Aprobe:
A Framework for Non-intrusive Software Instrumentation

Oliver Cole
President and Founder
OC Systems, Inc.

 www.ocsystems.com

Table of Contents

Instrumenting software applications

"Software instrumentation" usually refers to the chunks of code that developers insert in an application to record the values of function parameters, timing statistics, and other information necessary to debug and tune the application. These chunks of instrumentation code are not meant to be part of the finished application: their purpose is to debug the application, to find bottlenecks, and to solve similar problems.

Experienced development teams systematically add instrumentation code to their application while the application is being designed and written. Compile-time flags keep the instrumentation code out of the finished executables and support libraries. Run-time flags turn specific software instruments on and off. But that code is not always sufficient to debug every problem—particularly those issues that arise as a result of integration with other systems.

Adding software instrumentation to an application late in its development cycle can be quite disruptive to a project. Adding new source code expands the size of the machine-code libraries, which can cause transient problems to temporarily "disappear." You may need to create a separate build for instrumentation to allow groups of developers and testers to troubleshoot while other groups continue their normal work. If so, you then need to keep the two builds synchronized and do dual maintenance.

And if you are trying to solve a problem in a production system at a customer site, you may have the near-impossible task of duplicating the customer's production environment.

Aprobe technology provides a cleaner approach to software instrumentation.

Non-invasive, probe-based instrumentation

Aprobe is a patented software instrumentation framework that lets you add instrumentation in the form of "probes" to applications. It is designed for applications in the post-development environment but can be used in any stage of the software development lifecycle, from development through production.

The source files of probes are not part of the application's source files. The machine-language version of probes resides in special-purpose libraries, not the application's libraries. Aprobe inserts calls to probes into your application at runtime, while the application is in memory. The probes then execute as an integral part of the application.

You can access all parts of the application, including third-party code, shared libraries, dynamic components, Java Virtual Machines, compilers, application servers, browsers, and so on. The latest Aprobe version, Aprobe 5, even allows you to instrument the Linux Kernel!

Aprobe-based software instrumentation does not disrupt development projects or even production systems running at your customer's far-away site:

- You don't need to change any of the application's files stored on disk.
- You don't need access to source or object code; Aprobe can work with the actual delivered system software.
- You don't need to recompile or rebuild the application.
- You don't need to change how you start your application.
- With Aprobe 5, you don't need to restart the application.
- You can enable or disable probes dynamically, in real time, as the application runs.
- You can log results to memory or memory-mapped disk, using Aprobe's fast logging routines (described later in this document). This means that your application will be minimally slowed by the need to log results.

CASE STUDY

Tracking down an elusive memory leak

Project: NERC (New En Route Centre) is a major Air Traffic System for National Air Traffic Services in Hampshire, England. NERC provides en route air traffic control for all aircraft in UK airspace: a system that demands the utmost in reliability and response time.

Problem: NERC purchased a commercial X server to support its workstations. But its performance seemed sluggish. The X-server vendor blamed the center's in-house applications for poor code. Caught in a classic finger-pointing scenario, the developers needed to pinpoint the actual cause of the slowdown.

Solution: NERC's developers used Aprobe to verify how well the X server was operating. Their probes quickly revealed that the X server was spending 90 per cent of its time in an inefficient buffer allocate and de-allocate routine. Further probes showed that the buffers maintained by the vendor's X server were terribly fragmented.

OC Systems consultants wrote probes to replace the vendor's faulty buffer routine with a much cleaner one. CPU usage went down dramatically: the problem was fixed. NERC supplied the X-server vendor with indisputable evidence of the problem and the code to fix it. The vendor was able to repair the problem and incorporate the fix into their next build.

Remarks: NERC's developers left the probes in their system to fix the problem until they installed the new X server. Aprobe helped them move beyond finger-pointing to identifying and resolving the actual problem. Without access to source code, they were able to track down and fix a serious issue in commercial off-the-shelf software. ■

Aprobe is ideal for situations where your application is running in a production or production-like test environment—especially if your application is interacting with third-party software for which you do not have the source code.

Write probes that are specific to your problem

To use Aprobe, you need problem-specific probes written either in C or Java. Aprobe comes with a standard set of probes that perform tasks such as tracing, timing, detecting memory leaks, and logging data. You can use those probes right out of the box, modify them, or write your own probes.

These probes must specify the data to be collected or define the changes to the program's execution. If you are using C, the probes also specify where in the application these probes will be inserted. (If you are using Java probes, you specify the location using XML.)

Aprobe compiles the probes into machine code (or, for Java, into byte code). These machine-language probes are then inserted into the RAM image of the application, without modifying any files of the application.

Because probes are written in the full C or Java language, they can do anything that you can do in C or Java. You could write an entire application using probes. But, more likely, your probes obtain and log crucial data about your application as it executes.

For example, you can obtain and log:

- Parameter values on entry to a function or method.
- Return values on exit.
- Values of any variables at any line (or offset) in the function.
- Data about objects, structures, queues, stacks, and other constructs.
- Data about memory allocation and de-allocation.
- Timing statistics of selected functions, methods, or transactions.

Probes can also alter the behavior of applications; for example, by triggering exceptions or creating error conditions.

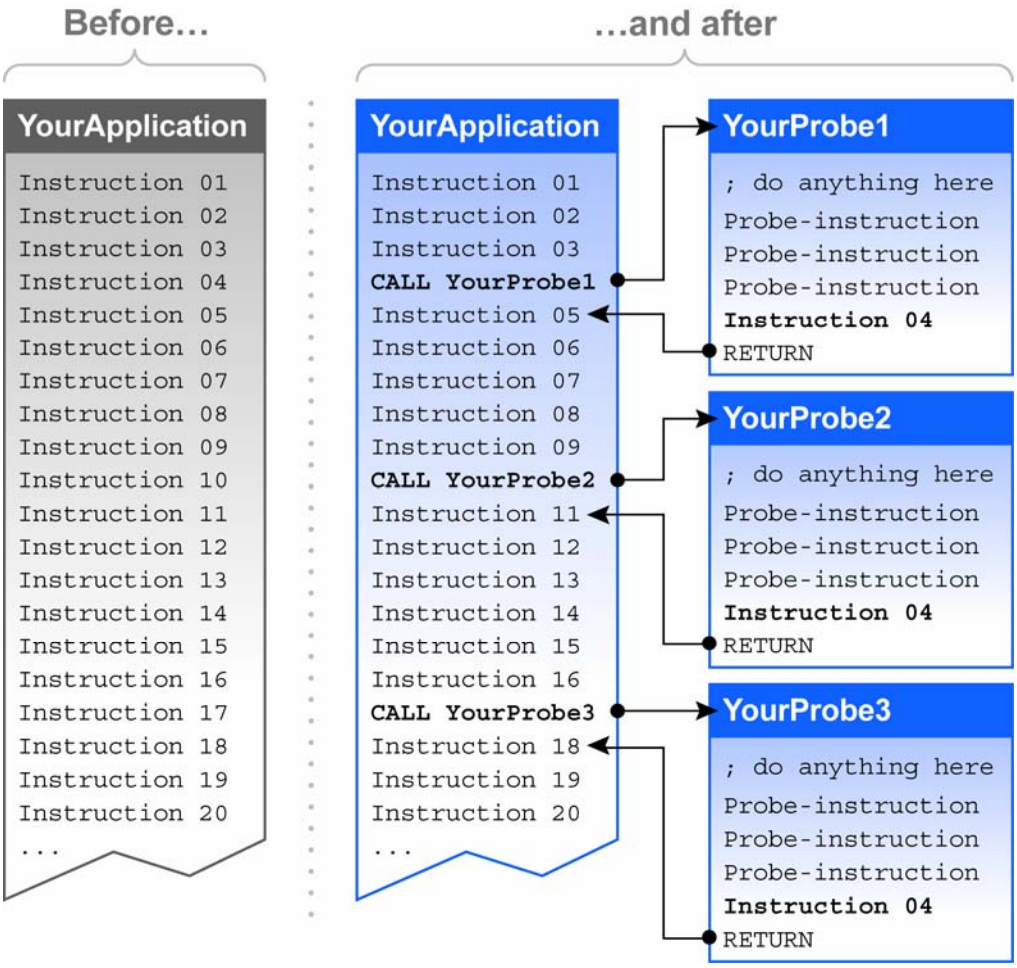
Probes can even add completely new functionality, such as integrating your application with a commercial, off the shelf product with no SDK.

OC Systems has created RootCause, RTI (RootCause Transaction Instrumentation), and other products on top of Aprobe. These software packages provide problem-specific solutions and a graphical user interface to probes written with Aprobe.

Aprobe is also used by other software vendors to extend the power of their software—and create new products—by writing probes specific to their application. It provides a powerful and flexible way to collect data about the application's execution, diagnose and repair problems in the field, and facilitate integration with other systems.

How Aprobe works

To use Aprobe, you must install it on the machine that is running the application that needs instrumentation.



Aprobe intercepts the application after it is loaded into memory, but before it starts. It then inserts calls to your probes into the application's image in memory. The calls are inserted at locations that you specify.

Aprobe never modifies the application's source files, executable files, or byte code files.

Writing your probes in C or Java

You can write your probes either in ANSI C (augmented by Aprobe directives) or in Java. In both cases, you can use the entire language: probes can open windows, read and write from sockets, call functions in the application directly, change the contents of buffers, get and set properties, trigger exceptions or error conditions, gather timing statistics, start threads and processes, and so forth.

CASE STUDY

Improving performance

Project: A multi-year, multi-billion-dollar project designed to create a Web-enabled system to process goods being imported into the USA.

Problem: Halfway through the schedule, bad news. The system could not handle the high throughput required. The development team had to do the impossible: improve performance at the same time as they coded the next release.

Solution: The project's prime contractor turned to OC Systems for help. OCS consultants took full responsibility for solving the performance problem. Using Aprobe-based performance tools, they integrated performance testing with the development project's daily regression testing, so no special performance testbed was needed: they were even able to use the project's change request process.

Aprobe allowed the consultants to be extremely specific in providing change requests; they even prototyped the recommended changes, then used Aprobe to try them out. The change requests were implemented as a matter of course during ongoing development. The system became faster and faster day by day, and the performance problem slowly disappeared.

Remarks: The flexibility of the Aprobe technology was key, since the development code was a complicated system that also included a lot of debugging code (which was to be removed in the final system).

In order to collect accurate performance data, the consultants had to turn off this sluggish debugging code. They used Aprobe to turn off the debugging code as well as to gather performance data. This allowed the system to go from debug mode to production mode with the flip of a software switch.

Because no separate performance testbed was required, the project saved $3 million. Even though performance tests were being run on the development system, this had no real impact on development efforts. All performance tests ran on the most recent build, so developers could work concurrently on improving performance and implementing new modules. No changes were needed to the existing test suite. Nor did any testers need to be retrained. ■

Two sample probes

Below are two sample probes written in C:

- The application-specific `probe thread` counts the number of times that a function is called.
- The application-specific `probe main` logs the count for later processing.

The application (not shown) is a Fibonacci number generator.[1]

```
probe thread
{
  int NumFibCalls = 0;

  probe "fib"
  {
    on_entry
    {
      NumFibCalls++;
    }
  }

  probe "main"
  {
    on_exit
    {
      log("For NumIterations = ", $NumIterations);
      log("the number of calls to fib = ",
         NumFibCalls);
    }
  }
}
```

Note: `$NumIterations` refers to the variable `NumIterations` in the Fibonacci application. Aprobe can reference application data using the identifier that was used in the original application source code.

Probes can do anything

Because probes are written in C or Java, you can write probes to do anything that these languages can do, including calling functions in your own application, calling functions in third-party applications or shared applications—even examining and modifying the computer's registers.

This means you can examine or change the contents of buffers, get and set properties, trigger exceptions or error conditions, gather timing statistics, start threads and processes, and so forth.

Probes are normally small, but nothing stops you from making them large enough to add completely new functionality.

[1] The Fibonacci numbers are a series of numbers starting with 1, 1; all subsequent numbers in the series are generated by adding the two previous numbers. The first seven numbers are 1, 1, 2, 3, 5, 8, and 13. Writing a Fibonacci number generator is a programming problem familiar to several generations of computer science students.

CASE STUDY

Slashing time-to-market for a software vendor

Project: A major embedded operating system vendor wanted to extend its toolset by providing enhanced management capabilities, including rapid diagnostics and repair capabilities for running software.

Problem: Time-to-market was critical. The vendor had the ability to build its own instrumentation in-house, but wanted to launch the new capabilities as quickly as possible. The company also needed to ensure that its solution would have comprehensive functionality, high levels of reliability and full scalability.

Solution: After an exhaustive search of instrumentation technologies and tools, upper-level management made the decision to partner with OC Systems.

OCS ported Aprobe to the vendor's operating system, customizing it to work in their Eclipse-based development environment. OCS worked with the vendor to define and deliver the proper sets of instrumentation that would support their product's value proposition.

Remarks: The vendor obtained a source-code license with full intellectual property rights. The two companies shared the risk of the project through royalty-based payments.

The project was delivered successfully, allowing the vendor to shave more than 30% off the schedule and get to market quickly with a proven instrumentation solution.

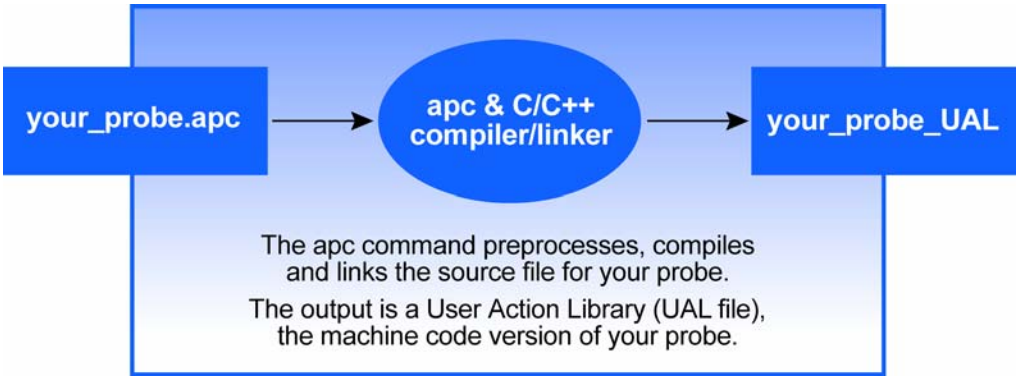
Compiling and linking probes

The C code for a probe contains several non-ANSI directives, such as:

- `probe`, `on_entry`, `on_exit`, `on_line` and `on_offset`, which specify where in the application the probe will be inserted.
- `log`, which logs data to buffers and/or log files.
- `$return`, `$myParm1`, `$myParm2`, `$$EAX`, and others which refer to return values, positional parameters, registers, and so forth. (The prepended "$" is followed by the identifier for functions, variables, and parameters in the target application.)

These directives are processed by **apc**, the Aprobe preprocessor for C. **apc** automatically generates pure ANSI C functions, and translates the directives to calls to the Aprobe API from your probes.

Your probes (now in the form of ANSI C functions) are then compiled by a standard C compiler, linked, and stored in a library called the User Action Library (UAL file).



The UAL file is implemented as a DLL on Windows and as a shared library on Unix/Linux.

In effect, the one or more probes (patches) that you write in C are translated into a shared library. That shared library contains not only the probes, but code that specifies where in the application to insert each probe.

For example, in the Fibonacci probes listed earlier in this document:

```
probe "fib"            // Insert the probe into function fib...
{
  on_entry           // ...at the first instruction.
  {
    NumFibCalls++;   // This is the actual probe to be inserted.
  }
}
```

CASE STUDY

Injecting faults to achieve comprehensive testing

Project: To achieve an extremely high level of software quality, the U.S. Federal Aviation Administration (FAA) requires the testing of thousands of modules created by different development teams.

Problem: To fully test an application, it must be tested under all possible conditions, including error conditions. But simulating errors like a disk offline or a disk broken can be difficult. The typical approach is to modify the source code temporarily before testing. However, this is so labor-intensive it is most often only done once. For this project, the FAA wanted a higher level of quality assurance, and asked the contractor to find a way to test many different error conditions.

Solution: The contractor chose Aprobe technology, since it can "spoof" a system to believe that any error has already occurred. Sometimes this involved using Aprobe to stub out the execution of specified methods and to return an error code instead. In other cases, the contractor used Aprobe to throw an exception. No application changes were necessary for these error tests. The fault injection process was so straightforward that it was added to the regression testing process and run on every build.

Remarks: Aprobe allowed the contractor to achieve comprehensive error testing. The quality of the final code was demonstrably improved.

Starting your application with Aprobe

Aprobe intercepts your application after it has been loaded into memory and before it starts executing. Under Windows, we use a device driver to implement this. Under Unix/Linux, we use features in the loader to get control each time a new process is created.

Or you can execute the **`aprobe`** command directly from the command line:

```
aprobe -u your_UAL_file.dll your_application
```

The above **`aprobe`** command will:

- Load your application into memory.
- Insert (into the memory-resident application) calls to the probes stored in your UAL file.

Then **`aprobe`** "goes away" and your application runs normally—except that it executes calls to the probes.

The **`aprobe`** command syntax allows you to specify parameters for your application, parameters for your UAL file, the number and size of Aprobe's log files, and so forth.

Inserting probes into your application

The **`apc`** command translates each C-based probe into an ANSI C function. For example, in the Fibonacci probe earlier in this document, **`apc`** generates a C function that implements the body of the probe for function `fib`, compiles it with your C compiler, and stores the resulting machine-language function in a User Action Library (that is, in a DLL or shared library).

```
probe "fib"           // This directive and
{
  on_entry            // ...this directive specify where to insert CALL.
  {
    NumFibCalls++;    // Body of probe is converted to ANSI C function.
                      // C function is compiled into a machine-code
                      // function that is the target of the CALL.
  }
}
```

When your application is loaded into memory, but before it executes, Aprobe inserts the machine-language equivalent of CALL statements into the locations specified by the probes that you wrote.

Each machine language CALL executes the body of the probe by calling the machine-language function that was previously compiled-and-linked from the probe's source code.

What happens to the instruction that we replaced with the CALL instruction? It becomes the last (or almost the last) instruction at the end of the C function.

Because Aprobe uses function calls, the size of your application does not increase.

CASE STUDY

Performing remote debugging at user sites

Project: A major contractor created a widely-distributed system for the U.S. Department of Defense designed to assess military readiness for a variety of emergency situations.

Problem: To remain effective, this system needed close to 100 percent uptime. But the budget wouldn't cover flying senior support engineers to multiple sites to track down every bug that appeared during operational test.

Solution: The contractor was already using Aprobe technology to find bugs in its integration testing lab. They soon realized its power could be extended to remote debugging.

Probes were defined by the contractor in the contractor's test lab, then sent by e-mail or ftp to user sites. A technician at each site loaded the probes into the Aprobe directory. As the application ran, trace data was logged. The trace captured code-level, system-level and hardware/software configuration details, minimizing the data each site had to supply manually.

At any point, the trace data could be e-mailed back to support staff, who would step through the trace. This helped them zero in on bugs quickly.

They would also use Aprobe to create a temporary patch to test a fix. When the fix worked, it could be left in place until the next build was ready.

Remarks: The contractor was able to debug the problem in the customer's environment without burdening the customer. Doing remote debugging avoided the high cost and delays of sending senior support staff to perform on-site troubleshooting. The probes had virtually no impact on system performance, so they could be safely left in the system should they be needed for future debugging. These application-specific probes are being used throughout the life of the military system to ensure rapid time-to-resolution of any issues.

Working with Java applications

If your application is written in Java, then so are your probes. The process of writing and deploying Java probes is essentially the same as with C probes. The most important differences are:

- You create Java probes by extending a class supplied with Aprobe.
- You must write XML-based deployment descriptors that specify the methods to be probed. Here is an example:

```
<probe_deployment>
  <probe class="YourProbe">                    // name of the probe.
    <target value="YourClass::yourMethod()"> // name of the method
                                               //   to be probed.
  </probe>
</probe_deployment>
```

Reporting the logged data

Probes usually report results by logging data. Aprobe provides the **`log`** directive to support logging. For example:

```
log("For NumIterations = ", $NumIterations);
log("the number of calls to fib = ", NumFibCalls);
```

Aprobe uses sophisticated logging mechanisms that provide very fast logging:

- All logging is done at full memory speed: Aprobe memory-maps the log files.
- You can configure the maximum size and number of log files.
- You can configure whether the log files will wrap.[2]
- We use a fast algorithm to prevent multiple threads from simultaneously accessing the logging code.[3]
- Logged data is stored in a proprietary binary format for performance reasons. You can use the **`apformat`** command to convert the binary data to ASCII. If necessary, you can then also use grep, perl, or third-party reporting software to further manipulate the logged data.

[2] If wrapping is enabled, logging can continue even after the maximum size of the log file is reached: the newest logged data replaces the oldest logged data in the log file.

[3] All of Aprobe is "thread-safe." That means that Aprobe's logging code (and all other parts of Aprobe) cannot be executed simultaneously by multiple threads. But we do not use thread locks (blocking algorithms) to protect data. Instead, we have carefully designed all of our stacks, queues, and other abstract data types to use faster, non-blocking algorithms (implemented using the CompareAndSwap machine instruction).

Calling the Aprobe API directly

As explained earlier, the **apc** command translates Aprobe directives into calls to the Aprobe API. You probably won't need to call the Aprobe API directly. But if you do, the API is fully documented (and available at www.ocsystems.com).

The API allows you to manipulate module IDs, manipulate function line numbers and code offsets, obtain tracebacks, manage threads and processes, insert probes, manage the logs, manage the UAL files, and inject numerous other useful utilities for software instrumentation.

There are separate APIs for Java and for C.

How to find out more

To learn more about Aprobe, go to www.ocsystems.com, where you will find more information, sample probes and full documentation for Aprobe. You can also arrange a personalized online demo over the Internet.

Or you can contact me, Oliver Cole, directly at +1 (703) 359-8160, or oec@ocsystems.com.

About OC Systems

OC Systems provides software tools, development environments, and services that help organizations maximize software quality and application availability for critical applications.

Founded in 1983, OC Systems originally developed compilers and other custom solutions for its clients.

In the mid-1990s, OC Systems evolved into a products company, first offering an integrated Ada development environment called PowerAda. Then it introduced Aprobe, the software instrumentation technology addressed in this white paper.

In 2001, the company launched RootCause, an application internals management tool that runs on top of Aprobe.

The RootCause Transaction Instrumentation (RTI) family of products provides accurate measures of response time for actual end-user transactions. The first RTI product was introduced in 2007. RTI for Internet Explorer delivers transaction response time for transactions that begin in the browser. RTI for ITCAM, launched in 2009, extends IBM's ITCAM RTT product for monitoring past the edge of the enterprise, all the way to the end user.

Clients include Lockheed Martin, IBM, Intel, SAIC, Sandia National Laboratories, Sun Microsystems, the U.S. Army, Northrop Grumman, Unisys, and SAS.

• • •