Performance Measurement Using System Monitors

Erwin M. Thurner

Siemens AG, ZFE ST SN 13
D - 81730 München

Abstract: *System monitors record inner states of computing systems. They are required for the debugging of computer systems as well as for the measurement of performance and they are used for the verification of system models, too. This paper first discusses the area of application of system monitors, and afterwards it introduces measurement-principles of the different monitor types:*

- *Software monitors, that either analyze the account-log, or that are available as event-driven monitors, as samplers or as profiling-monitors.*
- *Hardware monitors, using the measurement-principles logic-analyzer, events, sampling, and constructing classes of states.*
- *Hybrid monitors, which use the measurement-principles of hardware-monitors on the hardware part, but differ by the software part that generates the signals to be measured.*

At last, connections to performance-analysis based on models are discussed.

Key words: *System Monitor, Bus Monitor, Software Monitor, Hardware Monitor, Hybrid Monitor, Measurement of Performance, Debugging, Performance Models, Event Monitor, Sampling Monitor, Monitor based on Classes of States*

1 What does "System Monitor" mean?

1.1 Usage of System Monitors

A system monitor has the task to protocol inner states of a computing system. The results that can be obtained by system monitors allow to solve problems in a broad area of application:

- to find some errors in a computing system (debugging),
- to check the resource utilization and the job load of system components,
- to say something about the performance of the systems,

- to provide a base for building models of the computing system,
- to find system bottlenecks.

The broadness of the requirements necessitates that more than only one type of monitors is offered on the market. In the next sections, we will classify these different kinds of monitors under aspects of objects of measurement, type and principles of measurement.

1.2 Measurement Objects of System Monitors

For system monitors, typical objects of measurement are:

- *Bus Monitors:*
  The buses of a computer are the central connection elements between its system components. The bus activity is an indication of the sytem components' activity. By measuring the bus, we can make statements
  - about errors in the bus protocol,
  - about the kind and the amount of communication on the bus,
  - about system bottlenecks,
  - (indirectly) about the load of the sytem components.
- *Cache Monitors:*
  The behaviour of the cache is very important for the overall performance of high-performance-sytems. If the real system behaviour is known, one is able to tune the system purposefully.
- *CPU Monitoring:*
  By doing this, one can directly measure the CPU load. Measurements like these are particularly important for multi-processor systems, because the process scheduling depends on it. That is why the system must be checked continuously for busy and idle processors and which of them are able to perform a waiting task.
- *I/O Monitoring:*
  The I/O-system plays an important role for the overall-performance of a computer, as pointed out by Hennessy and Patterson [14]. Besides this, I/O-operations are very costly and therefore they are important for the accounting of the system. For this reason, I/O-operations are measured either via the system bus or on the components, via the working time of an I/O-processor or the duration of moving the harddisk arm.
- *Other System Monitors:*
  Basically, the performance of any system component can be measured by a particular monitor. In practice, only such components are measured which are suspected of being faulty or retarding for the overall-performance of the system.

1.3 Types of System Monitors

The types of system monitors are distinguished by the realization of their detecting element into: software monitors, hardware monitors, and hybrid monitors.
The evaluation on the one hand and the display of the measured system parameters can basically be done independently of the measurement itself. This fact enables the user to select the detecting element and the evaluation programs separately. Furthermore, the evaluation tools must be adapted both to the detecting element and the requirements.

2. Usage and Requirements of System Monitors

In this chapter, a survey is given about the usage of system monitors. The results will be summarized in a table at the end of this chapter.

2.1 Debugging of System Components and Interconnections

Particularly in the test-phase of the system components, system monitors are very often used for the debugging of system components and their connections.

- *Debugging of System Components:*
  Here, both the faults of the system components and faults that happen by the cooperation of the components in a system are analyzed. Due to the progress in simulation, a lot of faults of the system components can already be discovered and corrected in the design phase of the system. Nevertheless, in complex systems many faults of system components and their cooperation are not discovered before their integration into the system.
- *Debugging of Connections (Buses) and Protocols:*
  When system components are connected with the standard backplane bus of computing systems, the exact obediance to the bus protocol has to be verified. The same is true for other connections like LANs and wires. Besides a careful testing (e. g. by using a test suite, cf. [1]), Formal Description Techniques (FDTs) are used for this purpose. FDTs are available in languages Estelle, SDL or LOTOS (see [5] and [7]). FDTs are being developed even for bus protocols [30]. Also Petri Nets are used successfully for the verification of protocols [6].

For debugging, a *high temporal resolution* of the monitor is necessary, i.e. a sampling frequency that is as fast as the bus clock. A good *selectivity* is very useful, i. e. the monitor only records data when errors occur, but then the erroneous data have to be recorded with as many details as necessary. For

debugging monitors, a long record-time is usually not very important. For this purpose mostly event hardware-monitors or logic analyzers are used.

2.3 Load of System Components and Performance Measurement

To measure the load of system components, "software probes" are injected into the source code of device drivers. These probes measure when an I/O-operation is started and when it has been terminated (e. g. [9] and [24]).
The monitoring of system components can be done for several reasons:

- Accounting of the used CPU and the I/O computing power, to determine the cost of the computing for every user.
- The load distribution, the allocation and the migration of tasks in a multi-processor system or within loosely coupled systems.
- Optimization of the system performance by optimizing frequently used system resources or parts of a program.
- To get a performance profile of a system, i. e. continuous watching of interesting parts of a system and the overall-performance of the system.
- Response times of a system.

For load monitors, mostly *statistical* statements are made about a longer duration, rather than to get measurement values that are as exact and have as many details as possible. For load and performance monitors, event hybrid and software-monitors are mostly used; but there are also hardware monitors available e.g. the idle counter.

2.4 Building a System Model

The evaluation of the performance of computer systems requires the usage of system models, which describe the system's behaviour. These models are to represent realistically on the one hand all the system parameters of the computer system and on the other hand the system load. For the construction and the verification of system models, some data are to be measured concerning the characteristic load and the behaviour.
The evaluation of the system performance and other system parameters is unavoidable, if the system is not yet available as hardware, but some claims are to be made about it to make architectural decisions. This way is gone frequently nowadays for the analysis and evaluation of new architectural approaches, and the quantitative influence of architectural parameters on the system performance. Due to this, the load profile of this system must be available.
To make statements about the whole system, classes-of-states hardware monitors are very well suited. For partial aspects, event or sampling hybrid monitors or profiling software monitors are used. Table 1 gives a survey of the typical usage of the monitor principles.

Type of Monitor *Usage*	Hardware Monitor	Hybrid Monitor	Software Monitor
Debugging	event monitors, logic analyzers, classes-of-states monitor with attribute memory	—	—
System Load and System Performance	classes-of-states monitor, idle counter	event monitors, sampling monitors	analysis acc.-log, event-driven, sampling and profiling
System Bottlenecks	classes-of-states monitor, big logic analyzers	event monitors	profiling
Model Building	classes-of-states monitor	event monitors, sampling monitors	profiling

Table 1: Types of Monitors and Typical Usage

2.5 Classifying System Monitors

As mentioned above, the different classes of system monitors differ by
- their *temporal resolution*,
- their *duration of measurement*, and
- the *amount of measured data*.

So the selection of a monitor can be made by constructing a coordinate frame with the axes temporal resolution, duration of measurement, and amount of measured data (fig. 1). In this coordinate frame, the desired place of one's monitor can be determined. Of course, the cost of a monitor grows with growing coordinates.
Using this categorization, the different variants of monitors can be shown very well. The monitor with the best cost-performance-ratio for a distinct usage can be determined easily by assigning one's requirements in this parameter space to a suitable monitor in this space.
Let us illustrate the classification of monitors by looking at some examples, which are drawn in fig. 1: A *logic analyzer* has a temporal resolution of

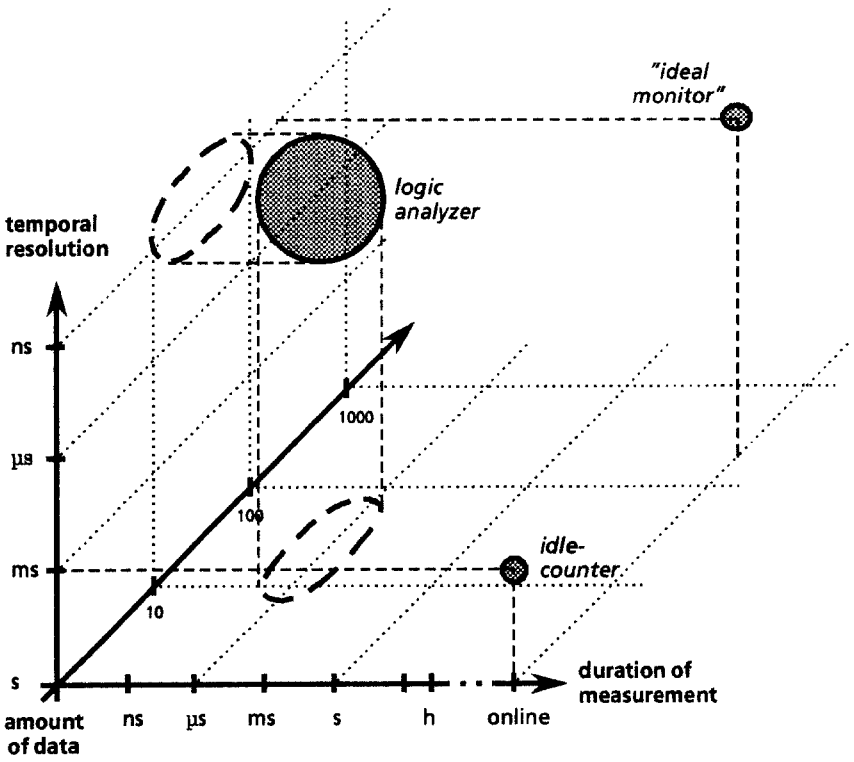

Fig. 1: Classification of System Monitors

some nanoseconds, a measurement duration of some microseconds, and it can display some 10 or 100 signals. An *idle counter* has the temporal resolution of about one millisecond, a very long duration of measurement (up to "online"), and it supports only one signal, namely idle. An "*ideal monitor*", i. e. a monitor that records and displays everything that happens within a computer system, would be drawn into the right upper corner of fig. 1. But: "Ideal monitors" like this are virtually never realized due to the enormous amount a data one has to cope with. Furthermore, the measurement results obtained by this method are not so much better than results that can be gained by monitors with reasonable limitations. That is why "ideal monitors" actually have a very bad price-performance ratio. This approach to collect as much data as possible and to reduce the amount of data not before the evaluation has been realized only in the very first generation of system monitors. Modern research in this area has the target of successive limitation, where the exact formulation of the data to be gained and the parts to be examined plays an essential role. Using the classification-diagram of fig. 1, also software and hybrid monitors can be categorized easily: At the

temporal resolution, hybrid-monitors begin in the microseconds, while software-monitors begin in the milliseconds.

2.6 Problems During the Measurement with System Monitors

Besides the requirements discussed above, there are some specific problems still to be considered which appear during the measurement.

Interference between Monitor and Measured System. The interference between the measuring monitor and the measured system is a very important criterion for the judging of system monitors. Generally spoken, *hardware monitors* have *no interference* with the performance of the measured system. (Interference given by impedances of the measuring device etc. may be neglected here. But the hardware monitor designer has to take care of these aspects, because ugly sporadic faults may be generated by such effects.)
Software monitors however are part of the system to be measured. Due to this, they consume system performance, namely between 3% and up to more than 50% of the whole system. *Hybrid monitors* typically need from 1% to 3% of the system computing power during the measurement.
Due to these reasons, measuring real time systems can only be performed by hardware monitors and – under some circumstances – by hybrid monitors. Anyway, the interference between the measuring monitor and the measured system does unfortunately exist, and it has to be considered both at the evaluation and when load models of the system are based on these measurements.

Measuring of Time. Another problem of system monitors is the measuring of time: The used time slots must be fine enough to record every single activity of a system. For hardware monitors and hybrid monitors, this is not a real problem, because an external clock can be provided, which can be selected freely. With software monitors however, the system clock is used for all purposes in the computer and it provides in many cases only the resolution of one second. This problem can be solved by introducing an additional process clock, which can be read via a special register (cf. [26] and [10], section 5.2.2).
An additional problem arises with multi processor system: Here, a global system clock must be provided, to correlate the activities of the processors. [15] proposes a synchronization signal, which is sent to the measuring devices via Ethernet. For other monitors, cables with exactly the same length from the measured system parts to the recording device are enough.

2.7 Other Criteria for the Judging of System Monitors

Evaluation Tools. After measurement, the collected data must be evaluated. So one can ask which criteria suitable evaluation tools have to fulfill:
- How good do the tools work together with the measuring devices?
- How can they be handled?
- Generally: How fast can I say something about the system's behaviour?

The requirements of the evaluation are hard to quantify. It seems that there is no alternative to considering every evaluation tool to be used, if it is really able to display what is needed for the desired application.

Flexibility. At system monitors, flexibility can be useful. Flexibility in this context means, that one type of monitor can be adapted easily to more than one bus or to more than only one object to be measured. So the user has to learn only one concept, one user menu etc. The manufacturer could offer only one type of monitors for a whole class of requirements.

Documentation and Archive Procedure of the Measurement Results. The ability for documentation and the functionality of the archive procedure of the measurement results is important, because usually many more than one measurement must be performed to say something about the behaviour of a computing system under several aspects.

3. Types of System Monitors

The types of system monitors are divided into
- software monitors (and firmware monitors),
- hardware monitors, and
- hybrid monitors.

For these types, the measurement principles and some typical implementation examples are given in this chapter.

3.1 Software Monitors

Software monitors were the first types of system monitors that have been developed. Their first task was to measure, how much of computing time, I/O throughput etc. are needed for a particular user resp. for each particular task. Based on this, the cost for the computer usage are assigned to the users. Aspects like performance came much later. In this section, first common principles and problems of software monitors are sketched. Afterwards, the measurement principles of software monitors are explained.

Principles and Problems of Software Monitors. Generally spoken, software monitors are a part of the measured system and it is unavoidable, that they interfere with it: They need memory, they use the CPU, and they perform I/O-operations. This fact must be considered when using software monitors. That is why there are several approaches to minimize the general load or one of the discussed parameters caused by the monitor.
For a software monitor, additional program code must be inserted into the system to be measured. Inserting additional code into the examined places of a program is called *instrumentation*. This can be done by three methods (cf. [10]):

- To use an additional program in the computing system:
  Such a program can cyclically evaluate data of the operating system and analyze them under some aspects. This approach is followed by sampling monitors and by the analysis of the account-log.
- Modification of the program to be measured:
  This method is mostly used by modern software and hybrid monitors. Here, the examined parts of a program – like procedure calls, basis blocks, program line, etc. – are "instrumented" by additional code, which produces a protocol about the dynamic run of the program. This method is used for the principles profiling and for event-driven software monitors.
- Modification of the operating system:
  This method is the least portable one, because it uses the internal data of the operating system. This approach is used for generating the account-log, but also for the interrupt-intercept approach at event-driven monitors.

When measuring multi processor systems, the problem of the global time for the whole system comes up once more. This problem can be solved either by some hardware measure such as a system-wide common clock, or by a synchronization signal for start and stop, from which the correct times can be computed.

Analyzing the Account-Log. The data of the account-log are recorded regularly at multi user systems. They include some details about duration of tasks, process load, usage of peripheral devices, login times etc. So the account-log can be used as a source for some statistics about the load of the computing system. It shows load peaks and it is a first indication for system bottlenecks. It has the advantage of giving no additional load to the system, because these values are always measured. This methods depend very much on the examined machine and the operating system.
In [25] a software monitor is introduced, which reads the account-log every day and computes the difference to the last account-log. Based on this, the monitor compiles some daily statistics about the usage of the hardware resources, like CPU, I/O, paging etc., and the offered "service", here defined as the response times at multi user mode.

Event-Driven, Interrupt-Intercept. An event is defined in [10] as any change of the state of a computing system. (This definition must not be confused with the notion of "event" at a hardware monitor!) An event-driven software monitor is a machine that records changes of the states of a computing system in a so-called event-trace. This approach has the disadvantage, that a complete trace (*"full trace monitoring"*) generates an enormous amount of data, so that the data flood is to be reduced by limitation on distinct aspects.
It suggests itself only to consider important actions of the operating system in the event-trace such as task switches or I/O-requests. Particularly for this request, interrupt-intercept monitors are used. In these kind of monitors the addresses of the interrupt routines are changed, so that every interrupt-call in reality first calls a monitoring routine and then jumps to the subroutine which actually handles the interrupt [17]. With this monitor, meaningful traces about important actions of the operating system can be captured.

Sampling Software Monitor. Sampling Monitors perform measurements in periodical time slots. The monitor is subdivided into two parts (cf. [10], section 5.2.1):

- The Extractor:
  It periodically generates an interrupt, say 1 to 20 times per second. In the software routine that handles the interrupt data are collected, which are meaningful for the system state. This interrupt needs a high priority, so that the interrupt routine cannot be interrupted and the watched system data cannot become corrupted by that interrupt.
- The Analyzer:
  It evaluates the data from the extractor under some aspects and shows relevant system data.

It is clear, that sampling monitors can make only *statistical* statements about system data. The exactness can be influenced in a broad range by the sampling frequency.

Profiling Monitor. Profiling is the *dynamical analysis* of a program. See the description of the tools `prof`, `pixie`, `pixstats` in [22] and `prof`, `lprof` in [27]. The opposite is the *static analysis* of the assembler-code, which provides the relative frequency of one instruction. The instrumentation of the examined program can be performed automatically and can simply be chosen by a compiler option.
In most cases, the time used to execute a procedure is measured, or the number of runs of every line or basic block of a program. By using data gained this way, frequently used procedures and program parts can be found, at which tuning will make sense. In addition, subroutines can be

found which are not called at all. This may be an indication of an error or of a lack of fault coverage.
With this kind of monitors, data or instruction profiling can be executed: With the MIPS-tool `pixie` traces can be made, which list the virtual addresses of program data and instructions in the temporal sequence they appear when the program is run. This list can be used as a base for a cache simulator (e. g. `cache2000` for MIPS computers), which computes the hit rates of the caches.

3.2 Hardware Monitors

Hardware monitors measure electrical signals, which come from distinct points of a computer (fig. 2). Such measuring points may be: bus signals, critical signals within a computing system, control signals of peripheral devices, e. g. the positioning arm of harddisks (see [10]), or even more complex signals, like the well-known "wait"-light of IBM /360 computers, (cf. [23], p. 54).
Due to the amount of data becoming huge, if all signals in every bus cycle are measured, after the probes of a hardware monitor a filter is installed, which limits the amount of data. This filter moves a real hardware monitor away from the "ideal monitor" of fig. 1 in at least one axis. The limitation can be performed either by limiting the recorded period of time, or by considering only a subset of the signals to be measured, or by reduction of the recorded number of cycles, or by limitation on some events etc. It must be guaranteed by this filter, that even in the worst case all measured parameters are recorded without loss. The overflow of an intermediate buffer may make a full measurement invalid.
For the *online evaluation* it must be ensured that the measured values can be read continuousely from the intermediate buffer and that they can be computed for the display. After displaying them, the computed values can be stored. The continuous display requires a high data reduction, because human watchers cannot follow fast changes of many signals. The *offline evaluation* uses stored data as a base. These data can be computed and displayed under a broad range of aspects.
The measured signals usually change with a frequency of some megahertz (signals inside a computer or bus signals), sometimes in the range of kilohertz (composed or peripheral signals). At the online evaluation the display typically changes once a second.
With hardware monitors *only low-level signals* can be measured. If statements about high-level processes are to be made, e. g. idle of the operating system, subroutine calls, duration of I/O-operations, task switch etc.), then the measured signals can be postprocessed or concentrated by suitable tools, or they must have been preprocessed before the measured value is

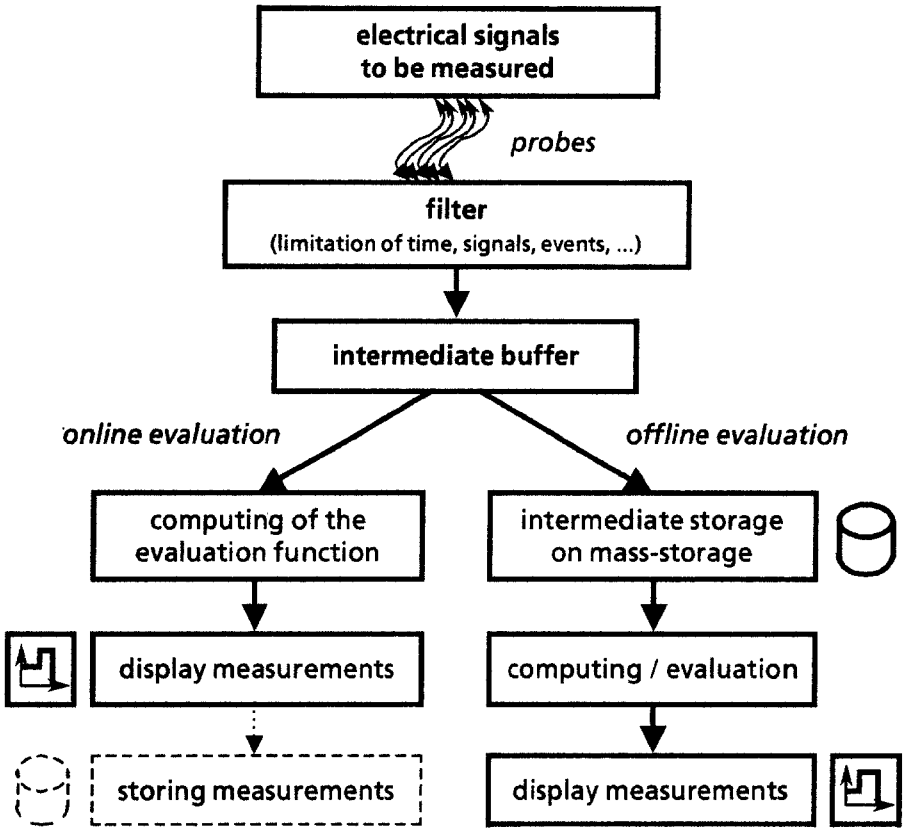

Fig. 2: Principle of a Hardware Monitor

taken, e. g. the wait-signal mentioned above or by the software part of hybrid monitors.

Logic Analyzer. A logic analyzer is the simpliest form of a hardware monitor: It records the signals to be examined with a variable resolution. The data are displayed on a screen, which represents the signals as a sequence of 0s and 1s. There are also triggering conditions available. With logic analyzers, it is possible not only to record the signals *after* the triggering conditions, but even *before* it. Progress in logic analyzers resulted in higher temporal resolution, more signals, longer traces, and the support of more complex triggering conditions.
For the measurement of the system load, these devices are not very suitable: On the one hand, every measurement causes a huge amount of data, on the other hand, the evaluation only consists in looking through the

traces. Tools for the fast and purposeful evaluation are not available or have to be written *ad hoc* for every usage.

Event Monitors. The most wide-spread measurement principle of hardware monitors is the event monitor. The data reduction is provided by recording signals not continuousely, but only when a distinct event appears. An event is a class of signal combinations, e. g. "write on I/O-address 123", "read from memory-address 500 to 1000". When this event appears, a definite action is performed. The events are a purely combinatorical expression of a subset of the examined signals. Events can be cut out under some circumstances.

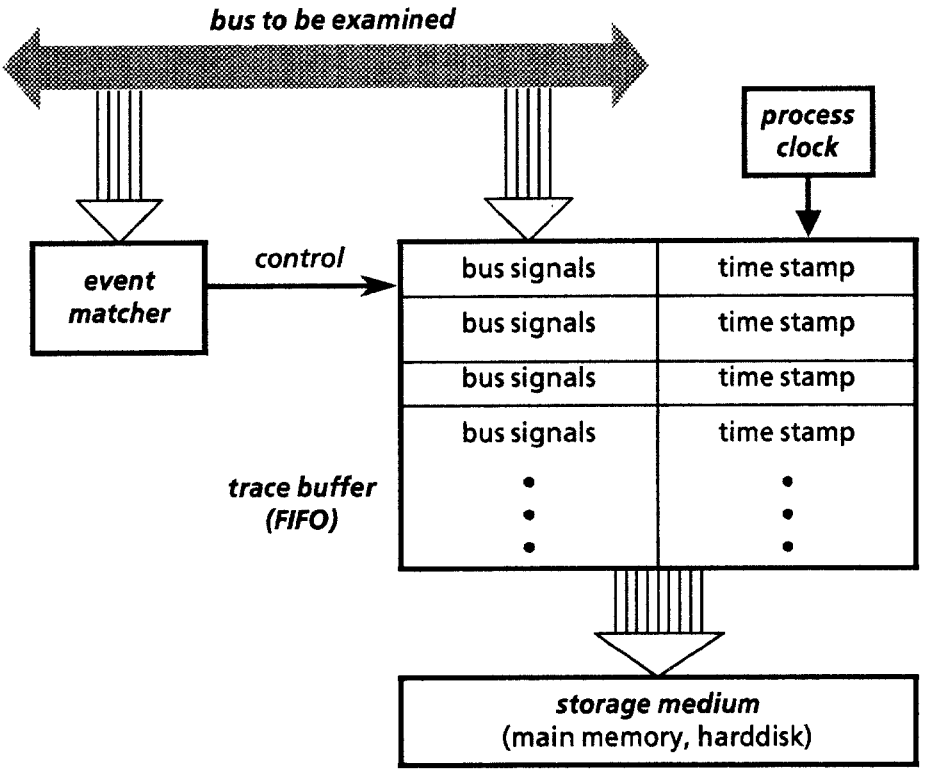

Fig. 3: Principle of an Event Hardware Monitor, Time Mode

If an event happens, then all the signals are stored in a FIFO, which is called *trace-buffer*. To get a temporal relationship, a *time stamp* is stored together with the event data. The events and the actions to be performed must be defined before the measurement takes place. The data reduction of event monitors happens by the fact, that due to their triggering conditions

only 1% to 0.1% of the cycles are recorded. The trace-buffer FIFO is necessary to deal with short peak-loads.
The action that is performed mostly at event monitors is to record all the signals or a subset of them when the event occurs (so-called *time mode*). Fig. 3 shows the principle of an event monitor: The event-matcher of the monitor recognizes the signals, adds a time stamp and stores them into the trace-buffer FIFO. The trace-buffer is read out either after the measurement or – for online and long-time measurements – during the measurement. The content of the trace-buffer is stored into a storage medium. The time mode needs about 64 to 128 bits for every line of the trace-buffer, and its length is about 8 K to 64 K entries. Instead of using the time mode it is possible only to count the number of events. This is performed by the *count mode.*
It is also possible to make this monitor programmable. When an event happens, one of a broad range of actions may be started. Anderson et al. [3] introduce a monitor, which performs the following actions:
- Increment or reset external or internal counters.
- Buffering of signals, maybe with a time stamp (similar to time mode).
- Writing data from the buffer to an external storage-medium.
- Set and reset of the event counter.

With these measures, an event monitor can be made very flexible.

Sampling and Cumulation. At the sampling mode of system monitors, significant values of the measured system are recorded in equidistant intervals – e. g. every millisecond, see fig. 4 – or in stochastic intervals.

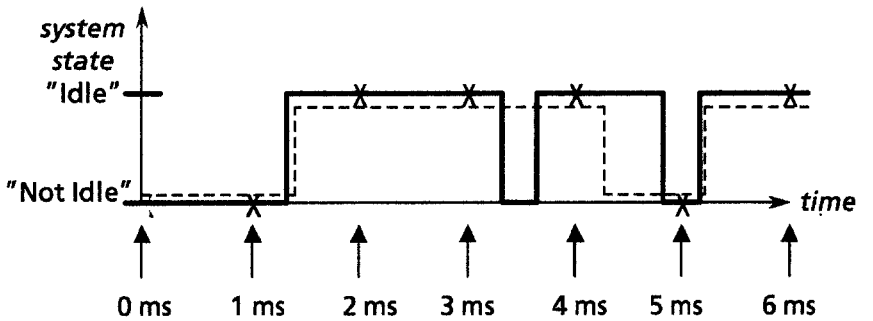

Abb. 4: Principle of a Sampling Monitor

With this kind of measurement, only statistical statements about the system can be made. Furthermore, measurement values are neglected. In fig. 4 the system is not idle between 3 and 4 ms, but this is not noticed by a

sampling monitor, on the other hand it records the short not-idle at 5 ms. The measured signals are shown by the thin broken line. With sampling monitors, long measurements and even online-measurement is possible.
A typical sampling monitor is described by Hattenbach [13]: Here every millisecond a measurement takes place. Recorded values are: the current op-code in the instruction register, i. e. which operation is being executed at the time of the recording, and the physical address of the main memory, to get an impression of the usage of the main memory. The measurements lasted over several hours resp. days. Based on this measurement data, claims are made about the floating-point load of the computer and the efficiency of paging. With these measurements some questions were to be made clear, e. g. if an additional CPU is necessary. Furthermore, a software monitor was checked by this sampling monitor.

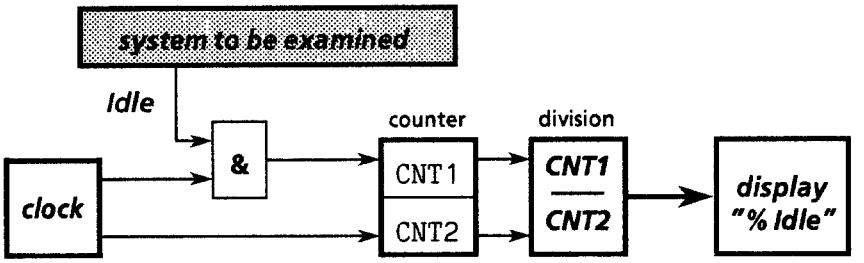

Fig. 5: Principle of an Idle Counter (Cumulation)

An example for the cumulation is the idle counter. Fig. 5 shows its principle: The idle counter consists of two counters CNT1 and CNT2. CNT2 is incremented continuously, CNT1 is incremented whenever the examined system is idle. After the end of the measurement, the relative load of the system can be expressed by the quotient <CNT1>/<CNT2>. After having read CNT1 and CNT2, the both counters are reset. For practical use, one can spare CNT2 and scale the percentage by a suitable choice of the measurement interval. It is also possible to measure more system parameters with this principle and to show the results with an Kiviat graph. Due to the low data rate, this principle is very suitable for online measurements.
Event monitors can be used for sampling and cumulation, if the event matcher is substituted by a timer-clock. The mesurement can be done without the time stamp then.

Classes-of-States Monitor. The principles eventing and sampling are not very suitable to find system bottlenecks and to look for their reasons. With these monitors it may happen, that short state changes of the examined object are averaged or vanish completely (sampling monitor), or state-

changes that are important for the performance are not recorded at all (event monitor), or the measurement interval is too short to answer the questions asked by this measurement (logic analyzer). So it is necessary to record every state-change of the system, i. e. to measure with the system clock, and a longer time interval, say one second or more, has to be recorded and analyzed. Therefore a Classes-of-States Monitor is very suitable (see [29]).

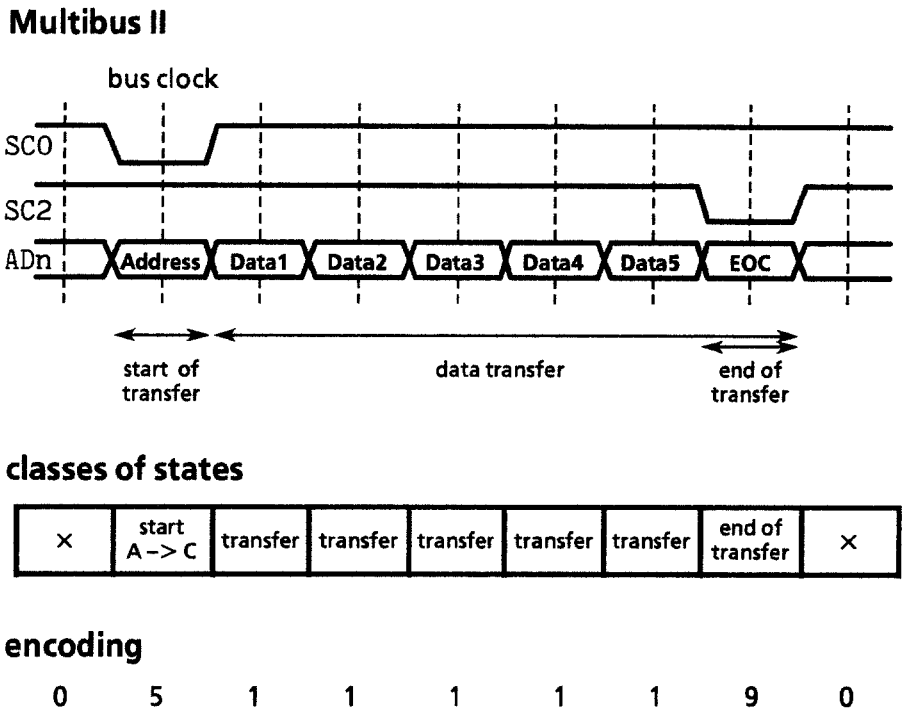

Fig. 6: Message Passing Protocol of the Multibus II

The basic idea of this monitoring principle is to look at the actions, which take place on a processor, on a bus or in a cache system, as a sequence of states. The Multibus II in a message passing operation has the state-sequence "idle, start of transfer A→C, transfer data1, transfer data2, transfer end, idle" (fig. 6, cf. [16]). A selected set of these states can be clustered to a class-of-states (CoS). The states in a class-of-states are different only by their attributes, which are neglected by this method. In the example of fig. 6 "start of transfer A→C" is shown as CoS 5, "transfer" as CoS 1, EOC (End-Of-Cycle, transfer end) as CoS 9, and the classes-of-states out of the transfer as as CoS 0. In the evaluation, the "Transfer A→C" can be searched by searching for the regular expression "5 [1]+ 9". The acquisi-

tion of the measurement values at classes-of-states monitor consists of storing the sequence of CoS in a memory; at 32 CoS, 5 bits for the encoding are necessary. The evaluation consists of the search for the patterns. The occurence of these patterns can be shown by histograms and load-diagrams. By this method the whole protocol of a bus – or all the states of a computing system – can be represented as a sequence of CoS. The definition of the CoS – i. e. the clustering of a set of states to one class-of-state – can be done as the user likes. Due to this, it can be adapted to every requirement. To keep the flexibility of this measurement principle even in the realization of the monitor, programmable hardware should be selected, such as PALs or LCAs/FPGAs (Logic Cell Arrays, Field Programmable Gate Arrays, see e. g. [8]). It is important, that all of the system states without gaps are encoded into classes-of-states. This can easily be guaranteed by using commercial tools for the hardwre synthesis.
For the encoding of the classes-of-states $s = \log_2(\text{CoS})$ signals are necessary, i. e. for 32 CoS, only $\log_2(32) = 5$ bits are needed. A time stamp can be avoided by using this method, because all cycles of the system are recorded without any interrupt. A big advantage of this method is the possibility to scrutinize the systems with the temporal resolution of the system clock and furthermore, that no states can be forgotten. (If at the CoS-tree in fig. 7 an important system state has been forgotten, then the CoS "REST" appears very often during the measurements.)

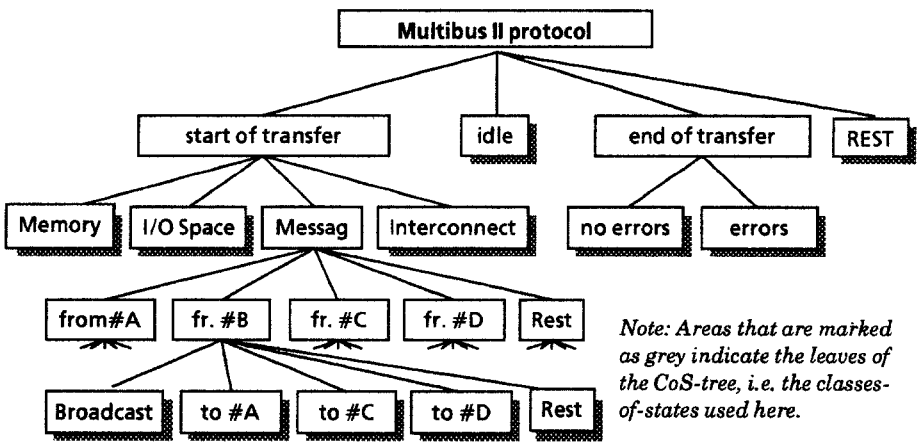

Fig. 7: CoS-tree of the Multibus II Protocol (32 CoS)

Fig. 7 shows as an example, how the protocol of the Multibus II can be divided into classes-of-states: First idle and transfer are distinguished. The transfer is subdivided into three parts (cf. fig. 5): the start of transfer, transfer itself, and the transfer end. At the start of transfer the address space can be distinguished etc. This refinement is performed as long as the protocol is

subdivided into all the CoS to be differentiated. Of course, every other CoS-tree than this one may be constructed as well as the one of fig. 7. Using this scheme, the CoS-tree can be constructed for any bus protocol, and even for the states of a computer system or of a processor cache.
Based on this principle, one can go one step ahead and record the *attributes* – i. e. the contents of address – in a special storage, the attribute-storage. By doing this, the amount of data increases, but the advantage of a simple and fast evaluation remains, and the debugging of systems becomes possible.

3.3 Hybrid Monitors

Hybrid monitors are hardware monitors with a software front-end (fig. 8). This software front-end generates some signals – in fig. 8 with the sub-routine `monitor()` – and the hardware part records these signals. If an event monitor is used as hardware part, then every call of the function `monitor()` can be considered as an event and can be recorded.

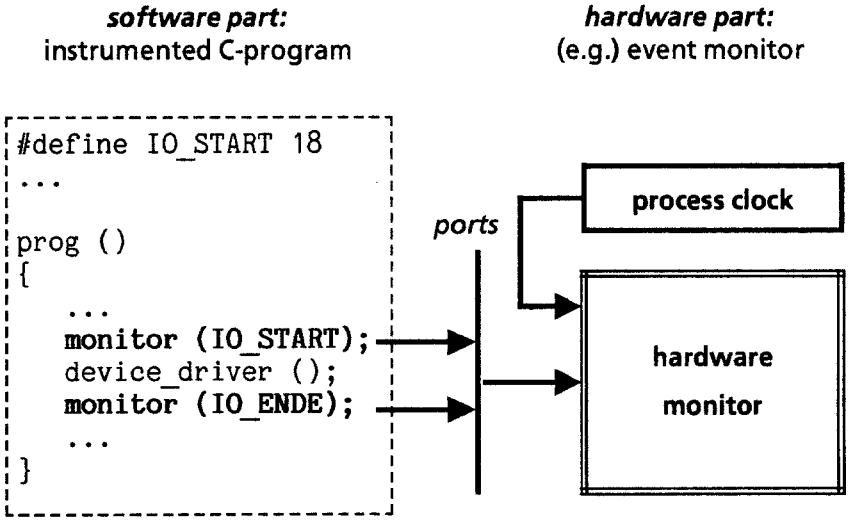

Fig. 8: Principle of a Hybrid Monitor

From the software part, the hardware of a hybrid monitor is a device which can be called by the C-function `monitor()`. The hardware part, however, only sees a lot of signals, which appear on a bus or on a port of the computer. That is why only the software part of a hybrid monitor – i. e. the procedure-call and the driver-part of the call – interferes with the system. The transfer of the signal, e. g. to a computer port, lasts some microseconds. The system load caused by a hybrid monitor is about 1% to 3%.

The examined software has to be instrumented, before a measurement can take place. Exactly spoken, in front of every interesting system-call, access to variables etc. the call `monitor()` has to be set. To differentiate the calls, a value is given with them, in fig. 8 the values `IO_START` and `IO_END`. If not only the starting time of a call is to be considered, but also the duration of system-calls, then the program has to be instrumented in front of the call and behind it. The instrumentation should be done automatically.
By the interconnection with the software it becomes possible to watch even high-level informations of the program, such as task switches, begin and end of subroutines, duration of I/O-operations (i. e. I/O-drivers), how much time a LAN-software needs in which OSI-levels (i. e. I/O-drivers), idle of the operating system, writing onto (shared memory) variables, etc.
Hybrid monitoring is not limited to the recording of software-triggered events. Virtually every hardware monitor can be used as a hybrid monitor, if a port for the output of the software information is provided. The software part can be used for a broad range of applications:

- as a single signal, e. g. as an event or as a trigger for the start and the end of the data record of a hardware monitor;
- the software-data can be treated as an additional information to the pure hardware information, e. g. to indicate the event number and types;
- in the software preprocessing can take place, and the hardware part is only to store the data.

The measurement principles of hybrid monitors are similar to those of hardware monitors, because both of these types use the same principles of recording, but they differ by the software preprocessing and the meaning of what they measure.

Software-Triggered Events. In its hardware part a hybrid monitor with software-triggered events is quite the same as an event hardware monitor. The difference to a hardware monitor is, that the monitor is not triggered by a hardware event, but by a special system-call. The system-call sends a certain "word" to a system part – e. g. to the bus – that is measured by the hardware part of the monitor. This word is considered as an event in the hardware part and it it is stored in the usual way into the trace buffer.
Quick [24] describes a hybrid monitor which examines the load of a multi processor system by setting software-probes to relevant parts in the operating system. The "relevant parts" are selected on the basis of the UNIX process model. The record (event lane) has 9 bits width and is recorded by a hardware monitor, whose clock has a temporal resolution of one microsecond. The execution of one measurement lasts between 7 µs and 15 µs. The interference with the measured system of course strongly depends on the frequency of the system-calls. Usually it ranges from 0.1% to 10%. The results of the measurement are displayed in several Gantt-diagrams.
Hofmann [15] introduces the next version of this monitor: The trace-buffer

has been enlarged to 96 Bit (40 Bit time stamp, 8 bit flags, 48 bit data) and has the length of 32 K entries. The trace-buffer can be transfered stage-by-stage, with a maximum of 10,000 events per second. The causal interdependencies between the activities of the processor – such as send and receive mechanisms – were considered as to be important. Due to this, the temporal resolution of the time stamp has been increased to 400 ns, and the clocks of the submonitors in every processor are synchronized via Ethernet by a special synchronization pulse and a pseudo-event.
In [11] the commercial "Software Analysis Workstation" of CADRE/Micro-CASE is used. The port to the monitor is a so-called "monitor-register", which can be accessed by special functions, i. e. software calls write into this register during the measurement, and the hardware parts read the words provided by the calls. One measuring event last "few microseconds". The program to be measured is instrumented by the function "write to monitor-register".

Cumulation. At the cumulation the measuring takes place in equidistant points of time. The difference to the sampling consists in the evaluation by the host before the measurement. Typically the load percentage of the system states "idle", "CPU busy" etc. is measured.

Software-Triggered Classes-of-States. The classes-of-states approach can also be used for hybrid monitoring, after some slight changes. In this case, the software-trigger is used to start and to terminate the measuring at a definite place. Between these points, the usual functions of the classes-of-states monitor are given. As long as no measurement takes place, a "pseudo class-of-states" PAUSE is written into the CoS-memory. This class-of-state PAUSE is used to keep the time correlation.
This measure makes sense if only some parts of a program, e. g. device-drivers, are to be examined. By the pseudo class-of-states PAUSE, combined with an efficient coding of the run length, the duration of the measurement can be enlarged very much.

4. System Monitors and Measuring of Performance

Considered historically, the usage of system monitors has changed its main focus: System monitors used to be used mostly for the accounting and for the debugging. Today, they are mostly used to measure the performance of a system. Even the measuring of performance has changed: Nowadays, performance measuring is not only done in computing centers, who want to have data about the system load of their machines, but more and more by programmers and system designers, who want to find performance bottlenecks in their system or program. A new usage arises to multi processor systems and LAN-coupled computers: To control efficiently the task

migration to a processor or computer in idle, a continuous survey about the system load has to be provided.

In parallel to the developments in the field of system monitors, many methods have been developed, which can be used to make statements about systems that not yet exist and to influence the design of this system before its developement. To do this, models of computing systems are built, which can be evaluated with several methods.

- *Simulation:*
  Bemmerl et al. [4] use models on which the run of several benchmarks is simulated. By varying some architectural parameters some claims about the effect of these parameters for the performance of the computing system can be made.
- *Analytical Evaluation:*
  Besides queuing models (see [12] and [2]), Timed Petri Nets are used to evaluate the performance and the reliability of new system architectures (cf. [2]). The tool which has been introduced by Klas and Lepold in [18] supports the definition and the analytical evaluation of Generalized Stochastic Petri Nets (GSPN). In [19], [20], [21] several examples are described for the usage of GSPNs for the analysis of performance and reliability of computing systems.

A problem that arises both at the simulation and at the analytical evaluation is the validation of the load profiles: How can I know that I really use a model which is near to reality? By the comparison with measured load profiles – measured by system monitors – these methods have been enhanced very much. Finally, the agreement resp. the difference to the assumptions made in the model with the real system has to be shown. Also for this case measurement data are necessary, which can be gained by system monitors.

References

[1] Adams, M.; Qian, Y.; Tomaszunas, J.; Burtscheid, J.; Kaiser, E.; Juhász, C.: Conformance Testing of VMEbus and Multibus II Products. IEEE Micro, February 1992, pp. 57-64

[2] Ajmone Marsan, M.; Balbo, G.; Conte, G.: Performance Models of Multiprocessor Systems. The MIT Press: Cambridge (Mass.), 1986

[3] Anderson, C. S.; Armstrong, K. J.; Borriello, G.: Proceedings of CS 586. PHM – A Programmable Hardware-Monitor. Technical Report # 89-09-11, University of Washington, Seattle (WA). August 1989

[4] Bemmerl, Th.; Karl, W.; Luksch, P.: Evaluierung von Architekturparametern verschiedener Rechnerstrukturen mit Hilfe von CAE-Workstations. In: Müller-Stoy, P. (Hrsg.): Architektur von Rechen-

systemen. 11. GI/ITG-Fachtagung. VDE-Verlag: Berlin 1990, S. 255-273

[5] Brinksma, E.: A Tutorial on LOTOS. Protocol Specification, Testing, and Verification, V, 1986, pp. 171-194

[6] Civera, P.; Conte, G.; del Corso, D.; Maddaleno, F.: Petri Net Models for the Description and Verification of Parallel Protocols. In: Barbacci, M. R.; Koomen, C. J. (Eds.): Computer Hardware Description Languages and their Applications. North-Holland 1987, pp. 309-325

[7] Dembinski, P.; Budkowski, S.: Specification Language Estelle. In: Diaz, M.; Ansart, J.-P.; Courtiat, J.-P.; Azema, P.; Chari, V.: The Formal Description Technique Estelle. Results of the ESPRIT/ SEDOS Project. North-Holland 1989, pp.35-76

[8] Conner, D.: High-Density PLDs. EDN, January 2, 1992, pp. 76-88

[9] Fehlau, F.; Simon, Th; Spaniol, O.; Suppan-Borowka, J.: Messungen des Leistungsverhaltens Lokaler Netze mit einem Software-Monitor. Informatik Forsch. Entwick. (1987) 2: 55-64

[10] Ferrari, D.; Serazzi, G.; Zeigner, A.: Measurement and Tuning of Computer Systems. Prentice-Hall: Englewood Cliffs 1983

[11] Föckeler, W.; Rüsing, N.: Aktuelle Probleme und Lösungen zur Leistungsanalyse von modernen Rechensystemen mit Hardware-Werkzeugen. In: Informatik-Fachberichte 218, 1989, S. 39-50

[12] Gross, D.; Harris, C. M.: Fundamentals of Queuing Theory. Wiley and Sons: New York 1985

[13] Hattenbach, J.: Hardware-Monitor-Messungen an einer SPERRY 1100/83. GWDG-Bericht Nr. 26, 1983, S. 1-21

[14] Hennessy, J. L.; Patterson, D. A.: Computer-Architecture: A Quantitative Approach. San Mateo (CA) 1990

[15] Hofmann, R.: Gesicherte Zeitbezüge beim Monitoring von Multiprozessorsystemen. In: Müller-Stoy, P. (Hrsg.): Architektur von Rechensystemen. 11. GI/ITG-Fachtagung. VDE-Verlag: Berlin 1990, S. 389-401

[16] Intel: Multibus II Bus Architecture Specification Handbook. Santa Clara (CA), 1984

[17] Keefe, D. D.: Hierarchical Control Programs for System Evaluation. IBM Systems Journal, Vol. 7, No. 2, 1968, pp. 123-133

[18] Klas, G.; Lepold, R.: TOMSPIN, a Tool for Modeling with Stochastic Petri Nets. Proc. CompEuro 92, The Hague (The Netherlands), May 1992

[19] Klas, G.; Wincheringer, Ch.: A Generalized Stochastic Petri net Model of Multibus II. Proc. CompEuro 92, The Hague (The Netherlands), May 1992

[20] Lepold, R.: Performability Evaluation of a Fault-Tolerant Multiprocessor Architecture Using Stochastic Petri Nets. Proc. 5th Int. Conf. on Fault-Tolerant Computing Systems. Nürnberg, September 1991

[21] Lepold, R.; Klas, G.: Generierung und analytische Auswertung stochastischer Petri-Netz-Modelle zur Bewertung komplexer Rechensysteme. In: Müller-Stoy, P.: Architektur von Rechensystemen. Tagungsband 11. GI/ITG-Fachtagung. vde-Verlag: Berlin 1990

[22] MIPS Computer Systems Inc.: RISC/os User's Reference manual, Vol. I (System V). Sunnyvale (CA), June 1990

[23] Nutt, G.: Tutorial: Computer System Monitors. IEEE Computer, November 1975, pp. 51-61

[24] Quick, A.: Synchronisierte Software-Messung zur Bewertung des dynamischen Verhaltens eines UNIX-Multiprozessor-Betriebssystems. In: Informatik-Fachberichte 218, 1989, S. 142-158

[25] Richter, E.: LS2 – Software-Monitor und Steuersystem für SVM. rechentechnik / datenverarbeitung 26 (1989) 3, S. 19-22

[26] Rosenbohm, W.: Messung von SVC-Ausführungszeiten mit Hilfe eines Software-Monitors. In: Mertens, B. (Hrsg.): Messung, Modellierung und Bewertung von Rechensystemen. Springer: Berlin, Heidelberg 1981, S. 58-72

[27] SCO (The Santa Cruz Operation): SCO Open Desktop Development System, Programmer's Guide, ch. 9: C Programmer's Productivity Tools. Santa Cruz (CA) 1989

[28] Svoboda, L.: Software Performance Monitors: Desing Trade-Offs. In: CMG VII Conference Proceedings, 1976, pp. 211-220

[29] Thurner, E. M.: Hardware-Monitor Using Classes of States to Detect Performance-Bottlenecks in Computer Systems. In: Krupat, C.: Proc. Supercomputing Symposium '92, Montreal 1992, pp. 328-339

[30] Thurner, E. M.: Formal Specification of Bus-Protocols and a Way to their Automatic Implementation. In: Eck, Ch. et al. (Eds.): Proc. Open Bus Systems '92. Zürich (Schweiz), 1992, pp. 123-128