

**Code Profilers:
Choosing a Tool for
Analyzing Performance**

A Metrowerks White Paper

By Rick Grehan

Code Profilers: Choosing a Tool for Analyzing Performance

A profiler is a development tool that lets you look inside your application to see how each component – each routine, each block, sometimes each line and even each instruction – performs. You can find and correct your application's bottlenecks.

How do they work this magic?

By Rick Grehan

A SOFTWARE APPLICATION usually progresses through a series of phases as it goes from concept to reality. Those phases -- analysis, design, implementation, and test -- have been (and continue to be) well-explored by hosts of software methodologists. However, you can reveal an omission to this list of phases if you translate them from their formal names into more casual terms:

- Decide what the program is going to do.
- Decide how the program is going to do it.
- Make the program do it.
- Verify that you made the program do what it was supposed to do.

The missing ingredient is one of degree, and occurs (or, should occur) at the last step: You've verified that the program does its job...but have you verified that it does it well? Is the program efficient? Have you verified that its execution speed cannot be improved? What tool could you use to answer these questions?

The answer to this last question -- and, therefore, that which will allow you to answer the other questions -- is a profiler.

The name "profiler" gives its purpose away: a profiler creates a "profile" of an application's execution. And, to explain what we mean by that, consider yourself to be in the situation we described at the outset. You've written an application, it works, and

now you want to improve the application's performance, to make it run faster. How do you do that?

Well, you could create a large amount of test data, sit down in front of a computer, grab a stopwatch, feed the data to the application, and time how long it takes to execute. This gets you partway to your answer, but there's a problem with this approach.

A program is not a monolithic entity; it is composed of routines that call other routines that call other routines, and so on in a complex web of inter-dependent functions. Also, if you've used statically-linked or dynamically-linked libraries in your application, then you've added more layers of functional inter-dependency that involves code for which you don't even have the source.

So, the problem is this: If your application is running slowly, what parts of it are running slowly? For the outside, it looks like the whole thing is running slowly. But, it could be that a single, fundamental algorithm used repeatedly throughout your application is the only bottleneck. That algorithm may constitute a small fraction of your source code, so that searching other parts of your application for opportunities of improvement is, literally, a waste of time. You need to focus on that algorithm exclusively.

To do that, you need a profile, a map of your application's performance; so that you can see the individual parts' contributions to the application's overall execution.

That's the job of a profiling tool. It lets you examine the execution behavior of your application in such a way that you can determine where your application is spending its time. Most profiling tools actually provide graphical output that lets you quickly identify performance "hotspots" so you don't waste time wading through source code that's not a problem in the first place.

In this white paper, we're going to examine how profilers do what they do. As you'll see, how a profiler works determines that profiler's capabilities. Not all profiling technologies are equal and, therefore, not all profilers are equal.

We'll begin by defining and describing the two broad categories into which profilers fall: passive profilers and active profilers. In our descriptions, we'll examine the characteristics -- positive and negative -- inherent in each category.

Passive Profilers

A passive profiler gathers execution information about an application without

modifying that application. Passive profilers are passive in that they "stand outside" the application and watch its performance from a distance, sort of like a coach standing on the sidelines and measuring the performance of his running athletes with a stopwatch.

This is in contrast to active profilers, which modify (or "instrument") the application. While a passive profiler works from the outside (of the application), an active profiler works from the inside. (We'll describe active profilers in more detail later.)

How It's Done - PC Sampling

Virtually all passive profilers gather data using a technique called "PC sampling." Don't be thrown off by the acronym "PC", which has several meanings. In this case, it means "program counter": the register inside the CPU that keeps track of a program's current execution location.

A passive profiler will typically install an interrupt service routine (ISR) that is triggered by a timer interrupt. (The ISR is installed prior to running the application.) As the application executes, the timer interrupt triggers the ISR, which wakes up, records the location of the program counter, then goes back to sleep, at which time the application resumes execution.

(Note: Most modern computer systems and their accompanying operating systems provide timer interrupts that are easy to "hook into". Creating a timer-interrupt-driven ISR is a straightforward task.)

Over a "long enough" period of time (that is, if the application runs long enough) the profiler will collect enough samples to paint a reasonably accurate picture of where the application is spending its time. Provided that the profiling tool has access to the application's symbolic information, the tool can -- for each sample -- deduce what routine is being executed. In fact, if enough samples are taken, and the locations of those samples are well-distributed throughout the application, the profiling tool can even generate execution information for individual instructions in the application.

More samples will be taken in code that consumes more CPU time; fewer samples will be taken in code that consumes less CPU time. This is the essence of PC sampling.

It's Statistical

PC sampling is inherently a statistical technique. The information it reports is an approximation, and the accuracy of that approximation depends on such factors as:

— **How many samples are taken.** Taking more samples will generate a more accurate execution profile. Unfortunately, there are no clear-cut rules for determining

how many samples are enough. Figuring that out is typically a matter of successive approximation: Take a passel of samples, record the results, then rerun the application, take twice as many samples as before, and see if the results from the two runs vary widely. If not, you're probably taking enough samples. If so, repeat (adding more samples to each run) until the differences between successive runs falls below a reasonable threshold (usually, around 5%). (Of course, taking more samples can mean two things: increasing the sampling rate, or running the application longer. You'll have to decide, based on your application, which is the course to take in a given situation.)

_ **The sample rate.** Again, this is not an easy thing to determine. The profiler might perturb the application's real behavior. Sample at too high a frequency and the system spends more time in the sampling ISR than in the application. Sample at too low a frequency and the data could be so "coarse" as to be useless.

_ **Other applications in the system.** PC-sampling profiles the application "live"; other applications and OS background tasks could be running during the profiling session. Hence, the resulting data can be affected by code outside the target application. If, for example, between two samples taken by the profiler, the application had been interrupted by an OS task, the profiler -- having no knowledge of that interruption -- would assume that the CPU's time between those samples had been taken up solely by the application.

We'll discuss this particular problem in greater detail later.

Now that we've examined the intricacies of passive profilers, and seen how they use PC sampling to monitor an application's performance from the outside, let's move over to active profilers.

Active profilers use different methods to monitor the application. They gather performance data from the inside of the application.

Active Profilers

Whereas passive profilers take an "I'll stand back and watch" approach, active profilers actually modify the application.

Jump back to the analogy of the coach. If the passive profiler is a coach observing from a distance with a stopwatch, then an active profiler is a coach who -- armed with the latest in medical telemetry equipment -- attaches instruments to his athletes to measure their performance. An active profiler attaches "probes" or "instrumentation code" to the application (within the application, in fact). This instrumentation code feeds performance data to the profiling tool. (The data is logged either in memory or to a file while the application runs. The profiling tool reads the logged information after the application completes.)

The fact that the profiler is modifying the application could cause problems. The profiler must make modifications in such a way that the application's performance is altered as little as possible. Back to the image of the coach attaching monitoring instruments to the athletes: too many instruments attached, or attached so that they encumber the athletes, and the information collected is useless. The athlete's performance will have been altered by the instrumentation. Similarly, if the instrumentation code added to the application impedes the application's execution, the collected data could be corrupted.

Developers of active profilers face the age-old problem of how to invade a system in order to observe it but, in so doing, not modify the system so that the observations are invalid.

How, then, do active profilers instrument the code?

Methods of Instrumentation

Active profilers attach instrumentation in two ways:

- _ Instrumenting source code.
- _ Instrumenting object (executable) code.

Each has its advantages and disadvantages. Also, there are variations on the fundamental theme of each technique.

Source Code Instrumentation

A profiling tool that uses source code instrumentation adds profiling code to the application's source. Hence, the profiling code is compiled into the final executable right alongside the application's source code.

There are a couple of ways that a profiling tool can instrument source code.

One way is via a collection of cleverly-defined macros that the user adds to the application by including a header file. These macros are defined so that they invoke calls into profiling library routines (which are linked into the application) at strategic locations in the code. (Note: This technique is usually used by code-coverage tools.)

Another way a tool can instrument source code is by using a pre-processor that parses the code before it is passed to the compiler. The preprocessor locates function calls and basic blocks, and adds calls at those locations so that -- at runtime -- when a function is called or a basic block is entered, a profiling library routine is called that logs the activity.

Instrumenting "By Hand"

Actually, source code instrumentation can be done manually. Most programmers have, at one time or another, probably fallen back on the "debug-by-printf()" technique: peppering code with strategically located printf() statements as a first attempt at locating where a crash occurs, or as a means of printing out suspect variables at important points in the application's execution.

Using an operating system's "get current time" functions to create "stopwatch" routines, a programmer could use a similar technique to measure duration between two points in a program's execution. One routine would start the stopwatch (record the current time); the other would stop the stopwatch (read the current time and calculate the time elapsed since the start stopwatch call) and write the results to a log buffer or file. Thus, the programmer would do manually what many profilers do automatically.

Of course, the foremost drawback to "hand" profiling is that it is tedious and error-prone. If you insert the stopwatch calls yourself, you have to remember to go back and take them out when you've finished profiling. Otherwise, you'll release an application that emits profiling information to an unsuspecting user.

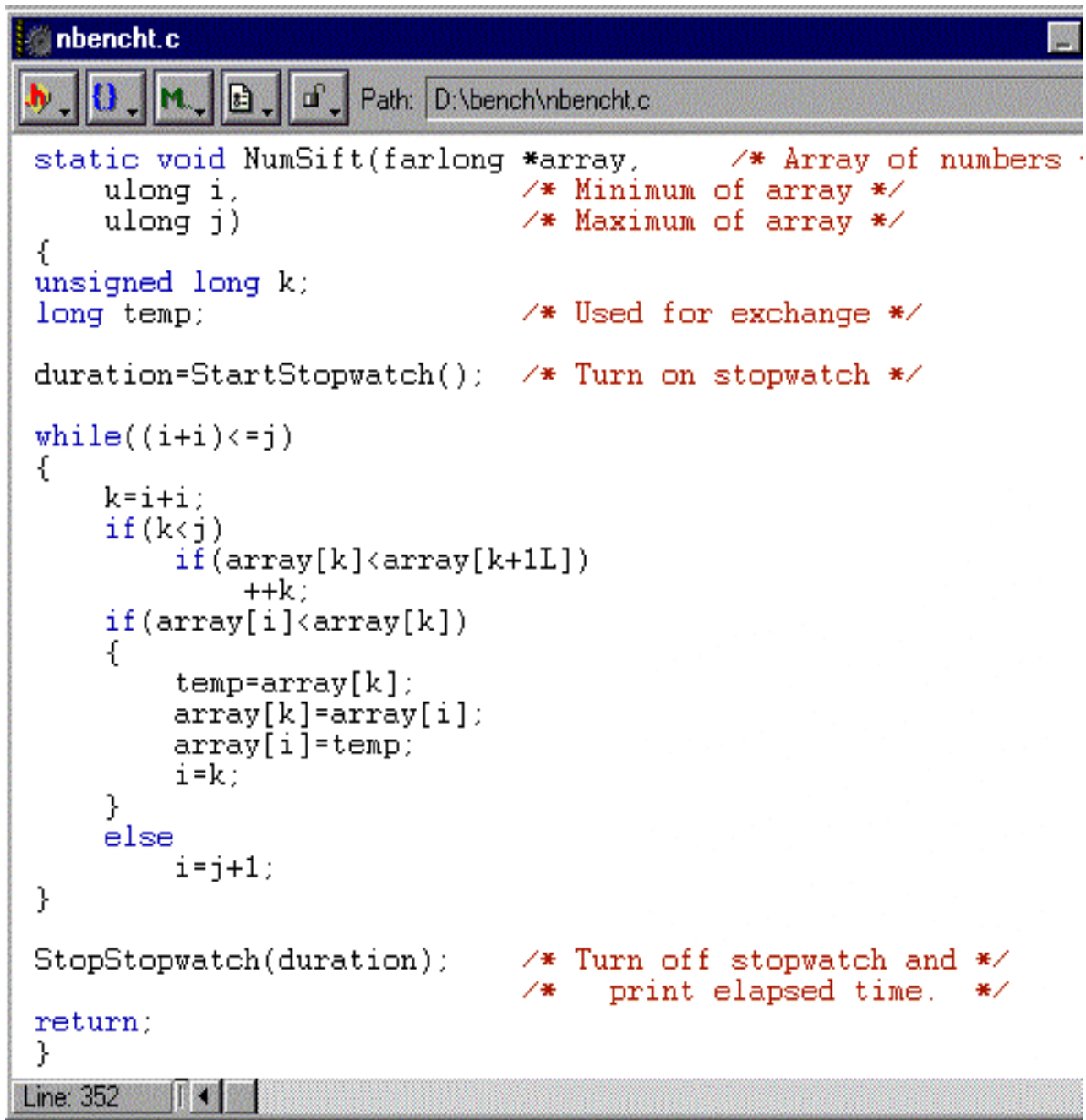
In addition, you have to be careful not to get too "enthusiastic" about adding measurement code. Depending on how many additional calls you insert, you could cause a substantial increase in executable size. And, if you collect a lot of time measurements and log the data to disk, you can significantly slow down the application.

Object Code Instrumentation

The second instrumentation technique is object code instrumentation. Tools that use object code instrumentation add their measurement code at a later step: after the executable has been created. Profilers that use object code instrumentation operate in one of two ways: direct modification and indirect modification.

Direct Modification

Direct modification is also referred to as "binary patching." A tool that uses binary patching employs a method similar to the method that debuggers use to set breakpoints in an application. A debugger sets a breakpoint in an application by replacing one or more of the application's machine instructions -- at the breakpoint location -- with a specific machine instruction called a "break" instruction. The application executes and, at some point, hits the break instruction. When the break instruction executes, it



```
static void NumSift(farlong *array,      /* Array of numbers
        ulong i,                        /* Minimum of array */
        ulong j)                        /* Maximum of array */
{
    unsigned long k;
    long temp;                            /* Used for exchange */

    duration=StartStopwatch();           /* Turn on stopwatch */

    while((i+i)<=j)
    {
        k=i+i;
        if(k<j)
            if(array[k]<array[k+1L])
                ++k;
        if(array[i]<array[k])
        {
            temp=array[k];
            array[k]=array[i];
            array[i]=temp;
            i=k;
        }
        else
            i=j+1;
    }

    StopStopwatch(duration);             /* Turn off stopwatch and
                                        /* print elapsed time. */

    return;
}
```

Line: 352

Figure 1. Hand-instrumenting source code. This heapsort algorithms has been hand-instrumented with the StartStopwatch() and StopStopwatch() user-written routines. StartStopwatch() reads the current time and stores it in the variable duration. StopStopwatch() also reads the current time, and uses the time stored in duration to calculate the elapsed time since the StartStopwatch() call. StopStopwatch() prints the elapsed time. Note that, for best accuracy, the user will have to find some means of accounting for the time taken by the StartStopwatch() and StopStopwatch() calls themselves.

causes what amounts to a jump into the debugger. Of course, the instruction that's been replaced by the break instruction is executed first, so that correctness of execution is maintained. The mechanics are similar if the application is being profiled. When the break instruction is encountered, program flow transfers into the profiler.

(Note: What we've described here is referred to as a "soft" breakpoint. A soft breakpoint works by modifying the code. However, some processors provide breakpoint registers that a debugger -- or profiler -- can set by loading the register with the address of the intended breakpoint. When a program's execution encounters the address, the CPU vectors into debugger code just as it would if a break instruction had been executed.)

The profiler patches the executable in this manner, placing "probes" at strategic locations within the code. Precisely where the probes are placed depends on the extent of the desired profiling. For example, if the user simply wanted to measure the execution time of a single function, the profiler would place test points before and after the function call.

Indirect Modification

A profiling tool that uses indirect modification instruments the executable in a three-step process. First, the tool reads the executable file and translates it into an "intermediate representation" (IR). This is sort of like translating the executable from one language to another (the source being machine language, the destination being the IR). However, the IR is not a language in the sense of C or assembly language. It consists of data structures that allow the profiler to store information regarding the overall organization of the application. (This information is gathered during the translation process.)

The profiler then instruments the IR, and re-translates the instrumented IR form of the application back into executable form. As a result, the executable has instrumentation code built-in.

Admittedly, this seems convoluted. Why convert to an intermediate representation, instrument that, then re-convert back to executable form? Why not instrument the executable directly?

First, indirect instrumentation gives the tool greater control over the structure of the final application. Since the tool is building the executable, the tool can control where in memory the instrumentation code is placed in relation to the application code. The tool can, for example, place all the instrumentation code at the end of the application's text area, so that the topology of the application -- that is, where the application's routines are placed in memory relative to one another -- is undisturbed.

In addition, the process of translating the executable into the intermediate representation provides useful information to the profiling tool. This information, used in conjunction with symbolic information (generated by the compiler) allows the profiler to identify, for example, basic blocks of code, which is how the profiler knows where to insert instrumentation code. (Note: The information that the profiler gathers while converting from the executable to the IR can be used to make other testing tools. For example, because the profiler translates the entire executable, it has all the information it needs to provide code-coverage testing.)

Note: Notice an important difference between the source and object instrumentation used by active profilers and the PC sampling technique used by passive profilers. PC sampling is inherently random; whatever routine is executing in the application when the timer interrupt goes off is whatever gets sampled. Active profilers place their instrumentation code in the application at "known" locations.

Object Instrumentation vs. Source Instrumentation

Object code instrumentation has some obvious (and non-obvious) advantages over source code modification that, we believe, argue in favor of object code instrumentation tools. With an object code instrumentation profiler:

— **You don't need source code to profile your application.** This advantage might, at first, seem specious. A developer will almost always have access to the source for the target application. However, there may be instances in which the developer will want to include third party libraries (for which source is not available) in the profiling. In those instances, object code instrumentation is the only choice. (Note: Of course, there are times when you want to exclude third-party libraries from the profiling. You may want to focus exclusively on the times generated by routines in your application - code you actually have control over.)

There's a less-apparent development time advantage associated with this. As you'll see, many active profilers offer more than one data-gathering technique. With an object code instrumentation package, you can change the data-gathering technique without having to re-compile the source code. And, if the profiler is intelligent enough to cache the IR of the target, switching to a new data-gathering technique is even faster.

— **An object code instrumentation tool can control the impact of the instrumentation code on the application's topology.** Source code modification adds instrumentation routines to the application's source code. Consequently, the in-memory location of the executable code generated by the instrumentation routines is at the discretion of the compiler and linker. The instrumented application's in-memory floorplan can be significantly different from that of the uninstrumented application. Consequently, the uninstrumented application's performance characteristics can be very different from the instrumented version. (The reason is that the source code inserted by the tool will cause the routines in the instrumented application to be linked

into a different memory location than in the uninstrumented form. The performance of the processor's instruction cache will be different for the instrumented application than for its uninstrumented version. Consequently, the developer might see -- and alter code in order to correct -- poor instruction cache behavior caused by the presence of instrumentation code.)

This is a classic example of how observing a system will perturb its performance. The performance analysis tool must be cautious about how it instruments the executable. The presence of the instrumentation code could change the performance profile of the executable so much that the reported data is misleading...and alterations that the programmer makes to improve the code can actually degrade the final executable.

Object code instrumentation modifies the executable, and therefore has access to the application's topology. If the instrumenting tool is "careful", it can place the instrumenting code after the application's text area. Consequently, the topology of the application is less affected.

Note: Most programmers can remember function nesting in their application to at least one level. That is, they know which child routines are called by a selected "parent" routine. However, to carry the family tree metaphor a bit farther, programmers are less aware of grandchild, great-grandchild, etc. relationships. A problem routine "down deep" in the hierarchy can impede the performance of higher-level routines that call it.

Data Gathering Techniques

Now that we have looked at the advantages that object code instrumentation profilers have over source code instrumentation tools, let's examine the data-gathering techniques that object code instrumentation tools employ. There are two such techniques: time-stamping and cycle-counting.

In a way, these techniques are complementary -- each reveals a different aspect of an application's performance. A deeper investigation into each technique will make this apparent.

Time-Stamping

Time-stamping is similar to the manual instrumentation that we described in the "Instrumenting by Hand" section above. It's like setting stopwatches throughout your code to measure execution durations. The profiling tool "brackets" regions of code in the target application with calls into the tool's library.

Time-stamping is sufficient for measuring the execution time of functions and "basic blocks" of code (i.e., for(), while(), and do() loops, as well as the true and false "arms" of an if() statement). You cannot, however, use time-stamping to measure the time taken by individual lines of code (much less a single instruction). The code overhead -- adding measurement code for each instruction -- makes that infeasible.

Because time-stamping measures actual time intervals while the application is running, it has some of the same advantages and disadvantages of PC sampling. If you want to see test application behavior while other applications are running, time-stamping will "feel" the effects of those other applications. As stated earlier, this could be good or bad, depending on your needs. Should you need to isolate measurements to the target application alone, time-stamping won't get you there.

But cycle-counting, the other data-gathering technique, will.

Cycle-counting

A system using cycle-counting actually counts processor cycles taken by individual machine instructions in the application. The tool can then determine the amount of time taken by a block of code by adding up the cycles of all the instructions in that block of code and multiplying appropriately to account for repeated execution of instructions in a loop.

Though this scheme might appear to produce exact results, the results are unfortunately approximate. A tool employing cycle-counting will use "static" instruction information to determine the cycle time for a specific instruction. This static information is no more than the information as reported by the processor's manufacturer in that processor's data sheets. Consequently, the cycle-counts used are optimistic: they do not account for run-time situations that can increase the number of cycles that the instruction requires. Such situations include cache misses, misaligned data, mispredicted branches, and so on -- things that happen at run-time and cannot be predicted by simply examining the individual instructions of the application.

(Note: We don't want to sound too negative, here. The static model used by most profilers is not uselessly simplistic. Some active profilers, for example, actually examine instruction sequences, will recognize pipeline stalls, and can factor in the time added by the stall.)

Note that cycle-counting is available only to tools that use object-code instrumentation. To use cycle-counting, the tool must have access to the machine code instructions of the executable. A source code instrumentation tool works on the "wrong side" of the compiler; it cannot know what instructions will be generated, and therefore cannot use cycle-counting.

Instruction	Clocks with aligned data	Clocks with misaligned data
MOV EAX,[EBX]	1	4

Figure 2. Missed by Cycle-Counting. The above table shows the 486 instruction for loading the extended AX register indirectly through the extended BX register. If the data is aligned – that is, if the address in the extended BX register is evenly divisible by 4 – then the fetch requires only 1 clock. (On a 25 MHz 486, one clock is 40ns.) However, if the fetch is at a misaligned address, the instruction takes an additional 3 clocks. (This is why aligning a large array to an address “amenable” to the array’s data type is so important – it can significantly improve processing time.) Unfortunately, a profiler that uses cycle-counting cannot know whether the address in the extended BX register is aligned; that information is not available until runtime. So, the profiler would count the instruction as taking 1 clock cycle, when – if the data is misaligned – it might take 4. Hence, such a profiler’s report is likely to be more optimistic than reality.

Comparing Passive and Active Profilers

We've looked at the characteristics of passive and active profilers. We've also examined the different data-gathering techniques that each employ. Let's recap, collect the facts we've seen so far, and do a comparison.

Passive Profilers - Advantages

There is at least one advantage that passive profilers have over active profilers: they are less intrusive. Passive profilers do not alter the structure of the application. The developer, at least, has the assurance that the application being tested is largely in the form it will be in at delivery time. (We say "largely" because it's likely that the developer will compile the application with symbolic information attached. This allows the profiler to match instructions in the executable to the lines of source code that generated them. However, some symbolic formats include the symbolic information in the executable, and consequently create a larger, slower-running executable than the release version of the application.)

Therefore, passive profilers that use PC sampling have -- as compared to active profilers -- a minimal impact on the performance of the application. Recall that a profiler using PC sampling will install a timer interrupt at application startup. After

that, the application is completely "unaware" of the profiler's presence. When the timer interrupt fires, the profiler's ISR wakes up, samples the application's PC, saves that in the log, then exits. Consequently, the profiler performs little processing at application runtime.

Passive Profilers – Disadvantages

On the other hand, passive profilers have several drawbacks:

— **The data captured is "flat".** Because the passive profiler uses random PC sampling, the data gathering ISR (awakened by a timer tick) is awakened at unknown locations in the application's execution. It cannot, then, decipher the contents of the stack to deduce the structure of the call chain. Simply put, the profiler can determine which routine is being executed, but it cannot know who called that routine.

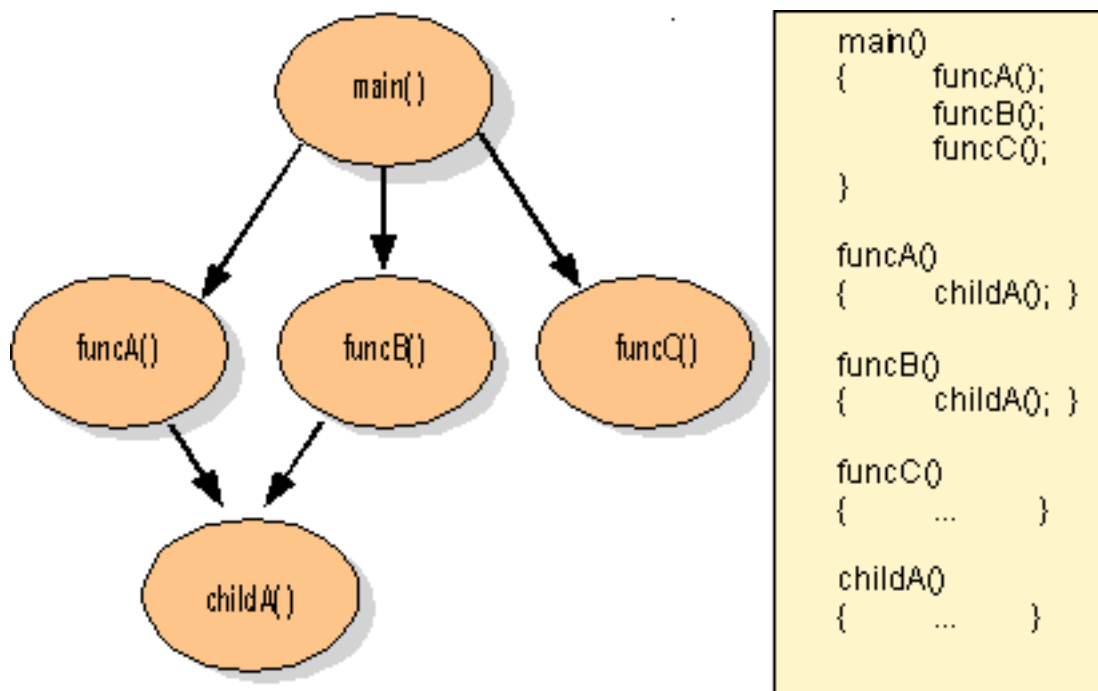


Figure 3. Multiple Parents. In many applications, a single routine will be called by several other routines. Put another way, a given child function will have multiple parents, as illustrated above. Passive profilers have a "flat" view of the functions within the application, and cannot know how much of `childA()`'s total time is spent as a child of `funcA()`, and how much is spent as a child of `funcB()`. This is information critical for a programmer; `childA()`'s execution time can be dependent on its input parameters, and `funcA()` may – likely will, in fact – call `childA()` with a different set of inputs than will `funcB()`.

So, if a single routine has more than one parent (called from different locations in the application), a passive profiler is unable to collect the information needed to determine that function's contributions to the execution times of its parents. A developer cannot determine if one parent is calling the routine more than any other; therefore, the developer can't get information needed to determine whether a fault lay in the parent routine (which could be calling the child routine an inordinate number of times) or in the child routine.

— **Reported times are affected by other applications.** A passive profiler using PC sampling installs an ISR that repeatedly wakes up and samples the program counter. If the application under test is running in a multitasking operating system (virtually all desktop operating systems have some level of multitasking), then it's possible -- likely, even, that code other than that in the application will be executing between samples taken by the profiler. The collected data will suggest that the application performance is below its true level.

In some cases, this is a plus. It can be worthwhile to see how the runtime of an application is affected by competing applications in the operating system, since that is the environment of the final application.

On the other hand, it can be a disadvantage if a programmer's interest is in the performance of the application only. Other application or system tasks simply pollute the test data, and make it all the more difficult to locate the application's problem areas.

— **Data capture is a statistical process.** To get an accurate picture of an application's profile, most passive profilers recommend you run several sample sessions, and at different sampling frequencies. Since passive profiling is not immune to other applications in the system, it's possible that one of the sampling sessions is "polluted" by the sudden execution of an OS task or another application consuming inordinate amounts of CPU time. A developer unaware of the intruding task or application could be thrown off by the data.

Active Profilers - Advantages

Having examined the advantages and disadvantages of passive profilers, let's turn our attention to active profilers.

— **Can use multiple data gathering techniques.** While passive profilers are more-or-less limited to PC sampling, active profilers can offer two data-gathering techniques: time-stamping and cycle-counting.

These two techniques provide different views into the performance of an application, and therefore give a clearer picture of what's going on than one view alone. As described earlier, cycle-counting's times reported are overly optimistic, showing the application's best-case performance. Time-stamping, on the other hand, gathers its data in real-time, and therefore returns times that are more in line with what the application will exhibit while it is executing in the real world. Time-stamping's reported results will be pessimistic as compared to cycle-counting's results. (Note: Not all active profilers offer both techniques.)

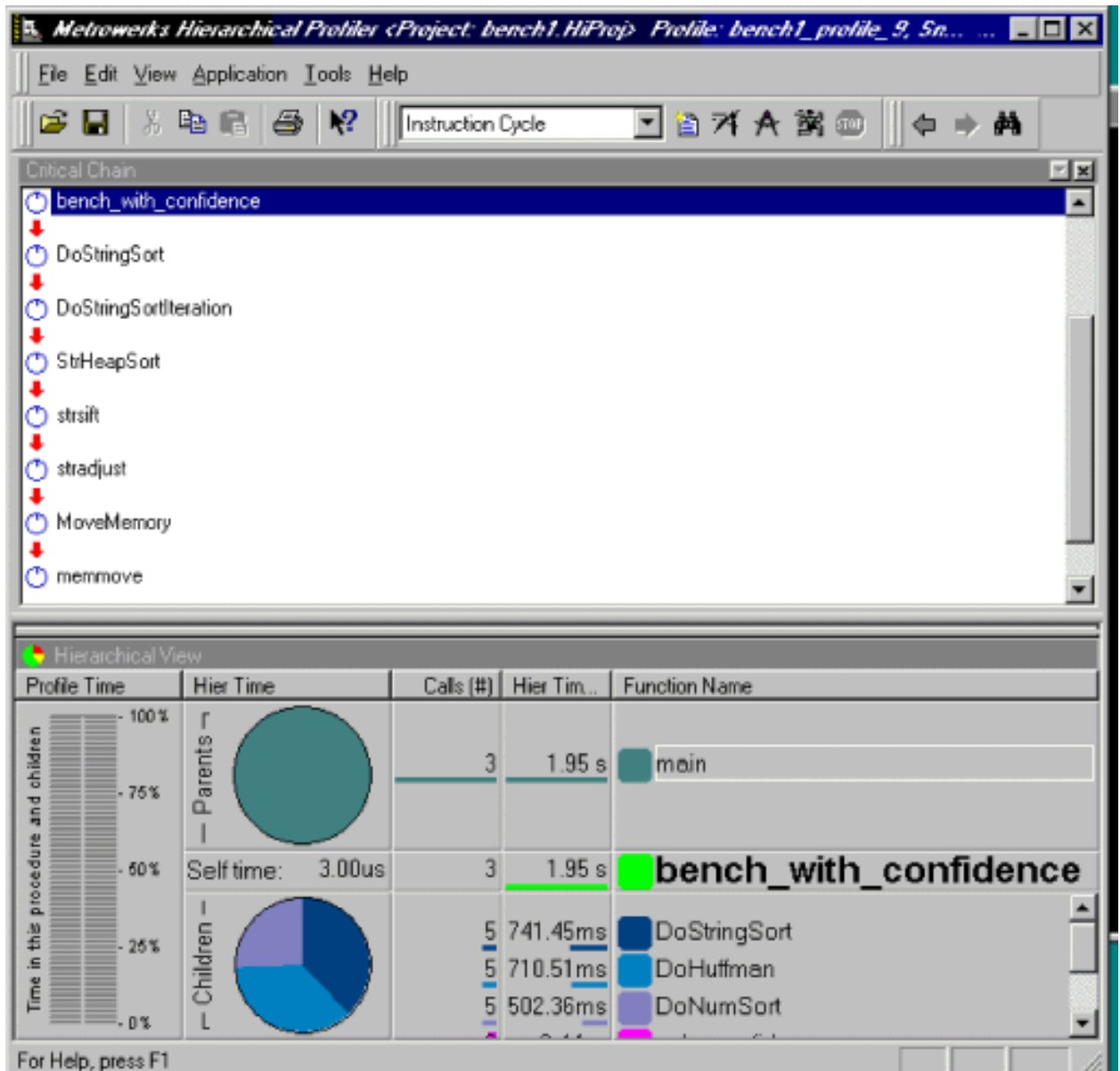


Figure 4. **Control flow.** This screen shot from Metrowerks' CodeWarrior Analysis tools shows a call chain view in the upper window. This particular view is referred to as the "critical chain" view, and shows the lineage of parent, child, grandchild, etc. routines that represent that branch of the function tree where the most time is being spent.

— **Can produce hierarchical information.** Active profilers have access to the internal structure of the application, and can apply this information at run-time. Consequently, they can determine who is calling whom, and build charts that show function call chains. In addition, they can annotate the function call chains to show how often a given child function is called by each parent.

— **May not require source.** If the active profiler uses object code instrumentation,

you don't need the source to profile an application. As we mentioned earlier, this can be a great benefit if you're working with third-party libraries for which you don't have the source. You can take into account the performance of the library.

— **Can build control flow graphs.** Since active profilers work either with the source (source-code instrumentation) or object and symbolic code (binary-code instrumentation), such tools have enough information to construct control-flow graphs. Control flow graphs allow the developer to see the hierarchy of routines within the application. The result is often displayed in the form of a tree, showing "parent-child" relationships.

In addition, there are advantages associated with the data-gathering techniques that active profilers use.

— **Cycle-counting is immune to other applications in the system.** If a profiler uses cycle-counting, the reported time is not affected by other applications in the system. Cycle-counting allows you to focus on the performance of single instructions.

— **Time-stamping accounts for run-time effects.** Since time-stamping runs the code "live", it captures the effects of run-time hold-ups such as cache misses, misaligned data, and so on. This, therefore, complements the view of the application as provided by cycle-counting.

Active Profilers - Disadvantages

To be fair, there are some disadvantages to using an active profiler.

— **Ability to profile third-party code may be of limited use.** If you don't have the symbolic information available for a third-party library, the profiler has to treat the library as something of a black box. True, you can perform cycle-counting profiling on the application, but this forces you to deal with assembly language. Reverse-engineering assembly language -- without the high-level source that generated it -- is anything but easy.

— **The profiler can increase application size.** Because the active profiler adds code to the executable, it increases the executable size. (If the profiler is "smart", the application portion of the instrumented executable will be unaware of the instrumentation code.) How much of an increase depends on how enthusiastic the developer gets about profiling. Time-stamping every function call and basic block can add a lot of calls. Cycle-counting increases the size of the application as well; more, in fact, than time-stamping, since cycle-counting adds instrumentation code for every basic block, rather than for every function.

— **Cycle-counting is static.** We've already stated that cycle-counting instrumentation is static in nature. It cannot measure run-time effects, and therefore gives an optimistic approximation to performance.

— **Time-stamping is affected by other applications.** Meanwhile, time-stamping suffers from much the same problem as PC sampling. Time-stamping cannot filter out the effects of other applications that may be running concurrently with the application under test.

In fact, time-stamping is less flexible than PC sampling. An "OS-aware" PC sampling system can gather data for all the applications running in the operating system. You can therefore get not only a profile of the execution of the target application, but of other applications and operating system processes as well. This would be handy, for example, in a situation where a separate application or OS process was actually the hold-up, and making your application seem pokey. By comparison, time-stamping code runs only within the context of the application being instrumented, and has no access to other applications or OS processes.

Conclusion

We began this whitepaper with a scenario that, we hope, illustrated the importance of profiling in the development process. Verification of correct execution is of paramount importance. Optimizing that execution can mean the difference between a working application that succeeds in the marketplace, and a working application that fails.

With that in mind, we embarked on a discussion of profilers. We showed that profilers can be either passive or active, and described the characteristics of each. We hope we've given an even-handed disclosure of the pros and cons of passive and active profilers, and of the various data-gathering techniques they employ.

It is our conclusion that, having examined all the facts, for fine-tuning the performance of an application, active profilers are superior tools to passive profilers. In addition, active profilers that employ object code instrumentation offer more advantages than source code instrumentation profilers.

In the final analysis, however, the tool you select depends on your profiling requirements. No single tool (yet) offers a one-stop-shopping solution to all conceivable profiling needs. If, for example, you need to profile the overall performance of a computer system -- OS, applications, device-drivers, the works -- then a PC sampling profiler is the only way to go. If, on the other hand, you need to step through your code with microscopic accuracy, fine-tuning individual machine instructions, then an active profiler with cycle-counting can't be beat. You will have to be the final judge; we can only hope that the information presented here helps in your search for the ideal profiler.

Addendum: Recursion and Hierarchies

We've established that the hierarchical view of function interaction as provided by active profilers can be a powerful tool in tracking down performance hotspots.

However, if the profiler doesn't handle hierarchical views properly, they can, in some situations, confuse an unwary user.

How? The answer comes when you realize that functions are not always associated in simple "parent/child" relationships. In a recursive algorithm, a function might call itself, either directly or indirectly through another function.

Consider how a profiler might provide a hierarchical representation of recursion, and - - more importantly -- how that profiler would go about reporting the results. In a simple parent/child relationship where function A calls function B, a profiler can report data that indicates that "function B -- contributes xx time to the overall execution of function A."

Simple enough -- but what if function A calls function B, which turns around and calls function A again, which then calls function C. If the profiler isn't smart enough, it could calculate a portion of function A's time twice -- once as a parent, and again as a child of function B.

As an example of how tools deal with this, Metrowerks' CodeWarrior Analysis Tools, keeps track of call graphs, and is able to identify loops that indicate that recursion is taking place. The CodeWarrior Analysis Tools "short-circuits" its own data-gathering when such a situation occurs, so that function A's time is not counted twice.

In conclusion, keep this in mind as you investigate profiling tools. Simply having hierarchical views is one thing; managing those views effectively in complex situations like recursion is another.
