Attention:

This material is copyright © 1995-1997 Chris Hecker. All rights reserved.

You have permission to read this article for your own education. You do not have permission to put it on your website (but you may link to my main page, below), or to put it in a book, or hand it out to your class or your company, etc. If you have any questions about using this article, send me email. If you got this article from a web page that was not mine, please let me know about it.

Thank you, and I hope you enjoy the article,

Chris Hecker
definition six, incorporated
checker@d6.com
http://www.d6.com/users/checker

PS. The email address at the end of the article is incorrect. Please use checker@d6.com for any correspondence.

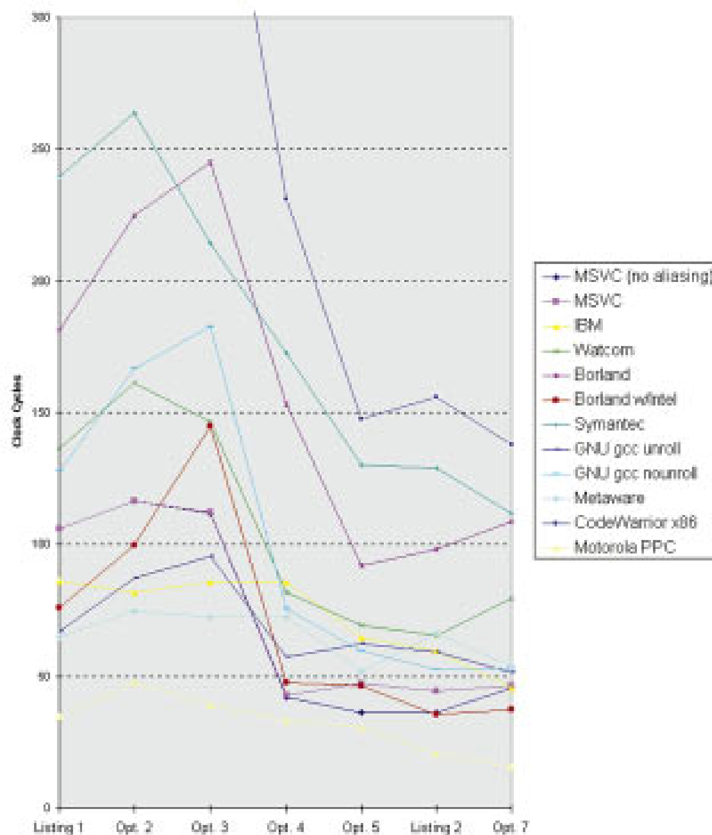
More Compiler Results, and What To Do About It

I sure am glad my full-time job isn't reviewing compilers, because their ubiquitous bugs and wacky user interfaces would drive me insane. However, evaluating compilers does have its moments, like when I found the following paragraph in the Watcom 10.6 compiler's help file under the heading, "What you should know about optimization":

"The C/C++ language contains features which allow simpler compilers to generate code of reasonable quality. Register declarations and imbedding [*sic*] assignments in expressions are two of the ways that C allows the programmer to "help" the compiler generate good quality code. An important point about the Watcom C/C++ compiler is that it is not as important (as it is with other compilers) to "help" the compiler. In order to make good decisions about code generation, the Watcom C/C++ compiler uses modern optimization techniques."

Considering Watcom did around the fourth worst overall in my simple performance test, and did a bad job compiling the texture mapper as well, it would behoove the Watcom compiler writers to refrain from reading their own help files and get back to work on those "modern optimization techniques." Of course, I shouldn't single out Watcom for abuse just because they handed me a convenient passage in their help files. Just like my last article "PowerPC Compilers: Still Not So Hot" (Behind the Screen, June/July 1996), all the compilers this time around deserve abuse, so let's get to it.

Figure 1. The Timing Results



The Contestants

This month, we'll finish up my two-part series on compiler optimizations. Last issue, I evaluated a bunch of C++ compilers for the Macintosh PowerPC platform, and this time I'll do the same for the current crop of x86 compilers. I hesitate to call these "reviews" since I'm not completely evaluating every compiler feature or every possible optimization. However, unlike most compiler reviewers, I'm actually focusing on the compiler—you know, that tiny part of the 200MB Integrated Development Platform and Suite of Accompanying Visual Applications that actually generates the computer code for your application. I'm assuming that since you're reading this magazine you're interested in fast code, and the compiler's the part of the above-mentioned 200MB that generates (or, as we'll see, doesn't generate) that fast code.

This month will be slightly more than an x86 version of last issue's column, however. I'll quickly recap the test results and then move on to what the results from the two articles mean to you as a performance-oriented game programmer.

Chris Hecker

Chris Hecker finishes up his two-part series on compiler optimizations and interprets what the test results mean to you as a performance-oriented game programmer.

I tested eight compilers this time around: Microsoft Visual C++ 4.0, the beta 4 of IBM's VisualAge for C++ for Windows V3.5, Borland C++ 5.0, Watcom C++ 10.6, Metaware C++ 3.32 for OS/2, Metrowerks CodeWarrior 8 in x86 cross-compilation mode, the Free Software Foundation's gcc 2.7.2 on Linux, and Symantec C++ 7.2.

Table 1 shows the eight compilers and their results on my test programs, plus one extra row for both Microsoft VC++ and gcc with different command-line switches. I also included one extra row for Borland using the Intel optimizing backend they supply, and just for kicks I've included the results from the Motorola PowerPC compiler from last time. The timing numbers are in clock cycles per iteration of the test loop, on my 133Mhz Pentium. The PowerPC results are on my 132Mhz PPC604, so it's a pretty fair comparison. I realized after writing the last column that a table full of numbers doesn't exactly tell the most interesting story, so Figure 1 is a graph of Table 1.

The Test

To test the x86 compilers, I used the same simple product of a three-by-three matrix and an array of three element vectors that I used on the PowerPC compilers. Listing 1 shows the first attempt at the code and corresponds to the first column of Table 1. As you move across the table, each column represents a new optimization I applied to the base code in an attempt to coax reasonable output from the compilers. The compilers did pretty poorly on Listing 1; most were two to three times slower on it than on their fastest code, which was usually attained on the function in List-

Table 1. The Timing Results

	Listing 1	Opt. 2	Opt. 3	Opt. 4	Opt. 5	Listing 2	Opt. 7
MSCV (no aliasing)	105.9	116.3	111.7	42.1	36.3	36.3	45.6
MSVC	106.1	116.5	112.1	42.8	47.1	44.4	46.4
IBM	86	81.9	85.7	85.8	64.5	59.6	45.6
Watcom	136.4	161.4	146.4	81.9	69.5	65.6	79.6
Borland	181.1	224.9	245.2	153.1	91.9	98.1	108.84
Borland w/Intel	75.9	99.7	145	47.7	46.4	35.4	37.5
Symantec	239.7	263.8	214.3	172.5	130.2	129	111.9
GNU gcc unroll	67.2	87.4	95.5	57.4	62.3	59.3	51.6
GNU gcc no unroll	127.6	166.9	182.6	75.7	59.6	52.6	53
Metaware	65.1	74.8	72.4	72.6	51.9	66.4	53.9
CodeWarrior x86	309.3	331.3	400.3	230.9	147.7	155.8	137.9
Motorola PPC	34.5	47.4	39.5	33.2	30.8	20.6	15.5

Note: For source code and optimizations, refer to the June/July 1996 issue.

ing 2 and whose results are recorded in the fifth column.

I'm not going to explain the different test programs in detail because I covered that last time. For the complete story and an explanation of the weird variable names in Listing 2, pick up the June/July 1996 issue. The final column of Table 1 shows the results of applying the optimization mentioned in the last paragraph of that article to Listing 2.

In brief, the same criticism I leveled on the PowerPC compilers applies to the x86 compilers: you have to spoon-feed them already optimized code to get reasonable results.

The biggest difference between the PowerPC compilers and x86 compilers is that while you can coax a bad PowerPC optimizer (such as Symantec's PowerPC compiler) into producing almost-optimal code, the same was not true of the bad x86 optimizers (such as Symantec, Borland, and CodeWarrior). I believe generating good PowerPC floating-point code is relatively straightforward compared to generating good x86—and especially Pentium—floating-point code. The wackiness of the Pentium Floating-Point Unit (FPU), with its `FXCH`es, stack-based operands, and stalls, makes optimizing difficult for the x86 compilers. Of course, you'll notice the difference in cycle counts between the PowerPC and x86 tests. My 132Mhz PPC604 is twice as fast as my 133Mhz Pentium at running this code. The speed difference is due to the flat register-based, floating-point architecture of the PowerPC, combined with a quaternary multiply-accumulate instruction. "Quaternary" instructions have four operands (d = a * b + c in the case of a multiply-accumulate); contrast this with the pathetic stack-based binary x86 instructions, where the compiler is forced to constantly move operands around, and you can see why there's a huge difference. Too bad about that annoying market share thing, huh?

It's always a good idea to try to calculate the optimal cycle count for your functions to see what kind of performance improvements are possible, so let's do that for the x86 and the PowerPC. We'll ignore loop overhead and any stalls and assume maximum throughput for this estimate to give ourselves a lower bound.

For the x86, my estimate for the minimum clock cycles to do our matrix multiply is 30 cycles: 9 multiplies at an optimistic 1 cycle each, 6 additions also at 1 cycle each, 3 stores at 2 cycles each, and 9 loads for the source vector because the binary x86 instructions don't let you keep an untouched copy of it in registers. For the PowerPC, we can load the whole matrix into registers before we start, so I count 15 cycles: 3 loads, 9 multiply-additions, and 3 stores. Both estimates are close to the best times we achieved, so we can be pretty sure we're not missing anything major in our analysis.

I suppose, if forced to pick a winner, I'd choose the Microsoft compiler. It seemed to do what it was told most of the time, so if you give it highly optimized code it does an okay job. The Borland compiler with the Intel backend did okay as well, but its quality seemed slightly more random (note the spike in Figure 1). I should also note the Intel backend wouldn't compile my texture mapper correctly, while Borland without the Intel backend compiled it correctly but generated the code quality you'd expect from Borland's position on Figure 1. The IBM and the Metaware compilers were the most consistent of the bunch, meaning they did better than most on the unoptimized functions, as in Listing 1. To me, this indicates both compilers recognize optimization opportunities at a high level but can't generate tight x86 machine code at the low level. Watcom was the most disappointing of the bunch, simply because the conventional wisdom says Watcom generates great code. I didn't see great code from Watcom in my tests.

The main point here is that you cannot expect the compiler to do much work for you beyond a rote translation of the code you write into native machine code. (With the incredible code generation bugs I've found, you sometimes can't even expect this.) If you write a loop that does one simple thing and you express it with 10 inefficient operations, the compiler will faithfully translate all ten operations for you, performance be damned.

With that in mind, let's discuss the kinds of optimizations you should be able to expect from the compiler but currently have to perform yourself.

Transformers, More Than Meets The Eye

When a compiler optimizes your program it (supposedly) does work at a number of different levels. At the lowest levels, it obviously needs to generate the fastest instruction sequence for a given atomic high-level language operation: a C addition of two integers shouldn't turn into much more than a machine code addition with a possible load or store. At a higher level, the compiler puts your code through a series of program transformations which turn the code you wrote into something more amenable to the lower-level code generator. These transformations aren't algorithmic changes. For example, the compiler won't change your $O(n^2)$ bubble sort to an $O(n \log n)$ quick-

Listing 1. The Initial Code

```
void TransformVectors0( 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 = 0.0f;
                        for(j = 0;j < 3;j++) {
                                Value += pMatrix[i][j] * pSourceVectors[j];
                        }
                        *pDestVectors++ = Value;
                }
                pSourceVectors += 3;
        }
}
```

sort; that part is up to you (and algorithm changes are still the most important part of optimizing with the sole exception of profiling your application to make sure you know where to optimize). There are a number of these transformations available to the compiler, but we'll discuss what I think are the five most important ones: alias analysis, code motion, common subexpression elimination, strength reduction, and loop unrolling. You can perform these transformations on your code better than the current crop of compilers, once you know how they work.

Alias Analysis

As we've seen in previous articles, memory is slow compared to registers, so it would be really nice if the compiler could keep all your active variables in registers and operate on them there. If it could do this, it wouldn't have to keep touching memory to reload everything after every store. With pointers, however, it's not that simple. If your code performs reads and writes through two pointers, the compiler needs to decide whether one pointer can point to the same object as the other, a phenomenon called pointer aliasing. For example, think about what would happen in Listing 1 if `pDestVectors` pointed into the middle of `pMatrix`; it certainly wouldn't behave the same as Listing 2. According to the ANSI standard, the compiler needs to be pretty conservative and assume the worst for pointers to variables of the same type. So, one of the first transformations I made to Listing 1 was to use local temporary variables to make explicit to the compiler where I could alias pointers. The compiler knows a write to a temporary cannot affect anything else if you've never taken the address of the temporary. I initially declared a temporary array (as you saw in the previous issue), so I didn't have to unroll the matrix multiply loop, but none of the compilers used this temporary array to eliminate spurious reloads. Apparently today's compilers can't do alias analysis on arrays. I also looked for a compiler switch to make the compiler assume I wasn't aliasing pointers. Most compilers have these switches, and I turned them on when I found them.

Table 1 contains results for the Microsoft compiler both with and without the "assume no aliasing" switch turned on, and you can see the switch makes a big difference on Listing 2. From looking at the disassembly, it looks like the speed increase is due to the compiler now having the ability to move the stores to `pDestVectors` around to better schedule the code. It didn't have this freedom when it had to assume writes to the destination could be writing into one of its source operands. However, you can also see it makes little difference on the unopti-

mized code; it would be hard to make that code any slower.

On the other hand, you don't necessarily want to copy all of your active variables into temporaries, as I found out the hard way. The final column in Table 1 shows the results of code that copies the entire matrix into temporaries before entering the loop. On the PowerPC, you can see this gave me a 25% speedup because the compiler could copy the matrix into registers and reduce the number of loads in the inner loop. On the x86, however, most compilers slowed down on this code because they actually implemented the copies to temporaries. This behavior is related to alias analysis, I believe. If the matrix is in stack-based temporaries in the source code, the compiler needs to prevent writes through `pDestVectors` from changing the matrix elements, so it makes a copy of the matrix in the generated machine code. The PowerPC compiler didn't have to do this because it knows `pDestVectors` can never point into the floating-point registers, where it's keeping the matrix. The x86 compilers couldn't put the matrix in the floating-point registers, so they needed to copy it. This is a particularly bad example, because it means our C level optimizations aren't portable across machines: the PowerPC version got faster while the x86 versions got slower on the same code.

As an aside, I wish the ANSI C++ standard would loosen up their requirements for compilers to support pointer aliasing so pointers to `const` could be assumed to not be aliased by pointers to non-`const` in a function. However, I'm sure this would break a ton of code that relies on aliasing, so it's not likely to happen. I'd say this code is poorly written and deserves to be broken, but aliasing is a complex issue and I might be missing a legitimate use of it.

Code Motion

When you move loop invariants out of the loop, you're performing code motion. A loop invariant is something that doesn't change during the life of the loop, so it makes sense to calculate it once outside the loop and store it rather than calculate it every time. Code motion can also mean rearranging code so that it pipelines better or accesses memory sequentially for better memory bandwidth.

Common Subexpression Elimination

A common subexpression is an operation that appears multiple times in your code. For example, if you compute x + y in two places, and neither x nor y can change between those two places, then x + y is a common subexpression. Usually it's faster to compute the expression once and store its result than to compute the result multiple times. Of course, there's an exception to every rule, especially in these days

of wicked fast processors and slow memory systems. Computing something and caching it might be slower than just computing it multiple times. Time your code, as always. By the way, I've seen the acronym "CSE" applied to both Common SubExpression and Common Subexpression Elimination.

Strength Reduction

The classic example of strength reduction is turning a multiply or divide by a power-of-2 into a shift. I'm not sure why it's called strength reduction, but the basic idea is to convert an expensive operation into a cheap one or a series of cheap ones. Taking the classic example a step farther, you can break up more complicated multiplies into simpler ones (for example, x * 6 = x * 2 + x * 4), which can again be strength-reduced to some shifts and adds. Some architectures might further benefit from strength-reducing shifts by 1 to an addition. Another subtle but powerful example of strength reduction is a Bresenham line drawer, or any kind of forward differencing algorithm. These algorithms convert linear or even arbitrary degree polynomial equation evaluations into a bunch of additions by computing the forward differences outside the loop.

Replacing a divide with a multiplication by the reciprocal is an optimization that could arguably be called strength reduction, but it could also be considered an example of a related transformation called algebraic identification. You can guess from the name what that means.

Loop Unrolling

Finally, we come to everyone's favorite optimization—loop unrolling. Here, we try to mitigate some loop overhead and perhaps open up possibilities for pipelining by duplicating the loop body and reducing the loop count to compensate. Of course, you have to deal with some setup issues if your loop count doesn't divide evenly by your unroll count.

I got another large speedup in our test code by unrolling the loop in Listing 1 on my way to Listing 2, and this speedup was particularly surprising because it's a no-brainer. Alias analysis and code motion are hard. Unrolling a loop is basically a cut-and-paste operation.

Listing 2. The Optimized Code

```
oid TransformVectors5( float  *pDestVectors,
const float  (*pMatrix)[3],
const float  *pSourceVectors, int NumberOfVectors )
{
    int Counter;
    float Value;
    float _Krr1;
    float _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;
    }
}
```

The biggest thing to watch out for when unrolling a loop, besides the setup overhead mentioned above, is code bloat. You can actually make your unrolled code slower by causing it to be so big that it doesn't fit in the cache or kicks other important code or data out of the cache. Jumps aren't as expensive as they used to be, so amortizing the loop overhead isn't a big win since there's less overhead to amortize. Jumps on the Pentium, for example, are only a half cycle under the right circumstances. Cache misses are a lot more than a half cycle.

The gcc compiler supplies a command-line switch to force unrolling, so I used it in the row labeled "GNU gcc unroll." As you can see, it's not always faster than the gcc without unrolling.

Final Output

If you want to learn more about compiler technology, the bible is *Compilers: Principles, Techniques and Tools*, by Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman (Addison-Wesley 1986), affectionately called "The Dragon Book" because the cover illustration is a dragon bearing the words "Complexity of Compiler Design" being trounced by a knight with a sword that says, "LALR Parser Generator." I agree its goofy, but it's a classic.

You should also spend time on the web; there's tons of compiler information out there. For example, http://www.nullstone.com/htmls/category.htm has a series of 40 understandable examples of program transformations. For some aggressive compiler optimizations on a supercomputer compiler, check out http://www.astro.ku.dk/~aake/optimize/options.html. It covers optimizations that don't even preserve the original meaning of the code, which we didn't get into.

I would be remiss if I didn't mention that my friend Mike Phillip of Motorola actually managed to equal my final optimization using only compiler switches and Listing 1; you'll remember I mentioned Mike at the end of the last article. He ran the code through KAP *twice* with the no alias switch and then through the Motorola PowerPC compiler. It just goes to show you that knowledge of how something works (in this case Mike's knowledge of how the compiler worked) is always a good thing. ■

Don't believe the hype—Chris Hecker can use all the help he can get at checker@bix.com or gdmag@mfi.com.