Design and use of a program execution analyzer

by L. R. Power

Execution analyzers are used to improve the performance of programs, operating systems, and hardware systems. This paper presents a general overview of these tools, especially those designed for use by application programmers. The design tradeoffs of a wide variety of execution analyzers are examined. In addition, the design and use of a new execution analyzer are presented; its purpose is to assist in the optimization of highly modular PL/I programs.

An execution analyzer is a tool for measuring specific details about the execution of programs, for example, pinpointing where a program spends its time, identifying what program paths are actually executed, or identifying what subroutines are called. An execution analyzer can reveal surprising facts about a program that a programmer can use to improve the program, sometimes dramatically.

This paper discusses the design of a program execution analyzer. A general definition of this class of software tools is presented, and a wide variety of techniques used by existing analyzers are surveyed. In addition, the design and use of a new program execution analyzer that we call the Experimental PL/I Execution Analyzer (EPLEA)[1] are discussed. The purpose of EPLEA is to support the optimization of execution time of highly modular PL/I programs. Experience with this tool is presented as a case study. The casual reader should gain a better understanding of the value and use of execution analyzers. The general-purpose architecture and survey of techniques should benefit anyone designing a special-purpose execution analyzer or evaluating existing analyzers. The case study should be of particular interest to PL/I programmers.

Although the primary intent of this paper is to discuss execution analyzers for use by application programmers, many of the design tradeoffs discussed apply as well to operating system performance tools and hardware monitors. Measurement tools for operating systems are readily available[2-10] and have received good coverage in the literature.[11-15] Hardware monitors have received less attention and will be mentioned briefly. Many more program execution analyzers have been developed than are mentioned here. This is certainly a testament to their usefulness, but it is also a result of the wide variety of user requirements, programming languages, and system dependencies that they must deal with. In fact, it is the need for a unique measurement tool that has attracted the author to this area of research.

Four rather different execution analyzers have been selected as examples for more detailed discussions. Two of them, TIMEIT[16] and the PL/I Execution Analyzer (PLEA),[17] periodically interrupt a program and record its current location. TIMEIT works for any program, whereas PLEA is specialized for PL/I programs. The Dynamic Analyzer (DYNA)[18,19] and XICOUNT[20] trace a program's execution, DYNA by modifying its source code and XICOUNT by means of a hardware assist. Each of these execution analyzers reports its results in a different manner, and each is tailored to a different use.

Each of these execution analyzers reports its results in a different manner, and each is tailored to a different use.

What is an execution analyzer?

In this section we characterize, in general terms, what sort of details are measured by an execution analyzer and how the requisite data are reported and used. As the name implies, an execution analyzer involves both program *execution* and *analysis* of data or code.

Programs have both *static* and *dynamic* properties. An execution analyzer is a mechanism for measuring a specific set of dynamic properties of an executing program. Static properties include program size, external references, static link-edit map (as generated by the OS/VS Linkage Editor[21] or the CMS LOAD command[22]), and static call graph.[23] Dynamic properties include CPU and storage usage, I/O demand, paging behavior, and dynamic call graph.[24] There is no general way to predict the dynamic properties of a program, which can be measured only by actually executing the program. Static analysis lacks the necessary performance data, such as loop iteration counts, and even the most careful estimates are often inaccurate.

Execution analyzers divide the execution of a program into such smaller units as statements, procedures, or user tasks, and extract and report measurements on these smaller units. Obtaining measurements may itself require a rather detailed analysis of the program and its run-time environment.[25] For example, it can be quite difficult to determine which procedure has control at each point during the execution of a large program. An execution analyzer often performs data reduction and analysis to enhance the usefulness of its reports. Here, for example, in addition to a detailed statement-by-statement report, an execution analyzer may aggregate data by procedure names to produce a shorter summary report.

The output of an execution analyzer is a report called an *execution profile*. Typically in the form of a table or graph, an execution profile is designed for interpretation by a programmer. Its purpose is to assist a programmer in making changes that will improve such specific dynamic properties of the analyzed program as making the program run faster. Use of an execution profile is a discovery process similar to program debugging. A programmer studies an execution profile in conjunction with a program listing to discover something that was previously unknown about the program. This knowledge is then used to tune the program, usually by making relatively small changes. For an operating system, this may mean discovering a bottleneck that can be resolved by adding additional I/O gear or by increasing data blocking. For an application program, this may mean identifying an inefficient use of storage that can be corrected by a minor algorithmic change.

There are a great variety of execution profiles. The intended use of each is reflected in the type of data it collects and its style of presentation. Figure 1 shows examples of execution profiles for TIMEIT, PLEA, DYNA, and XICOUNT. Each profile exhibits something about the execution of essentially the same program.[26] (See Appendix B for source code.) TIMEIT supports the optimization of execution time at the machine instruction level by using storage address ranges (the left column of Figure 1A) to identify segments of code. PLEA supports the optimization of execution time for PL/I programs by reporting its findings in terms of PL/I procedure names and statement numbers. DYNA supports test-

Figure 1(A) TIMEIT execution profile

```
020000-02000F  N=   0      0.00%
020010-02001F  N=   0      0.00%
020020-02002F  N=   0      0.00%
020030-02003F  N=   0      0.00%
020040-02004F  N=   0      0.00%
020050-02005F  N=   0      0.00%
020060-02006F  N=   0      0.00%
020070-02007F  N=   0      0.00%
020080-02008F  N=   0      0.00%
020090-02009F  N=   0      0.00%
0200A0-0200AF  N=   0      0.00%
0200B0-0200BF  N=   0      0.00%
0200C0-0200CF  N=   0      0.00%
0200D0-0200DF  N=   1      1.72%   **
0200E0-0200EF  N=   0      0.00%
0200F0-0200FF  N=   0      0.00%
020100-02010F  N=   0      0.00%
020110-02011F  N=   0      0.00%
020120-02012F  N=   3      5.17%   ******
020130-02013F  N=   3      5.17%   ******
020140-02014F  N=  15     25.86%   ******************************
020150-02015F  N=   6     10.34%   ************
020160-02016F  N=   3      5.17%   ******
020170-02017F  N=   4      6.89%   ********
020180-02018F  N=   3      5.17%   ******
020190-02019F  N=   4      6.89%   ********
0201A0-0201AF  N=   0      0.00%
0201B0-0201BF  N=   0      0.00%
0201C0-0201CF  N=   0      0.00%
0201D0-0201DF  N=   0      0.00%
0201E0-0201EF  N=   0      0.00%
0201F0-0201FF  N=   0      0.00%

         ...

021A00-021A0F  N=   0      0.00%
021A10-021A1F  N=   1      1.72%   **
021A20-021A2F  N=   1      1.72%   **
021A30-021A3F  N=   0      0.00%
021A40-021A4F  N=   0      0.00%
021A50-021A5F  N=   0      0.00%
021A60-021A6F  N=   0      0.00%
021A70-021A7F  N=   3      5.17%   ******
021A80-021A8F  N=   2      3.44%   ****
021A90-021A9F  N=   1      1.72%   **
021AA0-021AAF  N=   0      0.00%
021AB0-021ABF  N=   0      0.00%
021AC0-021ACF  N=   0      0.00%
021AD0-021ADF  N=   0      0.00%
021AE0-021AEF  N=   0      0.00%
021AF0-021AFF  N=   0      0.00%
021B00-021B0F  N=   0      0.00%
021B10-021B1F  N=   0      0.00%
021B20-021B2F  N=   0      0.00%
021B30-021B3F  N=   0      0.00%
021B40-021B4F  N=   0      0.00%
021B50-021B5F  N=   0      0.00%
021B60-021B6F  N=   0      0.00%
021B70-021B7F  N=   0      0.00%
021B80-021B8F  N=   0      0.00%
021B90-021B9F  N=   0      0.00%
021BA0-021BAF  N=   0      0.00%
021BB0-021BBF  N=   2      3.44%   ****
021BC0-021BCF  N=   0      0.00%
021BD0-021BDF  N=   0      0.00%
021BE0-021BEF  N=   0      0.00%
021BF0-021BFF  N=   0      0.00%
```

Figure 1(B) PLEA execution profile

```
Statement trap counts for main procedure PRIME

                            Percent      Percent
Statement     Trap          of this      of total
 number       count         procedure    program

    3           1             1.8          1.8
    5          19            35.1         35.1       ****************
    6          23            42.5         42.5       ********************
    8           3             5.5          5.5       **
   10           1             1.8          1.8
   11           7            12.9         12.9       ******

Interrupts for this procedure     54

This procedure consumed 100.0 percent of the total CPU time
```

Figure 1(C) DYNA execution profile

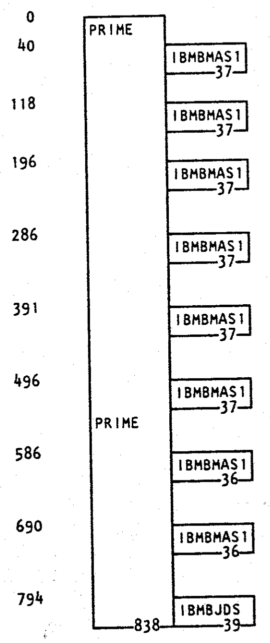
```
Line                                                               Monitor counts
number                       Source                              Total  Conditional

   1  C     Count the prime numbers between 1 and 10000.
   2  C     Count primes 1, 2, and 3.
   3        N=3                                                      1
   4  C     Start with 5 for the rest.
   5        DO 1 I=5,10000,2                                         1   executed:  4998
   6           A = I                                              4998
   7           A = SQRT(A)                                        4998
   8           K = A                                              4998
   9           DO 2 J=3,K,2                                       4998   executed: 55960
  10              IF (I-I/J*J) 1,1,2                             55960    label 1:      0
                                                                          label 1:   3771
                                                                          label 2:  52189
  11      2    CONTINUE                                          52189
  12  C        Count the primes.
  13           N = N+1                                            1227
  14      1 CONTINUE                                              4998
  15        WRITE (6,6) N                                            1
  16      6 FORMAT (I6,35H Prime numbers between 1 and 10000.)
  17        STOP                                                     1
  18        END
```

ing of FORTRAN programs by producing an annotated source listing with statement execution counts and conditional branch counts appended on the right. XICOUNT supports measurement of subroutine path lengths by reporting actual instruction counts grouped by subroutine calls, where each call is represented by a box.

A design for an execution analyzer

The outline of a hypothetical general-purpose execution analyzer is shown in Figure 2. We use this design to further describe the four analyzers just mentioned. The design has three component functions that are performed in the following order.

Figure 1(D) XICOUNT execution profile

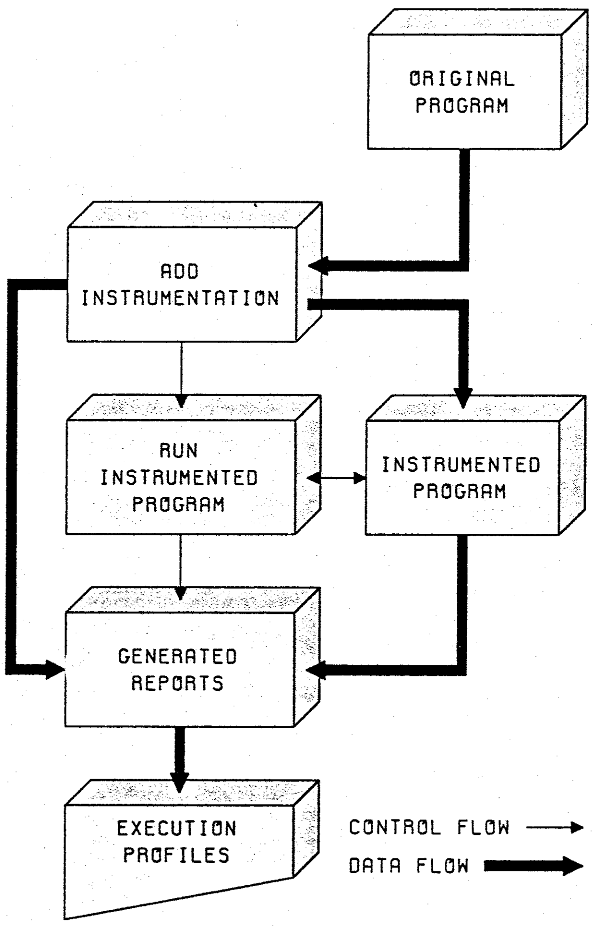


Routine	Address	Number of calls	Total instructions		Instructions at level 1	
PRIME	020058	1	838	100.0%	505	60.3%
IBMBJDS	0213D8	1	39	4.7%	39	4.7%
IBMBMAS1	0219F0	8	294	35.1%	294	35.1%
						100.0%

First, instrumentation is added to the original target program. The instrumented program is then run under the control of the execution analyzer. Finally, a report generator creates an execution profile. Designing a particular execution analyzer involves designing two pieces, the execution profile and the instrumentation mechanism. The execution profile addresses which dynamic properties are to be measured, and the instrumentation mechanism addresses how to get the measurement data. In practice, it is necessary to balance these two views. The details and intricacies of programming languages and operating systems often make it difficult to obtain precisely the desired data.

Model of program execution. Each execution analyzer is designed to serve a specialized activity (e.g., program testing, program optimization, system tuning, or hardware design) and must operate within a particular set of system, hardware, and programming language constraints. To this end, each analyzer presents to its users an abstracted model of program execution and uses a measurement technique appropriate to that model. TIMEIT and XICOUNT view program execution at the machine instruction level. In addition, XICOUNT recognizes subroutine calls based on standard linkage conventions.[27] PLEA and DYNA view program execution at the level of high-level language statements for PL/I

Figure 2 Design for an execution analyzer



and FORTRAN, respectively. System performance tools present a model of execution based on the state of user tasks and system resources (e.g., problem programs, system program, I/O devices, memory systems, etc.).[11]

The design or use of any particular execution analyzer requires an understanding of its underlying model of program execution and its measurement technique. The usability of an execution analyzer is determined largely by how natural and convenient these appear to its users. For example, one expects a PL/I programmer to prefer PLEA and an assembly language programmer to prefer TIMEIT because the different terms and concepts used by each are familiar to these respective users.

Measurement unit. In the general case, the execution of a program can be modeled as a conveniently defined, ordered sequence of *execution states.* TIMEIT and XICOUNT use a sequence of machine instructions. PLEA and DYNA use a sequence of high-level language statements. The model for a system performance tool may have several kinds of

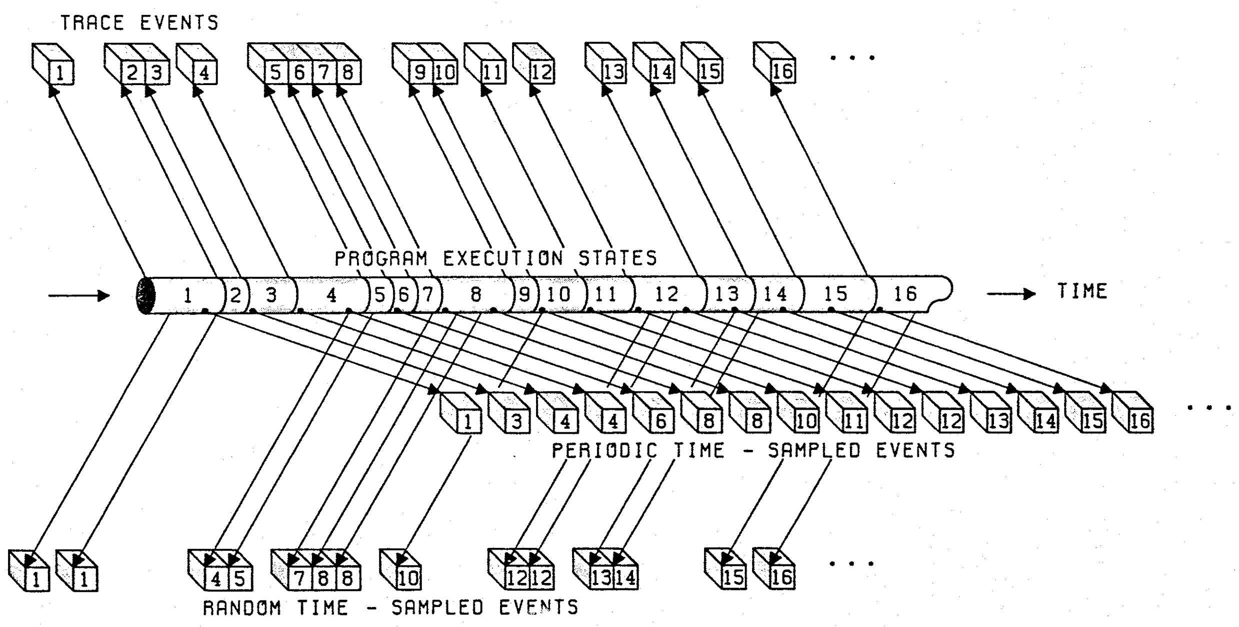
For tracing, each and every transition from one execution state to the next is captured by the execution analyzer as a measurement event.

execution states such as "device active or idle" and "CPU active or idle." Individual execution states have identifying attributes such as instruction location for TIMEIT, and statement number and procedure name for PLEA. Although the sequence of execution states occurs over time, it does not by itself provide a measure of time or resource utilization.

Execution states, along with their attributes, are used as *measurement units,* usually directly but sometimes in groups. PLEA, DYNA, and XICOUNT use their execution states directly for measure units. TIMEIT collects all instructions executed within a range of storage addresses, forming a single homogeneous measurement unit for a group of instructions.

Measurement techniques. Figure 3 illustrates two basic measurement techniques, *tracing* and *sampling.* There are also several varieties of sampling. Given an ordered sequence of execution states as previously discussed, the measurement technique defines an associated sequence of *measurement events* that are identified by their measurement units. This sequence comprises the measurement data produced by an execution analyzer.

Figure 3 Tracing versus sampling



For tracing, each and every transition from one execution state to the next is captured by the execution analyzer as a measurement event. This gives a complete trace of the executing program. Tracing is, consequently, especially well suited to program testing and debugging. DYNA and XICOUNT both use tracing as their measurement technique. Although tracing program execution by itself provides only an approximate measure of execution time (e.g., statement execution counts), it can be supplemented with actual timings to provide precise measurements of execution time. The primary disadvantages of tracing are the large amount of data and the associated processing time involved. Although a misnomer, the term *event-driven sampling* is sometimes used to describe the tracing (or counting) of important program and system events such as I/O requests, page faults, and various asynchronous activities. Most operating systems include some of these measurements as part of their accounting data.

For sampling, the transitions from one execution state to the next are ignored. Instead, the current execution state is observed (or sampled) from time to time. Each such observation becomes a measurement (or sample) event. Just which states are sampled depends on the kind of sampling. The term *time sampling* (either periodic or random) is used when there is a known relationship between time and the sampling technique. Random time sampling has useful statistical properties that guarantee that the sampling mechanism does not inadvertently synchronize with the program execution as it might with periodic time sampling. Such a synchronization, although unlikely in practice, would produce anomalous results. Sampling, as compared to tracing, generally provides less data at correspondingly lower overhead. Provided a sufficient number of sample events are obtained, sampling is a very effective technique for performance analysis. Heavily used portions of code are sampled frequently while lightly used portions are sampled infrequently or not at all. Time sampling is also used to measure the status of a program, for example, with regard to I/O waits and page waits. Within the limits of resolution of the timing mechanism, the number of sample events can be adjusted by varying the time interval between them. Time sampling is rather easy to implement in most operating systems. TIMEIT and PLEA both use periodic time sampling. The primary disadvantage of sampling is that it lacks the completeness of tracing. For performance analysis, however, this is generally a minor loss.

The instrumented program. In order to obtain measurement data, the target program must be augmented in some way. This *instrumented program* contains instruments to capture measurement data.

A program may be instrumented by augmenting source code, compiler-generated object code, the run-time environment, the operating system, or the hardware system.

It is the instrumented program that is actually executed, not the target program. This is important to note, because the instrumented program generally does not have precisely the same properties (static and dynamic) as the target program. For example, an instrumented program may be larger and take longer to execute than its target program. The degree of variation depends largely on the instrumentation mechanism.

A program may be instrumented by augmenting source code, compiler-generated object code, the run-time environment, the operating system, or the hardware system. Often a combination of these alternatives is employed. This augmentation adds counters and monitor code to collect measurement data. Other static information (e.g., procedure names) can often be taken directly from the source or object programs. TIMEIT adds a small monitor routine to the run-time environment. PLEA, in addition to adding a similar run-time monitor, uses augmented object code to obtain statement numbers.[28,29] DYNA augments the target source code by inserting counters that measure the frequency of statement executions. XICOUNT utilizes a hardware monitor facility supported by its operating system.

Event counting and execution profiles. Execution profiles are based on counting and tabulating similar measurement events. Two events are similar if they have the same value of some particular attributes. For example, PLEA counts events with the same statement numbers and PL/I procedure names. Most execution profiles produce a detailed report of the frequency of similar measurement events. Often events are counted and tabulated by several different attributes such as by I/O events and library subroutine calls. The granularity of these tabulations is determined by the underlying measurement units.

A report generator has two kinds of inputs: (1) measurement data and (2) various tables and dictionaries that associate the target program with these measurements. These latter data may be obtained by static analysis of the target program and may include information describing the target source program (e.g., FORTRAN statement numbers). Advantage can often be taken of some existing facility to greatly reduce the need for a new instrumentation mechanism. A link-edit map is an excellent example because it provides a ready-made dictionary correlating symbolic module names to storage addresses. A number of execution analyzers exploit a link-edit map or compiler listing as a source of static information.

Referring again to Figure 1, we can now state precisely what is being reported by each of these execution profiles. TIMEIT reports counts of periodic time samples localized by storage address ranges. PLEA reports counts of periodic time samples of PL/I statements identified by their statement numbers and procedure names. DYNA reports counts of traced FORTRAN statements. XICOUNT reports counts of traced machine instructions grouped by module names as obtained from a link-edit map. Finally, we look at the graphic techniques employed by these four execution profiles: TIMEIT and PLEA use histograms; DYNA augments the source program listing; XICOUNT depicts nested procedure calls by means of a tree of boxes and optionally provides a histogram (not shown).

These four examples all focus on program tracing or execution time measurement. Indeed, this is what most program execution analyzers do. Our design, however, is general enough to support other kinds of execution analyzers, such as one that traces storage usage[30] or one that samples active references to data structures. In practice, it is common for a program execution analyzer to incorporate some measurements of the operating system environment as well (e.g., I/O or paging activity).

Design alternatives

We have defined what an execution analyzer is and, on the basis of four selected execution analyzers, we have noted important variations in measurement units, measurement techniques, instrumentation mechanisms, and execution profiles. Appendix A is a table summarizing the characteristics of an expanded list of execution analyzers that are used in this section to examine more detailed design tradeoffs. Some of these analyzers are available commercially.[31-33]

To assist in this discussion, Figure 4 presents a more detailed design of the hypothetical execution analyzer shown in Figure 2. In addition to its run-time state, the instrumented program is shown with three additional pieces: *probe*, *extractor*, and *recorder*. The term probe, suggested by Cheatham[34] to designate an execution analyzer, identifies the most crucial piece of our design. The extractor and recorder are sometimes of trivial importance or are missing altogether.

The probe is added to the target program by the instrumentation mechanism. It gains control for each measurement and, in so doing, implements the measurement technique. The extractor's function is to assemble the attributes identifying each measurement event. If the measurement unit has a complex run-time structure (e.g., a PL/I statement), extracting its attributes (i.e., statement number and procedure name) is an equally complex process. All measurement events are fed to the recorder either to be added to an internal table or recorded onto permanent storage for later processing. In the case where all sampled data are maintained in storage, the recorder and the report generator may be combined.

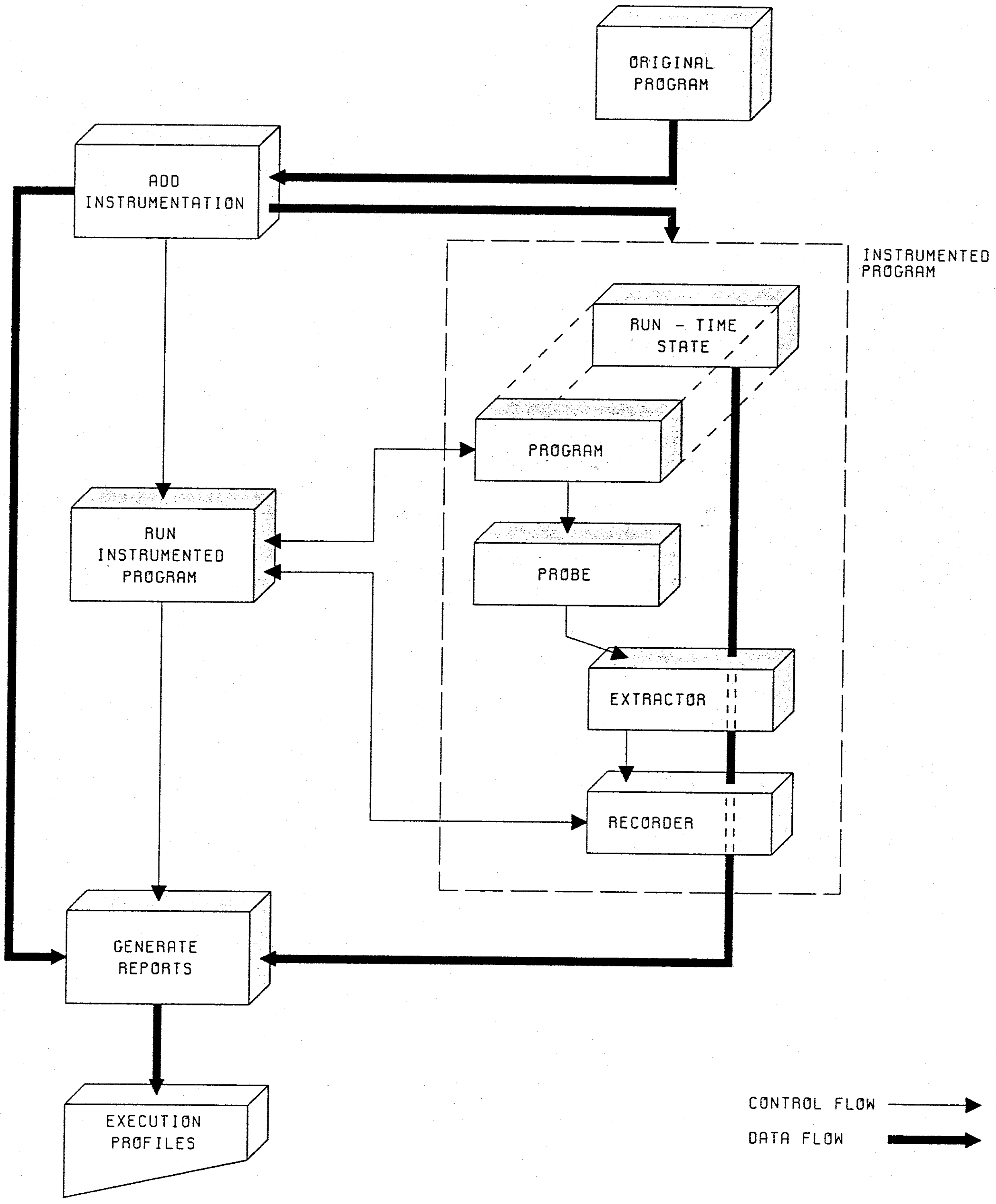
Instrumentation mechanisms. We compare three different instrumentation mechanisms: source, runtime, and hardware.

Source instrumentation. Source instrumentation is usually accomplished by a preprocessor that adds statements to the target source program. These probe statements typically define and increment counters interspersed with the target code, either at every statement or only at branch points. Since code motion can affect the semantics of these counters, compiler optimization of the instrumented code cannot be used. In some cases, this function is available as a compiler option, such as the PL/I COUNT option.[29] The instrumented program is usually considerably larger than the target program and executes more slowly. The author has observed an increase of six hundred percent in CPU time using the PL/I COUNT option. This extreme case occurs when the counters are implemented through subroutine calls. Direct in-line counting is more efficient. With source instrumentation, only one external procedure—compilable unit—is instrumented at a time. Instrumenting an entire system requires recompiling the entire system, which is often impractical. Moreover, since this mechanism is source language dependent, different preprocessors are needed for different languages and their variants. It is used with high-level languages only, and the measurement unit is usually the high-level language statement. The measurement technique is that of tracing, and the execution profile normally tabulates the count of statement executions.

With source instrumentation, only one external procedure—compilable unit—is instrumented at a time.

Since the amount of CPU time required for execution of different statements can vary considerably, this style of execution profile is not the best for performance measurements. This limitation has been addressed by several analyzers. Both FORTUNE[35] and ANATEMP[36] estimate the execution time for each statement by source analysis. Only FORTUNE, however, couples this with an execution analyzer using tracing. APAT[37] instruments the

Figure 4 Detailed design for an execution analyzer



source program so as to actually measure execution time for each statement. On some machines, the resolution of the timer is not fine enough for this technique. The distinctive advantage of source instrumentation is that the full trace of the execution of a program is of great value in testing and debugging and can report the statements that are never executed. DYNA in particular promotes this as a testing method to ensure that all program paths

There is considerable variety in run-time instrumentation with regard to measurement techniques and language and system dependencies.

have been executed during testing of a FORTRAN program. COBOLDAP[38] provides similar functions for COBOL programs.

Run-time instrumentation. Run-time instrumentation is accomplished by adding a probe monitor routine to the run-time environment of the target program. Generally the program need not be recompiled and can contain procedures written in different languages. There is considerable variety in run-time instrumentation with regard to measurement techniques and language and system dependencies. In addition to implementing the measurement technique, a run-time probe often assists in loading and initiating the target program. Both of these functions are highly system dependent.

Periodic time sampling is implemented through system-provided facilities that cause timer interrupts. METER[39] and PLEA use the MVS STIMER TASK facility,[40] and TIMEIT and SPYTIME[41] use the VM/SP virtual interval timer[42] and manage the External Interrupt PSW.[43] Use of timer facilities by the analyzer normally preempts their use by the target program. These analyzers also use the MVS LINK (or LOAD)[40] and CMS SVC 202[42] facilities, respectively, to load and initiate the target program. It is important to understand precisely how the system-provided timer facility works. For example, the VM/SP virtual interval timer cannot accurately implement periodic time sampling. Instead of causing its own interrupts, the VM/SP virtual interval timer merely adopts as its own other system interrupts, in particular those immediately following the expiration of the requested time intervals. Thus, actual time intervals vary upwards from the requested interval. In both VM/SP and MVS, timer interrupts may be postponed when they occur within supervisor routines. In spite of these timing distortions, the analyzers just discussed yield satisfactory results.

With time sampling, the time can be either *CPU time* or *real time*. At the end of each CPU time interval, the program is always observed to be executing some instruction in the program. The analyzers in the preceding paragraph use CPU time sampling. By contrast, the end of a real-time interval may find the program in a wait state (e.g., I/O wait, page wait, supervisor scheduler wait). PROGTIME,[44] Problem Program Evaluator (PPE),[32,45] and the STROBE analyzer[32] use real-time sampling. This can be implemented via the STIMER REAL facility[40] or an augmentation to the operating system. By observing the program during wait states, the PPE and STROBE analyzers can measure and report on I/O device and paging activities, in addition to normal program execution. These system-dependent measurements are useful in optimizing I/O and improving paging behavior. Since both CPU time and real time measurements can vary significantly, depending on other activities in a time sharing system, different measurements should be compared only when they are taken under comparable system loading conditions.

In general, the use of run-time instrumentation disrupts the execution of the target program less than source instrumentation. That is, it requires less additional storage, less additional CPU time, and little or no modification to the program itself. To the extent that time sampling is used and the system timer is accurate, the resulting measurements are closer to actual CPU usage. In practice, these analyzers are extremely effective in isolating the inefficient portions of a program. They are convenient to use, efficient, and above all they produce objective, hard facts about where a program spends its time.

Several analyzers have effectively combined instrumentation for both tracing and time sampling—the Optimizer III analyzer[31] and GPROF[46] are two examples. Optimizer III combines time sampling with statement tracing. The object code output by the COBOL compiler is both automatically optimized and instrumented to trace COBOL statements. By counting and measuring (via time sampling) statement executions, this analyzer exploits the strengths of both measurement techniques. GPROF also uses time sampling in conjunction with tracing. It traces all procedure invocations by calling a monitor routine at the entry to each procedure. These calls are compiler-generated. This mechanism is a form of *software hook*, typically used by debugging systems to gain control at particular statements within a program. The Software Measurement Tool (SMT)[47] is a rather general attempt at using software hooks for tracing and analyzing the execution of programs.

Hardware monitors are more commonly used for system tuning and for hardware and system design, but they have similar design tradeoffs.

Hardware instrumentation. Hardware instrumentation is accomplished by putting the probe into a separate processor. This is the least invasive mechanism, leaving the source code, object code, and run-time environment unchanged. Since this mechanism requires special hardware facilities, it is relatively uncommon for program analysis. Two rather different examples of program execution analyzers are noted. The XICOUNT analyzer uses the System/370 Program-Event Recording (PER)[43,48] hardware (via the VM/370 CP TRACE command).[49] PER is a built-in programmable hardware monitor that supports the tracing of the execution of either selected or all machine instructions. Unfortunately, tracing at the machine instruction level generates large amounts of sample data, requiring considerable processing overhead. The use of PER also slows down the machine hardware considerably. A second example of hardware instrumentation is the SPY[50] analyzer. It utilizes one of the peripheral processors of the CDC 6000 series as a probe, which in effect implements periodic real-time sampling of the target program executing in the main processor. This causes essentially no disruption of the execution of the target program.

Hardware monitors are more commonly used for system tuning and for hardware and system design, but they have similar design tradeoffs. A typical hardware monitor has an array of high-speed hardware counters (registers) that can be programmed in conjunction with boolean logic and clocks to count signals from hardware test points in the target machine. The set of test points can be selected manually to monitor a wide variety of hardware activities, for example, to identify the instruction types executed.[51,52] Much like an I/O device, the hardware monitor itself may be attached to the same or a different computer.

The System Performance Monitor (SPM)[53] is an example of a hardware monitor that has been used like a program execution analyzer (i.e., controlled by special codes from the target machine) to help improve the code generated by compilers. In addition to a hardware probe, the Hybrid Monitor System (HMS)[54] has a probe into the target machine's storage system and a set of fully programmable high-speed processors. HMS is designed for performance evaluation of complex multiprocessor operating systems.

Data extraction. Each measurement is identified by its measurement unit attributes (e.g., storage location, statement number, procedure name), which are obtained from the executing program. For source and object instrumentation, they are usually identified implicitly by the particular counter that is updated. Other instrumentation mechanisms require more complex processing. TIMEIT and SPYTIME, the simplest mechanisms, merely pick up the location counter at the point of the timer interrupt and convert it to the appropriate storage address range and place no requirements on the structure of the program's run-time environment. By contrast, PLEA uses the location counter, the call-frame stack, and the PL/I statement number table to locate the procedure name and statement number for the most recently executed PL/I statement. Incidentally, the PL/I statement number table is accurate even when code motion has occurred as a result of compiler optimization. PLEA and METER examine MVS control structures to obtain information about modules that are dynamically loaded during program execution. In addition to this information, the PPE and STROBE analyzers extract data from system control blocks concerning I/O activities and various wait states.

Poking around in language-dependent and system-dependent control structures at arbitrary times during program execution exposes an execution analyzer to the possibility of program interrupts, for example while trying to use an address pointer that has not been properly initialized. PLEA protects itself against this problem by setting up a program interrupt handler via the MVS SPIE facility.[40]

Although using the program location counter to identify the last instruction executed seems trivial, it has some serious problems. In MVS, for example, simply finding the interrupted location counter is a major undertaking. Moreover, if the last instruction executed was a branch instruction, it cannot be identified by the value of the current location counter. PLEA, assuming that branches are less frequent, always identifies the instruction immediately preceding the location counter. This causes anomalous results at heavily executed branch points. Another approximation used by TIMEIT and SPYTIME is always to identify the next instruction to be executed. This ignores the differences in execution time for different instructions (e.g., a long move versus a short move or a register load).

Data recording. Measurement events must be stored either in temporary tables or in permanent storage. If temporary tables are used, some data reduction (i.e., counting) can be done on-the-fly and later dumped to permanent storage or output directly as the execution profile. There are four tradeoffs involved here: (1) the use of additional storage during program execution, (2) the use of permanent storage, (3) the effect of additional I/O on the sampled program or the sampling technique, and (4) the ability to rerun the report generator on the same measurement data without having to rerun the execution analysis itself. If there are several execution profiles with different options, it is highly desirable to have a permanent copy of the measurement data available for rerunning the report generator. For long execution analysis runs, some on-the-fly data reduction may be needed merely to reduce permanent storage usage. Finally, certain programs may be especially sensitive to storage use or I/O activity. TIMEIT and SPYTIME both use temporary tables, from which TIMEIT produces its execution profile directly, and which SPYTIME dumps to permanent storage for later processing. PLEA and METER output one record for each measurement event and process the resulting data with a separate program. The tables of execution counts generated by DYNA, for example, are dumped to permanent storage at the end of program execution. This can be a problem if the program fails to terminate properly.

Report generator. A number of detailed execution profiles have already been presented. From our expanded list of examples, SPYTIME, METER, and TIMEMAP[55] use link-edit information for converting storage addresses to module names. TIMEMAP is actually a postprocessor for TIMEIT. METER extracts its link-edit information directly from the stored version of the load module.[21] The Optimizer III and STROBE analyzers use compiler listings to identify high-level language statements. There are two other styles of execution profiles: *summary* and *predictive*. Each requires additional processing by a report generator.

The summary profile quickly identifies the few detailed profiles that are of special interest.

Summary profiles. Execution analysis of a large program requires one or more *hierarchical* summaries, such as a summary by procedures where each procedure has its own detailed execution profile. The summary profile quickly identifies the few detailed profiles that are of special interest. Another form of summary is *structural* in nature. A structural profile distinguishes between resources used directly by a procedure (or task) and those used by subprocedures (or subtasks) that were invoked by it, either directly or indirectly. For CPU time these resources are called *self-time* and *inherited time*, respectively. The GPROF execution analyzer is designed especially to report self-time and inherited time. XICOUNT also distinguishes between self- and inherited instruction counts. Another feature of summary profiles is the ability to limit or expand the amount of detail (termed "zoom-in" and "zoom-out"). SPYTIME and XICOUNT provide control over this aspect.

Predictive reporting. After studying a particular execution profile, one might like to know the effect of making a change to the target program without actually having to implement the change. This would be done by rerunning the report generator with some additional parametric data. We call this *predictive reporting.* One known example of this is the computation of stand-alone run-time from data collected in a time-sharing environment.[11] This is done in the report generator by making certain assumptions based on an analysis of the measurement data. There is a close similarity between predictive reporting and the use by trace-driven simulators of measurement data to predict the behavior of a hypothetical machine when executing the measured program.[56]

The EPLEA execution analyzer

EPLEA was developed in support of a research project called ADAPT,[57,58] which is studying the use of data abstractions. Since data abstractions encourage a highly modular design with a superabundance of procedure calls, it is of some importance to know just how much these calls cost. EPLEA was developed to measure this overhead and to assist in the general optimization of PL/I code generated from data abstractions.

Requirements. EPLEA has several requirements not satisfied by existing execution analyzers. An individual procedure is normally optimized by making minor adjustments to its implementation or by replacing its algorithm altogether. Such changes are generally local to a procedure. Optimizing a large complex system of interrelated procedures, on the other hand, often involves changing the structure of procedural interdependencies, moving code across procedure boundaries, and eliminating some procedures altogether. To assist in this style of optimization, we need execution profiles that can summarize large numbers of procedures, show their interdependencies, and report both self-times and inherited times in a way similar to that of GPROF. Measuring the overhead involved in using procedure calls requires additional measurement attributes to identify the relevant pieces of code (e.g., procedure prologues and calling sequences). To a lesser extent, we are also interested in library subroutine, storage management, and supervisor service calls (primarily I/O). For usability, EPLEA profiles must be in terms of PL/I constructs. Ideally, EPLEA would be used to predict the effect of making a change to the target program, which requires accurate measurements of CPU usage.

Design and implementation. Since the PLEA execution analyzer comes close to satisfying the EPLEA requirements, its design was used as a basis for EPLEA.

CPU time inheritance was implemented by taking a "snap shot" of the entire call-frame stack at each timer interrupt.[59] The report generator was programmed to credit CPU time to the calling procedure and tabulate it in two ways—with respect to itself and with respect to the total program. For example, in Figure 5A, TEXTOU called TEXTOU.WRITE in statement 95. This subroutine accounts for 31.0 percent of total program execution time and 88.7 percent of the time taken to execute TEXTOU. That is, TEXTOU inherits 88.7 percent of its time from TEXTOU.WRITE.

Code-classification flags have been added to each statement to identify the underlying code as follows:

X	In-line compiled code.
P	Prologue code (i.e., procedure initialization).
C	Call/return sequence.
L	PL/I library subroutine code.
S	Supervisor service (e.g., I/O) subroutine code.
blank	Inherited from another PL/I subroutine call.

These codes are also used in all summaries. Figure 5 shows two examples of EPLEA execution profiles. Figures 5B and 1B are for the same program for comparison with the original PLEA. The L and S codes implement a special form of inherited CPU time from library and supervisor subroutines, respectively. The S code pinpoints and measures those statements responsible for I/O activity, a procedure which has proved to be effective in optimizing I/O usage. For our purpose, the sum of P and C is used as a measure of procedure call overhead.

Figure 5 (A) EPLEA execution profile for prime number program; (B) EPLEA execution profile showing inherited time

```
Traps for main procedure PRIME

                                  Percent      Percent
Statement  Trap     Trap      of this      of total
 number     count    weight   procedure    program

   1  P        1     2.0         2.4          2.4    *
   3  X        1     2.0         2.4          2.4    *
   5  X        7     2.0        16.8         16.8    ********
   5  C        1     2.0         2.4          2.4    *
   5  L        5     1.8        10.8         10.8    *****
   6  X       20     2.0        48.2         48.2    ************************
   7  X        1     2.0         2.4          2.4    *
   8  X        1     2.0         2.4          2.4    *
  11  L        3     1.3         4.8          4.8    **
  11  S        4     1.5         7.2          7.2    ***

Interrupts for this procedure       44 (1.9)

X: 72.3% (2.0)   P: 2.4% (2.0)   C: 2.4% (2.0)   L: 15.7% (1.6)   S: 7.2% (1.5)

This procedure consumed 100.0 percent of the total CPU time
```

```
Traps for secondary procedure TEXTOU

                                  Percent      Percent
Statement  Trap     Trap      of this      of total
 number     count    weight   procedure    program

  24  S        1     3.0        37.5          1.4
  90  X        2     1.5        37.5          1.4
  93  X        1     2.0        25.0          0.9
  95          58     1.1        88.7         31.0    *************** TEXTOU.WRITE

Interrupts for this procedure       4 (2.0)

X: 62.5% (1.7)   P: 0.0%   C: 0.0%   L: 0.0%   S: 37.5% (3.0)

This procedure consumed   3.9 percent of the total CPU time
```

An early version of EPLEA used periodic CPU time sampling under VM/SP. The VM/SP virtual interval timer distortions (discussed previously) exaggerated measurements for supervisor and PL/I library subroutines—depending on system load—because system interrupts are usually more frequent in this code. This problem was solved by implementing a *weighted time sampling*, which adds a weighting factor to each timer interrupt proportional to its actual CPU time interval. This factor is reported for each statement and also for summarized code classifications (in parentheses). Note in Figure 5 that supervisor and PL/I library weighting factors are generally lower than the others. Statement 24 in Figure 5B is an interesting exception in that it is an OPEN statement that is masked against timer interrupts for a relatively long period of time. When all percentages are computed on the basis of weighted time sampling, these measurements are quite accurate.

Two new summary profiles have been designed for EPLEA. With many hundreds of procedures involved in an execution profile, it is necessary to group them

Figure 6 EPLEA summary of procedure groups

```
Summary of procedure groups

Group          Percentages (by type)
name           X      P      C      L      S     Total

DMATCH        0.0    0.0    0.0    5.8   88.2    17.0   *****
EXTRACT     100.0    0.0    0.0    0.0    0.0     0.5
LINEIN        0.0  100.0    0.0    0.0    0.0     0.5
LINEOU        0.0   83.3    0.0    0.0    0.0     6.0   **
POOL        100.0    0.0    0.0    0.0    0.0     1.5
PROOFER      35.7    7.1    7.1    0.0   35.7     7.0   **
SORT          0.0    0.0    0.0    0.0    0.0     0.0
SORTMNG      10.0   90.0    0.0    0.0    0.0     5.0   *
TABLE         0.0    0.0    0.0   50.0   50.0     1.0
TEXTIN        7.6   23.0    0.0   11.5   42.3    13.0   ****
TEXTOU        9.0   31.8    0.0   13.6   40.9    11.0   ***
VMATCH       10.8    1.3    0.0    2.7   78.3    37.0   ************

Summary of trap types:  X: 12.1% (1.0)     P: 19.0% (1.1)     C: 0.5% (1.0)
                        L:  5.5% (1.0)     S: 63.0% (1.1)
```

according to some scheme. A *procedure group summary* profile supports the grouping requirements, whereby measurements for a user-defined group of procedures are reported under a single name. Figure 6 shows an example of this in which

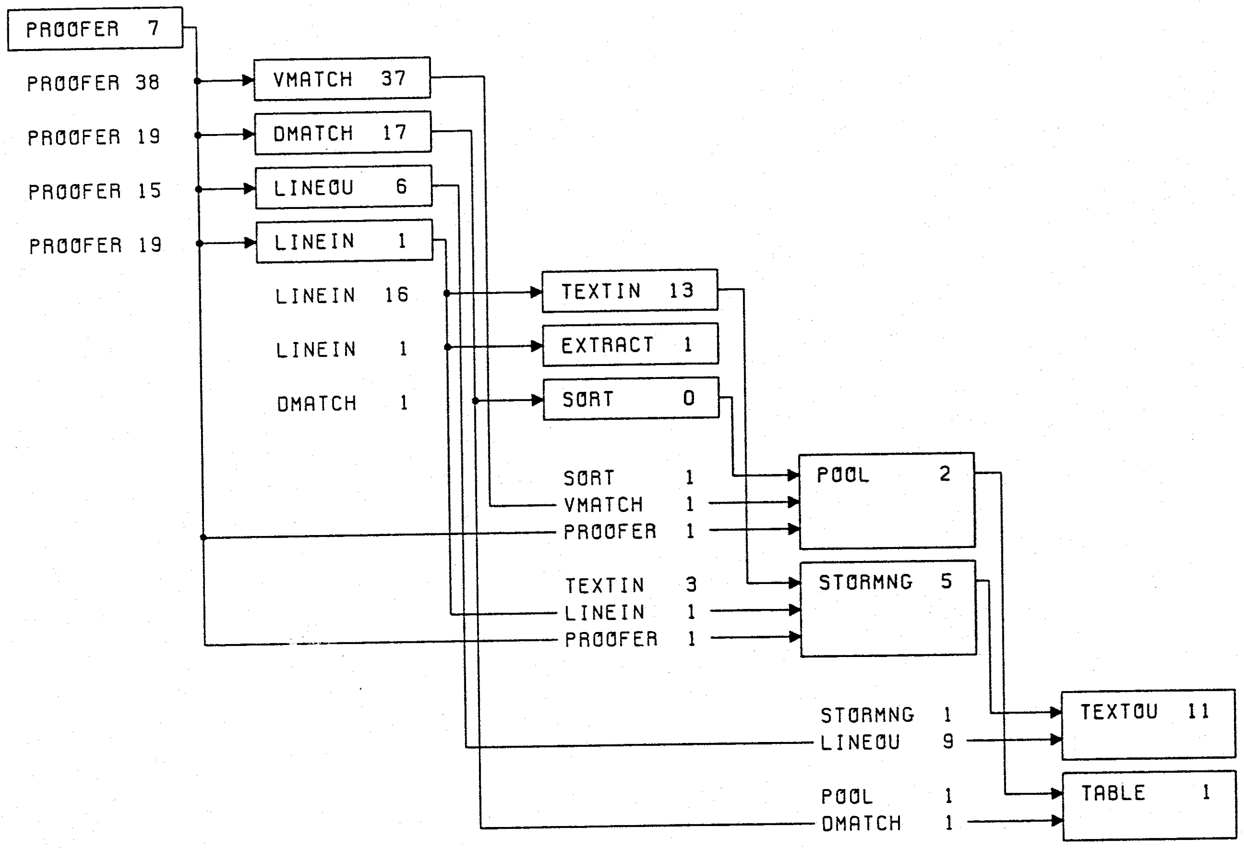
In practice, the majority of code in a large system has very little effect on its overall performance.

the names on the left represent groups of procedures. For our purposes, this feature supports the grouping of procedures by their data abstractions. The second summary profile, termed a *call dependency graph* and shown in Figure 7, depicts a dynamic call graph of the program.[60] This graph contains only those calls actually sampled by EPLEA. Each box denotes a single procedure or a group of procedures. Lines entering at the left of a box represent calls to the box, and lines leaving from the right represent calls from the box. All of the numbers on this graph are percentages. The numbers inside the boxes are self-times that (except for rounding errors) sum to 100 percent. The numbers outside the boxes are inherited times, and (except for rounding errors) the sum of the inputs to a box always equals the sum of the outputs plus the number inside the box. Each column of boxes is ordered by descending percentage of CPU usage. An important use of this graph is to report the structural dependencies between procedures, which is information not often available elsewhere.

User experience. Before presenting actual results using EPLEA, we make a few observations on the code optimization process.

Performance optimization and current practice. In current practice, questions of performance measurement and optimization often receive no attention at all. When they do, they are usually not addressed systematically. Rather, the answers often rely solely on experience and informed guessing. In the case of PL/I, this requires expert knowledge indeed,[61] involving knowledge of library subroutines, data representations, and data conversions. The rare expert PL/I programmer with this knowledge tends to apply it uniformly to all parts of a

Figure 7 EPLEA call dependency graph



program regardless of need. In practice, the majority of code in a large system has very little effect on its overall performance. Given the tradeoff between optimization and maintainability, it may not be cost-effective to optimize this code.

The use of execution analyzers can significantly alter this practice. By concentrating on producing a correct, well-written piece of software, most optimization issues can be deferred and addressed systematically later in the development cycle. As an unbiased measurement tool, an execution analyzer allows one to focus on those few areas of a piece of software that have the greatest potential for improvement. Regardless of methodology, an execution analyzer can uncover inefficiencies that often surprise even the most skilled programmer. Outlined here are the steps involved in using an execution analyzer for code optimization:

1. Execute the original program with the execution analyzer.
2. Examine the summary and detailed profiles to locate the program areas with the highest CPU usage.
3. Using the program listings, focus attention on these areas and determine whether a source code modification will improve performance. This may require an in-depth study of a few statements, possibly checking to see what the compiler is generating.
4. After implementing modifications, iterate the above steps until further improvements appear too small to be justified.

The first few iterations of this process usually accomplish the lion's share of possible improvements, usually involving only a very small fraction of the original program. Since the same program

Figure 8 (A) Fragment of EPLEA execution profile for the report generator; (B) EPLEA report generator code before optimization; (C) EPLEA report generator code after optimization

```
                                     Percent       Percent
Statement    Trap      Trap         of this       of total
 number      count     weight       procedure     program

  330  P       47       1.0           1.1           0.6
  333  X        9       1.0           0.2           0.1
  334  X       86       1.2           2.5           1.4
  335  X     1086       1.3          34.5          20.2   **********
  336  X      660       1.5          24.2          14.2   *******
  337  X      249       1.0           6.1           3.5   *
  338  X       43       1.0           1.0           0.6
  339  X      116       1.5           4.2           2.4   *
  341  X      504       1.3          16.0           9.3   ****
  342  X      157       1.6           6.1           3.6   *
  343  X        2       1.0           0.0           0.0
  350  X       11       1.0           0.2           0.1
```

```
Statement
 number

  334      DO I=0 TO LENGTH(HSTR)-2 BY 2;    /* Do for all characters */
  335        TT=(CB8(I)+M(I))*(CB8(I+1)+M(I+1));
  336        RR= MULTIPLY(TT,.0000000000001B,15,0) + MOD(TT,8192);
  337        IF RR>=8191 /* Mod too big? */ THEN
  338          DO; RR=RR-8191;
  339              IF RR>=8191 /* Mod too big? */ THEN RR=RR-8191;
  340          END;
  341        HB=BOOL(HB,RB,'0110'B);          /* Accumulate hash        */
  342      END;
```

```
Statement
 number

  334      DO I=0 TO 2*DIVIDE(NAMELGH-1,2,15,0) BY 2,/* Varying part */
                  LENGTH(NAME) TO LENGTH(HSTR)-2 BY 2:/* Fixed part   */
  335        TT=(CB8(I)+M(I))*(CB8(I+1)+M(I+1));
  336        RR= MULTIPLY(TT,.0000000000001B,15,0) + MOD(TT,8192);
  337        IF RR>=8191 /* Mod too big? */ THEN
  338          DO; RR=RR-8191;
  339              IF RR>=8191 /* Mod too big? */ THEN RR=RR-8191;
  340          END;
  341        HB=BOOL(HB,RB,'0110'B);          /* Accumulate hash        */
  342      END;
```

may behave quite differently, depending on its inputs, a variety of test runs is recommended. The total performance improvement is measured by observing total CPU time "before" and "after," without the use of the execution analyzer. Finally, execution profiles serve as documentation for the optimized program.

Use of EPLEA. As a challenging example, the author ran EPLEA against its own report generator, a PL/I program skillfully implemented according to current practice. A fragment of the resulting execution profile appears in Figure 8A. For large measurement files, the loop in statements 334–342 shown in Figure 8B used about 55 percent of the

Table 1 Details of typechecker performance improvements

	Seconds	
Algorithmic changes—10 percent improvement		
Symbol lookup routine	0.7	
Input file format	0.8	
Hash table dictionary	0.8	
TOTAL	2.3	
PL/I language usage—7 percent improvement		
(NOSUBRG)	0.5	
BIT(1) ALIGNED	0.5	
Resident transient routines	0.5	
TOTAL	1.5	
Repackaging—45 percent improvement		
Multi-ENTRYs (8)	1.4	
In-line expansions (12)	5.3	
Partial in-line expansion (1)	3.4	
TOTAL	10.1	
Procedure call overhead		
Original code	7.5	(33 percent of original CPU time)
Improved code	2.3	(27 percent of improved CPU time)

CPU time (computed by summing the values reported for those statements). This loop implements a hashing algorithm[62] over strings containing, among other things, PL/I procedure names. The body of the loop hashes two characters at a time with its iteration count fixed to a maximum length. A close examination of the body of the loop indicated that it was already highly optimized, offering no hope of further improvement. It was noticed, however, that the number of iterations through the loop could be reduced. The length of the string to be hashed had recently been increased to support a longer procedure-naming convention. Shorter names were padded with blanks before hashing. The algorithm was modified to hash only the non-blank part, making it a variable-length hashing algorithm as shown in Figure 8C. This modification reduced total CPU time by 48 percent. Although a relatively minor change, this was a real algorithmic change that could not have been done automatically. What EPLEA did was to focus attention on a small number of statements on the critical path. This experience mirrors that reported by Knuth,[44] where the speed of his execution analyzer was doubled by applying it to itself.

A larger example, making use of the unique features of EPLEA, is the optimization of the ADAPT typechecker. This is a small system of over 250 highly modular PL/I procedures written in the style of data abstractions. The iterative process just described was employed through twelve iterations over the course of one week without a detailed understanding of the original code. The improved system ran about three times faster than the original. Table 1 outlines the kinds of changes that were implemented. In each case, a concerted effort was made to apply the minimal changes to the original source code. In several cases the PL/I macro preprocessor was used to accomplish this.[63]

The salient features of this example are that the repackaging changes accounted for most of the performance improvements, and procedure call overhead was reduced by 69 percent. These results are more dramatic than those previously reported for in-line expansions.[64] A total of 21 small procedures were repackaged. Eight procedures were merely converted from secondary entry points to individual procedures. Twelve procedures were eliminated altogether, and their code was expanded

Table 2 Summary of PL/I call overhead tests

Test case	Percent					
	Overhead	X	P	C	L	S
PLEATAB	2.6	47.4	1.6	1.0	8.6	40.8
PLIDOC (without DRAW)	4.2	22.4	2.5	1.7	15.5	57.7
TIDY	8.1	57.0	5.2	2.9	6.0	28.7
LALR	11.0	42.2	8.1	2.9	34.4	12.2
GREENPRT	11.3	74.5	9.1	2.2	6.9	7.1
PROOFER	13.5	16.5	12.6	0.9	16.5	53.3
PACKPTAB	17.8	28.0	11.1	6.7	28.5	25.6
GRAPH (normal input)	22.7	27.6	20.1	2.6	25.8	23.6
ADAPT	25.0	26.5	19.8	5.2	32.7	15.6
PLOT (output of ADAPT)	43.1	26.0	32.8	17.1	22.3	1.6
GRAPH (large input)	62.5	22.8	53.5	9.0	6.8	7.6

in-line via preprocessor macros. One procedure was split into two parts, one part being expanded in-line and the remainder called as a subroutine. The author believes that this and other similar examples demonstrate the feasibility of optimizing PL/I code that is based on data abstractions.

The optimized typechecker still has about a 27 percent procedure call overhead. It this a lot? With the EPLEA execution analyzer, this is an easy question to answer. Table 2 summarizes a series of measurements made on PL/I programs readily available on the author's computing system. The range of this measured overhead is striking, from 2.6 percent to over 60 percent. The low-overhead programs have relatively simple procedure call hierarchies or are dominated by I/O. The high-overhead programs use extreme modularization or recursion. The in-between group of programs (i.e., 8–25 percent overhead) is quite diverse, including several large systems. This group probably represents the normal range for PL/I overhead as defined here.

Two other observations on these data are:

1. When a program is optimized, the percentage of overhead may very well increase, even though the program runs much faster. The actual overhead has not necessarily increased; it represents a greater proportion of a smaller total.
2. Program behavior can differ dramatically with differing inputs. The GRAPH test cases are the same program with different inputs. The program behaves quite acceptably with one input set and very inefficiently with the other.

Concluding remarks

We have presented a general-purpose design for characterizing existing execution analyzers. Several program execution analyzers have been discussed. Design tradeoffs concerning program instrumentation, data extraction and recording, and report generation have been discussed. A new execution analyzer, designed for use with data abstractions written in PL/I, has been described and its use demonstrated. It is clear that these tools can make an enormous difference in balancing performance requirements with the advantages of this new software technology. New execution analyzers will undoubtedly be built to deal with other aspects of this technology such as data representation and storage utilization.

Acknowledgments

The author is indebted to the authors of the programs discussed in this paper and has spoken with many of them about their contributions to this area. Some of these programs have not been documented previously in the form of reports or papers. They were simply programmed by their authors to solve a problem they had at the time. David A. H. Smith has been especially helpful to the author. His thoughts stimulated a number of the features of EPLEA and helped unravel the mysteries of timers. He offered a number of suggestions and encouragements on this work. The author also appreciates the thoughtful comments and suggestions of the reviewers; they greatly improved the content of this paper.

Appendix A: Features of execution analyzers

Execution analyzer	System dependency	Language dependency	Instrumentation mechanism	Measurement unit	Measurement technique	Additional inputs	Detailed profile	Profile features
TIMEIT[16]	VM/SP	None	Run-time	Machine instruction address range	CPU time sampling	None	Histogram by machine address	Combined with recorder
TIMEMAP[55]	VM/SP	None	None (Postprocessor for TIMEIT)	None	None	TIMEIT histogram; link-edit map	Histogram by machine address with module names	None
SPYTIME[41]	VM/SP	None	Run-time	Machine instruction address range	CPU time sampling	Link-edit map	Histogram by module names and offsets	Summary; rerun options; zoom
METER[39]	MVS	None	Run-time	Machine instruction in program or supervisor module	CPU time sampling	Link-edit module	Histograms by module names and offsets	Summary; rerun options; zoom
PROGTIME[44]	MVT	None	Run-time	Machine instruction address range	Real-time sampling	None	Histogram by machine address	
PPE[32,45]	OS/VS and DOS/VS family; VM/SP	None (COBOL option available)	Run-time	Machine instruction in program or supervisor; program wait states; I/O device	Real-time sampling	Link-edit module; COBOL listing optional	Histogram and graphs by module names and offsets for CPU, I/O, and paging	Summaries; supervisor routine usage; COBOL statement numbers optional
STROBE[32]	OS/VS and DOS/VS family	None	Run-time	Machine instruction in program or supervisor; program state; I/O device	Real-time sampling	Compiler listings	Chronological and usage charts, CPU by source names, I/O by file and cylinder	Summaries; supervisor routine usage; rerun options
SPY[50]	CDC 6000	None	Hardware	Machine instruction	Real-time sampling	Link-edit map	Histogram by machine addresses with module names	Summary
XICOUNT[20]	VM/SP	Standard System/370 linkage conventions	PER hardware feature	Machine instruction	Tracing	Link-edit map	Path lengths by subroutine; call graph; histogram	Summary; rerun options; zoom
PLI COUNT[29]	None	OS PL/I compiler	Object	PL/I statement	Tracing	None	Table by procedure names and statement numbers	None
DYNA[18,19]	None	FORTRAN	Source	FORTRAN statement	Tracing	Source code	Augmented source listing	Summary; combined runs
COBOLDAP[38]	None	COBOL	Source	COBOL statement	Tracing	Source code	Augmented source listing	Measurements saved on program abort
FORTUNE[35]	None	FORTRAN	Source	FORTRAN statement	Tracing and statically computed time estimates	Source listing	Augmented source listing	Combined with recorder
ANATEMP[36]	None	FORTRAN	None (Not an execution analyzer)	FORTRAN statement	Statically computed time estimates	Generated machine code listing	Augmented source listing	None
PLEA[17]	MVS	PL/I	Run-time; object	PL/I statement	CPU time sampling	PL/I statement number table	Histograms by procedure names and statement numbers	Summary; supervisor module counts
EPLEA[1]	VM/SP	PL/I	Run-time; object	PL/I statement	CPU time sampling	PL/I statement number table	Histograms by procedure names and categorized statement numbers	Summaries; rerun options; call graph with self- and inherited times
Optimizer III[31]	System/360 System/370	COBOL	Run-time; object	COBOL statement; non-COBOL machine instruction	Tracing and time sampling	COBOL listing	Augmented source listing tables for CPU and I/O counts	Summaries
GPROF[46]	Unix	None	Run-time; object	Procedure	Tracing and time sampling	Prologue monitor calls; static call graph	Caller/callee tables with self- and inherited times	Summary; rerun options
APAT[37]	None	APL	Source	APL function	Tracing and CPU time measurement	None	Table by function names	None

Appendix B

Source code for Figures 1A, 1B, and 5A

```
Statement
 number
         /* PRIME: Count the prime numbers between 1 and 10000.        */
   1     PRIME:
            PROC OPTIONS(MAIN) ;
   2           N = 3 ;                  /* Count primes 1, 2, and 3.   */
   3           DO I=5 TO 10000 BY 2 ;  /* Start with 5 for the rest.  */
   4              ISPRIME = 1 ;         /* Assume I is prime.          */
   5              DO J=3 TO SQRT(I) BY 2 ;  /* Look for divisors.     */
   6                 ISPRIME = I-FIXED(I/J)*J ;
   7                 IF ISPRIME=0 THEN
                        LEAVE ;         /* Found divisor, I not prime. */
   8              END;
   9              IF ISPRIME¬=0 THEN
                     N = N+1 ;          /* Count primes.               */
  10           END;
  11           PUT EDIT (N ||' Prime numbers between 1 and 10000.')(A);
  12        END;
```

Source code for Figure 1D

```
Statement
 number
         /* PRIME: Count the prime numbers between 1 and 20.           */
   1     PRIME:
            PROC OPTIONS(MAIN) ;
   2           N = 3 ;                  /* Count primes 1, 2, and 3.   */
   3           DO I=5 TO 20 BY 2 ;      /* Start with 5 for the rest.  */
   4              ISPRIME = 1 ;         /* Assume I is prime.          */
   5              DO J=3 TO SQRT(I) BY 2 ;  /* Look for divisors.     */
   6                 ISPRIME = I-FIXED(I/J)*J ;
   7                 IF ISPRIME=0 THEN
                        LEAVE ;         /* Found divisor, I not prime. */
   8              END;
   9              IF ISPRIME¬=0 THEN
                     N = N+1 ;          /* Count primes.               */
  10           END;
  11           DISPLAY (N ||' Prime numbers between 1 and 20.');
  12        END;
```

Cited references and notes

1. L. R. Power, *EPLEA—Using Execution Profiles to Analyze and Optimize Programs*, Research Report RC9932, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (1983).
2. *Virtual Machine Facility/370 Performance/Monitor Analysis*, SB21-2101, IBM Corporation; available through IBM branch offices. VMAP program number: 5798-CPX.
3. *Statistics Generating Package for VM/370 (VM/SGP) Program Description/Operations Manual*, SH20-1550, IBM Corporation; available through IBM branch offices. VM/SGP program number: 5796-PDD.
4. *OS/VS2 MVS Resource Measurement Facility (RMF) General Information Manual*, GC28-0921, IBM Corporation; available through IBM branch offices. RMF program number: 5740-XY4.
5. *Service Level Reporter General Information Manual*, GH19-6169, IBM Corporation; available through IBM branch offices. SLR program number: 5740-DC3.
6. *IMS Performance Analysis and Reporting System*, SB21-2140, IBM Corporation; available through IBM branch offices. IMSPARS program number: 5798-CQP.
7. *CICS/VS Performance Analysis Reporting System*, SB21-2495, IBM Corporation; available through IBM branch offices. CICSPARS program number: 5798-DAB.
8. *VS1 Performance Tool (VS1PT) Program Description/Operations Manual*, SH20-1837, IBM Corporation; available through IBM branch offices. VS1PT program number: 5796-PGL.
9. *VSE Performance Tool (VSE/PT) Program Description/Operations Manual*, ST40-2171, IBM Corporation; available through IBM branch offices. VSE/PT program number: 5796-PLQ.
10. *VTAM Performance Analysis Reporting System (VTAMPARS)*, SB21-2247, IBM Corporation; available through IBM branch offices. VTAMPARS program number: 5798-CTW.
11. W. H. Tetzlaff, "State sampling of interactive VM/370 users," *IBM Systems Journal* **18**, No. 1, 164–180 (1979).
12. P. H. Callaway, "Performance measurement tools for VM/370," *IBM Systems Journal* **14**, No. 2, 134–160 (1975).
13. R. M. Schardt, "An MVS tuning approach," *IBM Systems Journal* **19**, No. 1, 102–119 (1980).
14. T. Beretvas, "Performance tuning in OS/VS2 MVS," *IBM Systems Journal* **17**, No. 3, 290–313 (1978).
15. M. Deitch, "Analytic queuing model for CICS capacity planning," *IBM Systems Journal* **21**, No. 4, 454–470 (1982).
16. M. L. Joliat, *personal communication*. The TIMEIT execution analyzer was written by M. L. Joliat, who is currently with Intermetrics, Cambridge, MA.
17. R. K. Treiber and I. M. Cuthill, *personal communications*. The original PLEA execution analyzer for the PL/I F-level compiler was written for Standard Oil of California by R. K. Treiber, who is currently at the IBM San Jose Research Laboratory. It was later modified for the PL/I optimizing compiler by I. M. Cuthill, Statistics Canada, Ottawa, Ontario. PLEA is available from the SHARE Program Library Agency (SPLA), Triangle University Computing Center, P. O. Box 12076, Research Triangle Park, NC, by referring to Program Number 360D-04.2.008.
18. L. G. Stucki and H. D. Walker, "Concepts and prototypes of ARGUS," *Software Engineering Environments*, H. Hunke, Editor, North-Holland Publishing Company, New York (1981), pp. 61–79.
19. B. A. McIntosh, N. L. Skeels, and D. H. Springer, *DYNA User's Manual, EKS Version*. For further information contact L. Stucki, Manager, Software Engineering Technology, Boeing Computer Services, Seattle, WA.
20. B. W. Wade, *personal communication*. The XICOUNT execution analyzer was written by B. W. Wade, IBM San Jose Research Laboratory.
21. *OS/VS Linkage Editor and Loader*, GC26-3813, IBM Corporation; available through IBM branch offices.
22. *IBM Virtual Machine/System Product: CMS Command and Macro Reference*, SC19-6209, IBM Corporation; available through IBM branch offices.
23. A static call graph is one whose nodes are procedures and whose directed arcs represent all statically bound calls from one procedure to another (i.e., all calls except those to variables or parameters).[46]
24. A dynamic call graph is one whose nodes are procedures and whose directed arcs represent those calls that actually occur in a particular execution of the program; they can be either statically bound calls or calls to variables or parameters.[46]
25. In order to execute a program, the addition of a set of control blocks or data areas (e.g., register save areas and I/O control blocks) and library or supervisor subroutines are required. These data and subroutines constitute the run-time environment of the program.
26. The execution profiles for TIMEIT and PLEA (Figures 1A and 1B) were produced from the PL/I program in Appendix A. The DYNA execution profile was generated from the equivalent FORTRAN program, shown in the execution profile in Figure 1C. The XICOUNT execution profile was generated from a modified version of the PL/I program in Appendix B.

27. For XICOUNT, standard linkage conventions constitute the use of the BALR 14,15 instruction for calling a procedure and BR 14 for returning from a procedure.[43]
28. The PL/I optimizing compiler will generate a statement number table and add it to the object program if either the GOSTMT or GONUMBER compiler option is used.[29]
29. *OS PL/I Optimizing Compiler: Programmer's Guide*, SC33-0006, IBM Corporation; available through IBM branch offices.
30. A. P. Batson and R. E. Brundage, "Measurement of the virtual memory demands of ALGOL-60 programs," *SIGMETRICS Symposium 74, ACM Performance Evaluation Review* **3**, No. 4, 121–126 (December 1974).
31. "System Software," *Auerbach Technology Reports*, Volume J, P. Nesdore, Managing Editor, Auerbach Publishers, Inc., Pennsauken, NJ (March 1983). Report 635.2040.020 (December 1981) describes the Optimizer III analyzer, which is marketed by Capex Corp., 4125 N. 14th St., Phoenix, AZ 85014.
32. *Datapro 70*, Volume 3, M. C. Heminway, Managing Editor, Datapro Research Corp., Deran, NJ (January 1983). Report 70E-098-01 (September 1981) describes the TSA/PPE analyzer, which is marketed by Boole and Babbage, Inc., 510 Oakmead Parkway, Sunnyvale, CA 94086. Report 70E-692-01 (September 1982) describes the STROBE analyzer, which is marketed by Programart Corporation, 30 Brattle Street, Cambridge, MA 02138.
33. "Software," *Data Sources* **2**, Sections K–M, No. 3, G. L. Fisher, Editor, Ziff-Davis Publishing Co., New York, NY (Spring 1983).
34. T. E. Cheatham, "Comparing programming support environments," *Software Engineering Environments*, H. Hunke, Editor, North-Holland Publishing Company, New York (1981), pp. 11–25.
35. D. Ingalls, "The execution time profile as a programming tool," *Design and Optimization of Compilers*, R. Rustin, Editor, Prentice-Hall, Inc., Englewood Cliffs, NJ (1971), pp. 107–128.
36. S. L. de Freitas and P. J. Lavelle, "A method for the time analysis of programs," *IBM Systems Journal* **17**, No. 1, 26–38 (1978).
37. *APL Performance Analysis Tools*, SH20-2620, IBM Corporation; available through IBM branch offices. APAT program number: 5796-PPJ.
38. N. Fujimura and K. Ushijima, "Experience with a COBOL analyzer," *IEEE Computer Software and Applications Conference*, Chicago (October 1980), pp. 640–645.
39. E. Mahoney and M. Henderson, *METER—A Tool for Evaluating Application Programs*, Technical Report TR03.070, Santa Teresa Laboratory, San Jose, CA 95150 (August 1979).
40. *OS/VS2 MVS Supervisor Services and Macro Instructions*, GC28-0683, IBM Corporation; available through IBM branch offices.
41. R. G. Scarborough, *personal communication*. The SPYTIME execution analyzer was written by R. G. Scarborough, IBM Scientific Center, Palo Alto, CA 94304.
42. *IBM Virtual Machine/System Product: System Programmer's Guide*, SC19-6203, IBM Corporation; available through IBM branch offices.
43. *IBM System/370 Principles of Operation*, GA22-7000, IBM Corporation; available through IBM branch offices.
44. D. E. Knuth, "An empirical study of FORTRAN programs," *Software Practice and Experience* **1**, No. 2, 105–133 (April 1971).
45. K. W. Kolence, "A software view of measurement tools," *Datamation* **17**, 32–38 (January 1971).
46. S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: A call graph execution profiler," *Proceedings of the Sigplan 1982 Symposium on Compiler Construction, ACM Sigplan Notices* **17**, No. 6, 120–126 (June 1982).
47. D. Ferrari and M. Liu, "A general-purpose software measurement tool," *SIGMETRICS Symposium 74, ACM Performance Evaluation Review* **3**, No. 4, 94–105 (December 1974).
48. Access to the PER hardware is available under VM/SP via a facility documented in "PER Debugging for Virtual Machines." For further information contact K. Anderson, University of Maine, Orono, ME, or T. Johnston, Stanford Linear Accelerator Center, Stanford, CA.
49. *IBM Virtual Machine/System Product: CP Command Reference for General Users*, SC19-6211, IBM Corporation; available through IBM branch offices.
50. S. Jasik, "Monitoring program execution on the CDC 6000 series machines," *Design and Optimization of Compilers*, R. Rustin, Editor, Prentice-Hall, Inc., Englewood Cliffs, NJ (1971), pp. 129–136.
51. M. R. Sinnott and A. J. Bytheway, "What is going on inside the machine? (The effectiveness of hardware monitoring as a measurement tool)," *Computer Performance Evaluation*, Online Conferences Limited, Uxbridge, England (1976), pp. 371–388.
52. C. D. Warner, "Monitoring: a key to cost efficiency," *Datamation* **17**, 40–49 (January 1971).
53. R. Ibbett, "The hardware monitoring of a high performance processor," *Computer Performance Evaluation*, N. Benwell, Editor, Advanced Book Program, Cranfield Institute of Technology, UK (December 1978), pp. 274–292.
54. R. W. Hadsell, M. G. Keinzle, and K. R. Milliken, *The Hybrid Monitor System*, Research Report RC9339, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (1983).
55. R. G. Scarborough, *personal communication*. TIMEMAP is a postprocessor to the TIMEIT execution analyzer written by R. G. Scarborough, IBM Scientific Center, Palo Alto, CA 94304.
56. S. W. Sherman, "Trace driven modeling: An update," *Proceedings of Symposium on Simulation of Computer Systems, Conference, Boulder, Colorado* (August 10–12, 1976), pp. 85–91.
57. J. L. Archibald, "The External Structure: Experience with an automated module interconnection language," *The Journal of Systems and Software* **2**, No. 2, 147–157 (June 1981).
58. B. M. Leavenworth, *ADAPT Reference Manual*, Technical Memo No. 19, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (June 1981).
59. D. A. H. Smith, *personal communication*. D. A. H. Smith, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, modified the PLEA execution analyzer to implement inherited CPU time and to make it run on VM/SP.
60. H. A. Ellozy, *personal communication*. The EPLEA call dependency graph[24] is an adaptation of the GRAPH program written by H. A. Ellozy, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598. The GRAPH program was originally implemented to support the ADAPT External Structure.[57]
61. M. A. McNeil and W. J. Tracz, "PL/I program efficiency," *Sigplan Notices* **15**, No. 6, 46–60 (June 1980).

62. J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing, Conference, Boulder, Colorado* (May 2–4, 1977), pp. 106–112.
63. *OS PL/I Checkout and Optimizing Compilers: Language Reference Manual*, GC33-0009, IBM Corporation; available through IBM branch offices.
64. R. W. Scheifler, "An analysis of inline substitution for a structured programming language," *Communications of the ACM* **20**, No. 9, 647–654 (September 1977).

Leigh R. Power *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Since joining IBM in 1963 as a member of the Service Bureau Corporation, Mr. Power has designed and implemented systems for I/O control, interactive computing, information retrieval, text processing, and sorting. Since 1970, Mr. Power has been a research staff member at the IBM Thomas J. Watson Research Center, where he is currently working in the field of software technology. In addition to contributing to the development of the ADAPT tools, Mr. Power has become especially interested in the performance implications of data abstractions. Mr. Power received his B.A. degree in physics from Cornell University in 1963 and attended the IBM Systems Research Institute in 1967 and 1969.

Reprint Order No. G321-5196.