An effective method for measurement and analysis of system software performance

by JOHN R. RUMSEY and DAVID W. ABMAYR
Harris Corporation
Dallas, Texas

ABSTRACT

The performance measurement and analysis of software operating systems which extend basic computing machinery is discussed. The description of an external monitoring technique which facilitates the correlation of hardware events with software functions without the need for software monitors is presented. A time related Event is defined to provide the basis for the technique used to implement the monitor system. In addition, Event analysis methods are introduced which allow a software system execution profile to be constructed.

INTRODUCTION

The existence of complex software systems to extend basic computing machinery is well established and procedures for implementing these systems are well known. Many of these extended machines, however, are dynamically responsive so that a predictable system execution path does not always exist. The system path is normally a function of the occurrence of a real-time event or the result of a user system resource request. The occurrence of either of these events is usually asynchronous to the system software. Considering the unpredictable and complex nature of system software execution paths, the empirical determination of these paths for the evaluation of extended machine performance is generally considered a complex and time consuming task.

Several techniques have been proposed for measurement and analysis of the performance of extended machines. Nutt\textsuperscript{2} surveys the three classical types of computer system monitors: pure software, pure hardware, and hybrid monitor systems. Generally, hardware monitors are considered to be most adept at detecting a wide variety of events but are limited in their ability to relate the events to a specific software function. Software monitors, on the other hand, are better suited for identifying the source of an event than responding to a large class of events. Svobodova\textsuperscript{3} indicates that system measurability can be achieved only with an integrated hardware-software approach in which the external monitoring device complements the internal monitor.

To developers of operating system software, the need to identify and analyze various areas of the software is most important. Items of interest include task dispatching overhead, the response time required to support a disk access, and many other similar functions. It seems, however, inappropriate to burden system software development with the added task of developing and maintaining software monitoring support. Similarly, a hybrid monitor system requires imbedded software to identify certain key functions and has limited flexibility. A technique, therefore, is needed for externally monitoring system software execution.

The objective of this paper is to describe an external monitoring technique which facilitates the correlation of hardware events with software functions without the need for any software monitors, and allows more than one view of machine instruction execution. To completely characterize machine execution, the gathering of several different types of data is required. The data types include:

1. Time related execution state events—to facilitate analyzing program execution
2. Selected address references—to allow reference counts to be accumulated
3. Instruction operation codes—to allow analysis of the distribution of instructions executed
4. Instruction execution and address reference traces—for analyzing branch patterns and to aid software debugging.

Each data type provides a different view of the execution of a sequence of machine instructions. This allows the implementation of a comprehensive program analysis.

EVENT DEFINITION

Conceptually it is desirable to monitor the execution of software with no impact on the system whatsoever. This includes the elimination of any background trace functions or special instructions to identify the occurrence of particular events. To accomplish this goal, it is necessary to define an autonomous device that is capable of capturing states of
execution associated with the software system. This, of course, could be an overwhelming task if no method could be devised to concisely represent the set of possible execution states. An approach to this problem has been defined which effectively reduces the set of execution states to a manageable size.

Consider that any function in a program can be associated with one or more address references. To completely classify an address reference, however, it is necessary to identify the reference type, to define the data associated with the address, and to distinguish multiple references with a time value. A set of primitive elements for defining execution states, therefore, can be used to define Events in terms of an address reference five-tuple.

\[
\text{Event} = E = (A, R, D, C, T)
\]

where
- \( A = (a_1, a_2, a_3, \ldots, a_n) \) is an address space
- \( R = (r_1, r_2, r_3, \ldots, r_j) \) is a non-empty set of address references
- \( D = (d_1, d_2, d_3, \ldots, d_k) \) is a set of data types associated with an address
- \( C = \text{(Fetch, Read, Write, \ldots, I/O)} \) is a set of reference classes
- \( T \) is a time interval in which two consecutive address references contained in \( R \) must be guaranteed to occur

Any Event \( t \) occurs at real-time modulo \( T \).

A profile of the execution of any program can be described in terms of Events. The granularity of the profile is controlled by the magnitude of the Event set. Figure 1 illustrates an application of the Event definitions. A memory map of a skeletal representation of an operating system and associated asynchronous processes is shown. The address space of interest in this example contains the operating system and its associated job scheduling process. The selected Events segment the operating system into specific functions, but the granularity of coverage is very loose. The auxiliary Event associated with clock service guarantees that an Event will always occur within the Event time interval \( T \). Since an Event is defined to always occur within an interval real-time modulo \( T \), a time value may be associated with each Event. Events, therefore, not only provide execution path data but also provide sufficient information to determine the time required to execute the functions described by the profile. Therefore, if the address space is defined so that all software functions are included, then the Event set will allow direct determination of processor utilization for any function of interest. For a particular execution of the software, the Events illustrated in Figure 1 are sufficient to describe the relative utilizations in the defined address space.

The determination of the absolute address associated with each Event of interest may seem to be an arduous task. This is not the case, however, because Events may be defined in terms of relative displacements associated with the set of software functions to be measured. These relative addresses are then easily converted to match a particular implementation.

**IMPLEMENTATION**

The implementation of the mechanism required for gathering the various data types is based primarily on a function \( f(E) \) with a domain consisting of a set of time related events \( E \). A set of data is collected by applying \( f_i \) to \( E \). In particular, for the four data types previously described, the functions are defined as follows:

\[
f_i(E) = \text{insert } a_i \text{ and real-time modulo } T \text{ into the data set if}
\]

- \( a_i \in R \)
- \( c_j \in C_j \)

where \( C_j \subseteq C \)

\[
f_i(E) = \text{counter}_j \leftarrow \text{counter}_j + 1
\]

if

- \( a_i \in R \)
- \( c_j \in C_j \)

where \( C_j \subseteq C \)

\[
f_i(E) = \text{counter}_i \leftarrow \text{counter}_i + 1
\]

if

- \( c_i = \text{Fetch} \)

where \( j = f(d_j) \) which identifies the instruction code

\[
f_i(E) = \text{insert } a_i \text{ into the data set if}
\]

- \( c_i = \text{Fetch} \)

* Notation for unordered cross product taken from Reference 1.
The implementation of these functions requires that the monitoring system must be capable of capturing addresses, verifying reference types, and associating time values with the address data. A device to accomplish these functions is not difficult to implement, consisting only of a logical four-way switch, simple address translation, incremental counting, and the associated memory and function generator. A simplified block diagram is provided in Figure 2.

The monitor device is interfaced to a standard production system which specifies the data path in the monitor and performs the analysis of the stored data. Since address, data and reference type can be captured from the Test System, each of the required data types can be accumulated. The function \( f_1 \) is implemented by translating monitored memory addresses to specify an appropriate reference type comparison. The Type Comparator, which is dynamically programmable, provides indication of a match between address and reference type. If a match occurs the address data and the current value of the time counter are inserted into a queue in Temporary Storage. Subsequently the queued data is written to Auxiliary Storage. The second function \( f_2 \) is also implemented by translating monitored memory addresses. In this case, however, the translated address specifies the location of a counter in Temporary Storage which is to be incremented by one. Function \( f_3 \) is similar to \( f_2 \) since both specify that a Temporary Storage Counter is to be incremented. The third function, however, invokes translation of memory data if and only if the reference type is a Fetch. The address generated by the Translator becomes a function of the instruction operation code. The last function \( f_4 \) is a special case of \( f_1 \). The address (without time counter) is queued if the reference type is a Fetch.

EVENT ANALYSIS

Experience with the monitor system has shown that the most extensively used function is \( f_1 \), i.e., gathering Events for profiling system execution. While the method for accumulating Events is interesting, the techniques used to analyze the Event sequence to extract system utilization and performance information may be more significant.

For the purpose of analysis, each Event in a sequence of Events may be considered to represent a start, end, or single Event. Start and end Events are used to define intervals, the durations of which are of interest. A single Event has no interval connotation but implies merely a count of occurrences of some address reference by specific type. With respect to intervals, several things must be considered:

(1) Intervals may be nested. Let \( A \) represent a set of start Events and \( B \) represent a corresponding set of end Events. Then it is permissible for \( A_j \) to occur simultaneously with or after \( A_i \) but before \( B_i \), provided \( B_i \) occurs concurrently with or before \( B_i \). The allowable nesting level depends upon the total number of single Events and Event pairs. The following example illustrates timing calculation. Suppose the following Event sequence occurs:

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
A_1 & A_2 & B_1 & A_3 & A_4 & B_3 & B_4 & B_5 \\
\hline
\Delta t_1 & \Delta t_2 & \Delta t_3 & \Delta t_4 & \Delta t_5 & \Delta t_6 & \Delta t_7 \\
\end{array}
\]

Then the intervals associated with each Event pair are as follows:

<table>
<thead>
<tr>
<th>Pair</th>
<th>Interval Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_1 ) ( B_1 )</td>
<td>( \Delta t_1 ) + ( \Delta t_3 ) + ( \Delta t_5 )</td>
</tr>
<tr>
<td>( A_2 ) ( B_2 )</td>
<td>( \Delta t_2 )</td>
</tr>
<tr>
<td>( A_3 ) ( B_3 )</td>
<td>( \Delta t_4 ) + ( \Delta t_6 )</td>
</tr>
<tr>
<td>( A_4 ) ( B_4 )</td>
<td>( \Delta t_7 )</td>
</tr>
</tbody>
</table>

(2) An Event may define
a) the start of \( n \) intervals
b) the end of \( n \) intervals
c) the start of one interval and the end of another interval
d) the start and end of an interval

Where \( A_i \) starts several intervals, the interval count is accumulated only until the first end occurs and is associated only with that specific start-end pair.

(3) The percent utilization associated with an interval is calculated as follows:

If \( T_{ij} \) denotes the \( i \)th interval time associated with the \( j \)th Event pair and only the \( j \)th Event pair, then the percentage of time spent in the \( j \)th interval is

\[
\% = \frac{\sum_i T_{ij}}{\sum_j \sum_i T_{ij}}
\]
Considering the above analysis rules, an effective method of determining performance is based upon constructing a profile of system activities. The process of defining a system profile consists of selecting appropriate Events associated with the software system. These Events may then ultimately be used to define intervals of time which collectively account for the total real-time associated with a particular period of system operation. After selected programs are characterized with the profile, performance characteristics may be calculated.

In addition, a data profile may be defined which is a subset of the system profile. A data profile is generated when a data sample accumulated with a system profile is analyzed. Various data profiles may be generated from a single data sample. The one-to-many mapping from system profile to data profile facilitates multiple analyses of a single data sample. For example, data profiles may be generated for selected combinations of Events in the system profile to allow analysis of processor utilization, interpretive instruction execution time, disk system physical access time, etc.

When a data profile is generated certain information relating to intervals is accumulated. The minimum, average, and maximum interval times are maintained with a count of the number of times each interval was entered. In addition, a multipoint interval time distribution is calculated which provides the basis for typical weighted interval residence times.

An example of a system profile is provided in Figure 3. Each item of coverage implies that two Events must be defined to represent the associated interval with the exception of count only items which are implicitly defined by single Events. Associated with each address is the reference type designation.

A data profile may be generated with any subset of the system profile. Utilization summaries extracted from the various data profiles allow an extensive performance analysis. For example, considering a disk based operating system, it is of interest to determine the distribution of the time required to physically access the disk. This is easily accomplished by generating a data profile containing an interval which begins when a Start I/O to the disk is issued and ends when the first instruction of the corresponding interrupt service is executed. A data profile is illustrated in Figure 4. The interval counts have a dimensional value of two microseconds.

Data profiles allow a comprehensive look at any part of the system software. Figure 5 illustrates a summary of the performance of one of the access methods supported by the File System. Data profiles similar to Figure 4 were used to compile the statistics.

The Event analysis method described is certainly not unique. Function \( f_i \) generates a sequence of Events that are written to auxiliary storage. The analysis of these Events is in no way confined to the method described; however, this technique has proven to be one of the most useful. An alternative analysis method has been used which does not consider intervals as nested. The time for any interval is accumulated from the first instance of the start Event until the corresponding end Event is detected. This alternate technique is useful for determining the frequency of occurrence of intervals.

### OTHER DATA TYPES AND ASSOCIATED ANALYSES

From the viewpoint of performance analysis, the second most utilized statistics have been derived from machine instruction operation code data. A simple but very useful analysis counts the occurrence of each operation code type. When the data sample is exhausted, a summary is generated indicating the percentage of execution associated with each instruction. Weighted instruction execution times are calculated which clearly define processor utilization in terms of each machine instruction. Excessive use of high cost instructions is quickly identified.

The benefits of generating an address trace concurrent with the execution of system software are immediately
An Effective Method for Measurement and Analysis of System Software Performance

<table>
<thead>
<tr>
<th>Functions</th>
<th>Transfer Rate</th>
<th>System Utilization</th>
<th>Buffering Level</th>
<th>Avg Disk Accesses per Transfer</th>
<th>Avg Access Rate(^2)</th>
<th>Avg Access Time(^3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA → DISK</td>
<td>551.56 RPM(^1)</td>
<td>28.59 %</td>
<td>0</td>
<td>0</td>
<td>2.34</td>
<td>21.55</td>
</tr>
<tr>
<td>DATA → DISK</td>
<td>581.69</td>
<td>19.30</td>
<td>0</td>
<td>1</td>
<td>0.42</td>
<td>4.03</td>
</tr>
<tr>
<td>DATA → DISK</td>
<td>579.87</td>
<td>18.51</td>
<td>0</td>
<td>2</td>
<td>0.22</td>
<td>2.16</td>
</tr>
<tr>
<td>DISK → DISK</td>
<td>381.49</td>
<td>23.54</td>
<td>0</td>
<td>0</td>
<td>3.83</td>
<td>24.34</td>
</tr>
<tr>
<td>DISK → DISK</td>
<td>638.25</td>
<td>24.42</td>
<td>0</td>
<td>1</td>
<td>1.89</td>
<td>20.08</td>
</tr>
<tr>
<td>DISK → DISK</td>
<td>744.57</td>
<td>27.21</td>
<td>0</td>
<td>2</td>
<td>1.72</td>
<td>21.29</td>
</tr>
<tr>
<td>DISK → DISK</td>
<td>452.80</td>
<td>22.07</td>
<td>1</td>
<td>0</td>
<td>2.86</td>
<td>21.61</td>
</tr>
<tr>
<td>DISK → DISK</td>
<td>478.36</td>
<td>22.48</td>
<td>2</td>
<td>0</td>
<td>2.68</td>
<td>21.40</td>
</tr>
<tr>
<td>DISK → DISK</td>
<td>1087.45</td>
<td>26.72</td>
<td>1</td>
<td>1</td>
<td>0.95</td>
<td>17.21</td>
</tr>
<tr>
<td>DISK → DISK</td>
<td>1360.39</td>
<td>27.79</td>
<td>2</td>
<td>2</td>
<td>0.56</td>
<td>12.62</td>
</tr>
</tbody>
</table>

\(^1\)RPM — Records per minute
\(^2\)Accesses per second
\(^3\)Milliseconds

Figure 5—File system performance statistics

apparent to programming personnel. Performance data, however, related to programming methodology can also be extracted from address data. Consider the simple problem of searching an array \(A\) for a value \(x\). Examples 1a and 2 of Reference 2 provide two solutions with differing efficiencies. The second is considered the more efficient but it also may execute a greater number of Branch instructions. The increase in the number of Branch instructions executed may or may not have an effect on performance, depending upon the environment. For example, if the architecture of the processor executing the software is based on a pipeline technique, then it is possible that a high frequency of Branch instructions could certainly affect system performance. Branch analysis of address trace data, therefore, could reasonably be used to predict performance problems.

CONCLUSIONS

The performance monitoring technique described has been implemented and successfully used to evaluate the performance of a general purpose, disk based operating system and related asynchronous processes. The most extensively utilized operating system functions were quickly identified. Critical real time dependent functions were isolated and the functions for which execution times were marginally acceptable were rewritten to provide sufficient safety margins. Each release of the operating system or associated applications is currently profiled to ensure consistently reliable extended machine performance.

REFERENCES
