Using Performance Analysis Tools

A Metrowerks White Paper

By Rick Grehan

Copyright © 1999 Metrowerks, Inc. All Rights Reserved.

Using Performance Analysis Tools

Is your program "soup" yet?

Eliminating bugs is a critical step in the development process, for sure. But when that last bug goes belly up, there is still work to be done.

by Rick Grehan

When a program has been written and tested, is the programmer's job finished? Is verification of correct execution sufficient to conclude work or a given program?

Real-life examples familiar to any programmer will tell you that the answer is "no." For instance, how many times have you needed a sort routine, and reached for the quick-and-dirty bubble-sort? It works, but at what performance cost? And how many times has a second look at your code revealed that a cleverly-rewritten loop or re-arranged conditional statement will boost a frequently-called routine, thereby benefiting the entire application?

The former example (the sorting routine) is a case of a gross inefficiency. Your best weapon against gross inefficiencies is your own wits: choose good algorithms, code defensively, etc. Tackling a situation as illustrated in the latter example, however, sometimes requires more resources than wits can provide. If you need to find those hard-to-find inefficiencies, you have to get under the code's hood. You need to be able to locate the performance bottleneck – usually a single routine or basic block of code that's holding up everything else. Then, you need to be able to "zoom in" on the bottleneck and explore the code's plumbing to find the one or two lines (sometime, a single instruction) that are eating processor time.

In this article, we're going to examine this oft-encountered task of performance-boosting with the help of a case study – a real-world application. This application is a relatively straightforward "compression" program that uses a specific compression algorithm called the "Huffman" algorithm (named after its inventor, D.A. Huffman of M.I.T.) The program works by reading through the data that's targeted for compression, and – using frequency information gathered from the data -- building a special data structure called a Huffman tree. Guided by the Huffman tree, the algorithm compresses the input data. The

resulting compressed data can be decompressed using the same Huffman tree (we'll go into the details later).

We've written the program so that it compresses and decompresses about 10K of data. So that we can get an idea of the runtime of the algorithm, we've written the program so that it compresses and decompresses the data 4 times for a single iteration. The program runs a number of these iterations inside of a timed loop. At the end of 10 seconds of execution, we can determine the algorithm's iterations per second.

On a 150 MHz Pentium system, the first version of the application performed about 16 iterations per second.

We added debugging code to verify that the program was indeed creating a Huffman tree, compressing the data, and properly decompressing the result. The program runs correctly, but are there areas of improvement? If so, how do we find them?

Notice that the question of performance leads into an area where debuggers are of little – if any -- use. If a program works, then in one sense it isn't broken (unless the system is a real-time system, and execution within a specified window is mandatory). Debuggers can find dangling pointers, array bound violations, loops with erroneous bounds, and so on. But a debugger simply isn't equipped to help a programmer fine-tune an already functioning application.

The Right Tool

What we need is a performance analysis tool that will enable us to zero in on those spots in the code that are taking the most time. That way, rather than having to scan through the whole program, we can focus our attention on the trouble sites. Time spent scrutinizing such hot-spots is far more rewarding than perusing the entire body of source code, which can waste time analyzing code that does not contain performance bottlenecks in the first place.

Now, we could do our own performance analysis. Indeed, since the program calculates iterations per second, we have a degree of performance analysis in place already. We could use the same functions that provide that elapsed time calculation to bracket suspected blocks of code and print out the elapsed time with a `printf()` statement. (This is known as time-stamping the code.)

However, it's unlikely that we will live long enough to use such a "do-it-yourself" technique effectively. It would involve our adding time-gathering and data output routines throughout the code. Then we'd have to spend more time pouring over the

numbers, trying to make sense out of them.(Editing source-code like that is always chancey. Try adding a single line after an `if()` statement and then forget to put that line inside the bracket and see what happens. And when you get everything working, you have to go *back* through the code and delete it all. Or perhaps you can bracket all the timing routines in `#ifdef` statements and write a Perl program to surgically remove it when you want to ship the source. Do you really want to do all that?)

Happily, there are already performance analysis tools on the market that can do most of the dirty work for us. We don't have to invent this wheel – we just have to figure out which is the best wheel to buy.

Peformance Analysis Tools

When you're determining which performance analysis tool to acquire, the fact to keep at the head of your list is that such tools use different techniques to gather analysis data. Each technique has advantages and disadvantages – which we'll explain in a moment. It's important to know the techniques and their advantages so you can make an informed buying decision.

Some tools use what is referred to as "PC (program-counter) sampling". The tool installs a timer-triggered interrupt that "wakes up" at regular intervals while the target program is executing and samples the state of the program counter. The idea is that, if enough samples are taken over a long enough period of time, and if the sampling interval is small enough, the result is a reasonably good picture of the distribution of execution time within the application.

What, precisely, makes for a "small enough" sampling interval is not easily determined. Tools that use PC sampling often recommend that you try a range of intervals; not too coarse else the resulting data be simply useless, and not too fine else the program's behavior be altered by the frequent interruptions by the timing routine.

Other analysis tools work their magic by "instrumenting" the code. That is, they modify the code in ways that are similar to our adding the time-gathering and printf() statements as described in the timestamping technique above. Precisely how the code is instrumented differs from tool to tool. Some actually insert source code during the compilation process. The modified source is compiled and linked with specific libraries Other tools work their magic using a technique known as "binary code instrumentation" (BCI), in which they actually modify the final executable code.

Precisely how BCI works is beyond the scope of this paper. In a nutshell, a tool that uses BCI translates the executable code to an intermediate representation known as "register

transfer language" (RTL). RTL is a kind of combination data structure and pseudo-assembly language, a form of which many compilers use while translating a high-level language to the target machine code. The BCI tool then instruments the RTL representation of the program, and translates that result back into an executable form.

There are several advantages that BCI has over source-code modification. First, since the tool does not modify source code, it doesn't require the developer to have source code available. That means that third-party libraries (whose source code the user may not have on hand) can be instrumented as well as the user's application. Nevertheless, BCI is at its best if you have source and symbolic code available; that allows the profiling tool to display times associated with functions, basic blocks, and even individual lines in the source code file.

Second, BCI can instrument code to provide either timestamping or – since BCI operates on the executable – cycle-counting. Cycle-counting means that the tool determines the cycle time for each machine instruction (usually, by referring to an internal database) The tool can then run the program and, by counting the number of times each instruction is "visited", calculate the total time each instruction contributes to the overall runtime of the application. This allows the tool to provide aggregate figures for loops, basic blocks, functions, and so on.

Cycle counting's downside is that the tool's database can only include "static" cycle counts as provided by the processor manufacturer's specifications. The true cycle count that an instruction takes can vary. For example, an instruction that includes a fetch through an index register will execute in a different number of cycles depending on whether the data address is aligned to the data item being read, whether the data is in cache, etc. Such information is not available to the instrumentation tool.

Cycle counting's upside is that – unlike PC sampling and timestamping – it is immune to the effects that other applications and tasks within the operating system might have on the runtime of the application under scrutiny. It also offers very accurate resolution; you can examine the time taken by individual lines of code.

A good BCI tool will provide a combination of cycle counting and timestamping, allowing you to apply the strengths of each to cancel the other's weaknesses. Having both techniques available gives you a clearer picture of your application's performance.

Such a tool exists: the CodeWarrior Hierarchical Profiler, a member of the CodeWarrior Analysis Tools suite from Metrowerks, Inc. The CodeWarrior Hierarchical Profiler provides performance analysis for C/C++ applications running on a Win32 platform (Windows NT, 95, or 98), and is compatible with Metrowerks own CodeWarrior Professional development environment (release 4), as well as Microsoft's Visual C++ 2.x

or later and Microsoft's Developer Studio 4.x or later. The CodeWarrior Hierarchical Profiler uses BCI technology to provide timestamping and cycle-counting.

(Author's note: The CodeWarrior Analysis Tools also provide code coverage analysis, useful in the testing phase of application development. Code coverage is not the topic of this paper, however, so we won't describe it here.)

We'll use the CodeWarrior Hierarchical Profiler to examine our Huffman compression/decompression application, and to guide us through improving its performance. First, we'll investigate the program more closely; understanding how the algorithm works will be important as we fine-tune its performance.

Get With The Program

The Huffman compression algorithm is a character-based compression scheme. That is, the algorithm reads in a character, compresses it, outputs the result, reads the next character, compresses it, and so on. (Other compression schemes, such as LZW, compress entire strings at a time.) The Huffman algorithm performs its compression by replacing each character with a unique bit string whose length is determined by the character's frequency of occurrence. High-frequency characters are replaced by short bit strings; low-frequency characters are replaced by long bit strings.

It works like this: You pass the algorithm a buffer holding the uncompressed data (what we'll call the "plaintext"). The routine passes through the plaintext, counting the occurrence of each byte in the data, creating a 256-entry frequency table. (Each entry in the table corresponds to a byte value.) Guided by the information in this table, the algorithm builds a kind of binary tree data structure called a Huffman tree. The leaf nodes of the tree consist of the original 256-entry table. Each node in the tree (with the exception of the leaf nodes) has one parent and two children. One edge from parent to child is marked "0", the other is marked "1" (representing binary digits).

Once the tree is built, the compression algorithm is fairly straightforward. The routine reads an input character from the plaintext, and locates that character's corresponding node at the tree's root. Next, the compression routine travels "up" the tree, from leaf to root, keeping track of each edge (whether it is a "0" edge or a "1" edge) as it moves from node to node. In so doing, the routine builds a bit string that is the Huffman code for the plaintext character. (The bit string is backward, since the routine travels from leaf to root, but reversing the string is easily taken care of in software.)

Uncompressing the data amounts to following edges through the tree in reverse. The

algorithm reads bits in sequence from the compressed data and, starting at the root, follows a trail of “0” and “1” edges (depending on the bit value that’s been read) to a leaf node, which corresponds to the uncompressed character. The algorithm outputs the corresponding character, returns to the root, extracts the next bit from the compressed data, and continues.

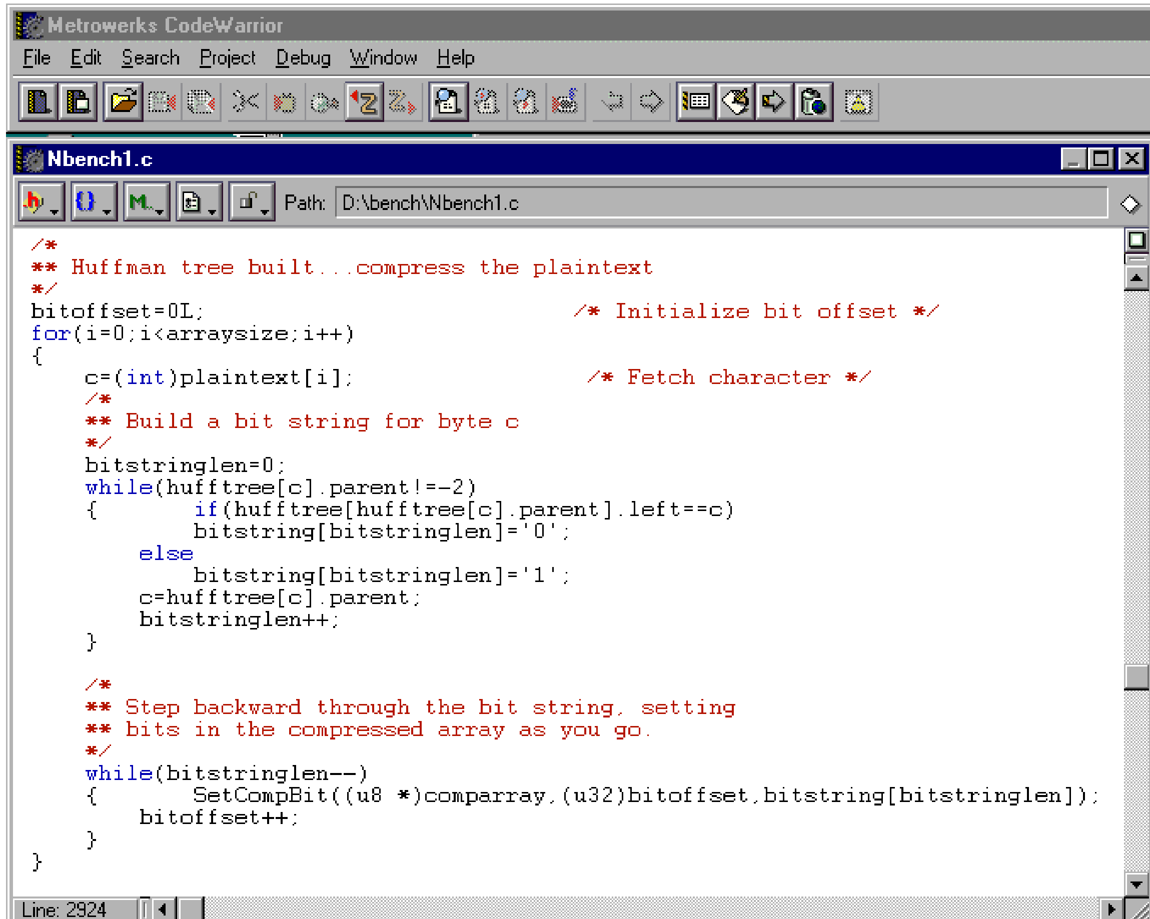

Figure 1. The Huffman compression algorithm. At this point, the Huffman tree has already been built. The algorithm reads the uncompressed data (in the plaintext[] array) and compresses it into the buffer pointed to by comparray.

The portion of the algorithm that performs the actual compression is shown in Figure 1. The code is composed of two `while()` loops within a single `for()` loop. The outer `for()` loop steps through each character of the plaintext buffer. The first `while()` loop reads “up” the tree, building the bit string that represents the Huffman code (reversed) for the character. The next `while()` loop steps backward through that bit string, setting the appropriate bits in the array that holds the compressed data (`comparray`) with the help of the `SetCompBit()` function.

Since compression is the first function of the program, we'll focus our attention on that part of the code for our first run. First, however, we need to create a project within the CodeWarrior Hierarchical Profiler.

This is the easy part. The only prerequisite is that we have a compiled executable on hand, and – for best results – the source and symbolic files. (Since we created our executable using the CodeWarrior Pro 4 C/C++ IDE, creating the symbolic information needed was a simple matter of re-building the executable with debugging on.) The CodeWarrior Hierarchical Profiler provides a setup wizard that hand-holds you through the process of creating a project. It prompts you for the destination directory (where the profiling data files will be stored), prompts for the initial instrumentation technique, lets you enter any command-line arguments that should be passed to the application at runtime, and even allows you to select which modules within the application are instrumented. (The default is all modules.) Figure 2 shows a screen shot of the CodeWarrior Hierarchical Profiler setup wizard in action.

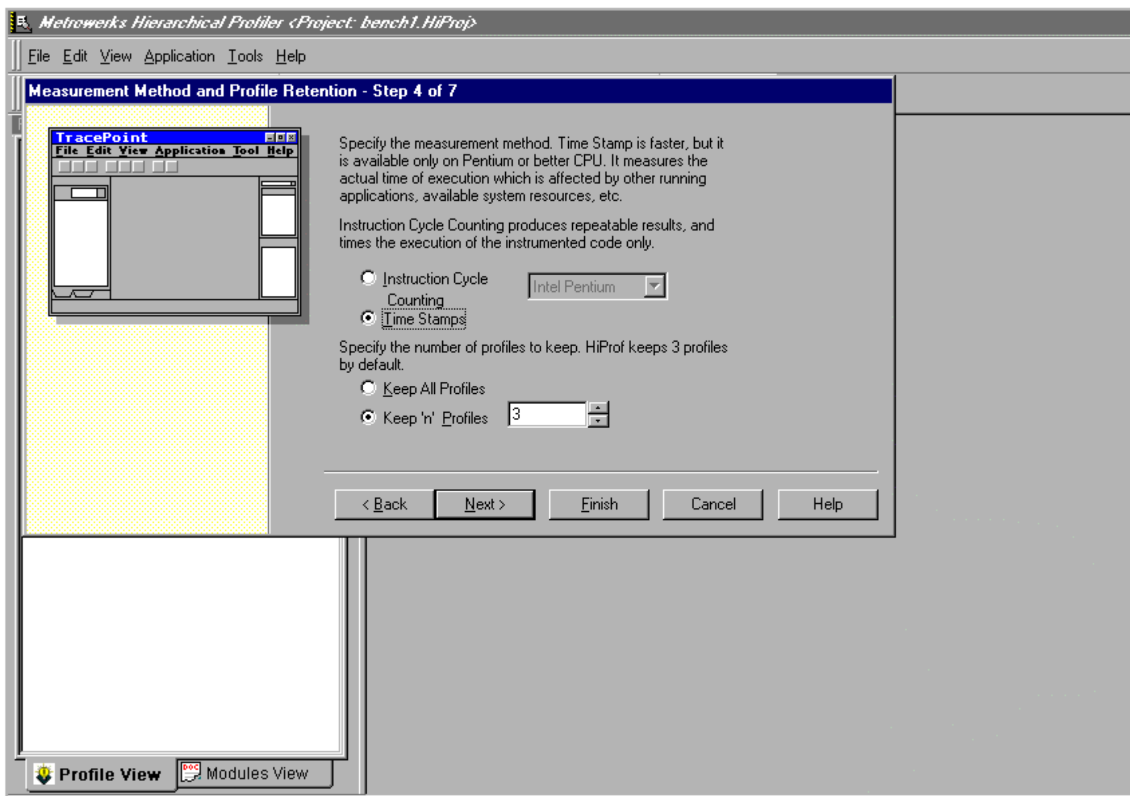

Figure 2. The CodeWarrior Hierarchical Profiler's setup wizard in action.

With the project built, instrumenting and running the application is a simple selection from a menu. Because our application had not yet been instrumented, we selected

"Instrument & Run". The CodeWarrior Hierarchical Profiler builds a separate, instrumented executable, then launches it. As the application runs, you can watch its activity in a stack crawl window provided for each thread in the application. Figure 3 shows the stack crawl window for the Huffman application. (While our program is single-threaded, the CodeWarrior Hierarchical Profiler handles multithreaded applications as well.)

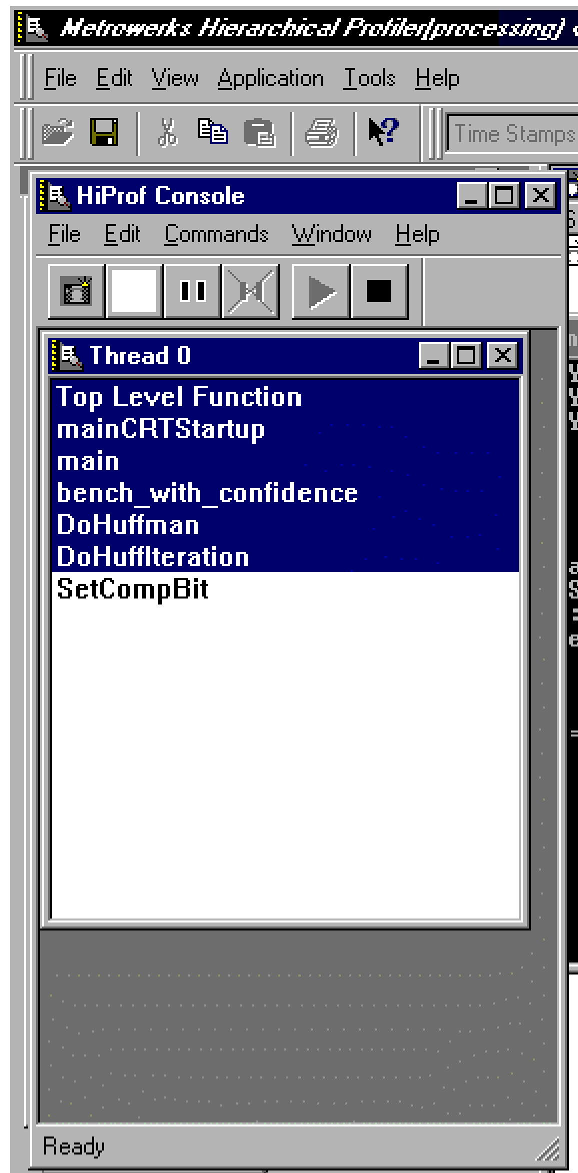

Figure 3. Running the application in the CodeWarrior Hierarchical Profiler. The Profiler tracks the execution in a stack crawl window. You can see the call chain of routines within each thread of execution.

The results of our first run using timestamps is shown in Figure 4. This is the “modules” view of the collected data. The leftmost window shows the various source-code and library modules in the program. In the upper right, the function view shows – among other things – the hierarchical time taken by each function in the selected module. For example, we can see that the `DoHuffIteration()` function consumes 12.11s of the total 12.14s taken by the `DoHuffman()` function. “Hierarchical time” means that the time shown is only the amount of time within the function hierarchy that we’re viewing. In other words, the time shown for `DoHuffIteration()` is only the amount of time the function consumes where called from `DoHuffman()`. Any time taken by `DoHuffIteration()` as called from other functions is not shown in this view (even though, in this particular application, no other functions call `DoHuffIteration()`).

Focusing in on the hierarchical view of the `DoHuffIteration()` function (the middle window showing the two pie charts) we turn our attention to the lower pie chart. This shows the amount of time taken by the inline code of `DoHuffIteration()`, as well as the time taken by its children routines. There are 4 child routines involved: `SetCompBit()`, `GetCompBit()`, `StopStopWatch()`, and `StartStopWatch()`. We’re not interested in the “stopwatch” routines, they are part of the code that calculates the iterations per second that the program reports. Also, as can be seen from the pie-chart, they have little effect on the execution of `DoHuffIteration()`.

What we are concerned with, however, is how much time the `SetCompBit()` and `GetCompBit()` routines are taking. The hierarchical window shows us that, while the inline code of `DoHuffIteration()` takes 4.57 seconds, `SetCompBit()` takes 3.86 seconds, and `GetCompBit()` takes 3.68 seconds. The `SetCompBit()` and `GetCompBit()` routines involve little computation; the former sets bits in the compressed array, the latter reads bits; one bit per call, in each case. Yet, each consumes nearly one-third of the overall runtime of the `DoHuffIteration()` function. Such simple routines shouldn’t take so much time.

We need to focus in on the execution of the `SetCompBit()` and `GetCompBit()` routines, so we can see what parts of them are taking the time. To do this, we need re-instrument the code so that the profiler will capture instruction-cycle counting, then re-run the application and focus in on the suspect routines.

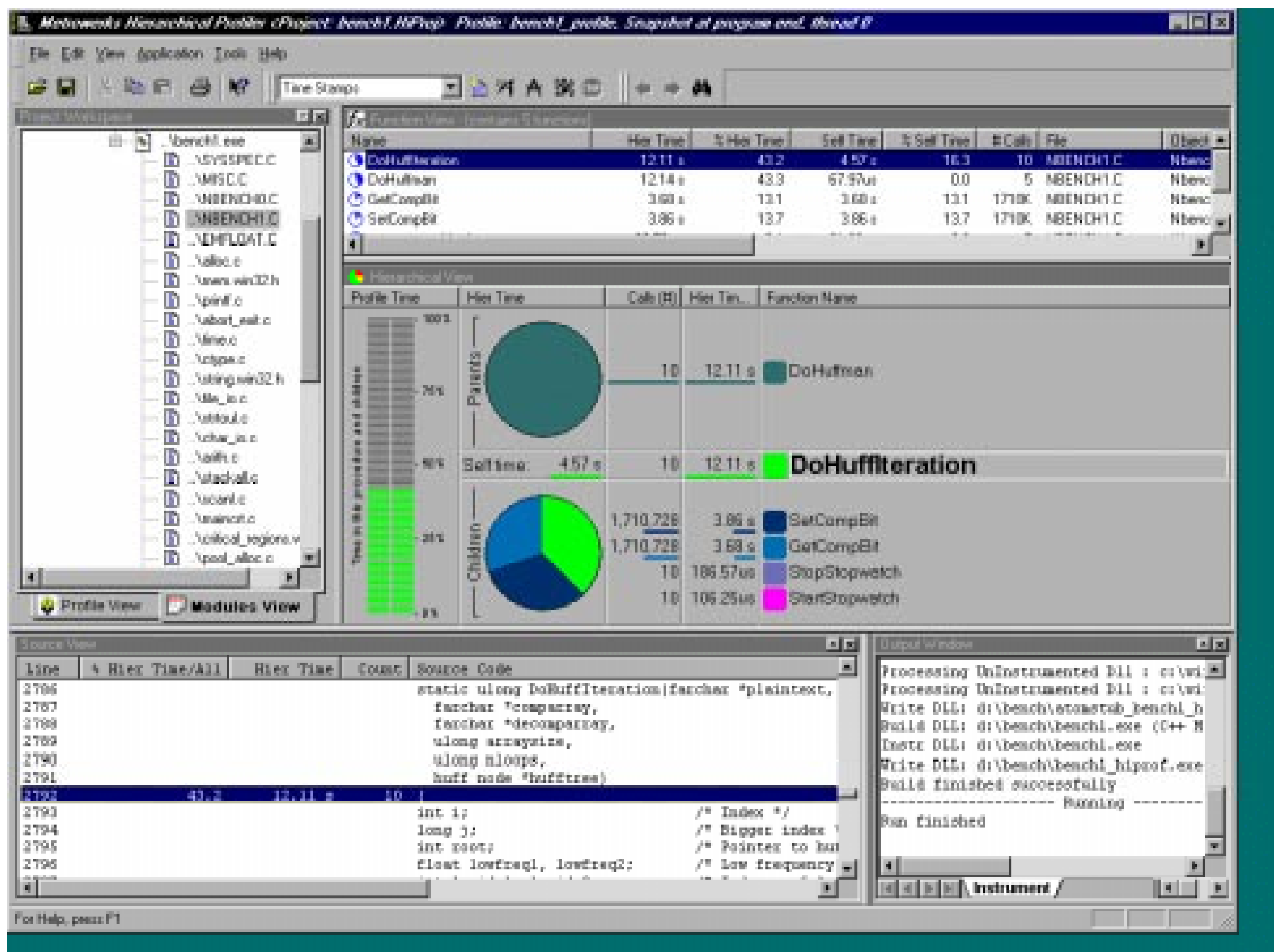

Figure 4. The CodeWarrior Hierarchical Profiling output display for the timestamped Huffman algorithm. The `DoHuffIteration()` routine builds the Huffman tree, compresses and decompresses the data four times (counted as a single iteration of the test). Note that large portions of time within the `DoHuffIteration()` routine are consumed by two child routines – `SetCompBit()` and `GetCompBit()`.

Though we set the instrumentation technique to timestamping when we built the project, CodeWarrior's Hierarchical Profiler lets us modify the project parameters through a project settings dialog. It's an easy matter to change the timing method from timestamping to instruction cycle counting, re-instrument, and re-run the code. (We don't have to recompile the application to do this.)

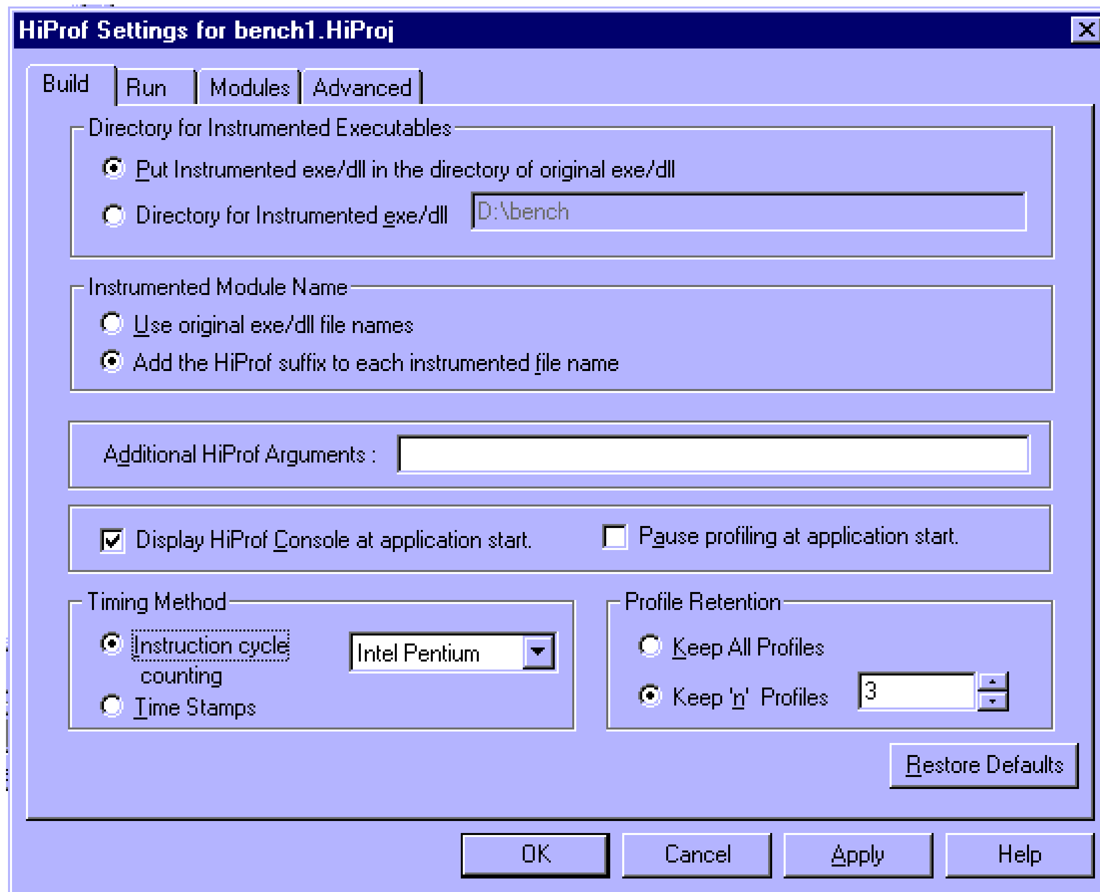

Figure 5. The project settings dialog box within the CodeWarrior Hierarchical Profiler. Note that we can select instruction cycle counting or time stamping data gathering techniques.

From the project settings dialog (shown in Figure 5, above) we select instruction cycle counting in the Timing Method box. An adjacent drop-down pick-list allows us to select the Intel Pentium as our target processor. (The other choice is Intel 486. The Hierarchical Profiler is expected to provide additional processor support in the near future.)

With the new timing method set, it's a simple matter to re-instrument the application and re-run it; we use the same menu selection we used to the first time we ran the application. (Re-instrumenting takes about 5 seconds for a 255K executable on our 150-MHz Pentium system).

After we re-run the application, we move to the source view window in the CodeWarrior Hierarchical Profiler as shown in Figure 6. (The source window was also in Figure 4, in the lower left hand corner. For the sake of simplicity, we've shown only the source window in Figure 6.)

Source View

Line	% Hier Time/All	Hier Time	Count	Source Code
2976				char bitchar)
2977	0.0	22.81ms	855364	{
2978				u32 byteoffset;
2979				int bitnumb;
2980				
2981				/*
2982				** First calculate which element in the co
2983				** alter. and the bitnumber.
2984				*/
2985	0.0	11.40ms	855364	byteoffset=bitoffset>>3;
2986	0.2	250.91ms	855364	bitnumb=bitoffset % 8;
2987				
2988				/*
2989				** Set or clear
2990				*/
2991	0.0	17.11ms	855364	if(bitchar=='1')
2992	0.0	27.87ms	464540	comparray[byteoffset]|=(1<<bitnumb);
2993				else
2994	0.0	23.45ms	390824	comparray[byteoffset]&=~(1<<bitnumb);
2995				
2996	0.0	28.51ms	855364	return;
2997				}

For Help, press F1

Figure 6. The source view of the SetCompBit() routine. With cycle counting enabled, we can see the execution times of individual statements in the source code. Note the comparatively large amount of time taken by the statement in line 2986.

As you can see in Figure 6, we have scrolled down to view the `SetCompBit()` function. We instantly see that a single line -- line 2986 -- takes more time than all other lines combined. This is obviously a line that deserves our attention. Is there anything we can do to reduce its execution? To answer that, we have to figure out what that line does in the routine.

Recall that the job of `SetCompBit()` is to set the value of a bit in the output (compressed) array, an array of 8-bit characters (bytes). One of the incoming arguments to `SetCompBit()` is `bitoffset`, which contains the address of the bit in the output array. Obviously, lines 2985 and 2986 work together to resolve the bit address in the array to a byte address (which is calculated in line 2985 and stored in `byteoffset`) and a bit address (`bitnumb`) within the target byte. The `SetCompBit()` routine calculates `byteoffset` by a right-shifting `bitoffset` by 3 bits, effectively dividing it by 8 to derive the byte offset. Similarly, `SetCompBit()` calculates `bitnumb` by taking the remainder of `bitoffset` divided by 8 (using the modulus operator). Obviously, line 2986 takes so long because it uses a division instruction. Division instructions are notoriously slow as compared to other processor instructions.

But, wait. The modulus operation performed in line 2986 has a base of 8 – a power of two. We can calculate the remainder of `bitoffset` divided by 8 just as easily using a

simple bitwise AND operation. Replacing line 2986 with:

```
bitnumb=bitoffset & 7;
```

will yield the same answer, as the current line 2986, but without resorting to division. A bitwise AND operation is an extremely fast processor instruction.

Armed with this new insight, we return to the source code and make the change. We recompile, re-instrument (still using instruction counting) and re-run the application from within the CodeWarrior Hierarchical Profiler (see Figure 7). As before, we turn our attention to the source window for the SetCompBit() routine.

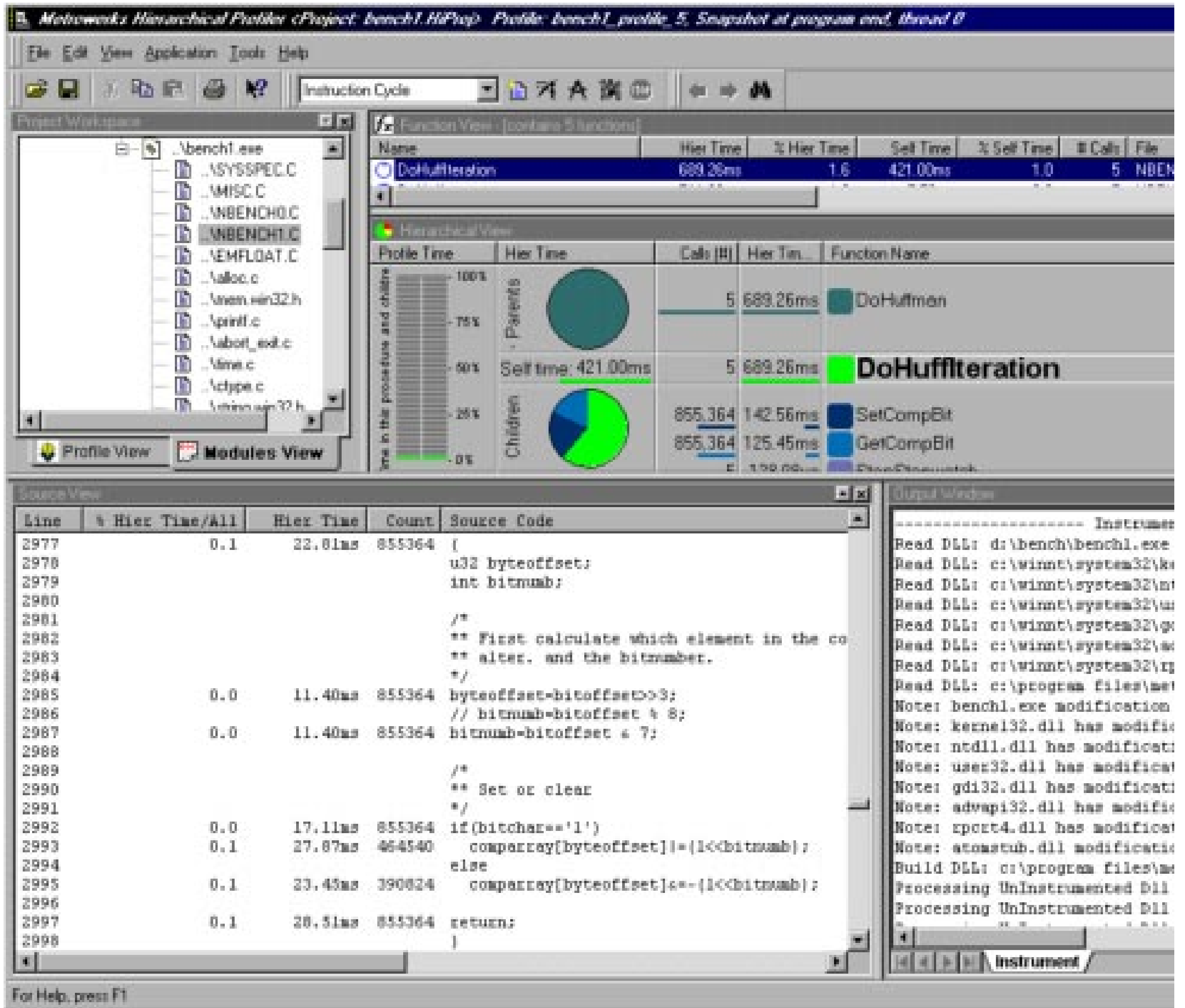

Figure 7. Re-execution of the code after fixing a single line of code. We can see that the amount of time taken by the new statement is down to 11.40ms from 250.91ms (in Figure 6). Also, notice that the pie chart in the hierarchical view shows that the SetCompBit() and GetCompBit() routines take less of the entire execution time of DoHuffIteration().

We commented out the old line; its replacement is line 2987. The difference in execution is significant – we've reduced an execution time of over 250ms to an execution time of just above 11ms (reduced by a factor of 20). Our analysis of the situation appears to have been correct. (Note: Not shown was the fact that a corresponding remainder operation was executed in the `GetCompBit()` function. We replaced that line with an equivalent bitwise `AND` operation. We didn't show all that because it was simply a repeat of what we did to the `SetCompBit()` function.)

If you look in the hierarchical view window, you'll see that the relative portions of the time taken by the `SetCompBit()` and `GetCompBit()` functions has been noticeably reduced. Compare the lower pie-chart in the hierarchical view window with its corresponding chart in Figure 4. (Note: You might be wondering why the times shown in figure 7 are so much smaller than the original times shown in figure 4. It appears that the entire execution time of the application has been remarkably reduced. However, the timings shown in figure 4 were gathered with code that was instrumented for timestamping, while the timings shown in Figure 7 were gathered with code instrumented for instruction cycle counting. Recall that timestamping gathers information as the program runs in real-time; cycle counting, on the other hand, estimates instruction cycles of machine language executed for each line of code. Consequently, we should expect the recorded times to be different depending on the style of instrumentation.)

To verify that the changes have indeed improved the execution of the code, we re-run the modified (non-instrumented) executable and discover that our Huffman routine now executes 24.16 iterations per second – an improvement of 51 % -- and all from changing a single operator used in 2 lines of code!

But Wait, There's More

Anxious to continue our improvement of our application, we return to one of the earlier profiles. This allows us to recall performance information that's been gethered on earlier runs.

If you look at Figure 8, you'll see that we've selected `SetCompBit()` in the functions view window. The hierarchical view window tracks our selection, showing information for the `SetCompBit()` routine. You can see that `SetCompBit()` is not only a "leaf" routine (that is, it calls no other routines), but it is also called from one location only. `SetCompBit()` has only one parent – `DoHuffIteration()`. (We know that `SetCompBit()` has no children because the lower pie-chart is solid:

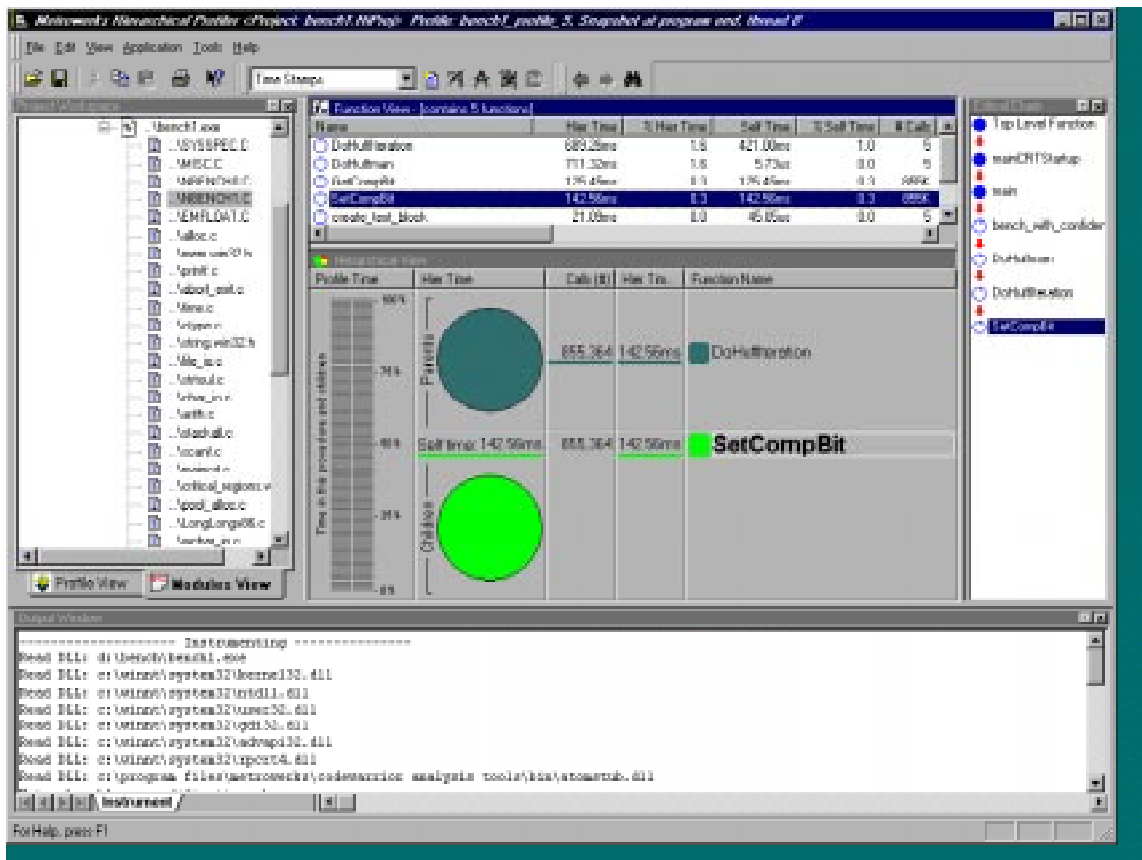

Figure 8. The hierarchical view “inside” `SetCompBit()`. `SetCompBit()` is called by a single parent function – `DoHuffIteration()` – and calls no children functions. Hence, `SetCompBit()` is an ideal candidate for inlining.

`SetCompBit()`’s time is all “self” time. We know that `SetCompBit()` has only one parent because the upper pie chart is also solid. All `SetCompBit()`’s time is attributed to calls from `DoHuffIteration()`.)

Consequently, `SetCompBit()` is an excellent candidate for inlining. Moving the function of `SetCompBit()` into `DoHuffIteration()` will not only save execution time on account of the removal of the “call” and “return” and all related stack management instructions, it will give us an opportunity for further execution improvement by buffering some information that couldn’t be buffered because of the decoupling between `SetCompBit()` and `DoHuffIteration()`.

Here’s what we mean by that last statement: If you look back at Figure 6, you can see that `SetCompBit()` modifies each byte in the output array by reading the byte, setting the appropriate bit, and writing the modified byte back out to memory. This read-modify-write cycle must take place for each bit written into the output array. If we look back at Figure 6, we can see that `SetCompBit()` is called approximately 855,364 times – that’s 855,364 read-modify-write operations.

However, if we inline the activity of `SetCompBit()`, we can improve performance by writing the byte out only it must be written out. That is, after all its bits have been modified. Furthermore, since writing the compressed data involves overwriting the entire output array, we can eliminate reading bytes altogether. (This is not a read-modify-write operation, it's simply a write operation.)

Metrowerks CodeWarrior

File Edit Search Project Debug Window Help

Nbench1.c

Path: D:\bench\Nbench1.c

```
/*
** Huffman tree built...compress the plaintext
*/
bitoffset=byteoffset=0L;                    /* Initialize offsets */
tchar=(u8)0;                                /* Clear temp byte */
for(i=0;i<arraysize;i++)
{
    c=(int)plaintext[i];                    /* Fetch character */
    /*
    ** Build a bit string for byte c
    */
    bitstringlen=0;
    while(hufftree[c].parent!=-2)
    {       if(hufftree[hufftree[c].parent].left==c)
                bitstring[bitstringlen]='0';
        else
                bitstring[bitstringlen]='1';
        c=hufftree[c].parent;
        bitstringlen++;
    }

    /*
    ** Step backward through the bit string, setting
    ** bits in the compressed array as you go.
    */
    while(bitstringlen--)
    {   if(bitstring[bitstringlen]=='1')
            tchar|=(1<<bitoffset);
        bitoffset++;

        if(bitoffset==8)          // Past end of byte?
        {   comparray[byteoffset]=(char)tchar;
            bitoffset=0;
            byteoffset++;
            tchar=0;
        }
    }
}
// Write out last compressed byte just in case we didn't end
//  on a byte boundary
comparray[byteoffset]=tchar;
```

Line: 512

Figure 9. The modified compression algorithm with the functionality of SetCompBit() merged into DoHuffIteration (compare with Figure 1, above).

The modified portion of the Huffman algorithm is shown in Figure 9. Note that we've added a couple of variables, and changed the meaning of a third. We added the

`byteoffset` variable, which tracks which byte in the output array the algorithm is currently processing. We've also added a variable called `tchar`, which is a temporary holding variable that holds the value of the output byte currently being processed. The `bitoffset` variable, which before held the address of the bit in the output array, now holds the address of the bit within the current byte; that is, he bit within the byte addressed by `byteoffset`.

This temporary variable allows us to eliminate redundant write operations. The output byte is "built" in `tchar`, and written into the array only when necessary (when the `bitoffset` variable's value equals 8). Also notice that the temporary variable is initialized to zero. This allows us to eliminate half of an `if()` statement that existed in the original `SetCompBit()`. Only "1" bits need be set in the output byte.

Recompiling the application and re-running it yields profound results. The number of iterations per second reported jumps to 50.16. We have tripled the performance of our application. And, given that the application both compresses and decompresses the test data, we can improve overall execution even more by inlining the counterpart to `SetCompBit()`, `GetCompBit()`. The resulting modification is so similar to what we've shown above, that we won't cover it here.

Conclusion

In this paper, we've seen how a performance analysis tool can help us "hot-rod" an application. In particular, we've seen how CodeWarrior's Hierarchical Profiler enabled us to zero in on problem routines; single, problem lines, in fact. We've seen how easy it is to make a modification, recompile, and know whether the modification was successful or not. The Hierarchical Profiler's ability to switch between timestamping and instruction cycle counting made this process even easier.

Since the profiler was hierarchical, we could easily identify "parent and child" relationships among routines (who is calling whom); that information lets us identify routines called exclusively from a single site, and which could be inlined for substantial performance gains.

Our conclusion is that a good performance analysis tool should be in every programmer's toolbelt. The decisive word, here, is "good." You should choose a tool that provides more than one data-gathering technique. Sometimes you'll want to use timestamping, and sometimes you'll want to use cycle counting. Equally important is the tool's data display capabilities. The CodeWarrior Hierarchical Profiler's data display provided a module view, a function view, a hierarchical view (of parent and child function data), and even a source-code view.