PROFILING, PERFORMANCE, and PERFECTION

Robert Bernecky

Research Department
I.P. Sharp Associates Limited
2 First Canadian Place, Suite 1900
Toronto, Ontario M5X 1E3
Canada
(416) 364-5361
FAX: (416) 364-2910

INTRODUCTION

A *profile* is "a set of data often in graphic form portraying the significant features of something" [We88]. Profiles can help us to quickly understand a person or entity better. In the development of computer-based applications, profiles are invaluable. They help us to understand the application – how it works, how well it works, whether it in fact works as we think it does, and whether it is still working the same way it did last month.

For our purposes, profiling is *the analysis of a running computer program in order to determine its actual, rather than predicted, behavior*. Profiling may be performed manually, or automatically, with the aid of hardware or software. The data collected by a profiling activity depends on the type of analysis to be performed, but typically will allow determination of instruction mix, storage reference patterns, and instruction reference patterns.

This tutorial presents several tools for profiling of APL and non-APL languages and discusses their utility in improving the quality and performance of applications. As well, several case studies are presented, which are intended to provide some insight into how profiling tools might profitably be used in your own work.

THE BENEFITS OF PROFILING

Profiling is beneficial to the software developer in a number of ways, including:

- locating performance hot spots
- predicting performance of new applications
- performing long-term performance monitoring
- as an aid in performing quality assurance.

Let's look at each of them in turn.

Locating Performance Hot Spots

One of the most common, and most beneficial, uses of profiling is to determine how an application's performance might be significantly improved with relatively little effort. In its crudest form, this might be done by a frustrated user who complains to an application designer that the application is "taking a long time to run." The designer interrupts the application and observes that the interruption occurred at line *x* of verb *y*. If the situation recurs, and the interruption is always at line *x* of verb *y*, the designer might get the idea that line *x* has some characteristic which makes it execute slowly. This may lead to an examination of that line, and a rewrite of it, intended to produce an improvement in the application's performance.

Another industry favorite is the "honcho" or "guru" approach to performance improvements. In this case, the user approaches the application designer (the honcho) with a problem. The honcho makes an immediate guess as to the cause of the problem, then runs off and rewrites some hunk of code, believing that this will fix everything. Of course, without some research into the problem, the honcho is working on the basis of guesswork or hunches, and may in fact be working in a totally unproductive area. Perhaps "huncho" is a better term for this mode of dealing with performance problems.

One problem with both these approaches is that they are not quantitative. In neither case does the designer have anything concrete to say about performance, other than that complaints have ceased. There is no assurance that things have not, in fact, gotten worse in some other area. There is also no quantitative statement that can be made about the extent of the claimed improvement.

These haphazard approaches can be replaced by one in which the hardware or application is instrumented to provide information about what the application is doing. In the case where no pre-existing tools can assist in this instrumentation, the application writer must insert code to collect that information. For example, each user-defined verb might have code added to it which writes a record to a file each time a specific line of the verb is executed. Assume the record contained the time of day, CPU time used thus far, and the verb name and line number. Once the application was then run to gather this information, a post-processor could analyze it to determine:

- which user-defined verbs were executed
- which lines of each verb were executed
- how many times each line of each verb was executed
- how much CPU time each user-defined verb consumed
- how much CPU time each line of each verb consumed

If this data were to be sorted by CPU time, it would give an ordered list of where the largest gains in performance might be made. A commonly accepted rule in computing, known variously as Pareto's rule or the 80/20 rule, is that 80% of the processing of an application occurs in 20% of the code. Assuming this rule holds, the list obtained above will become uninteresting after the first few entries – improvements made to code which is executed rarely or not at all will not make a measurable difference in performance.

Software-based profiling has a number of flaws:

Inaccuracy. The ability to exit from a verb in mid-line, rather than by falling out of the bottom of the verb, can cause data to be lost or to be misleading.

Heisenberg effects. The time required to sample clocks, write data to file, and so on may interfere substantially with the operation of the application. In the case of real-time applications which must react to external stimuli at rapid intervals, the measurements may be skewed to the point of meaninglessness. By the time one event completes, including time spent monitoring it, it is time to deal with the event again. This is similar to the Uncertainty Principle described by the physicist Heisenberg: it is not possible to measure something without altering its behavior in some way.

Introduction of errors. The act of altering the application in order to instrument it is liable to introduce errors into the application. The errors may be due to incorrect installation of the monitoring code, or to assumptions made in the original application, which render it sensitive to certain types of modifications. Use of absolute line numbers is an obvious form of a poor programming practice which could lead to failures of this type. The real problem with application alteration is that it constitutes a form of maintenance. The probability of getting a maintenance change correct on the first try has been measured at roughly 50% if fewer than 10 lines of code are changed, and much less as the volume of the change grows [Ma83].

Performance. The overhead of the instrumentation software may be unacceptably high. If monitoring a critical transaction-based system causes the transaction time to rise from .5 seconds to 10 seconds, then software-based profiling may be unacceptable. This is an extreme form of the Heisenberg effect.

Although these flaws cannot be totally removed, they can be managed by providing hardware or software assistance. These will be discussed in a later section.

Performance Prediction

When designing new applications, it is often important to be able to predict how well they will perform as the size of the problem or associated databases grow. A profiler can assist in this process. As an example, consider an application in which news stories from a wire service are archived in a data base, to be accessed by users who would use full text searching. The user might make the following query: "Show me all stories containing the phrase 'cold nuclear fusion', but not the word 'palladium'." Assuming the database will continually grow in size as more articles are added to it, it is critical that the time taken to perform such a search grow no worse than linearly with the size of the data base. In fact, one would attempt to design it so the search time grows sublinearly or not at all.

Assume that the application exists, and we wish to ensure that we have achieved our goal. To this end, we make repeated runs of the application under control of a profiler, with a number of growing databases. We then plot the CPU time used in each line against the database sizes. Any non-linear growth will be obvious, to be attended to before it reaches crisis proportions. In a related fashion, changes that improve performance will be quite visible as a reduction in the slope of the lines.

Profilers can also be of use in predicting performance during the course of program development: Karl Dawson implemented *ebar*, an array-searching primitive verb, for SHARP APL [Da88], by first modelling it in APL. He then used a SHARP APL profiling facility, □*fm*, in conjunction with the APL model, to predict actual interpreter performance. This methodology allowed him to create complete test scripts before any actual code had been written, and gave considerable insight into the value of several proposed special cases for the primitive.

Long-term Performance Monitoring

Performance monitoring of a running application is often abandoned, on the assumption that it's working perfectly, until the day when the users come screaming that *your* application is running so slow that it's unusable, and that they are going to:

– take you to court
– not pay their bill
– demand a refund
– go to your most unfavorite competitor
– all of the above.

On that day, management is likely to suggest that a quick fix of some sort be cobbled together *right now*. This will usually result in code changes which end up being too expensive, poorly designed, a maintenance nightmare, and inadequate except as a temporary circumvention. Significant effort will be required later, not only to correct the original problem, but to undo the quick fix.

Robert Bernecky

An automated monitor in place on such a critical application could call attention to a mounting problem long before it reaches crisis proportions. The problem could then be addressed in a cool and collected fashion, without earning the ire of the users, and without developers developing ulcers from working 24-hour days.

An automated monitor could execute typical transactions at regular intervals under the control of a profiler, recording the profiler results for each transaction. Another task would periodically analyze the profiler information and send an electronic mailbox message or other alarm to appropriate parties if elapsed or CPU times began to edge up to unacceptable levels. It's advisable to both log and plot profiles, for at least two reasons. First of all, the visual nature of a plot will make performance trends stand out clearly. Second, if for some reason the profiler, logger, or plotter ceases to work, you'll find out about it sooner. There is nothing more frustrating than going to a log file to look for historical data that will pinpoint a problem, and discovering that no data has been collected for six months because the log file was full.

A profiler should allow information to be collected easily without altering the application. This encourages its use in conjunction with the actual production code, rather than with a modified version that may not reflect reality. In addition, developers are more likely to make use of profilers if they are extremely easy and convenient to use, than if they require tedious planning and effort to use.

Data collection at the line by line level allows early detection of potential bottlenecks before they are visible in the aggregate of total CPU or elapsed time. For example, in a transaction which takes ten seconds, the time required to hold a file to prevent concurrent updates might only take a tenth of a second. Normal variations in aggregate execution time might be more than a second. If the time to perform the hold started to grow exponentially, it might have to increase by almost a second before a human would notice, at which time it might be too late to take thoughtful corrective action. Analysis of detailed information of this sort can provide valuable *early warnings* which make everyone's life easier. The idea is to be able to *predict* changes in usage patterns, and changes in load patterns before they become crises, by watching for non-linear growth trends.

Profiling activities shouldn't be restricted to observations of processor time only. Elapsed time variations can be very enlightening. By showing where real-time system delays are occurring, they can highlight problems in such areas as file system I/O queueing, shared variable processor or other communication bottlenecks, and locking delays on files or other serialized resources.

Finally, if a real problem surfaces, and no historical information is available to support analysis, the power of profiling tools as dynamically alterable instrumentation can help to pinpoint the hot spot quickly and precisely.

Quality Assurance Tools

There is no practical way to prove that a computer program will function correctly. The best we can do today is to employ the best designers and programmers we can find, prototype our designs, ensure that designs and code are meticulously vetted by independent, objective judges, and perform quality assurance tests as a verification step, to further support our belief that the application works correctly.

In the past, courts have been rather lenient on the computing industry as a whole, probably because of the infancy of the profession. However, as the industry matures, it will have to accept a larger measure of responsibility for errors caused by computer-based applications. If a bridge collapses, the engineers who designed it are probably in very hot water. Similarly, if a fault in a computer program causes death, injury, or significant financial loss, and the vendors of that program cannot show that they took all reasonable efforts to ensure the correctness of that program, then the vendors, and perhaps the designers themselves, are in line for civil and perhaps criminal action.

Performing quality assurance procedures on computer programs pays handsome divdends. It is well known that the cost of repairing a fault in a program increases by orders of magnitude as the implementation proceeds. An error discovered in the design phase is relatively inexpensive to remedy. An error discovered during development costs perhaps ten times that much to correct [Bo76, Br74]. An error found after product shipment is extremely expensive – customers get upset, and *n* copies of software, rather than one, have to be repaired. This of course offers opportunities for new problems to creep in – fix not applied, fix applied incorrectly, and so on.

Given these changing times, it behooves software vendors to take whatever steps are required to ensure that their products are as predictable as possible. Profiling can help this process in two ways: It serves as a mechanism to support claims about product reliability, and it ensures that performance claims will be met now and in the future.

Test Suites

Test suites are scripts written by software developers to support their claims that their programs in fact operate as designed. However, software developers are Panglossian by nature, and rarely exhibit an appropriate degree of skepticism about the reliability of their products. Casually designed test suites, therefore, may in fact deal with only those areas of a program about which the developer was concerned, and ignore large areas which "couldn't possibly have bugs in them."

A profiler can be a valuable tool in assisting software developers to remove the blinders from their eyes. They are secure in their knowledge that their code is bulletproof, secure in their knowledge that their test suites are a complete test of their code. Now, introduce a profiler, and ask no more than that the developer prove, rather than claim, S0 coverage; that is, mere execution of all instructions in the program.

Robert Bernecky

The results are eye-opening: Test suites rarely cover all the code. Developers are often at first puzzled by this revelation. Next, they take steps to correct the problem, and rewrite the suites to cover the missing areas. When they see the results, they become converts.

Developers often consider formal test suites to be a waste of (their) time, because "It's going to bring my development work to a halt! We'll never deliver on time!" Also, they are often offended by their manager even suggesting that they write test suites, considering this to be an attack on their competence. It is difficult to sell them on the idea. In my experience with a large development group, the only effective approach has been: "Just try it this once, and see how you like it, ok?" Once developers realize how their code quality has improved, they buy in readily. People like to do the best job they can; they'll use tools if they see a real benefit in doing so.

Besides serving as obstacle courses for system alterations, test suires also provide a handy benchmark for performance analysis purposes. If your new, improved system survives the obstacle course, but takes twice as long (or even 5% longer!) to run it, are you likely to knowingly unleash it on an unsuspecting public?

Finally, complete test suites allow those poor souls who are responsible for product support to have some faith that a new product may in fact work as advertised or better. The ability to rigorously test a system you're going to end up supporting, rather than taking a developer's word on its robustness and correctness, offers some peace of mind.

Suitably written test suites allow obstacle courses and performance measurements to be automated, reducing the human effort required to support an otherwise labor-intensive activity. Such tests can be run before and after each system change is released, to nip problems in the bud. Chasing performance problems months or years after they were introduced into a system is extremely difficult – old code simply stops working, due to lack of storage space for backup copies, incompatible operating system upgrades, and other such mundane but all too real concerns.

A FEW PROFILERS

A number of profilers of various degrees of sophistication and convenience are available for most computing languages and systems on the market today. What follows here is not a survey. Rather, it is intended to describe a few of the capabilities and limitations of several profiling tools (and facilities which have been bent into profiling tools) that we at I.P. Sharp Associates have used for our own work.

Profiler Environments

Profilers perform their work within a specific computing environment. In APL, they are associated with a specific APL task and application under the control of one user. The tool provided with MVS (an

IBM operating system for large computer systems) is usually associated with a specific *address space*, which might represent an entire collection of users. In VM (another IBM large system environment), it is associated with a *virtual machine*, which may represent either one or many APL users.

The APL-based tools are oriented toward APL application writers, and have all those characteristics that APL users expect and enjoy: ease of use and human-comprehensible results.

The operating system-based tools tend to be oriented toward assembler code programmers, and have all those characteristics that their audience has grown to expect. They're not exactly suited to the job, but with enough effort, you can bend them to work. Sort of...more on this later.

APL Profiling Tools

SHARP APL's Function Monitor, □*fm* [Sh87], provides the following information for any user-defined verb which it monitors:

- Line counts provide information on how often each line in the verb was executed.

- Elapsed and processor time is supplied for each line, including and excluding time spent in verbs invoked from that line.

- Configurable design gives you control over which verbs are to be monitored, and can control the level of detail of information to be collected – summary information of one line per verb, or highly detailed information on a line by line basis. The design is flexible enough to allow easy and consistent extension in the future to support new monitoring capabilities.

- Timings are precise to the level of the underlying system processor timer, typically within a microsecond. You don't have to run benchmarks for long periods of time to get meaningful results.

- Monitored information is correct whenever it is sampled, even within recursive, pendent, or suspended verbs.

The power of □*fm* became obvious to developers here on the day it was released on our internal SHARP APL system. The elapsed time from the point when the VIEWPOINT development team obtained the □*fm* documentation until they had used □*fm* to obtain a 25% CPU time reduction in the VIEWPOINT Report Writer was *three* hours!

STSC offers □*MF* [St85], a Monitoring Facility for their APL systems. □*MF* provides a subset of the services provided by □*fm*.

IBM offers a performance monitoring tool with APL2 called TIME [Ib87]. As of the publication

deadline for this tutorial, I was unable to find any documentation describing its precision or ability to handle recursion.

Operating System Profiling Tools

IBM provides VM TRACE for VM, and SLIP/GTF for MVS. SLIP is a generalized tool which IBM often uses as a problem determination aid; i.e., "When does my program get to instruction x?" When profiling, the question more often asked is: "When does my program get to instructions $x+\iota n$?" The Generalized Trace Facility (GTF) is used in conjunction with SLIP to intercept and process the events generated by SLIP.

One way SLIP/GTF can be used to perform profiling is as follows: The user configures SLIP to interrupt the executing program whenever an instruction of interest is executed. SLIP does this by conditioning the underlying S/370 PER (Program Event Recording) hardware to perform that task.

When a PER interrupt occurs, GTF processes it, takes some action, and returns control to the executing program. In spite of its name, GTF's capabilities in this regard are quite limited, and about the only practical action which can be taken for instruction tracing is apparently to ask GTF to write a trace record to disk or tape for each instruction intercepted.

For any realistic profiling work, VM TRACE and SLIP/GTF are inadequate. To see why, consider a real example chosen from the I.P. Sharp archives.

A performance problem had been reported in the SHARP APL newly released R19.0 interpreter: A user claimed a specific application ran slower than it did in the previous release. In the course of studying the problem, I had gotten to the point where it was clear that a problem existed, but I had been unable to determine what change in software was responsible for the problem. A search of our software change log, SOFTLOG, showed that hundreds of software changes had been made to the interpreter since the earlier release and that factoring out the changes by backing off each set of interdependent changes and running the affected application was simply impractical.

We decided that an instruction trace of the application running on the two software releases might provide enlightenment. The tool closest to hand was VM TRACE, so we started with it.

VM TRACE was designed as an aid for programmers, to let them step through programs and display the result of executing each instruction on a terminal or to write a line on the "vm print spool queue," traditionally used as the repository for data destined for a physical printer. Given a suitable amount of monkeying around, it is possible (not convenient, but possible) to capture this printer file data and copy it to a normal file where it can be analyzed by a program.

Knowing that the application was likely to execute *lots* of instructions, and being concerned about

having to analyze a large quantity of data (one record per instruction executed!), I chose a *very* small application test, which normally ran in about ten seconds on an IBM 3090 class processor. I started a test system with only one user on it, configured VM TRACE as required, and started the application. However, there was a problem – the disk space assigned for printer files in most shops is painfully inadequate for the quantities of data being called for here, and when a trace fills up that disk space, everything stops. Not just your job, but every job in the shop which wants to print something. It's quite user-hostile, and not the kind of thing which exactly endears one to the operations staff, so we gave up on VM TRACE.

Next, we tried SLIP/GTF, which at least has the ability to write its trace information on magnetic tape. We restarted the application after configuring GTF and SLIP appropriately, at which time, GTF happily started to spin tape, writing trace records. After a while, it started to write a second reel of tape. Then a third. We left for dinner at this point, and returned several hours later to find a mound of tapes, a slightly disgruntled computer operations staff, and tapes still spinning. At this point, I decided to cut my losses and analyze what I had obtained thus far.

The results were less than encouraging for someone who was hoping to make practical use of instruction tracing as a way to solve all the problems known to mankind. That particular ten-second test would have run for about two weeks of dedicated processor time, and have written roughly 600 reels of tape, or 120 gigabytes of trace information! The war stories and jokes which arose from this event led, in a day or so, to the design and development of SPY by Leigh Clayton, as a result of a suggestion from Kirk Iverson.

The basic problem with the IBM-provided tools is that they were not really designed for the kind of use to which we were putting them. They were designed to be used with events which occur infrequently – once per minute or hour – instead of at megahertz frequencies. The PER hardware was doing its job quite well, but the operating system supports for that hardware were simply inadequate. However, I.P. Sharp was, and is, a society of toolmakers. When we find available tools lacking, we build new ones to meet our needs.

In 1974, I created a "PSW Sampler" as a way to do statistical profiling. Similar in spirit to hitting "break" on a running application periodically "to see how it's doing," the PSW Sampler used timer delays to interrupt the running APL system and build a histogram of PSW values encountered in APL at those times. In the S/370, the PSW (Program Status Word) is akin to)SI in APL: It tells you what program is executing, and where it was when you interrupted it. Using the PSW Sampler, we determined that syntax analysis in APL consumed 10% to 15% of the processor time associated with most applications, with storage management functions running a close second.

The problems with the PSW Sampler grew out of its statistical nature. It gave a fairly good gross picture of the system, but wasn't reproducible, couldn't perform total code coverage, and was unable to monitor certain parts of the system, such as serialized code. Because of this, the PSW Sampler got dusty, and

stopped working around the time we switched from SHARP DOS to MVS. But it wasn't really missed until Leigh recalled the PSW Sampler's technique of using a histogram, rather than writing data to a file. Realizing that this approach could be used with SLIP to efficiently achieve most of our requirements, Leigh designed and wrote SPY.

SPY runs in conjunction with SLIP, intercepting the PER interrupts and processing them itself, to build a storage-resident histogram of instruction execution. The requirement specifications for SPY were that it minimize the number of instructions required to trace a single instruction, that it not perform I/O while tracing, and that it produce a histogram, rather than a history trace, to minimize post-processing requirements.

SPY is a hybrid system, written partly in S/370 Assembler code, to handle the sensitive PER interrupts, and partly in SHARP APL, to provide back-end analysis capabilities such as instruction count summaries, instruction count detail reports, and code coverage reports. The use of APL also offers significant flexibility and convenience in modifying and extending SPY's capabilities. For example, it was trivial to extend SPY to include reporting to the assembler code label level, once we obtained documentation from IBM on the format of AUTOTEST [Ke88] output.

SPY has been a wild success from day zero. It allowed us to gather the precise statistics we required in a few minutes, rather than weeks, and the post-processing task requires only seconds to process roughly a megabyte of data, instead of hundreds of gigabytes. Thanks to Leigh's expertise, SPY's overhead is quite low – a factor of about 30 slowdown instead of thousands. This allows it to be used for high-volume event tracing, and to be realistically used to monitor real applications.

According to Leigh, SPY doesn't stand for anything beyond its obvious cloak-and=dagger denotation. I prefer to think of it as the Sharp Performance Yardstick, but who am I to dispute the meaning of a name? Juliet?

Case Study - Storage Manager

The performance problem mentioned earlier is a prime example of how suitable instrumentation allows rapid focusing on the true, rather than suspected cause of a problem. In software of any complexity, a long-standing performance problem that cannot be attributed to any specific change to the system is hard to track down. Queries to involved developers of the form "Do you recall *any* changes you made which have even the most remote possibility of causing the problems we're seeing?" result in a "No flies on me, mate!" response from all of them: they *know* their code works perfectly – the problem must be someone else's. There is no way to assess which area is at fault except by backing off changes one at a time until you find the offending one. In a complex system, where changes often interact, this is a problem of Gordian-Knot complexity, and is simply impossible to solve.

SPY allowed us to nail the interpreter performance problem in a few hours – we had been chasing the

problem unsuccessfully for more than a week.

We obtained a copy of the application that the user claimed ran degraded under the new release. Then, with the help of Karen Brant and Gary Wride, both of whom were then members of the Software Coordination Team, we ran the application under both releases. Our measurements substantiated the user's claim of degraded performance.

This step was required because the application itself had been substantially rearchitected in the interval, and we wished to make sure the problem was really there, as described. We then reran the application under SPY with both releases of the system, to determine where the discrepancy in performance was coming from.

The results were eye-opening. Within an hour, we determined that a change to the storage manager had introduced a fault which rarely affected applications, but when it did, the effect was dramatic. Figure 1 gives a comparison of the first few lines of output from both runs. Let's do some quick analysis of these results. "TOTAL" is the count of actual relevant instructions executed. It is quite clear that an undesirable change has occurred in this test – 49 million versus 53 million instructions – degradation of more than 10%.

A look at the more detailed information below shows that AFMT is clearly the heavy user of cycles on this job, weighing in at 14.24 million instructions. Since both counts are within a tenth of a percent of each other, we know that AFMT is not our problem. Ditto for SABI and FUNSCAN. But what about SBREAL? It went from 3.3 million to 8.3 million instructions, clearly a change we didn't expect, in spite of the fact that other system changes resulted in many fewer calls to it (248 versus 406). This made it abundantly clear that our problem lay in SBREAL.

A detailed instruction trace in SBREAL showed an obvious fault. The fault was quickly corrected, and the system tested again. Results were gratifying. Not only did the application run faster now than in R17.0, but we *proved* that we had resolved the problem by direct, quantifiable comparison of instruction counts on a function by function basis. The instruction count for SBREAL dropped to 2.8 million, in line with the author's expectations, which were that SBREAL in R19.0 should cost less than it did in R17.0.

In the next months, we used SPY extensively on a number of benchmarks of our own, and on a number of benchmarks supplied by OEM and inhouse SHARP APL sites, with the intent of improving SHARP APL performance so that all benchmarks would perform significantly better than before. This effort paid off handsomely, as Figure 5 shows. Release 19.12 of SHARP APL offered substantially improved total CPU time reductions for a wide range of applications, all achieved in a short period of time. With SPY, we rapidly located hot spots in the system brought on by a number of very different applications, and quickly cool them off.

Case Study - Formatter

I was able, in the course of a few hours, to improve the performance of $\square fmt$, at a time when that wasn't even my prime concern. I had made some rather trivial changes to the formatter, and was writing an S0 coverage test suite for it. S0 coverage is by no means a complete test, but it has a well-deserved reputation for exposing code faults. In the course of writing the suite, I stumbled onto a program fault which had been in the code, undetected, for eleven years. After fixing this, I ran my completed suite under SPY.

Figure 2 shows a fragment of the result of "SPY-ing" in order to achieve S0 code coverage. An APL function called COVER analyzes a SPY histogram, an assembler code listing, and a load module, to produce a report highlighting the lines of code which were not executed during a particular test suite. The missed lines are indicated by question marks.

The version of COVER described here annotates the entire assembler code listing. A variant of COVER, which I find more convenient to use, produces only the non-covered lines as its output. When the output is an empty array, you're done. This empty array is no joke!

It is often enlightening to browse through code coverage results in detail, as potential bottlenecks, obvious bugs, or shortcomings may come to light. The fragment of code in Figure 3 is taken from an old version of $\square fmt$. I have annotated the listing by replacing the assembler-generated code with the instruction counts which resulted from executing a simple format expression on a 5000-element array.

In Figure 3, it is obvious that the four lines starting at L5H are being executed far more often than any other in the code fragment. In fact, several minutes of analysis showed that the loop performed by those lines could be replaced with non-looping code which indexed a table. An hour of coding and testing produced a significant speedup of the entire formatter.

Although the above example shows inline code, I usually sort the report by instruction counts, to quickly highlight the large CPU burners (a la Pareto.)

Reports of the above nature are valuable when developing code. If I believe that a specific piece of code should be executed *n* times per data element, then I will be able to predict what the report contents should be when executed with a specific test suite, and thus verify that things are working as expected. Deviations from the expectations are cause for alarm and careful analysis to resolve the differences.

S0 coverage also turned up some unreachable code, which proved to be faulty search loops. I was able to replace these by one line of inline code -- the fastest line of code is the one that isn't there. It's also the one that fails the least!

On the performance front, it is often handy to be able to obtain real instruction counts when comparing

two versions of a component, because the amount of other changes involved (other development work, operating system upgrades, faster processors) may be such that it is impossible to obtain meaningful measurements otherwise. It also offers a way for developers to quantify their improvements beyond vague measures of CPU time. Figure 4 contains excerpts from the APL function SPYSUMM executed against two versions of $\square fmt$, before and after the changes described earlier.

Several items are worth noting here: First, we timestamp all data sets, so that we have a way of ensuring that we are in fact analyzing the correct data. In a large development shop with many operating systems running at once, it is easy to get confused and analyze old data or someone else's data by mistake. Second, note that the number of distinct instructions executed has decreased slightly. If I had not made changes that would lead me to expect a discrepancy of that sort, I would go back and look more closely, to see what else had unexpectedly changed, perchance to turn up a bug. Finally, the most interesting part: the relative instruction counts of 5.5 million against 4.3 million show a clear performance improvement of 22%. Not bad for a few hours' work!

A lesson to be learned here is that although anybody could have looked at that code and immediately seen how bad it was, nobody had done so. (I feel compelled to point out that the formatter code in question here was not written by an I.P. Sharp employee).

A benefit of profiling is that it directs your attention to the root of a problem immediately and forcefully. The importance of this cannot be overstated. In the $\square fmt$ study, changing 10 lines out of 200,000 made a significant improvement in performance. The television repair technician who charged 50 dollars for two minutes' work to replace a tube (this is an old story) said "That's 1 dollar for the tube and 49 dollars for knowing which one to change." It's the same with performance improvements: you have to know where to look, and profilers are a very inexpensive way to learn where to look.

Case Study - Cross Sums

Although supercomputer applications such as computational fluid dynamics, ray tracing, and petroleum exploration are getting a lot of press lately, other major computing challenges are worthy of our attention. Some of these Everests, including Conway's Game of Life [Du71, Mc88], New Eleusis [Be81], and Rubik's cube [Pe84], have been addressed in the literature, but others have inexplicably remained unchallenged. One such problem is Cross-Sums.

Cross-Sums resemble crossword puzzles, except that:
- the words are integers
- the clues are the sum of the digits in the word
- no word contains a zero
- no word contains duplicate digits.

Cross-Sums puzzles can be found in almost any Crossword Puzzle magazine.

Since duplicates and zeros are forbidden, a Boolean dictionary of valid words for Cross-Sums is:

```
bd←(9⍴2)⊤⍤¨1 0⍳2*9
```

The set of ω-digit words summing to α is:

```
((α=bd+.×⍳9)∧ω=+/bd)⌿bd
```

One heuristic in Cross-Sums is determining the set of digits which may validly occur in a particular square. And-ing vertical and horizontal sets, using ∧⍤¨1 and inner product, eliminates impossible choices. Process of elimination is the key. For many puzzles, this suffices. For harder ones, recursion based on guesswork is called for.

In order that data entry for the puzzle should be simple – many puzzles are harder to type in than they are to solve – I chose complex numbers as an obvious (to me) way to enter values. (It turns out there is a much simpler method, but I won't discuss that one.)

My function solved most puzzles adequately, but it was slow. I decided to apply profiling with ⎕*fm* to determine where the time was going. I used the following crude APL functions:

```
∇ setall
[1] ⍝1 ⎕fm ⎕nl 3
∇

∇ r←getall
[1] r←⎕nl 3
[2] r←(⌊(2↑⍤¨1⊢¯1 ⎕fm r)×⍤¨1⊢ 1 1e6)⊃r
[3] r←⍕srtc r
∇

∇ setfn ω
[1] ⍝63 ⎕fm ω
∇

∇ r←getfn ω
[1] r←⌊ 1 1e6×⍤¨1⊢2↑⍤¨1⊢¯62 ⎕fm ω
[2] ω←' ',(50-¯1↑⍴r)↑⍤¨1 ⎕cr ω
[3] r←r⊃('[55]'⍕⍪(-⎕io)+⍳1↑⍴r),ω
[4] r←⍕srtc r
∇
```

```
∇ r←srtc ω;i;⎕io;b
[1] ⎕io←0
[2] i←⍒b←,0 1↓>ω[0]
[3] b←(b[i]≠0)/i
[4] b←(6⌊⍴b)⍴b
[5] r←(>ω[0])[b;]⊃(>ω[1])[b;]
∇
```

Let's apply these to the Cross-Sums program on a typical puzzle. Since we don't know anything, we start by monitoring everything:

```
setall ⋄ ⊣cs 'n1' ⋄ getall
 30 1315808 adjust
  2  460844 fillh
  1  106521 cs
116   88739 white
 15   26492 hnum
 15   26334 vnum
```

The first column is the number of times the verb in the third column was invoked. The second column is the number of CPU microseconds used in the verb. Clearly, *adjust* is the major eater of resources here. Taking a closer look at it, by monitoring just that one verb in detail, we see:

```
setfn 'adjust' ⋄ cs 'n1' ⋄ getfn 'adjust'
244  748889 [28] l←(sums=s[i;¯1+''⍴j])≠seqs
244  141429 [33] ib[;i;j]←ib[;i;j]∧⍤1 0 ∨≠ l
244   62208 [32] l←((+/m)=l+.∧m)≠l
```

Again, the first column is the number of calls, and the second is processor time used. The third column is, of course, the text itself. The first line, which is the largest consumer, is puzzling (no pun intended) – it is fairly simple APL, and not the sort of thing one would expect to cause problems. Here is another lesson of profiling: Bottlenecks occur where they will, not where you expect.

Decomposing line 28 into two lines, to determine which part of the line is the offending one, yields the following output from *getfn*:

```
244  707791 [29] l←sums=l
244  144244 [36] ib[;i;j]←ib[i;j]∧⍤1 0∨≠l
244   62117 [35] l←((+/m)=l+.∧m)≠l
```

Since the rest of the old line 28 doesn't even appear, it's clear that the problem lies in the =. But how can this be, since equality is such a simple function? Maybe it has something to do with data types? In fact, when we look closer, we see that *l* is a complex array, and *sums* is an integer array. I forgot

to convert the complex input data to integers during setup. Now, modify things to ensure everything is integer, and try again, from the top:

```
setall ◇ →cs 'n1' ◇ getall
 30   658678 adjust
  2   461912 fill1
  1   102782 cs
116    88496 white
 15    26688 hnum
 15    26672 vnum
```

That's better – we chopped the cost of *adjust* in half! It turns out that there are other complex number problems still lurking about in *fill* and *fill1*, but we'll correct those behind the scenes. Going after other game, we pick on *adjust* again. It's still the major CPU user, and hence is the area we can still probably squeeze the hardest. Here's the *getfn* results:

```
244   136209 [33] ib[;i;j]←ib[;i;j]∧⍤1 0 ∨⌿l
244    95867 [28] l←(sums=s[i;¯1+''⍴j])⌿seqs
244    61620 [32] l←((+/m)=l+.∧m)⌿l
```

The problem here is that this release of the system didn't special case $\alpha \wedge\!\ddot{\circ} k\ \omega$, but ran it through a general adverb support facility. The proper solution to this problem is to lean on the software vendor, but maybe we can do something quicker. Let's just reshape the arguments so they match, and see how that does:

```
244    96285 [28] l←(sums=s[i;¯1+''⍴j])⌿seqs
244    81575 [33] m←ib[;i;j] ◇ ib[;;j]←m∧⍉(⌽⍴m)⍴∨⌿l
244    61878 [32] l←((+/m)=l+.∧m)⌿l
```

Not bad – we knocked that one down a fair bit as well. How did we do? Well, final comparisons showed a reduction in processor time from 212177 to 80692 – a factor of 2.6 improvement in about a half hour of analysis and programming time. This sort of improvement is typical for small applications, but larger applications will take more time for changes of this magnitude – 10% of a large amount of code is still a large amount of code, and the time spent in analysis, planning, measuring, and actually changing code will be proportionately larger.

PERILOUS PITFALLS OF PROFILING

Profiling is not without its pitfalls. The Heisenberg interactions of almost any profiler can produce misleading results. For example:

- Storage required for profiler history information can alter the behavior of a system. In APL with □*fm*, this might manifest itself as either WS FULL or degraded performance.

- Real-time systems may exhibit extreme sensitivity to performance degradations caused by profilers. In the case of some tests I have run on a time-sharing APL scheduler, the profiler overhead was such that the test would *never* complete, because the timer-event-driven scheduler kept getting further and further behind in time.

- When using profilers to gain confidence about the validity of test suites, it is important not to treat profiler results as a panacea. Test suite results must be verified to assert that the program is producing correct results as well. This may seem obvious, but it is something that is often overlooked by suite authors. Regression tests should be added to test suites, to ensure that program faults which were not picked up by extant test suites have in fact been repaired, and *stay* repaired.

- Don't use profilers to improve performance until *after* ensuring that you have a proper design. Tolerating poor design on the assumption you can profile your way out of any performance problems which occur is naive and shortsighted. As a good example, a lot of the code we optimized in the Cross-Sums example is wasted effort: that code is continually recomputing numbers which could be computed once, during initialization, if a slightly different design was chosen. Instead of making it run faster, we should instead have *removed* it – the fastest code is the code that isn't there!

SUMMARY

- Profilers can assist in making dramatic performance improvements in applications.

- Performance problems occur where they will, not where you think.

- Use profilers, logging, and plots to monitor critical applications over time, to predict problems before they become crises.

- Use profilers as one more element of your Quality Assurance toolkit.

Now, go forth and multiply your productivity with profilers!

ACKNOWLEDGEMENTS

The tools and techniques described in this tutorial represent the research and development efforts of a number of APL professionals once known collectively as the APL Systems Development Department of I.P. Sharp Associates Limited. Without their ideas and devotion to excellence, this tutorial could not exist. I am further indebted to Elena Anzalone for her meticulous assistance in editing this tutorial.

Robert Bernecky

BIBLIOGRAPHY

[Be81] Berry, M.J.A., *APL and the Search for Truth: A Set of Functions to Play New Eleusis*, APL Quote-Quad, Vol. 12, No. 1, 1981.

[Bo76] Boehm, B.W., *Software Engineering, IEEE Transactions on Computers*, C-25, 1976

[Br74] Brooks, F.P., *The Mythical Man-month: Essays on Software Engineering*, Addison-Wesley, 1974.

[Da88] Dawson, K.W., private communication, 1988.

[Du71] Duby, J., *Algorithm 70, "Conway's Game 'Life' "*, APL Quote-Quad III, 2 & 3, 1971.

[Ib87] *APL2 Programming: Using the Supplied Routines*, SH20-9233-1, IBM Corporation, 1987.

[Ke88] Kerr, D.W., private communication, 1988. AUTOTEST is a capability of the IBM S/370 Assembler, which was introduced in the mid-60's as a debugging aid, but which has not been supported on any recent operating system offered by IBM. It took a fair bit of hunting to locate anyone who knew anything about AUTOTEST, but IBM finally came through for us.

[Ma83] Martin, J., & McClure, C., *Software Maintenance: The Problem and Its Solutions*, Prentice-Hall, 1983.

[Mc88] McDonnell, E.E., *Life: Nasty, Brutish, and Short*, APL Quote-Quad, Vol. 18, No. 2, 1987. The bibliography in this paper refers to several earlier APL versions of Life.

[Pe84] Peelle, H.A., *Representing Rubik's Cube in APL*, APL Quote-Quad, Vol. 14, No. 4, 1984.

[Sh87] I.P. Sharp Associates Limited, *□fm Function Monitor Facility User Guide*, Publication Code 0842 99-1 E1, 1987.

[St85] STSC, Inc., *APL*PLUS Enhancements*, 1985.

[We88] *Webster's Ninth New Collegiate Dictionary*, Merriam-Webster Inc., Springfield, Mass., 1988.

	Release 17.0			Release 19.0		
Function	**Total Ops**	**Fn Calls**	**Ops/Call**	**Total Ops**	**Fn Calls**	**Ops/Call**
TOTAL	**49026703**	-	-	**53612664**	-	-
AFMT	14240525	15	949368	14243964	15	949597
SABI	7229089	1531	4721	7241263	1531	4729
FUNSCAN	3444068	1694	2033	3444068	1694	2033
SBREAL	*3384237*	*406*	*8335*	*8618659*	*248*	*34752*
SYNTXXA	2190024	109	20091	2145202	109	19680
SBDEAL	2134061	49288	43	1872554	48748	38
LOSYFN	1802568	1325	1360	1778102	2671	665
DBLOWUP	1257081	7018	179	1257632	7005	179
SLOPTOP	1206161	109651	11	1206315	109665	11
XRHONOPC	1020284	18415	55	1018385	18380	55
BLDCONZ	913129	38	24029	933100	38	24555

Figure 1 — SPY-ing for Performance

```
0002D0  5800 B958          00958                    L     R0,DVISVC
0002D4  5900 AE66          01E70                    C     R0,=F'4'
0002D8  4740 AA42          01A4C                    BL    LISER22
0002DC  5900 AE6A          01E74  *****              C     R0,=F'11'
0002E0  4720 AA42          01A4C  *****              BH    LISER22
0002E4  5800 B954          00954          CISFMTX1  L     R0,DVISVR
0002E8  1200                                        LTR   R0,R0
0002EA  4780 AA42          01A4C                    BZ    LISER22
```

Figure 2 — SPY-ing for Code Coverage

Robert Bernecky

COUNT	INSTRUCTION AND COMMENTS		
15000	L5J	BAL	RF,SKBL
15000		ST	7,ERPOS
15000		MVC	RTXA(LQUAL),DQUAL
15000		MVC	HCV(LHDCV),HDCV
15000		XC	GTXA(GDAZ-GTXA),GTXA
15000		BAL	RF,NUM
15000		LPR	1,1
15000		ST	1,N
25000	L5F	MVC	FC(1),SBRHO+4-WS(7)
25000		LA	1,AFCZ
25000		LCR	2,1
260000	*L5H*	*BCTR*	*1,0*
260000		*CLC*	*0(1,1),FC*
260000		*BH*	*LQPX*
250000		*BNE*	*L5H*
15000		MVC	FCF,AFCFZ-AFCZ(1)

Figure 3 — SPY-ing on □*fmt*

SPY WROTE SPYOUT DATASET AT 1989-01-21 19:00:34
Total distinct addresses executed 469
TOTAL OPS = 5552816
Document= AFMT

SPY WROTE SPYOUT DATASET AT 1989-01-22 02:54:56
Total Distinct addresses executed 458
TOTAL OPS= 4323199
Document= AFMT

Figure 4 — SPY-ing on □*fmt*

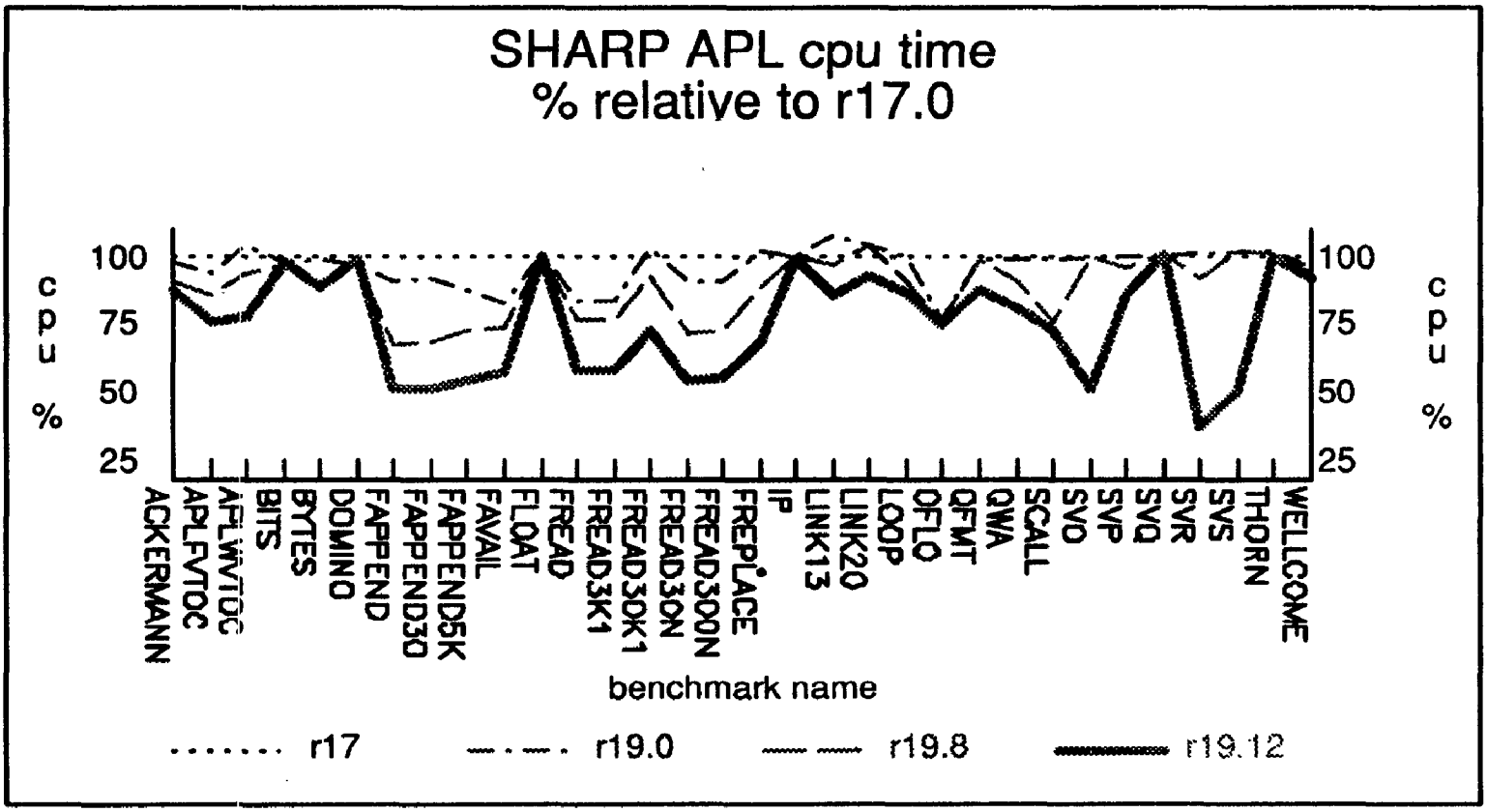

Figure 5 — Release 19.12 Performance

Robert Bernecky