Program Execution Monitors (Analyzers) / Program Optimizers

(An Example SW Monitor)

ECE 560 Handout

(L#2)

There are many reasons for monitoring the execution of a program. Some of these are the following:

- *Tracing*: To find the execution path of a program.
- *Timing*: To find the time spent in various modules of the program.
- *Tuning*: To find the most frequent or most time-consuming sections of the code.

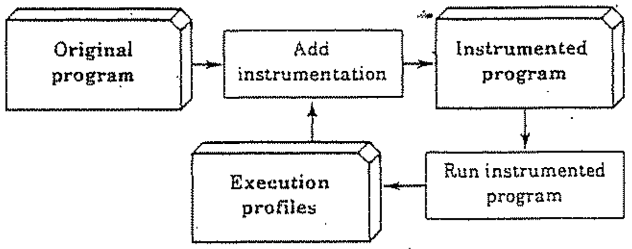

FIGURE 8.1 Steps in program execution monitoring.

- *Assertion Checking*: To verify the relationships assumed among the variables of a program.
- *Coverage Analysis*: To determine the adequacy of a test run.

Notice that not all applications of program execution monitors are related to the program's performance, although that may be the most common use of these monitors.

The programs to be monitored and improved should be chosen based on a number of criteria. The first criterion is the time criticality. Some programs are very time critical, and it is important to find out where the time is being spent so that the response can be improved. The second criterion is frequency of use. Programs used with high frequency should be optimized first. Finally, programs consuming the highest percentage of resources should be optimized. The resources include CPU time, I/O time, or elapsed (people) time. The most expensive resource should be optimized first. With the decreasing cost of computing resources, the people time is becoming the most expensive resource and may need to be optimized first.

Figure 8.1 shows the typical steps involved in program execution monitoring. First, instrumentation (or hooks) are added to the target program. The instrumented program is then run under the control of the execution monitor. Finally, the reports generated by the monitor are examined. Often the procedure is repeated several times and new instrumentation is added as more information about the execution profile of the program is obtained.

8.1.1 Issues in Designing a Program Execution Monitor

In designing an execution monitor, most of the issues to be considered are similar to those discussed in Section 7.3.1 on software monitor design. In addition, there are a number of issues that are specific to program execution monitors. These issues are as follows:

1. *Measurement Unit*: The execution monitor divides the program into smaller measurement units such as modules, subroutines, high-level language statements, or machine instructions. The data related to the execu-

(over)→

tion of each unit is recorded and shown in the final report. The lower the level of the unit, the more the overhead of monitoring. Lower level reports (such as machine instruction execution profiles) may be too detailed for some applications. Some monitors use higher level language statements, such as COBOL or PL/I statements, as a measurement unit, but then they also become language dependent. As a result, a program written in a mix of languages may not be correctly observed by such monitors.

2. *Measurement Technique*: The two basic measurement techniques are tracing and sampling. Tracing can be performed by using either explicit hooks such as trap instructions or by the trace mode of the processor. Using the trace mode produces too much unwanted data and is suitable only for monitors operating at machine instruction level. The sampling monitors make use of the system timer facilities and record the program states at periodic intervals. The interval may be specified in terms of CPU time or in terms of elapsed time. If CPU time sampling is used, the program is always found in the execution state. On the other hand, if elasped time sampling is used, the program may be in a wait state, waiting for I/O completion or for some other event.

3. *Instrumentation Mechanism*: A program has to be compiled and linked before it can be executed. The instrumentation can be added before compilation, during compilation, after compilation (before linking), or during run time. In other words, a program can be instrumented by augmenting the source code, the compiler-generated object code, the run time environment, the operating system, or the hardware. Often, a combination of these techniques is used. Source code instrumentation requires the addition of high-level procedure call statements at strategic locations in the program. The call statements transfer control to the monitor routines, which collect the data. Run time instrumentation is accomplished by adding a sampling monitor to the run time environment of the program.

   Execution monitors may also make use of additional information. For example, the link-edit map produced by the linker is an excellent source of symbol-to-address map that is used by most execution monitors.

4. *Profile Report*: Most program monitors produce an execution profile showing a frequency and time histogram. For large programs, several summaries at different levels of hierarchy may be presented; for example, summaries by modules, and then for each module by procedures, and for each procedure by statements. The procedure profiles may distinguish between resources used directly by a procedure and those used by a subprocedure that was invoked by it. For CPU time, these resources are called **self-time** and **inherited time**, respectively. Many monitors have the ability to limit or expand (zoom in or zoom out) the amount of detail.