(cr2, 0, &(C[i+2][j]));
            vec_st(cr3, 0, &(C[i+3][j]));

        }
    }
}
```

In essence, this optimized matrix-matrix multiply breaks down our matrix into 4x4 blocks, made up of four vector floats, and computes the solution for each block in turn. Figure 11-25 depicts graphically the breakdown of which vector components from each matrix contribute to the final result for each block. The outer two loops move this 4×4 block around the result matrix, using the rows from A and columns from B to compute the solution. The one tricky aspect of this implementation is the fact that matrix B is really stored in row-major format, as are all of our matrices. This means that the blocks from matrix B must be transposed in order to be able to operate on them as four vector float objects. Here we use our handy `vec_transpose()` routine presented earlier. As an example, if we have matrices $A_{8\times8}$ and $B_{8\times8}$, and partition them into 4×4 blocks so that:

```
\left[
\begin{matrix}
C_{11} & C_{12} \\
C_{21} & C_{22} \\
\end{matrix}
\right]
=
\left[
```

Figure 11-25. Vector Components Contributing to the Matrix-Matrix Multiply Solution




```
\begin{matrix}
A_{11} & A_{12} \\
A_{21} & A_{22} \\
\end{matrix}
\right]
\left[
\begin{matrix}
B_{11} & B_{12} \\
B_{21} & B_{22} \\
\end{matrix}
\right]
```

then we can compute each block in C by:

```
C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j}
```

Just for reference, I would like include a copy of what the non-optimized matrix-matrix multiply looks like:

```
- (void)multiply:(RRMatrix*)rhs in:(RRMatrix*)place
{
    int i, j, k, p = [rhs columns];
    float **C = [place data];
    float **A = [self data];
    float **B = [rhs data];

    for(i = 0; i < m; i++) {
        for(j = 0; j < n; j++) {
            C[i][j] = 0.0;
            for(k = 0; k < p; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

As a programmer coming along two years after-the-fact, which routine would you prefer to have to deal with? If this example doesn't send you screaming for the exits, nothing will. Vectorized code, like all optimization, can be rather obtuse to the casual observer. Many times when working on AltiVec code, you could (and still can) find me hunched over for hours with pencil and paper in hand drawing funny blocks and arrows trying to figure out what a piece of code was actually doing. Check out the resources listed at the end of the chapter, and make sure to check the www.altivec.org website and mailing list. There is a very useful community of developers that have probably already dealt with the same problem you might be having, and it is all archived for you to use.

11.5. Determining When to Vectorize

When considering whether code is a good candidate for vectorization, it almost always comes down to data.

- Do your algorithms lend themselves to implementation with the vector routines?
- Does your data, and more importantly its memory layout, lend itself to vectorization?
- Know where your speed problems exists!

It is often the case that traditional algorithms are presented in a way that does not lead to a simple implementation using a vector engine, as most of them were developed for traditional scalar arithmetic units. Of additional critical importance to those working with floating point numbers is AltiVec's single-precision floating-point limitation. Are the condition and stability of the algorithms you are using suitable for single-precision? Most traditional algorithms were designed when using double-precision floating point was a huge performance hit - or was not even available. *Gaussian elimination* with partial or complete pivoting is a good example of what can be done to try and control floating point error. Even so, some analysis reach the limits of floating-point sooner than others. *Bayesian* techniques, for example, will reach the limits of float-point accuracy very quickly. In the next chapter, we will take a look at how to detect and deal with the single-precision AltiVec limitations in your code.

When programming to take advantage of the AltiVec unit, or any vector machine for that matter, there is rarely a one-to-one correspondence between an algorithm or equation and its implementation. In almost all cases, you will find that the vector operations in the AltiVec implementation take one component from each of two vectors, perform an operation on them, and put the result in the equivalent location in the resultant vector:

```
c_i = a_i\ \textbf{op}\ b_i
```

This may lead to awkward data layout for vectorization. don't be afraid to change your data format. Instead of vectors that look like: `[w, x, y, z], [w, x, y, z], [w, x, y, z], [w, x, y, z]` try instead organizing your data like: `[w, w, w, w], [x, x, x, x], [y, y, y, y], [z, z, z, z]`. If you find that you are using many permutation operations to get your data into the right form, then you either need to rethink your data layout, or your problem may not be a good candidate for vectorization. The matrix-matrix multiply makes a good example as it requires only one transformation per pass to get the data in the right shape - and this transformation is only needed for one of the matrices.

The AltiVec engine needs data--lots of it. For it to work at top speed, you will need to make sure you are feeding it data as fast as possible. In this age of write-once, run-anywhere, AltiVec programming seems a throwback. To get the most advantage out of AltiVec, you will need to concern yourself with things like pipelines, clock cycles, caches and memory bandwidth. While daunting at first, there are only a few fundamental concepts to watch out for, and the

rewards can definitely be worth it as we saw with our optimized AltiVec dot product implementation. In the next chapter we will look at how memory, and its direct management can have an enormous impact on the success of your AltiVec code.

Primarily, this chapter has been about the *hows* of AltiVec programming--that is, how to get something done. This, it turns out, is only half the battle. In the next chapter we are going to look at the more advanced issues intrinsic in programming for AltiVec, and what is required to make sure your code is getting the most benefit from vectorizing. The irony of course is that we turned to AltiVec in order to optimize our scalar code, and now we are going to start optimizing our optimization! Will this madness never stop? Probably not--which is exactly why optimization should only be carried out after we understand our code's behavior intimately, and can make the determination that it is actually worth the effort. Otherwise, as we get distracted by ever tinier minut\'{e}, we never get to the point of solving the problem we were looking to fix in the first place.

11.6. Errors

Here are some common errors you might run across in ProjectBuilder when programming for AltiVec that have been mentioned throughout the chapter.

`illegal statement, missing ‘;’ after ‘vector’`
`‘vector’ undeclared`

: You have not set the `-faltivec` flag in ProjectBuilder, or from the command-line.

`too few initializers`.

: If you receive this error, it means that you have not provided enough values in your vector literal statement. For example, an vector unsigned char expects either 1 or 16 values. A vector float expects 1 or 4.

`warning: unknown conversion type character ‘v’ in format`,

: the release notes for the Apple developer tools say it best: “library support for vec_new(), vec_malloc(), vec_free(), etc., as well as vector extensions to standard I/O functions such as printf(), are not present.”

`no instance of overloaded builtin function [function name] matches the parameter list`,

: This typically means that you are trying to pass in an argument that AltiVec doesn't accept for the operation. For example, you might be trying to send a vector float into an integer only operation.

11.7. References

For AltiVec, the last word is from the source itself:

AltiVec Technology Programming Environments Manual

: http://e-www.motorola.com/brdata/PDFDB/docs/ALTIVECPEM.pdf

AltiVec Technology Programming Interface Manual

: http://e-www.motorola.com/brdata/PDFDB/docs/ALTIVECPIM.pdf

You will also probably want to have handy the user manual for your processor. While not necessary, it can come in handy when you want to look up items such as the binary representation of an opcode that gdb is not displaying properly, or the number of cycles an instruction takes to complete, which will be dependent on your specific processor version.

MPC7410 Microprocessor User's Manual

//e-www.motorola.com/brdata/PDFDB/docs/MPC7410UM.pdf

MPC7450 Microprocessor User's Manual

//e-www.motorola.com/brdata/PDFDB/docs/MPC7450UM.pdf

Some additional Web resources:

- Apple's Velocity Engine site is a great resource for AltiVec on OSX, and OS9, with tutorials, sample code, and additional resources: //developer.apple.com/hardware/ve/
- The Altivec.org web site: //www.altivec.org. has many useful links, as well as a must-read mailing list if you are serious about using AltiVec.
- Ian Ollmann has a very good AltiVec tutorial, along with sample code at: //idisk.mac.com/simd/Public

Chapter 12. Advanced AltiVec Techniques

12.1. Need to move over to XML

todo

```
\chapter{}\label{AAT}

- java mode
- detecting processor

\section{AltiVec Problems: Exceptions and Debugging}

-altivec debug tools

\section{Performance Considerations}

- pipelinning

- cycle time. latency vs. throughput

- processor differences (7410 vs. 7450) look at dispatch reservation units into VPU and VALU. more p

- a C op might map to more than one assembly. this means you need to

\subsection{Profiling AltiVec Code}

simg4

\subsection{Feeding The AltiVec Engine}

- do an image of stalling the pipeline

- once you start using altivec, the problem becomes one of feeding it fast enough.

\section{Dealing With AltiVec's Limitations}

talk about limitations of single precision vs. double, and that altivec only supports single precisi

- there are many ways around the problems of using single precision floating point numbers. Go t

\section{Using the vecLib Framework}

Thanks for the testing and benchmark results.  My understanding is that
Apple did extensive optimizations in vDSP for single -and- double precision,
independent of the AltiVec support for single precision.  So even in double
precision, we should be getting a pretty hefty speed boost.  I'm glad to see
that your benchmarks demonstrate that!

\subsection{Basic Routines}
```

```
\subsection{Vector Operations}
\subsection{vBLAS}
\subsection{vDSP}
```

Chapter 13. Working With Threads

13.1. Why Multithreaded?

todo:

13.2. Timers

Use NSTimers instead of threads: From Kirk Kerekes========== "Somewhere in Hillegass's CocoaProgrammingforMacOSX, Aaron makes the comment that most uses of NSThread can be better handled with creative use of NSTimer. Sage advice indeed. I wish I had listened more carefully. If the inability to actually release an NSTimer bugs you (as I know it bugs me), consider going a step lower to NSObject's -(void)performSelector:(SEL)aSelector withObject:(id)anArgument afterDelay: (NSTimeInterval)delay and -(void)performSelector:(SEL)aSelector withObject:(id)anArgument afterDelay: (NSTimeInterval)delay inModes:(NSArray *)modes -- which I infer contain NSTimer's core functionality. I ripped a bunch of crash-prone threads out of my current app and replaced them with NSTimer and performSelector techniques, and the perceived speed of the app increased dramatically. Add some judicious use of NSNotifications to handle document closing and application quitting, and the solution was complete. It never crashes inexplicably, and I don't have to worry about "thread safe" considerations. ==========

from ondra: > Are sequential performSelector:withObject:afterDelay: scheduled performs > guaranteed to be performed in the same order? The documentation doesn't say > one way or the other. Therefore they are not. I believe they actually do work that way currently (mainly since they, so far as I know, are stored with an absolute time computed as "now+delay", and of course "now" for subsequent messages differ), but you can't depend on this behaviour.

> What is the 'proper' way to shutdown down a thread? [NSThread exit] from within the thread you want shut down takes care of all the issues you mention.

> Option 1: Every user action that requires talking to the server spawns a new > thread, which attempts to communicate to the server, but is locked until the > server socket is free. This has the benefits of being easy to code: Expect > that I got my data back from my socket request, and update the UI on a > successful response. This has the downside that I will be creating many > threads, though I'd be surprised if any user could get more than two or three > threads going at a time (most server request / responses have little data > associated). > Option 2: Set up a communication thread when the program starts up, and set > up a queue to handle server requests. This has the benefit that only one > extra thread will be around. It has the downside that it makes it much more > difficult to know what response I'm receiving, and makes it more difficult to > keep the UI in sync with the server state. You can also create a thread pool which is a mix of option 1 & 2 and bound how big this thread pool gets. It also avoids having to create threads again and again.

13.3. Spinning Off Threads

todo

13.4. Data Contention

todo

http://developer.apple.com/technotes/tn2002/tn2059.html Using the Cocoa collection classes in mt apps.

I read that the "[[object retain] autorelease]" cycle was used for multi-threading safety. Is this a guaranteed behavior of NSEnumerator? ... On Tuesday, September 17, 2002, at 12:05 PM, John C. Randolph wrote: Okay, I just checked. NSEnumerator does retain and autorelease any object it returns from -nextObject. -jcr Fabien

```
Read the artcle:

I did. The part that makes it thread safe is having the retain in the lock, not the autorelease. You

      //    Get the dictionary’s value, and then try to message the value.
        [aDictionaryLock lock];
        anObject = [aDictionary objectForKey: KEY];
        [[anObject retain] autorelease];
        [aDictionaryLock unlock];

        [anObject description];

While that prevents the crash, this works also, without cluttering the pool:

      //    Get the dictionary’s value, and then try to message the value.
        [aDictionaryLock lock];
        anObject = [[aDictionary objectForKey: KEY] retain];
        [aDictionaryLock unlock];

        [anObject description];
        [anObject release];

Along with the same in doSets():

        [aDictionaryLock lock];
        [aDictionary setObject: anObject  forKey: KEY];
        [aDictionaryLock unlock];

        [anObject description];
        [anObject release];

The important thing is that you retain when inside the lock and release when you’re done. While the

matt.
```

http://www.mulle-kybernetik.com/artikel/Optimization/opti-4-atomic.html great example of using a little assembler to speed up locks around incrementing and decrementing, instead of using NSLock.

Basically, I have the producer / consumer problem. Thread number 2 needs the data and is writing it out to disk. Conceivably, this disk write could take seconds. I cannot have thread number one blocked for seconds, or it will loose data.

13.5. Communicating with Distributed Objects

todo

13.6. Multithreading and AppKit

todo

Found this rather vague piece of info in the Jaguar release notes. Does this mean we no longer have to use DO for threads to use the AppKit? You don’t have to anyways, because there’s now a new NSObject method called -performSelectorOnMainThread:withObject:waitUntilDone: (see Foundation Release Notes). http://developer.apple.com/technotes/tn2002/tn2053.html Threading Support (AppKit) Threading support in AppKit. NSWindows can now be created in threads other than the main thread. (r. 2887016).

Chapter 14. Clustering

14.1. Why Cluster?

todo

14.2. Distributed Applications

On 10.1 you cannot use the dotted-number IP address format - you need to use a FQDN.

It sounds like your object is getting dealloced and then you attempt to access it later. Try running with the following environment variables set: NSZombieEnabled = YES NSAutoreleaseFreedObjectCheckEnabled = YES You should see the code complaining if you attempt to access a dealloced object.

The docs for -[NSRunLoop run] state "If there are no input sources in the run loop, it exits immediately". So, removing your port from the run loop should cause it to stop running.

If you remove all runloop sources from a run loop, it will automatically exit.

On Wednesday, August 14, 2002, at 06:15 , Brian Webster wrote: The docs for -[NSRunLoop run] state "If there are no input sources in the run loop, it exits immediately". So, removing your port from the run loop should cause it to stop running. Yes, that is what I read too. So, I tried this but I must be doing something wrong, because it does not work. Well, you have to remember that you are not the only client of the run loop. The libraries you link with may put sources of their own in the run loop. This is one of the strengths of the run loop model, where the list of sources to be waited upon is managed independently of the actual blocking for events, and different parts of a program can operate with event handling independently of each other. If you want the run loop to terminate, you shouldn’t use the run-forever convenience method, -run. Instead, use one of the other run methods and also check other arbitrary conditions of your own, in a loop. A simple example would be: BOOL shouldKeepRunning = YES; // global NSRunLoop *theRL = [NSRunLoop currentRunLoop]; while (shouldKeepRunning && [theRL runBeforeDate:[NSDate distantFuture]); where shouldKeepRunning is set to NO somewhere else in the program.

14.3. Remote Ports: NSSocketPort

todo

14.4. Finding Objects Out on the Network

todo

14.5. Mixing Local and Remote Port Connections

todo

14.6. Applying Patterns for Distributed Computing

todo

Chapter 15. AltiVec Operations

15.1. Conversion Operations

`vec_ctf`
: convert from fixed-point word

vec_cts
: convert to signed fixed-point word saturated

vec_ctu
: convert to unsigned fixed-point word saturated

Mathematical Operations

vec_abs
: absolute value

vec_abss
: absolute value saturated

vec_add
: add

vec_addc
: add carryout unsigned word

vec_adds
: add saturated

vec_avg
: average

vec_ceil
: ceiling

vec_expte
: 2 raised to the exponent esimate floating-point

vec_floor
: floor

vec_loge

: log_2 estimate floating-point

vec_madd

: multiply add

vec_madds

: multiply add saturated

vec_mladd

: mulitply low and add unsigned half word

vec_mradds

: mulitply round and add saturated

vec_msum

: multiply sum

vec_msums

: multiply sum saturated

vec_mule

: multiply even

vec_mulo

: multiply odd

vec_nmsub

: negative multilpy subtract

vec_re

: reciprocal estimate

vec_round

: round

vec_rsqrte

: reciprocal square root estimate

vec_sub

: subtract

vec_subc

: subtract carryout

vec_subs

: subtract saturated

vec_sum4s

: sum across partial (1/4) saturated

vec_sum2s

: sum across partial (1/2) saturated

vec_sums

: sum saturated

vec_trunc

: truncate

Logical, Comparitor and Permutation Operations

vec_and

: AND

vec_andc

: AND with complement

vec_nor

: NOR

vec_or

: OR

vec_xor

: XOR

vec_cmpb

: compare bounds floating-point

vec_cmpeq

: compate equal

vec_cmpge

: compare greater than or equal

vec_cmpgt

: compare greater than

vec_cmple

: compare less than or equal

vec_cmplt

: compare less than

vec_max

: maximum

vec_min

: minimum

vec_all_eq

: all elements equal

vec_all_ge

: all elements greater than or equal

vec_all_gt

: all elements greater than

vec_all_in

: all elements in bounds

vec_all_le

: all elements less than or equal

vec_all_lt

: all elements less than

vec_all_nan

: all elements not a number

vec_all_ne

: all elements not equal

vec_all_nge

: all elements not greater than or equal

vec_all_ngt

: all elements not greater than

vec_all_nle

: all elements not less than or equal

vec_all_nlt

: all elements not less than

vec_all_numeric

: all elements numeric

vec_any_eq

: any element equal

vec_any_ge

: any element greater than or equal

vec_any_gt

: any element greater than

vec_any_le

: any element less than or equal

vec_any_lt

: any element less than

vec_any_nan

: any element not a number

vec_any_ne

: any element not equal

vec_any_nge

: any element not greater than or equal

vec_any_ngt

: any element not greater than

vec_any_nle

: any element not less than or equal

vec_any_nlt

: any element not less than

vec_any_numeric

: any element numeric

vec_any_out

: any element out of bounds

Permutation Operations

vec_mergeh
: merge high

vec_mergel
: merge low

vec_pack
: pack

vec_packpx
: pack pixel

vec_packs
: pack saturated

vec_packsu
: pack saturated unsigned

vec_perm
: permute

vec_rl
: rotate left

vec_sel
: select

vec_sl
: shift left

vec_sld
: shift left double

vec_sll
: shift left long

vec_slo
: shift left by octet

vec_splat
: splat

vec_splat_s8
: splat signed byte

vec_splat_s16
: splat signed half word

vec_splat_s32
: splat signed word

vec_splat_u8
: splat unsigned byte

vec_splat_u16
: splat unsigned half word

vec_splat_u32
: splat unsigned word

vec_sr
: shift right

vec_sra
: shift right algebraic

vec_srl
: shift right long

vec_sro
: shift right octet

vec_unpackh
: unpack high element

vec_unpackl
: unpack low element

Memory Operations

vec_dss
: data stream stop

vec_dssall
: data stream stop all

vec_dst

: data stream touch

vec_dstst

: data stream touch for store

vec_dststt

: data stream touch for store transient

vec_dstt

: data stream touch transient

vec_ld

: load indexed

vec_lde

: load element indexed

vec_ldl

: load indexed LRU

vec_lvsl

: load for shift left

vec_lvsr

: load for shift right

vec_mfvscr

: move from vector status and control register

vec_mtvscr

: move to vector status and control register

vec_st

: store indexed

vec_ste

: store element indexed

vec_stl

: store indexed LRU