PowerPC Compilers: Still Not So Hot

I believe it was Theodore Roosevelt who first called the presidency of the United States a “bully pulpit,” which is a catchy way of saying that the president can rant on a subject, people will actually listen, and maybe those people will even do something about whatever the topic of the rant happens to be. Magazine columns can be bully pulpits as well, and, while a computer magazine column is clearly not a pulpit on the same level as the White House, I don’t expect to hear Bill Clinton taking compiler vendors to task about lame optimization quality in the next State of the Union Address, so I might as well do it myself.

Review Problems

This article started out as a comparative review of compiler optimizations, but the more I learned about the various compilers and how they did or did not optimize, the more the article turned into an exploration of how we as programmers have to help the compilers do a good job with our code. So while I’m still going to talk about five compilers and give comparison charts like a normal review, I’m actually going to concentrate on how our source code changes affect the assembly the compiler generates.

Most other compiler reviews focus on the compiler's integrated development environment, on the fancy editor with color syntax highlighting that doesn’t even let you write a simple macro, on the debugger’s silly ToolTip windows (that pop up over variable names with their values if you hold your mouse there forever), and on the compiler supplied class library that violates just about every precept of good object-oriented design in C++ and is bloated and slow to boot. Wow! As you can see, I’m no fan of compiler reviews—I believe most are written by either nonprogrammers or nonproduction programmers writing toy programs. Compilers themselves are written for those reviewers, and so we end up with the current mess, where compiler vendors focus on silly new features to please silly reviewers instead of focusing on things that actually help production programmers do their jobs well.

When I evaluate a compiler I look for two things: C++ compliance and code optimization. The former is basically a lost cause at this point because the C++ draft standard is still a moving target and there’s no solid conformance suite. I pray this will change soon. By contrast, compiler writers have had years to work on compiler optimizations, and not much has changed since the early days.

By focusing on optimizations, we’ll not only learn which compilers optimize the best, we’ll also learn what we can do to help a compiler do its best with our code. This time, we’ll be covering compilers for the PowerPC chip on the Macintosh, and next time we’ll cover the Intel x86. Even if you don’t program for the PowerPC, reading this will help you learn a lot about compilers and how they optimize, and that knowledge will carry across to whatever CPU you care to program.

The compilers we’ll cover this issue are: Metrowerks CodeWarrior 8, Symantec C++ 8, version 1.0f3e2 of Apple’s MrCpp compiler (which is included with the Symantec compiler), Motorola’s 2.1.1 PowerPC C++ compiler, and the Microsoft Visual C++ for Macintosh 4.0 cross compiler.

The Test Code

We’ll use a simple inner product of a three-by-three matrix and a three element column vector to evaluate each

Table 1. Transform Cycle Counts

Compiler	Listing 1	Listing 2	KAPed 1 (not shown)	Listing 4	Listing 5	Listing 6
CodeWarrior	40.7	50.5	50.9	34.3	29.7	19.6
Symantec C++	76.6	94.9	82.8	50.9	31.9	25.7
Motorola C++	34.5	47.4	39.5	33.2	30.8	20.6
Apple’s MrCpp	52.0	65.0	56.2	36.1	28.8	19.5
Microsoft VC++	41.6	49.3	42.8	31.9	21.9	22.7

Chris Hecker

Compilers. What are they good for? Chris Hecker steps to the bully pulpit to rant about the state of current PowerPC compilers. Sadly, these days, most compilers need a lot of help optimizing code.

compiler's optimization quality. Obviously, a single function is not going to tell the whole optimization story, but it should give us an idea of what sorts of optimizations we can expect from today's compilers.

Listing 1 shows the function `TransformVectors`. I made it transform an array of vectors so the compilers would have to work a bit harder, but, even so, the code is trivial. I used 1,000 calls to this function with 500 transforms on each call to gather timing information. The first column of data in Table 1 shows the approximate cycle counts for each product measured with the MacOS call `Microseconds` for the various compilers on my Power Computing 604. I turned on all the optimizations I could find on each compiler to gather this data. I made sure my test program was producing correct results on every compiler by making the source vectors eigenvectors of the transform matrix and checking to see if the transformed vector was the same as the source—it's always a good idea to make sure neither you nor the compiler has introduced any bugs while optimizing.

Anti-Alias

If you've looked ahead at the other results in Table 1 and the other listings, you're probably wondering what the second column of data means, and why Listing 2 is almost identical to Listing 1. Even though you and I know we wouldn't call `TransformVectors` from Listing 1 with the source or destination pointing to the same vector, or, worse yet, with the destination pointing into the middle of the matrix somewhere, the compiler doesn't know this, so it can't assume we didn't do something silly. When a variable points to another live variable in the function, it's called "pointer aliasing," and when the compiler sees a write through a pointer, it needs to assume that the data could have landed anywhere, including into variables it's already loaded into registers. This means

Listing 1. The Test Function

```
void TransformVectors( float *pDestVectors,
        float const (*pMatrix)[3], float const *pSourceVectors,
        int NumberOfVectors )
{
        int Counter, i, j;
        for(Counter = 0;Counter < NumberOfVectors;Counter++) {
                for(i = 0;i < 3;i++) {
                        float Value = 0;
                        for(j = 0;j < 3;j++) {
                                Value += pMatrix[i][j] * pSourceVectors[j];
                        }
                        *pDestVectors++ = Value;
                }
                pSourceVectors += 3;
        }
}
```

Listing 2. The Non-Aliasing Test Function

```
void TransformVectors2( float *pDestVectors,
        float const (*pMatrix)[3], float const *pSourceVectors,
        int NumberOfVectors )
{
        int Counter, i, j;
        for(Counter = 0;Counter < NumberOfVectors;Counter++) {
                float aTemp[3];
                for(i = 0;i < 3;i++) {
                        float Value = 0;
                        for(j = 0;j < 3;j++) {
                                Value += pMatrix[i][j] * pSourceVectors[j];
                        }
                        aTemp[i] = Value;
                }
                pSourceVectors += 3;
                for(i = 0;i < 3;i++) {
                        *pDestVectors++ = aTemp[i];
                }
        }
}
```

the optimizer has to continually reload variables into registers in case we're aliasing parameters, so I wrote `TransformVectors2` in Listing 2 to give the compilers some help. Since `aTemp` is defined local to our function, the compiler knows it can't be aliased, so writes to `aTemp` shouldn't cause spurious register reloads.

Well, at least that's what I thought, anyway. As you can see from the timings, all the compilers got slower because not only did they still reload all the registers, they also naively implemented the copy loop at the end of Listing 2!

Let's look at the code generated by the winner of this round, the Motorola C++ compiler. Listing 3 shows the PowerPC assembly language generated for `TransformVectors2`, our supposedly non-aliased function. Despite some odd ways of moving values into registers, this code is a pretty straightforward translation of our source into assembly language, which is disappointing. For example, the compiler doesn't bother to load the source vector into registers outside the loop, even though it's used three times and cannot be aliased because of our temporary results array. Also, instead of leaving the temporary results in registers, it actually copies them out to the stack and then copies the stack to the destination.

It even increments the destination pointer in the loop with three discrete instructions instead of using offsets and doing one addition at the end, or even using the PowerPC's autoincrement instructions. The Motorola compiler also produced the fastest code for Listing 1, and the difference between the timings for Listings 1 and 2 can be attributed to the naive compilation of the temporary copy loop at the end of Listing 2 (even though the temporary loop was supposed to help by eliminating the possibility of aliasing). Overall, not a great showing, even by our winner in this round. Clearly, the compilers need more help.

Bust a KAP

The Motorola compiler ships with an interesting tool, called the Kuck and Associates Preprocessor for C (KAP). Basically, KAP compiles your C code (it doesn't support C++), optimizes it, and then generates C code as its output

Listing 3. Motorola C++ Assembly for Listing 2

```
TransformVectors2__FPfPA3_CfPCfi.b:
        cmpi    0x7,0x0,r6,0     ; compare count to 0
        addi    r11,r0,0         ; Counter = r11 = 0
        bc      0x4,0x1d,L..11   ; bail out if count = 0
        addi    r8,sp,24         ; allocate some stack
L..8:   ori     r9,r4,0x0        ; r9 = pMatrix
        addi    r10,r0,0         ; r10 = 0
        ori     r7,r10,0x0       ; r7 = 0
        subfic  r10,r10,3        ; r10 = 3
        mtctr   r10              ; ctr = 3
L..9:   lfs     f1,0(r9)         ; f1 = pMatrix[0]
        lfs     f2,0(r5)         ; f2 = pSource[0]
        lfs     f3,4(r9)         ; f3 = pMatrix[1]
        lfs     f4,4(r5)         ; f4 = pSource[1]
        fmuls   f1,f1,f2         ; f1 = f1 * f2
        lfs     f2,8(r9)         ; f2 = pMatrix[2]
        lfs     f5,8(r5)         ; f5 = pSource[2]
        fmadds  f3,f3,f4,f1      ; f3 = f3 * f4 + f1
        addi    r9,r9,12         ; pMatrix->next row
        fmadds  f2,f2,f5,f3      ; f2 = f2 * f5 + f3
        stfsx   f2,r8,r7         ; *(stack + r7) = f2
        addi    r7,r7,4          ; r7 next float
        bc      0x10,0x0,L..9    ; branch if (--ctr)
        lfs     f1,0(r8)         ; f1 = stack[0]
        lfs     f2,4(r8)         ; f2 = stack[1]
        lfs     f3,8(r8)         ; f3 = stack[2]
        stfs    f1,0(r3)         ; pDest[0] = f1
        addi    r3,r3,4          ; pDest++
        addi    r11,r11,1        ; Counter++
        stfs    f2,0(r3)         ; pDest[1] = f2
        addi    r3,r3,4          ; pDest++
        cmp     0x7,0x0,r11,r6   ; flags = Counter <
NumVecs
        addi    r5,r5,12         ; pSource += 3
        stfs    f3,0(r3)         ; pDest[2] = f3
        addi    r3,r3,4          ; pDest++
        bc      0xc,0x1c,L..8    ; branch if (Counter <
NumVecs)
L..11:  addi    sp,sp,48         ; clear stack
        bclr    0x14,0x0         ; return
```

Listing 4. The KAPed Listing 2

```
void TransformVectors2( float *pDestVectors,
         const float (*pMatrix)[3], const float *pSourceVectors,
         int NumberOfVectors )
{
    int Counter, i, j;
    float aTemp[3];
    float Value, _Krr1, _Krr2, _Krr4, _Krr5;
    long _Kii1, _Kii2;
    for ( Counter = 0; Counter<NumberOfVectors; Counter++ ) {
        _Krr1 = 0.0F; _Krr2 = 0.0F; Value = 0.0F;
        _Kii1 = Counter * 3;
        _Krr1 +=  pMatrix[0][0] * pSourceVectors[_Kii1];
        _Krr2 +=  pMatrix[1][0] * pSourceVectors[_Kii1];
        Value +=  pMatrix[2][0] * pSourceVectors[_Kii1];
        _Krr1 +=  pMatrix[0][1] * pSourceVectors[_Kii1+1];
        _Krr2 +=  pMatrix[1][1] * pSourceVectors[_Kii1+1];
        Value +=  pMatrix[2][1] * pSourceVectors[_Kii1+1];
        if (1) {
            _Krr1 +=  pMatrix[0][2] * pSourceVectors[_Kii1+2];
            _Krr2 +=  pMatrix[1][2] * pSourceVectors[_Kii1+2];
```

instead of assembly language. When I first got the Motorola compiler, I figured my test would be so simple that there was no way KAP could help out, but after looking at the results we just discussed, I figured anything was worth a try. Listing 4 shows the output of running Listing 2 through KAP (they call the processed code KAPed). If you've never seen machine-generated C code, don't be surprised by stuff like the `"if (1)"` block—compilers output weird stuff like that for bizarre reasons. However, you should be surprised at how poor the code is. It unrolls the loop, which is fine, but why does it go to the trouble of putting the temporaries in `aTemp` and then looping over `aTemp` to copy them into the destination? More absurd yet is that something as mundane as unrolling a loop in this simple function actually helped the compilers produce faster code.

You can see the timing results in Table 1. The KAPed Listing 1 is not even worthy of print, and as you can see the compilers all got slower on that version. The KAPed Listing 2 (shown in Listing 4) actually made a positive difference, even on the best compilers, and it made a huge difference on Symantec. Even so, if you thought it was bad that KAP generated the redundant loop at the end of Listing 4, it's even worse that every compiler generated actual assembly language code for that loop! The worst offender is clearly Symantec. Symantec ships a prerelease version of Apple's MrCpp compiler with their package, so it's unclear if I should even review Symantec on their optimization quality because I think they expect you to use MrCpp if you care about run-time speed. However, MrCpp and Symantec's main-

Listing 4. Continued from p. 16

```
                Value +=  pMatrix[2][2] * pSourceVectors[_Kii1+2];
            }
            aTemp[0] = _Krr1; aTemp[1] = _Krr2; aTemp[2] = Value;
            _Kii2 = Counter * 3;
            for ( i = 0; i<=2; i++ ) {
                _Krr5 = aTemp[i]; _Krr4 = _Krr5;
                pDestVectors[_Kii2+i] = _Krr4;
            }
        }
        pSourceVectors +=  NumberOfVectors * 3;
        pDestVectors +=  NumberOfVectors * 3;
    }
```

Listing 5. Hand-optimized Listing 1

```
void TransformVectors( float *pDestVectors,
        const float (*pMatrix)[3], const float *pSourceVectors,
        int NumberOfVectors )
{
    int Counter;
    float Value0, Value1, Value2;
    for ( Counter = 0; Counter<NumberOfVectors; Counter++ ) {
            Value0 = pMatrix[0][0] * pSourceVectors[0];
            Value0 += pMatrix[0][1] * pSourceVectors[1];
            Value0 += pMatrix[0][2] * pSourceVectors[2];
            *pDestVectors++ = Value0;
            Value1 = pMatrix[1][0] * pSourceVectors[0];
            Value1 += pMatrix[1][1] * pSourceVectors[1];
            Value1 += pMatrix[1][2] * pSourceVectors[2];
            *pDestVectors++ = Value1;
            Value2 = pMatrix[2][0] * pSourceVectors[0];
            Value2 += pMatrix[2][1] * pSourceVectors[1];
            Value2 += pMatrix[2][2] * pSourceVectors[2];
            *pDestVectors++ = Value2;
            pSourceVectors += 3;
    }
}
```

line compiler are not C++ feature-equivalent, so I'm not sure how they can expect you to freely exchange them.

A Helping Hand

At this point, it was clear that the compilers by themselves—and even with the help of KAP, for what it's worth—were not going to be able to produce reasonable code for these functions, so I had to step in and give them a hand. I looked at what kind of improvement KAP got from using what I had assumed were brain-dead rewrites (I can't even bring myself to call them optimizations), and I decided to hand code the `Transform` functions to see what would come out. Listings 5 and 6 contain the hand-optimized versions of Listings 1 and 2, respectively. You can see from the

Listing 6. Hand-optimized Listing 2

```
void TransformVectors2( float *pDestVectors,
    const float (*pMatrix)[3], const float *pSourceVectors,
    int NumberOfVectors )
{
    int Counter;
    float Value, _Krr1, _Krr2;
    for ( Counter = 0; Counter<NumberOfVectors; Counter++ ) {
        _Krr1 =  pMatrix[0][0] * pSourceVectors[0];
        _Krr2 =  pMatrix[1][0] * pSourceVectors[0];
        Value =  pMatrix[2][0] * pSourceVectors[0];
        _Krr1 +=  pMatrix[0][1] * pSourceVectors[1];
        _Krr2 +=  pMatrix[1][1] * pSourceVectors[1];
        Value +=  pMatrix[2][1] * pSourceVectors[1];
        _Krr1 +=  pMatrix[0][2] * pSourceVectors[2];
        _Krr2 +=  pMatrix[1][2] * pSourceVectors[2];
        Value +=  pMatrix[2][2] * pSourceVectors[2];
        *pDestVectors++ = _Krr1;
        *pDestVectors++ = _Krr2;
        *pDestVectors++ = Value;
        pSourceVectors += 3;
    }
}
```

timing results in the last two columns of Table 1 that it made a big difference on all the compilers.

Why did it make such a big difference? I have no idea, and the only explanation I can come up with is that you need to hold your compiler's hand on any piece of code you care about. The changes I made for Listings 5 and 6 are very obvious (to a human, if not a compiler). I'm basically just stating explicitly where variables are accessed, where possible aliasing can occur, and which variables are constant throughout a loop iteration. These are all things the compiler is supposed to do for us, so we can work on more important stuff, like design and algorithms, or assembly language code for our most inner loops. We're supposed to trust the compiler will do a respectable job, without having to optimize every line of our code (an impossible task for all but the smallest programs).

Now, if you're like me, you've been waiting to say something about all this loop unrolling for a while now. You're waiting to say that you don't actually want the compiler to unroll loops all over the place, because that makes your code bigger and probably slower. Hah! I was waiting for you to say that, because the most amazing thing of all about this test is that a couple of the compilers produced code for Listing 6 that is smaller than the code for the original non-unrolled Listing 2. Listing 7 shows the

CodeWarrior version of Listing 6; it's at least 6 instructions smaller than any of the compiled versions of Listing 2, and about two to five times as fast. Motorola produced similar code. (MrCpp decided now was the time to unroll the entire function, which tripled the size of the code for absolutely no performance increase.) Don't for a minute think this is a compliment for CodeWarrior or Motorola, it's really a damning insult to all the compilers: on maximum optimizations they didn't find the smaller and faster version of a basic function like a matrix transform. Heck, just by inspection I can see how to save a couple more instructions in Listing 7. And people actually say that writing assembly language is a dying art.

You Lose Some

If this was a normal compiler review, it would be time to pick a winner, but, instead, it's time to point out that you and I are the losers in this situation. Pundits have been saying that assembly language is dead—especially on RISC chips like the PowerPC—and it should be eminently clear from the listings in this article that those people have no clue what they're talking about. Even a beginning assembly language programmer could produce better code than any of the compilers for Listings 1 and 2, and this is simple code. While you might not choose to write your code in assembly language, you end up with C code that looks like assembly language if you want respectable performance, like Listings 5 and 6.

If I had to choose a winner, I'd pick the Motorola C++ compiler, because it seems like the least incompetent optimizer of the bunch. The Microsoft compiler showed some promising aggressiveness by loading the entire matrix into registers once at the top of the loop for its version of Listings 5 and 6. Microsoft has an option I turned on that tells the compiler there's no pointer aliasing that allowed them to perform this optimization, but they didn't take advantage of the assumption anywhere else that I could see. Given this optimization, I'm not sure why their version of Listing 6 wasn't faster than the others…it may have been a pipelining issue, or the loop might have been cache bound. As soon as I learn a bit more about the subtleties of the PowerPC 604, I'll get back to you on this one.

In the next issue, I'll quickly cover a bunch of Intel x86 compilers, but we will still have room to talk about some optimization programming techniques of our own, because the compilers clearly aren't going to do it for us.

This Just In: I was keeping Mike Phillip at Motorola's compiler group posted on my results, and he just got back to me with a command line switch for KAP that will assume there's no aliasing. When you turn it on, KAP produces something resembling Listing 6,

Listing 7. CodeWarrior version of Listing 6

```
TransformVectors2__FPfPA3_CfPCfi
     mr      r0,r6       ; r0 = NumVecs
     cmpwi   r6,0        ; flags = NumVecs == 0
     mtctr   r0          ; ctr = NumVecs
     blelr              ; bail if(NumVects == 0)
L1:  lfs     fp1,0(r4)  ; fp1 = pMatrix[0][0]
     lfs     fp3,0(r5)  ; fp3 = pSource[0]
     lfs     fp0,12(r4) ; fp0 = pMatrix[1][0]
     lfs     fp2,24(r4) ; fp2 = pMatrix[2][0]
     fmuls   fp7,fp1,fp3; fp7 = fp1 * fp3
     lfs     fp1,4(r4)  ; fp1 = pMatrix[0][1]
     lfs     fp5,4(r5)  ; fp5 = pSource[1]
     fmuls   fp8,fp0,fp3; fp8 = fp0 * fp3
     fmuls   fp6,fp2,fp3; fp6 = fp2 * fp3
     lfs     fp0,16(r4) ; fp0 = pMatrix[1][1]
     lfs     fp4,28(r4) ; fp4 = pMatrix[2][1]
     lfs     fp2,8(r5)  ; fp2 = pSource[2]
     fmadds  fp7,fp1,fp5,fp7
                        ; fp7 = fp1 * fp5 + fp7
     addi    r5,r5,12  ; pSource += 3
     lfs     fp3,8(r4) ; fp3 = pMatrix[0][2]
     fmadds  fp8,fp0,fp5,fp8
                       ; fp8 = fp0 * fp5 + fp8
     lfs     fp1,20(r4); fp1 = pMatrix[1][2]
     fmadds  fp6,fp4,fp5,fp6
                       ; fp6 = fp4 * fp5 + fp6
     lfs     fp0,32(r4); fp0 = pMatrix[2][2]
     fmadds  fp7,fp3,fp2,fp7
                       ; fp7 = fp3 * fp2 + fp7
     fmadds  fp8,fp1,fp2,fp8
                       ; fp8 = fp1 * fp2 + fp8
     fmadds  fp6,fp0,fp2,fp6
                       ; fp6 = fp0 * fp2 + fp6
     stfs    fp7,0(r3) ; pDest[0] = fp7
     stfsu   fp8,4(r3)
                      ; pDest[1] = fp8, pDest++
     stfsu   fp6,4(r3)
                      ; pDest[1] = fp6, pDest++
     addi    r3,r3,4   ; pDest++
     bdnz    L1        ; branch if(--ctr)
     blr               ; return
```

so all the compilers do well. However, not to be out-done, I decided to take this no-aliasing assumption to the limit and explicitly load the matrix into temporaries (much like the Microsoft compiler tried to do). The result: another 25% speedup, with times around 15 cycles, for something the compiler could have done itself. The compilers lose again. ■

Chris Hecker tries to live an optimized life, but he does about as good of a job on his own life as the current crop of compilers does on his code. Contact him at gdmag@mfi.com.