ZM4/SIMPLE: a General Approach to Performance-Measurement and -Evaluation of Distributed Systems

**P. Dauphin, R. Hofmann, R. Klar,
B. Mohr, A. Quick, M. Siegle, F. Sötz**

Erlangen, Januar 1991 Technical Report 1/91

**To appear in:
T.L. Casavant and M. Singhal, eds.,
Advances in Distributed Computing: Concepts and Design,
IEEE Computer Society Press, 1992.**

ZM4/SIMPLE: a General Approach to Performance Measurement and Evaluation of Distributed Systems

Peter Dauphin, Richard Hofmann, Rainer Klar, Bernd Mohr,
Andreas Quick, Markus Siegle, Franz Sötz

Universität Erlangen-Nürnberg, IMMD VII, Martensstraße 3, D-8520 Erlangen, Germany
Tel: +49–9131–857411 — Fax: +49–9131–39388,
email: quick@immd7.informatik.uni-erlangen.de

Abstract

The performance of parallel and distributed systems is highly dependent on the degree of parallelism and the efficiency of their communication systems. Both, efficiently parallelizing big jobs and successfully designing high-speed communication systems, need insight into the dynamic behavior of at least two computers at a time. Getting insight is usually needed in debugging, here it is a means for improving performance.

First, we present a comprehensive methodology for monitoring and modeling programs in parallel and distributed systems. In using event-driven monitoring and event-oriented models there is a common abstraction, the event, which enables us to integrate both approaches.

A second part describes implementation concepts in hardware and software which render the methodology generally applicable and fruitful for practical performance evaluation problems. The hardware monitor ZM4 uses many distributed monitor agents and a global clock mechanism for achieving generality, and the evaluation environment SIMPLE uses new trace description and access principles which allow for accessing arbitrarily formatted traces, thus making standardized formats in the measured event traces superfluous.

Keywords:

parallel systems, distributed systems, hardware monitoring, hybrid monitoring, event-driven monitoring, event trace, modeling, instrumentation, performance evaluation, debugging, program visualization.

1. Introduction

Parallel and distributed data processing is intended to increase performance by distributing a workload on many computers. This way of getting high-performance from multicomputer architectures is a big leap forward; but it also produces new problems like races between concurrent programs, mutual waiting of processes, or access conflicts on interconnection networks. Obviously, it is highly desirable to understand why and where such problems exist. Event-driven measurements with appropriate monitors can provide insight and knowledge about the dynamic behavior of parallel activities and the communication between them.

To appear in: T.L. Casavant and M. Singhal, eds., Advances in Distributed Computing: Concepts and Design, IEEE Computer Society Press, 1992.

One basic idea of our performance evaluation methodology is to do more than just monitoring, but to integrate performance monitoring and modeling. Both rely on the same abstraction of dynamic program behavior: strategic points are represented as *events of interest* and the overall dynamic behavior as an *event trace*[1]. Thus, the dynamic behavior of the program(s) is abstracted to an event trace. In this paper we show how models support a systematic event specification for monitoring and how monitoring validates the models. Monitoring helps to identify *current* performance problems and to find hints for tuning (e.g. improved scheduling, mapping). The integration of monitoring and modeling is the basis for extending the (measured) knowledge about implemented programs in existing computer systems via models onto *performance prediction* of future programs and of programs in future systems. Using measured parameters makes the performance prediction more relevant.

Another basic idea of our performance evaluation concept is to bring performance evaluation out of the ghetto of splendid isolation [Fer86]. A systematic methodology is indispensable. However, it does not automatically solve real world problems. It is the desire of our research to extend the theoretical relevance of the methodology to practical use. We agree with Ferrari who argues that in the past "*the study of performance evaluation as an independent subject has sometimes caused researchers in the area to lose contact with reality*". Practical relevance means tools which help to make performance evaluation a natural part of system design and software engineering. Therefore, a set of tools puts our methodology into effect: we built a distributed hardware monitor (ZM4) and implemented a software tool environment for performance evaluation (SIMPLE).

Two architectural features have been used in the design of the ZM4 monitor. They enable ZM4 to measure multiprocessor systems as well as computer networks. The first is a distributed and open-ended monitor architecture which matches the usually distributed architecture of the observed system. The observed computer system is called *object system*. The second feature is a high-precision global monitor clock mechanism which provides globally valid time stamps. This allows for simultaneous observation of many computers *over a common time scale*. Our decision to use a hardware monitor does not mean that we are primarily interested in measuring hardware events. On the contrary, it is the dynamic behavior and the performance of concurrent software in multicomputers we are interested in. Therefore, hybrid monitoring is our favored method: the object software initiates source referenced event tokens and the hardware monitor ZM4 collects them.

The tool environment SIMPLE is a modular, comprehensive set of tools for performance evaluation and visualization based on event traces, be they monitored or generated by simulators. These event traces of arbitrary structure, format, and representation are described by the versatile event *T*race *D*escription *L*anguage TDL. All tools of SIMPLE use the *P*roblem-*O*riented *E*vent *T*race access interface POET, a library of procedures for accessing and decoding information in event traces. The paper emphasizes the importance of this access interface TDL/POET as a means for evaluating event traces of arbitrary origin, for making trace format standardization superfluous, and for enabling the evaluation environment SIMPLE to cope with the flexibility of the distributed monitor ZM4.

The paper is organized as follows: the next section describes fundamentals of event-driven monitoring. Section 3 deals with the integration of monitoring and modeling. Section 4

---

[1] In our project, event traces are analyzed in terms of performance. However, there is an interesting bridge to *debugging of concurrent programs*. Event-based debuggers examine recorded event histories, i.e. traces, for finding errors [MH89]. Performance evaluation tools do almost the same. They examine event histories for getting insight into how and where to improve performance.

describes the hardware monitor system ZM4 and section 5 the tools of the evaluation environment SIMPLE. In section 6 we briefly present three typical applications. It is shown that SIMPLE may be operationally independent of ZM4 and that using ZM4/SIMPLE in combination provides successful evaluation of high-speed communication software as well as of parallel programs.

2. Fundamentals of Event-driven Monitoring

Monitoring is either time-driven or event-driven. *Time-driven monitoring* (sampling) only allows statistical statements about the program behavior [Svo76, FSZ83]. *Event-driven monitoring*, however, reveals the dynamic flow of program activities represented by sequences of events. An *event* is an atomic instantaneous action. It can be represented as a particular value on a processor bus or in a register, or a certain point in a program. With this monitoring method the dynamic behavior of the program is abstracted to a sequence of events.

There are three monitoring techniques: hardware, software, and hybrid monitoring. Using *hardware monitoring* the event definition and recognition can be difficult and complex. An event is defined as a bit pattern and detected by the hardware monitor's probes, and it is difficult to find a problem-oriented reference[2] to the monitored programs. Using software or hybrid monitoring the events are defined by inserting instructions into the program to be measured. This is called *program instrumentation*. These measurement instructions write event tokens to a hardware interface which is available for a hardware monitor (*hybrid monitoring*) or into a reserved memory area (*software monitoring*). Program instrumentation is not needed in sampling, it always implies the use of event-driven monitoring. If events are defined by instrumenting a program, each measured event token can be clearly assigned to a point in a program, it provides a problem-oriented reference. Thus, the evaluation can be done on a level familiar to the program designer.

The essential questions, however, which occur in event driven-monitoring — regardless of the monitoring technique — are:

- *What* is the aim of measurement?
- *Which* events are necessary for modeling the functional behavior?
- *Where* should the program be instrumented so that the modeled behavior can be monitored?

As the CPU time overhead increases with the number of events detected, the instrumentation of events must be limited to those events whose tracing is considered essential for an understanding of the problems to be solved. Therefore, to define events systematically one needs knowledge about the aim of measurement ("The most important questions to be answered before attempting to monitor a machine are *what* to measure and *why* the measurement should be taken." [Nut75]). Also, knowledge about the functional behavior of the program is necessary ("The workload and its evolution in time must be at least roughly known." [FSZ83]).

Using event-driven monitoring, the dynamic behavior is represented by events which are stored as an *event trace*. Whenever the monitor device recognizes an event, it stores a data record. We call such a data record an event record or *E-record* for short. It contains the information *what* happened *when* and *where* and consists of at least an *event identification (token)* and a *time stamp*. This time stamp is generated by the monitor and does not reflect a duration but the

[2] In many cases identifiers in the source code of the object program like procedure names are already intelligible problem-oriented references. Then, "problem-oriented" and "source-reference" are synonymous. Sometimes an interesting event has no problem-oriented identifier as a counterpart. Then, it is necessary to give the respective event a problem-oriented name which is not yet defined in the source code of the object.

acquisition time of the event record. Beside these fields, an E-record contains optional fields describing additional aspects of the occurred event, e.g. a field describing the processor where the event token came from. An important performance measure is the runtime of a program. A program or a program part delimited by two events is called *activity*. The duration of an activity is defined as the difference between the time stamps of its end- and start-event.

3. Integrating Monitoring and Modeling

Being interested in the functional and dynamic behavior of parallel and distributed systems and in getting insight, we decided to use function-oriented models which explicitly model the functional interdependence of activities.

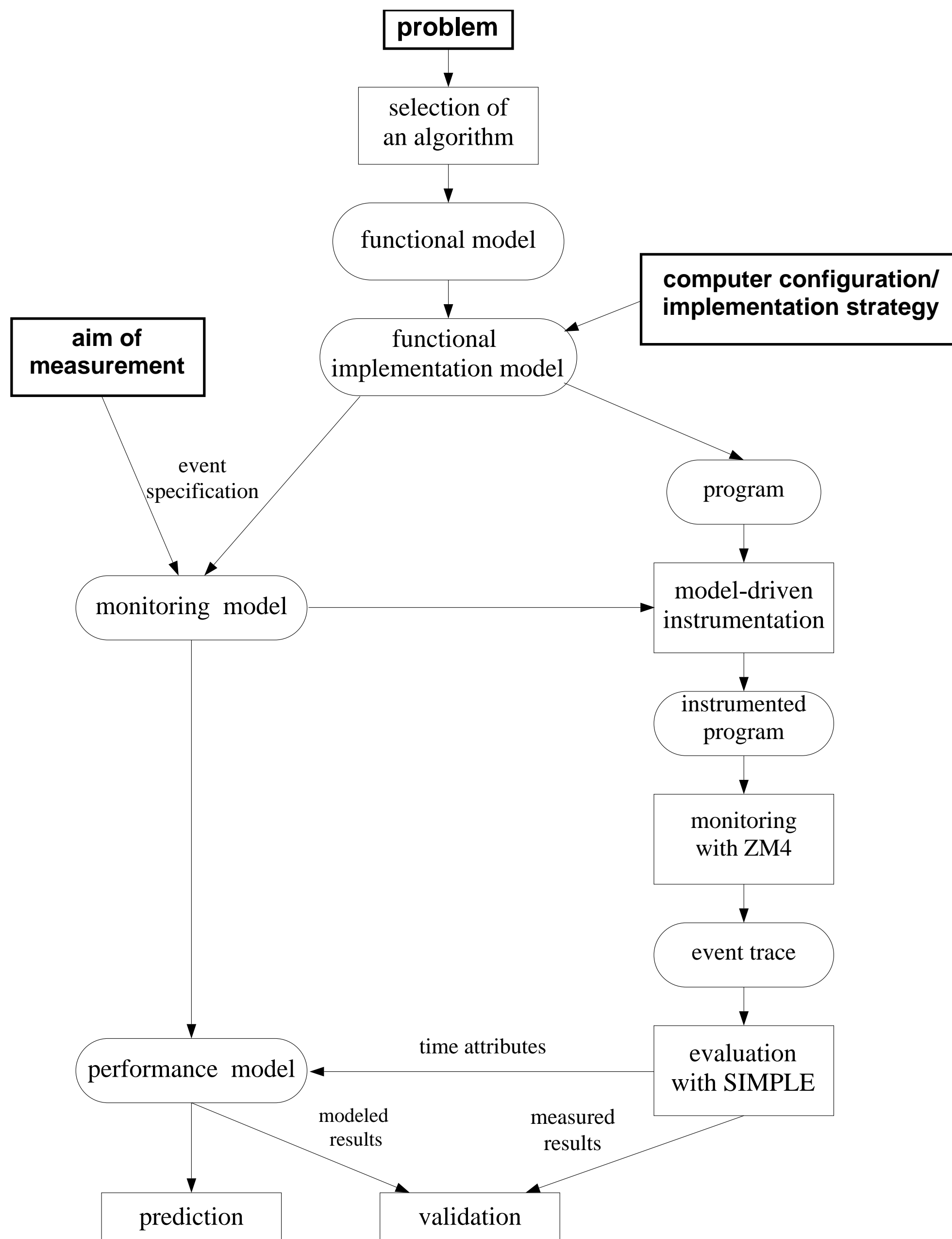

Figure 1: Model-driven monitoring

Fig. 1 shows the use of models in monitoring: specification of the problem and selection of an algorithm are a prerequisite for building a *functional model* which disregards all implementation aspects. In this model, properties of an algorithm which determine the functional behavior of a program are described. Mapping the functional model onto a given computer configuration leads to one or several implementation strategies which are modeled in the *functional implementation model*. The functional implementation model forms the basis for the implementation of the program and for the *monitoring model*. The monitoring model is a subset of the functional implementation model, i.e. it covers some but not all of the details of the implementation model. It describes the functional dependencies of the implemented program on that level of abstraction, on which the program should be monitored. The chosen level of abstraction is determined by the aim of measurement. The instrumented program results from an already implemented program and the respective monitoring model. Running the instrumented program, i.e. execution and measurement, produces an event trace as a result. The subsequent evaluation of the event traces provides overall results for validation and detailed performance parameters for assigning realistic time attributes to the program's activities. Monitoring model and time attributes build a *performance model* [Lut89]. The performance model is a prerequisite for tuning and predicting the performance of not yet implemented systems. Beside systematic event specification and performance prediction, the monitoring model can be used for automatic trace validation and as a graphical template for dynamic trace visualization (animation).

3.1. Systematic Event Specification

Instrumenting a program for event-driven monitoring means not only defining events but also defining the respective level of abstraction[3]. Here the relationship between monitoring and modeling is obvious: both techniques are based on the same definition of events. In the same way as modeling describes the flow by transitions between model states, monitoring describes transitions between program activities by events. Due to the very close connection between the flow description in modeling and monitoring, it is desirable to use the same set of events for modeling and for monitoring. The monitoring model describes the flow of the program on a level of abstraction desired for measurement and performance evaluation. The level of abstraction described in the monitoring model depends on the intention of the analysis. Thereby it is useful to first model and observe the flow on a coarse level, e.g. process level. This model reveals process concurrency, process interdependence, and process interactions but dispenses with instrumenting all procedures at the same time. Coarse level monitoring avoids an intractable flood of events. Therefore, stepwise refinement, usually used in software engineering, should also be applied for monitoring.

With modeling and monitoring integrated, the instrumentation will be simplified since the important phases of the program are regarded as black boxes and represented by states in the model. In this case the difficult question of how to instrument a program is implicitly already answered by the monitoring model, and it is tempting to derive the instrumentation automatically from the model. Thus the instrumentation need no longer be an intuitive action, it may be done systematically. A systematic procedure offers two great advantages:

- Program instrumentation can be carried out automatically with the support of tools.

[3] The idea of analyzing parallel programs in different levels of abstraction is also called "analysis using multiple views" [LMF90].

- The necessary input parameters for a performance model, e.g. runtime distributions of program phases or transition probabilities between them, can be derived from a measured event trace.

The systematic model-driven instrumentation guarantees by construction the same set of events in the monitored event trace as in the model. There is a one-to-one mapping between the model events and the measurement events. Therefore, no distinction is made between them.

The automatic instrumentation of a multigrid algorithm implemented on a multiprocessor system in the programming language C is described in [KQS91]. The modeling is done with stochastic graph models which can be generated and evaluated by the tool PEPP (*P*erformance *E*valuation of *P*arallel *P*rograms). Beside modeling, PEPP provides a command file for the model-driven automatic instrumentation with the tool AICOS (*A*utomatic *I*nstrumentation of *C O*bject *S*oftware). AICOS can be used for automatic instrumentation of programs written in C as a preprocessor. Both PEPP and AICOS have been developed at the University of Erlangen.

Another advantage of this method is the direct feedback between the monitoring results and the model. Building a model always includes intuitive deliberations. Intuition, however, can fail. Therefore it is absolutely necessary to validate the model with the help of the monitoring results. Due to the same set of events in monitoring and modeling validation is possible, i.e. it can be checked whether the model matches with the monitored behavior of the program. If the model does not match the monitored behavior there are two kinds of feedback between monitoring and modeling, i.e. for bringing monitoring into line with modeling and vice versa:

- If it may be assumed that there is a correct implementation which is to be described by a model, the original model has to be adapted to the implementation.
- If a specification given in a model is correct and should be implemented, then the implementation must be changed according to the model.

Such a correction demands a change of the program instrumentation before the next measurement. The advantages of using systematic and tool-supported automatic event specification and instrumentation are obvious: instead of a demanding and fault-prone manual re-instrumentation of the object program we can modify the monitoring model. Then a systematic, automatic instrumentation of all states described in the model is carried out. This tool support is especially helpful for the instrumentation of all ending points of a C-function which is too tedious and error-prone a task using a text editor.

The model-driven approach enables us to instrument arbitrary statements in the program under investigation on the desired level of abstraction. So, the overhead caused by instrumenting all procedures as in [AL89] can be significantly reduced. Also, monitoring is not restricted to interprocess-communication as it is done in [HC89], [JLSU87]. In [MCH$^+$90] an automatic approach to program instrumentation is presented which causes an overhead of up to 45%. We use the same multiple view concept as Leblanc [LMF90], but we support it by model-driven instrumentation which allows very easy re-instrumentation of a program. In addition to finding suspicious behavior as in debugging [MH89] and finding performance bottlenecks [BM89], the integration of modeling and monitoring enables us to predict the performance of not yet available systems and implementations.

3.2. Performance Prediction

Adding runtime distributions to the activities of a functional model, we get a performance model. Modeling enables us to compare the performance of various implementations and mappings. The use of measured distribution functions means model evaluation with realistic data and more relevant results. One application of integrating monitoring and modeling is shown in the following example.

> We used stochastic graph models as performance models to predict the speed-up of a parallel multigrid implementation on a 16–processor system. Each node of the model represented an activity like relaxation and interpolation. The analysis was started by implementing and measuring a parallel algorithm A_1. The achieved speed-up was disappointing. Analyzing the measured data, we detected a poor parallelization strategy being responsible for the bad speed-up. Using the same level of abstraction in the monitoring model as in the graph model, the measured distribution functions of algorithm A_1were valuable parameters for modeling different implementation alternatives. We were able to predict the speed-up of an improved implementation. The predicted speed-up could be confirmed by further measurements. In addition to that, the runtime of the algorithm on not yet available configurations with more than 16 processors was predicted based on the measured runtimes. According to this prediction the speed-up for this algorithm cannot be further improved by using more than 20 processors.

In this way performance prediction is integrated with monitoring. Dependent on the modeled problem, we use various modeling techniques like queueing models, timed petri nets, or stochastic graph models. Here, we describe the tools we currently use for modeling parallel programs.

For modeling parallel programs we prefer stochastic graphs. A stochastic graph consists of nodes and arcs. The nodes represent the activities of the parallel program. An arc from activity A to B means that activity A must be finished before activity B can be started. The runtime behavior of each activity is modeled by a distribution function.

Analyzing a graph we compute the runtime distribution function or the mean runtime of the modeled program. Let us consider problems which can be modeled with seriesparallel graphs[4]. If we have a seriesparallel graph, the overall runtime distribution function can be obtained by reducing the graph to one single node. The operators convolution (series reduction) and product (parallel reduction) are applied to the activities' runtime distributions. The operator convolution can be applied to the nodes A and B if the only successor of A is B and A is the only predecessor of B. The parallel reduction can be applied to the nodes A and B if the predecessor and the successor nodes are the same.

There are two ways of using the measured data for performance prediction.

1. The empirical distribution functions are approximated with appropriate parametric distribution functions.
2. The operators are directly applied to the empirical distribution functions.

Good approximations can be obtained by using exponential polynomials [Sah86], branching Erlang distributions like in MEDA [Sch87, Sch89], or distributions consisting of one deterministically and one exponentially distributed phase like in PEPP [Söt90]. The problem is that in many cases of practical relevance the analysis must be done by simulation because of the high computational costs of the known mathematical methods.

---

[4] For analyzing non-seriesparallel graphs we use the state space analysis.

The advantage of the second method is that the activities' runtime distributions may be arbitrarily distributed. In spite of this advantage the graph can be evaluated very efficiently [Kle82]. In order to compare the execution time of different implementation schemes we developed the tool SPASS (Series PArallel Structures Solver) [Pin88]. Using the empirical method, we can evaluate graphs consisting of 1000 and more nodes [SW90] in some minutes, independent of the runtime distribution shape.

4. ZM4 – a Universal Distributed Monitor System

4.1. Demands and Conceptual Issues

A monitor system, universally adaptable to computer systems with more than one processor, must fulfil several architectural demands. It must be able to

- deal with a large number of processors (nodes in the object system)
- cope with spatial distribution of the object nodes
- supply a global view of the object system
- be adaptable to different node architectures

In order to deal with a large number of processors *the monitor should be decentralized into a network of an arbitrary number of nodes* instead of being one huge monitor for the whole object system. Extending the concept of decentralization to *spatial distribution of the monitor nodes* allows the monitor system to cope with spatial distribution of the object system, too.

Supplying a global view needs methods for showing causal relationships between activities in different processors. The following considerations show that a *global clock with an accuracy better than 500 ns* provides a means to do this in any of today's parallel and distributed systems.

In distributed systems as well as in multiprocessors there is an event stream associated with each processor. As the processors are co-workers on a common task, they have to exchange information about each other, resulting in an interdependence of their event streams. *In order to globally reveal all causal relationships it suffices to order the events internal to each processor locally, and to order events concerning interprocessor communication globally.* Local ordering is automatically achieved if the events are recorded in the order of their occurrence.

A global ordering of the communication events can be achieved by the inherent causality of SEND- and RECEIVE-operations in systems communicating via message passing [Lam78]: a message can only be received after it was previously sent. But monitoring also has to show performance indices, introducing the necessity of physical time. Duda et al. describe a mechanism to estimate a global time from local observations in systems communicating via message passing [DHHB87]. Systems communicating via shared variables lack this easy mechanism to globally order events and to derive a global time. Here one processor's change of a shared variable alters the state of all processors using this variable, too. As the sequence of such accesses is arbitrary, it cannot be foreseen and there is no means to globally order such events without a global time scale.

The change of a shared variable actually affects another processor's state when it reads this variable. As the read-access to this variable can immediately follow the (state-changing) write-access, two consecutive accesses to a shared variable must be ordered correctly. Due to the asynchronous nature of multiprocessor and multicomputer systems, access conflicts can occur.

They are solved with an arbitration logic, which serializes conflicting access requests, but needs time to reach a stable state. So, a monitor clock with a global accuracy better than about half a microsecond allows to globally order communication events in systems with shared variables. As these demands on time resolution exceed those from ordering SEND-/RECEIVE-events by orders of magnitude, a monitor using a clock with that accuracy can be used universally.

A powerful monitor system should not be dedicated to just one object computer architecture. In order to enable an easy adaptation to arbitrary object systems and to fulfil the already mentioned demands, one prerequisite is a distributed architecture with a global clock. Other capabilities must be provided by the monitor nodes. These can be functionally separated into the following tasks

- *interfacing to the object system*: The monitor system must be adapted to the object system, i.e. appropriate signals in the object system must be transformed to be compatible with the monitor kernel.
- *event detection*: There must be a unit in the monitor system which is responsible for recognizing the predefined events. This unit has to prepare the event-defining signals in such a manner that the later steps of the monitoring process are supplied with compact and useful information about the events occurring in the object system.
- *time stamping and event recording*: Each event must instantaneously be assigned a globally valid time stamp allowing the ordering of all events with global interdependence. Recording of the events is necessary for post-processing, and multi-stage buffering is necessary to uncouple the event rate of the object system from the speed for writing high-volume trace files.

A pragmatic aspect is the handling of the monitor system. It must be applicable to large (i.e. many processors and/or spatial distribution) object systems as well as to small object systems. Therefore it must be flexible enough to support upgrading from small and rudimentary monitor configurations to very large ones.

Combining the features of typical monitor systems, these demands can nearly be fulfilled. Plattner developed a hardware monitor for monitoring software in real time [Pla84]. His monitor is dedicated to a single processor. This is fully adequate for his investigation: he shows that non-invasive monitoring of systems with dynamic resources, i.e. procedures with local variables, recursive calls, dynamic memory allocation etc., is very complicated.

Tsai et al. describe a monitor system which is also called a non-invasive monitor. It is aimed at monitoring of multi-microprocessors with the MOTOROLA 68000, which neither uses virtual addressing with memory protection nor caching mechanisms [TFC90]. This monitor works with a shadow-processor for each processor in the object system. Once armed, it is loaded with the internal status of the object processor and then runs in parallel with it. After the specified trigger condition is met, the status of the shadow processor, which is identical with the status of the object processor, can be investigated without disturbing the object system. The arming for the next investigation is done by issuing an interrupt to the object processor, which transfers its internal status to the shadow processor. The authors restrict the range of possible investigations to software without dynamic resources, and there is no discussion how to establish a global view of the object system.

The advantages and drawbacks of hardware, software and hybrid monitoring are analyzed by Mink et al. [MCNR90]. The authors state that hybrid monitoring allows investigations which

are not possible with pure hardware monitoring, e.g. when caches are involved. They prefer hybrid monitoring since it causes little interference on the object system. Their monitor system is built of measurement nodes which carry out the data collection. Together with a central analysis computer they are interconnected with a VME bus. A measurement node consists of a set of VLSI chips, responsible for gathering the event-defining information from the object system, time stamping the generated event records, and data buffering. As a special feature, event counters are implemented in one of the VLSI chips in order to reduce the amount of data to be transferred and evaluated. With a time resolution of 100 ns, this monitor system allows to correctly order all communication events in locally concentrated multiprocessor systems.

NETMON-II [ESZ90] is a hybrid monitoring tool for distributed and multiprocessor systems. It is a distributed master/slave system with monitor stations (slaves) and a central control station (master). Each monitor station contains a monitoring unit, a load generation unit, and a network interface for the communication with the central station, responsible for controlling the measurement and for data evaluation. The monitoring unit is implemented as an add-on card for PCs, which is dedicated to hybrid monitoring, and has an 8 bit wide Centronics printer port as the interface to the object system. Thus, interfacing, event detection, and event recording, i.e. all tasks of a monitor node, are combined on one board.

An autonomous clock with a resolution of 8 μs is part of each monitoring unit, making the monitor suitable for object systems which communicate via Send/Receive-Mechanisms. In order to establish a global timebase, these clocks are corrected every 15 ms via the time channel which connects all monitoring units. As this correction is carried out by directly accessing registers from a signal which is distributed over distances in the LAN-area, erroneous corrections due to spikes on the time channel can occur. This results in incorrect time stamps, which cannot be detected, because the clock circuitry does not distinguish between correct and incorrect pulses on the time channel.

In our opinion universal monitor systems need two more features:

- modular design of interfacing, detection, and time stamping in order to provide easy adaptability to arbitrary object systems.
- a global clock mechanism which combines high resolution, precise synchronization over large distances, and detection of synchronization errors.

4.2. Architecture of the ZM4

We have designed and implemented a universal distributed monitor system, called ZM4 (see fig. 2), which fulfils all the previously mentioned demands. It is structured as a master/slave-system with the *central control and evaluation computer* (CEC) as the master, and an arbitrary number of *monitor agents* (MA) as slaves. The distance between these MAs can be up to 1,000 meters. Conceptually, the CEC has the task to build the user interface of the whole monitor system, i.e. control the measurement activity of the MAs, store the measured data, and support the user with a powerful and universal toolset for evaluation of the measured data (see section 5).

The MAs are the nodes of the distributed monitor system. They are equipped with up to 4 *dedicated probe units* (DPUs). The MAs control the DPUs and buffer the measured event traces on their local disks. The DPUs interface the nodes of the object system and are responsible for event recognition, time stamping, and event recording with the first, high-speed stage of

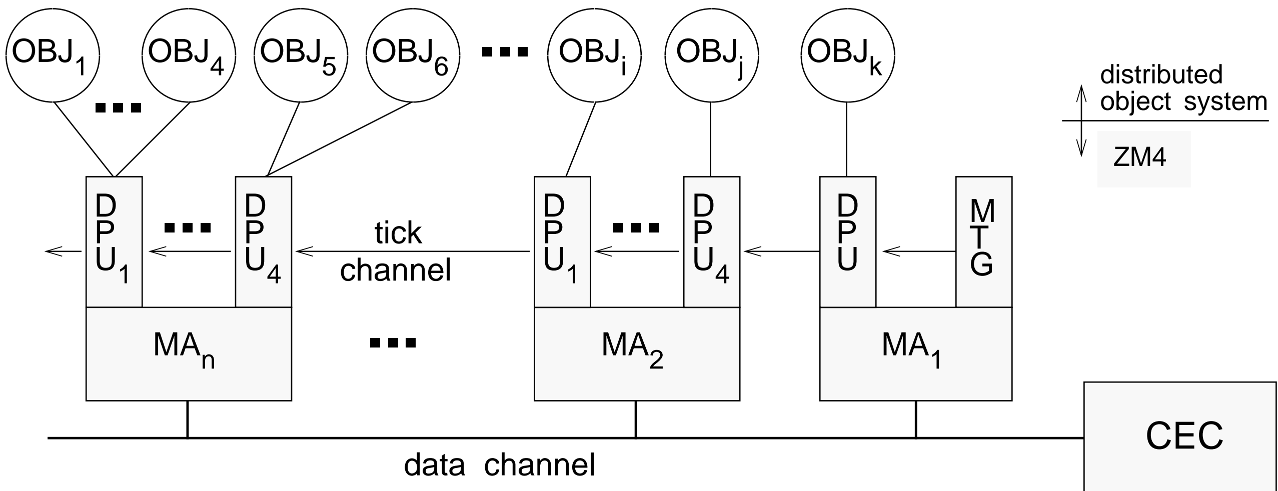

Figure 2: Distributed architecture of the ZM4

buffering. As can be seen in the following section, the DPUs are separated into specialized and general parts.

In order to establish a global time scale with the necessary resolution, a time stamping mechanism is integrated into the DPU by combining a globally synchronized clock with the event recording mechanism. The clock of each DPU gets all information necessary for preparing precise time stamps via the tick channel from the *measure tick generator* (MTG) which forms the master part of the clock synchronization.

While the tick channel together with the synchronization mechanism is our own development, we used commercially available parts for the data channel, i.e. Ethernet with TCP/IP. The data channel forms the communication subsystem of the ZM4 and it is used to distribute control information between the MAs and the CEC as well as measured data. Together with the MAs, the MTG, and the CEC, the general part of the DPU forms the universal kernel of the ZM4.

The ZM4's architectural flexibility has been achieved by two properties: Easy interfacing and a scalable architecture. The DPU can easily be adapted to different object systems, see section 4.3. ZM4 is fully scalable in terms of MAs and DPUs. The smallest configuration consists of one MA and one DPU, and can monitor up to four object nodes. Larger object systems are matched by more DPUs and MAs respectively.

4.3. Dedicated Probe Units in the Monitor Agent

The monitor agents are standard PC/AT-compatible machines. We use their expandability for adapting the kernel of the ZM4 to the various object systems. Each PC/AT provides processing power, memory resources, a hard disk, and additionally a network interface for access to the data channel.

In order to achieve the goal of a universal monitor system, the DPUs physically implement the demanded functional separation (see general DPU in fig. 3). According to the three tasks of event processing, there are also three levels of achievable generality. The *interface* has a tight connection to the object system, so it will never be universal (except probes similar to those unhandy ones of Logic Analyzers). Interfacing the object system is usually a small fraction of the whole monitoring effort and it can be done without interaction of the MA's processor. As

Figure 3: Monitor agent equipped with DPUs

At the output of the event detector a stream of events, consisting of a bit pattern and a signal for their occurrence, is available for the *event recorder*. This part is needed in general and if implemented carefully it can be used for any type of event and object (see next section).

Using hybrid monitoring, the object system itself carries out the event recognition and sends suitable event tokens to the monitor. In this case the interface and event detector can be combined to a hybrid interface which captures the information from the object system and transforms it to the protocol used by the event recorder (see simple DPU in fig. 3).

In a simple version, which we successfully use, a hybrid interface is a board containing four input connectors for a printer port on one side and the connectors for interfacing the event recorder on the other side. Between these connectors the signal lines for the data are directly routed, while the strobe lines, signalling the occurrence of the events, are filtered and buffered before routing them to the event recorder. An example of a more complex interface is our socket adapter for hybrid monitoring of Transputers. Here the signal pins of the Transputer are monitored in order to capture memory write cycles of the processor, triggered by the software instrumentation of the program.

4.4. Universal Event Recorder with a Global Clock

The event recorder has to fulfil a task common to every kind of event-driven monitoring: assigning globally valid time stamps to the incoming events, thereby building event records, and supplying a first level of high-speed buffering. We designed and implemented the event recorder for arbitrary monitoring applications. This is made possible by the architecture shown in fig. 4 and the decisions for dimensioning this component.

We start our description with one of the key features of this event recorder, i.e. the interface to the event detector. This interface is built by two functionally disjoint bundles of signals, the data path being responsible for the event description itself, and the control path signalling the occurrence of events. The signalling path is connected to the capture logic and mainly consists of four request lines (Req_i), each of them servicing an asynchronous and independent event stream. That means, up to four object nodes can be monitored with only one DPU. Additionally, each request line is paired with a grant line (Gnt_i). The grant line signals the capture of an event record. While ignoring the grant line makes no problem, Gnt_i can be used to facilitate the design of the event detector or the hybrid interface. We used this feature to ease the task of writing out event tokens via parallel ports which autonomously handle a request/grant protocol. This helped to cut down an instrumentation statement to a single output statement.

Each of the four event streams can be furnished with an arbitrary fraction of the data field, which in total supplies 48 bits. The decision how to split up this datafield and do the assignment to the event streams is postponed to the definition of the event detector's architecture. For example the event recorder can be used for one event stream with an event coding scheme using the whole width of the data path, or four event streams with eight or 16 bits, or any combination thereof.

If at least one of the request lines signals an event, the capture logic latches the data field into the data buffer in order to establish a stable signal condition for further processing of the event record. The event record is composed of the output of the data buffer, the flag register, and the clock's display register. It is written into the FIFO-memory within one cycle of the globally synchronized clock of 100 ns.

Each event stream is associated with a bit in the flag register whose active condition in the event record signals that its event stream contributed to the recording of this event (E_1 to E_4). This mechanism allows for recognizing the relevant part of an event record and ignoring the rest of the data field. A fifth event stream — internal to the monitor system — is established for the information transmitted via the tick channel (E_s). The concept of these synchronization events is described later. Coincidence of events simply causes more than one bit in the flag register to be set, meaning that their corresponding parts in the data field are valid event descriptions[5].

[5] The meaning of the E_S bit is defined by itself and has no corresponding part in the data field

Figure 5: The flag register

The overflow bit (Ov) means that at least one event has been lost due to buffer overflow. S_{OK} signals the correct operation of the event recorder clock (see later).

The synchronized oscillator together with the display register forms the slave part of a master/slave clocking scheme which is responsible for preparing globally valid time stamps with a resolution of 100 ns. This clocking scheme works on two levels, the PLL level and the token level. The PLL level is implemented as a distributed frequency synthesizer [Gar79], allowing to choose the clock frequency of the master (1 MHz) and the slave (10 MHz) according to their individual needs. Especially the data transfer rate on the tick channel (100 kHz) can be chosen to meet specifications for low-cost cabling and interfacing (RS 485) without significantly affecting the clock's precision. We have analyzed this scheme, and the measurements taken confirm a clock skew of less than 5 ns in the worst case. On the PLL level the availability of the signal on the tick channel and the lock condition of the PLL circuitry are supervised, and combined into the S_{OK}–bit in the flag register. The token level of the synchronization uses the PLL level for

the decoding of tokens, which are Manchester coded and distributed by the MTG via the tick channel. We use two different tokens, the `start_token` for starting the measurement and the `stop_token` for terminating it. While the PLL level of the clocking scheme ensures that all clocks run at the same rate, the token level is responsible for globally starting all clocks at the same 100 ns interval, thus creating a unique time scale over all event recorders.

In order to ensure the correctness of the generated time stamps the clocking scheme was extended by the concept of synchronization events, which use the previously mentioned internal event stream in the following fault tolerant protocol:

1. On a command from the CEC, transmitted via the data channel, all monitor agents reset their event recorders. This resets all clocks to zero and arms the event recorders waiting for the `start_token`.
2. After all event recorders have been armed the MTG sends the `start_token` to all event recorders via the tick channel.
3. On the receipt of the `start_token` all event recorders start their clocks and start recording events.
4. The receipt of a `start_token` creates an event on the internal event stream which is recorded as an event record with the E_S–bit in the flag register set. If this `sync_event` coincides with events from other streams, also the corresponding bits E_i of the active channels are set.
5. After fixed time intervals (selectable from 2 ms to 65536 ms in ms steps) the MTG repeats broadcasting the `start_token` which results in the corresponding `sync_events` at the event recorders.
6. A measurement is globally terminated by the MTG which broadcasts the `stop_token` to all event recorders.

In this fault tolerant protocol, the concept of `sync_events` allows to prove the correctness of all time stamps at `sync_events`, because the correct time stamp assigned to a `sync_event` is known a priori as a result of the fixed intervals for generating them. Supervising the state of the synchronization and recording this in the flag field for each event allows the extension of the proof for `sync_events` to any event between them.

If a `sync_event` is lost the interval for proving the correctness of time stamps is prolonged until the next `sync_event`. Unless there is a synchronization error in the prolonged interval, loosing a `sync_event` is irrelevant. Additionally, error recovery for corrupted time stamps is possible by this means.

Providing a bandwidth of 120 Megabytes/s at the input to the FIFO-memory, the event recorder has a peak performance of 10 million events/s. The high-speed buffering having a depth of 32 K event records not only allows hybrid monitoring but works for all kinds of event-driven monitoring which always deals with deliberately selected events and the resulting comparatively low event rates. The event rate exceeding this bare minimum necessary for event-driven monitoring can be used to record additional information. For example, if the program is instrumented in order to get a global overview and additionally a detailed view on a certain procedure, then a burst of events will be generated each time the procedure is executed. Within these bursts, the mean event rate will be exceeded by orders of magnitude.

Going down one step in fig. 4 leads us to the host interface, which is used for configuring the event recorder and for reading out the collected event records. This read-out can only be done

at the maximal rate of the host-PC/AT, resulting in a mean event rate of about 10000 events/s for one monitor agent. The buffering mechanism of the FIFO-memory allows high peak event rates; the ability to read out the FIFO-buffer while monitoring removes the restriction on the maximal length of a trace. So, a high input event rate on the one side and online buffering on the other side add to the universality of this event recorder.

The control logic is responsible for the correct operation of the event recorder. It handles the configuration information transferred over the host interface and supplies the monitor agent with status information. The configuration information defines the actually used width of the data field (16,32,48 bits), and other parameters for setting up a measurement or correctly terminating it. As status information, the clock is monitored as well as the amount of accumulated data in the FIFO-memory.

5. SIMPLE - a Performance Evaluation Environment

SIMPLE is a tool environment designed and implemented for performance evaluation of arbitrary event traces. We use it on our central evaluation computer of the ZM4. The name SIMPLE (*S*ource-related and *I*ntegrated *M*ultiprocessor and -computer *P*erformance evaluation, mode*L*ing, and visualization *E*nvironment) indicates that it is easy to use. SIMPLE has a modular structure and standardized interfaces, so that tools, which were developed and implemented by others, can be integrated into SIMPLE very easily.

5.1. The Concept for a General Logical Structure of Measured Data - the Basis for Independence of Measurement and Evaluation

The design and implementation of an evaluation system for measured data is too complex and expensive a task to be done for one special object system or monitor system only. But if the evaluation system is able to handle data produced by monitoring arbitrary parallel and distributed computer systems, the three following requirements are essential:

- *monitor independence*: As there is a great variety of parallel and distributed computer systems and applications, it is necessary to use different monitoring techniques and methods. But the measured data, recorded by different monitor devices and therefore usually differently structured, formatted, and represented, should be accessible in a uniform way.
- *source reference*: Data recorded by monitor systems are usually encoded and in a compressed form. But in the analysis and presentation of the data, the user wants to work with the problem-oriented identifiers of hardware and software objects of the monitored system.
- *object system independence*: There are many differences in structure and function of the single nodes and in the configuration of the interconnection system. There are a variety of operating systems and applications. But an evaluation system should be applicable to differently configured computer systems with a wide variety of functions.

To handle these requirements, we have to look at the measured data because this is what the evaluation system sees of the monitored system. All requirements mentioned have an effect on the structure, format, representation, and meaning of the measured data. In order to abstract from these properties we have to find a general logical structure for all the different types of

measured data. This logical structure can then be used to define a *standardized access method* to the measured data.

Using event-driven monitoring, the data resulting from the monitor is a sequence of E-records, each describing one event. An E-record consists of an arbitrary number of components, called *record fields*, each containing a single value describing one aspect of the event that occurred. In most cases an E-record has record fields containing the event identification and the time the event was recognized. It is also possible that a record field or a group of record fields is not always present in the current E-record or that a record field is interpreted differently, depending on the actual value of another record field. Therefore, it is possible that E-records have different lengths even in one event trace.

E-record fields can be classified in four basic *field types*:

TOKEN Record fields of type token contain only *one* value out of a fixed and well-defined set of constant values. A token record field is a construction similar to the enumeration types in the usual programming languages. They can be used to describe encoded information like event or processor identifications. Each value has a special, fixed meaning called interpretation.

FLAGS Record fields of type flags are like token record fields but they *can* contain *more than one* value out of a fixed well-defined set. This is done by encoding the individual values as bits which are set or not set. Similar to token values, each bit set or not set can have a special meaning also called interpretation.

TIME Record fields of type time are used to describe timing information contained in an E-record. This timing information can be of arbitrary resolution and mode (point in time or distance from previous time value).

DATA Record fields of type data in most cases contain the value of a variable of the monitored software or the contents of a register of the object system. They can be compared with variables in programming languages. It is only specified how to interpret their value. This format specification is a simple data type like integer, unsigned, or string.

Additionally, there are other types of E-record fields which are only relevant to the decoding system: first, there are record length fields, which contain the length of the current or previous E-record, and checksums. Second, fields containing irrelevant or uninteresting data, like blank fields are called filler.

Now, if during the measurement one stores the event records sequentially in a file (*event trace file*), one gets a sequence of E-records sorted according to increasing time. A section in the event trace which has been continuously recorded is called a *trace segment*. A trace segment describes the dynamic behavior of the monitored system during a completely observed time interval. The knowledge of segment borders is important, especially for validation tools based on event traces. It is possible that each trace segment begins with a special data record, the so-called *segment header*, which contains some useful information about the following segment, or is simply used to mark the beginning of a new trace segment.

With the hierarchy *event trace* / *trace segment* / *E-record* / *record field* we have a general logical structure which enables us to abstract from the physical structure and representation of the measured data. An E-record with its fields represents an event with its assigned attributes, and the event trace file the dynamic behavior expressed in streams of events.

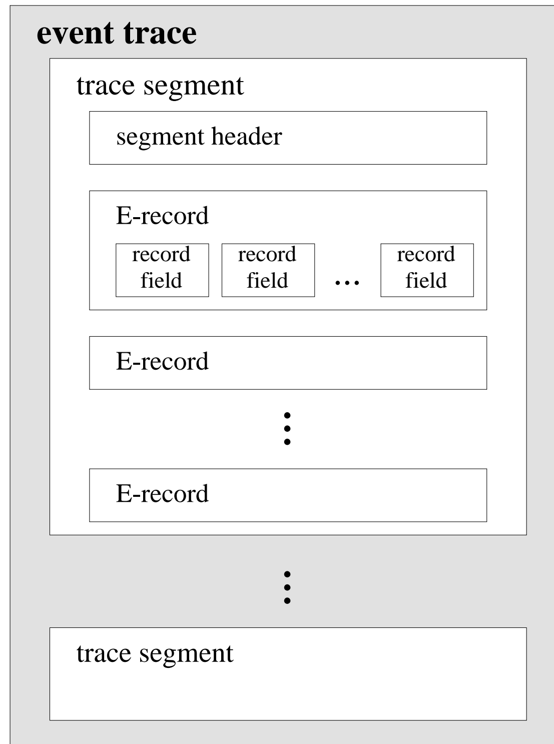

Figure 6: General event trace structure

5.2. TDL/POET - a Basic Tool for Accessing Measured Data

Based on the logical structure introduced in the last section, we designed and implemented the TDL/POET tool in order to meet the requirements listed in section 5.1. The basic idea is to consider the measured data a generic abstract data structure or an object as in object oriented programming languages. The evaluation system can access the measured data *only* via a uniform and standardized set of generic procedures. Using these procedures, an evaluation system is able to abstract from different data formats and representations and thus becomes independent of the monitor device(s) and of the object systems monitored. The tool consists of three components as shown in fig. 7:

- POET (*P*roblem *O*riented *E*vent *T*race interface): The POET library is a simple and monitor-independent function interface which enables the user to access measured data, stored in event trace files, in a problem-oriented manner. In order to be able to access and decode the different measured data, the POET functions use the *access key file* which contains a complete description of formats and properties of the measured data. For efficiency, the key file is in a binary and compact format. In addition to describing data formats and representation of the single values, the access key file includes the user-defined (problem-oriented) identifiers for the recorded values. These identifiers can now be used by the evaluation tools, thus enabling the required source reference. There is a great variety of POET functions: for example, there are functions to process the E-records in an event trace in any desired order. It is possible to process the E-records in an event trace in the order they have been recorded (`get_next`), or to move the current decoding position in the event trace relative (`forward`, `backward`) or absolute (`goto`) to a desired E-record. For each type of E-record fields POET provides an efficient and representation-independent way of getting the decoded values of a certain E-record field

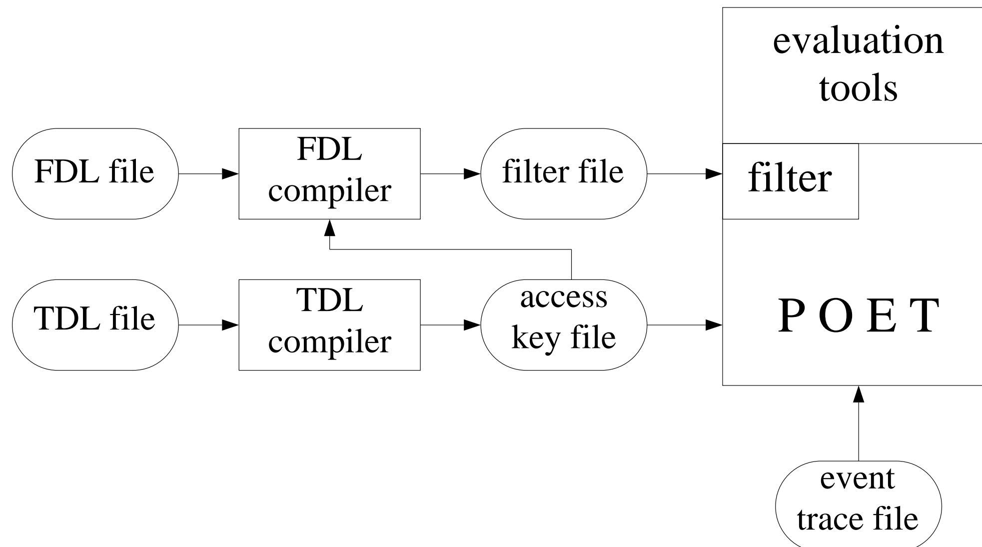

Figure 7: Event trace access with TDL/POET/FILTER

(`get_token`, `get_time`, ...). POET also provides a user-friendly way of handling time values (`set_resolution`, `print_time`, ...).

- TDL (event *T*race *D*escription *L*anguage): In order to make the construction of the access key file more user-friendly, we developed the language TDL which is designed for a problem-oriented description of event traces. The TDL compiler checks the TDL description for syntactic and semantic correctness and transforms it into the corresponding binary access key file. The development of TDL had two principal aims: the first was to make a language available which clearly and naturally reflects the fundamental structure of an event trace. The second was that even a user not familiar with all details of the language should be able to read and understand a given TDL description. Therefore, TDL is largely adapted to the English language. The notation of syntactic elements of the language and the general structure of a TDL description are closely related to similar constructs in the programming languages PASCAL and C. By writing an event trace description in TDL one provides at the same time a documentation of the performed measurement.
- Beyond that, we use a similar approach for filtering event records depending on the values of their record fields. There is an additional function to the POET library (`get_next_filtered`), which can be used to move the current decoding position within the event trace to an E-record which matches the user-specified restrictions given in a so-called filter file. These filter rules can be specified in FDL (*F*ilter *D*escription *L*anguage). Since the FDL compiler does not only read the filter description (FDL file) but also the event description (key file), the problem-oriented identifiers of the TDL file are also used for filtering.

The monitor independence enables us to analyze measured data with SIMPLE which were recorded by other monitor systems like network and logic analyzers, software monitors, or even traces generated by simulation tools. We are independent of all properties of an object system, especially of its operating system and the programming languages used. In order to adapt our environment to another kind of measurement, one only has to write a TDL description of the event trace to be analyzed. Being independent of the object system and the monitor device(s), the TDL/POET interface inherently has another advantage: as it provides a uniform interface,

the evaluation of measured data is independent of their recording. This enabled us to design and implement the tool environment SIMPLE in parallel to the design of our distributed monitor system ZM4. Additionally, POET is an open interface. This means that the user can build his own customized evaluation tools using the POET function library.

The tools TDL/POET/FILTER are implemented under the operating system UNIX in the programming language C. A prototype was designed and implemented in 1987. The growing interest and two years of experience in the use of the tool led to a complete redesign and reimplementation of the language and the related tools in 1989. The now available version 5.2 is much faster and provides more functions than the prototype [Moh90] (for details see [Moh89]).

5.3. Rating of the TDL/POET Approach

The idea of using configuration files or some sort of data description language, in order to make a system independent of the format of its input data, is used very often. Our work on TDL was inspired by the ISO standard ASN.1 (*A*bstract *S*yntax *N*otation One), which is used in some protocol analyzers to describe the format of the data packets. To the best of our knowledge, the first to use a description language for describing and filtering monitoring data was Miller in the DPM project (*D*istributed *P*rogram *M*onitor) [MMS86]. His language allows the description of name, number, and size of the components in an E-record. The description of trace structures like segments and of the physical representation of data values are not supported. Its main targets are distributed systems with send/receive communication. Unfortunately, regarding today's great number of evaluation tools, each depending on its own trace format, his approach seems not to be noticed.

In our opinion, the most important work on describing events was the definition of the event trace description language EDL by Bates and Wileden [BW82]. They also introduced the term behavioral abstraction. Their work inspired many others, among them our group. The main purpose of EDL is the definition of complex events out of primitive events. In EDL, attributes of the primitive events can be defined, but not their format or representation [Bat89].

Finally, a word on standardization: At the moment, efforts are taken to standardize the format of event traces for debugging and evaluation systems [Utt90]. We feel that standardization of the event trace format is not the right approach. No standard format can be flexible enough to represent all possible event trace formats unless format information is included in the trace, which is somewhat unhandy. Also, there is a great variety of existing (hardware) monitors which cannot produce a standardized format. Therefore, many conversion programs would have to be implemented. The TDL/POET interface shows that a generalized access method for arbitrary event traces works well. The only assumption about the trace is that it is a sequence of records each of which is a sequence of a variable number of fields. No further assumptions are made. This is flexible enough to handle all existing and future event trace formats. So, instead of standardizing the trace format we plead for standardizing the event trace access interface.

5.4. The Performance Evaluation Tools of SIMPLE

Performance evaluation of measured data, especially in large projects, can only be done if a powerful set of tools is provided. In this section, we give a short overview of the main components and the flow of data within the SIMPLE environment (see fig. 8). For a more complete overview and an example of how to use these tools see [Moh91].

Sometimes the measured data are recorded with one monitor only, but using a distributed monitor system returns a set of more than one independently recorded event traces. The first step is to generate a global event trace in order to have a global view of the whole object system (merging). It is necessary to have such a global view in order to detect and evaluate the interactions between the interdependent activities of the local object nodes. This task can be done by the tool MERGE. It takes the local event trace files and the corresponding access key files as input and generates the global event trace and the corresponding access key. The E-records of the local event traces are sorted according to increasing time.

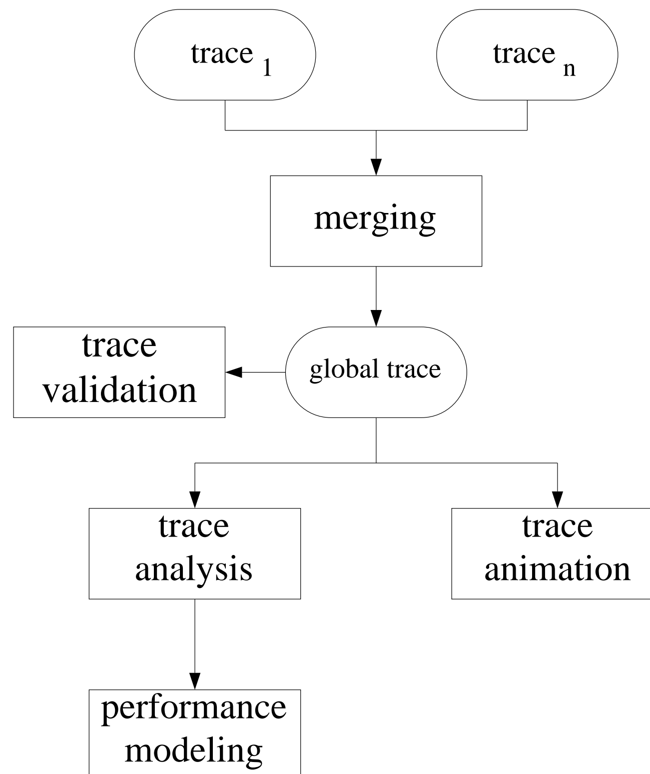

Figure 8: SIMPLE: overview

The next step is often forgotten but nevertheless necessary. Before doing any analysis it should be tested whether the measurement was performed without errors and the monitor devices have worked correctly (*trace validation*). There is a tool CHECKTRACE that performs some simple standard tests which can be applied to all event traces, and the tool VARUS (*VA*lidating *RU*les checking *S*ystem) in which the user can specify some rules in a formal language (assertions) to validate the event trace. If the validation tests were successful, we can start to analyze the data. There are three basic uses for event traces:

- The main use is what we call *trace analysis*. This can be a *standard trace analysis* for the generation of readable trace protocols (tool LIST) and computation of frequencies, durations, and other performance indices (tool TRCSTAT). The standard tools work well if overall questions are to be answered. If more complex investigations have to be done, it is better to analyze the measured data interactively and with graphics support. We call this approach *interactive trace analysis*: the event trace is stored in a relational data base (we use the commercial data analysis package S from AT&T [BCW88]) and can be used

to analyze the dynamic behavior as well as to compute performance indices. We extended the S package with additional functions to access the event traces and there description via the TDL/POET interface. The user can analyze the data interactively with a high-level programming language and has powerful graphical methods to visualize the data like histograms or time-state diagrams. An example taken from the BERKOM study, referred to in section 6, can be seen in fig. 9. Here, the transfer of one data block from the sender (host attachment) to the receiver (workstation) is depicted in time-state diagrams with a common time scale.

- The trace analysis gives performance measures such as frequencies of events and runtime distributions. These results can then be used for *performance modeling and prediction* as described in section 3.
- Third, the event traces can be used for *trace animation*. The dynamic visualization of an event trace presents the monitored dynamic behavior in a speed which can be followed by the user, exposing properties of the program or system that might otherwise be difficult to understand or might even remain unnoticed. By only displaying a single instant of time, more state information can be displayed simultaneously than in a time-state diagram. We developed the tools SMART (*S*low *M*otion *A*nimated *R*eview of *T*races), which can be used on any character-oriented terminal, and VISIMON, which offers enhanced graphic capabilities and is based on X-Windows. Here the user can specify the course and the layout of the animation in an animation description language.

For each measurement one gets a lot of related files like event trace, key, filter description, and VARUS assertion files. For the *administration* of all these files SIMPLE provides an additional tool (ADMIN). It is based on the UNIX filesystem and has a menu-driven interface. It classifies all files in the hierarchy *project* / *experiment* / *measurement* and stores additional information like date, reason, and experimenter of a measurement.

6. Experiences and Conclusion

In this article, we presented the monitor system ZM4 and the performance evaluation environment SIMPLE. The design of these tools has been guided by the idea of integrating monitoring and modeling. ZM4 is a universal hardware monitor which we have used for measurements on several parallel and distributed computer systems. Due to the modular structure of the DPUs, we could easily adapt it to different object systems. Also SIMPLE, a set of tools, proved to be capable of evaluating event traces either recorded by the ZM4 or originating from other hardware and software monitors. Three applications are now briefly presented:

Synchronous software monitoring of a parallel operating system [Qui89]

Accompanying the implementation of a parallel version of the UNIX operating system for a Concurrent 3280 MPS multiprocessor, the behavior of that system was analyzed. The 3280 MPS is a three-processor asymmetric architecture, in which only one processor can do I/O. For our analysis we used a software monitor which was implemented as an operating system function executable on all three processors independently. From each processor an event trace was written into a reserved memory area. Since the 3280 MPS has a common clock with a resolution of 1 microsecond which is available on all three processors, there exists a common time scale for all three event traces. In the 3280 MPS system a process can be scheduled to run on all three

processors alternately. One aim of the measurement was to verify and improve the operating system's scheduling strategy. This goal was achieved by detecting portions of inefficient code in a mutual exclusion mechanism of the operating system. Furthermore, an implementation bug that caused unfairness was detected by trace evaluation, and could be removed.

Monitoring a high-speed communication system [HKL+90]

As a second example, we present a study of the BERKOM[6] network, which was carried out together with the European Networking Center of IBM. BERKOM is a government-funded project for the development of fiber-based Broadband-ISDN. For this study, event traces were recorded using the ZM4 hardware monitor. The workload considered for our measurements was the transmission of rasterized images from one network node (host attachment) to another (workstation). Fig. 9 shows a typical result of this analysis.

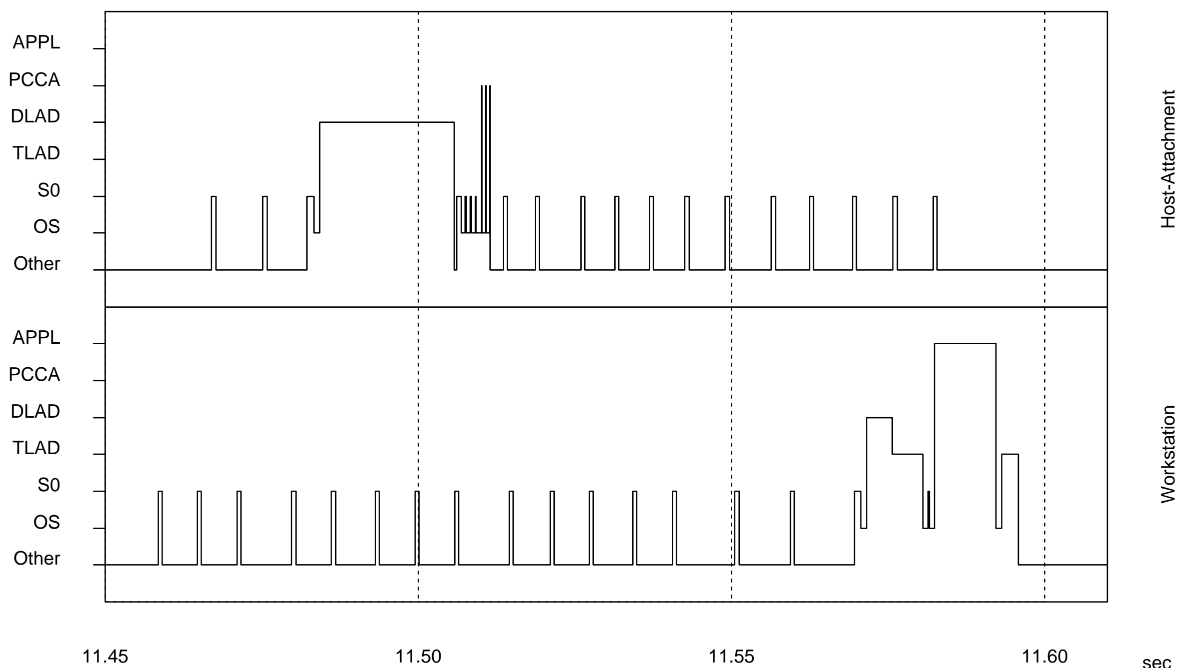

Figure 9: Example of time-state diagram

Luttenberger and Stieglitz [LS90] report that monitoring helped to eliminate two major bottlenecks in the communication system: the amount of memory-to-memory copy operations in the receive path of the transport system could be reduced. Also, the message-oriented buffer management scheme, which caused interrupts on every "buffer-return" message, could be replaced by a procedure-oriented buffer management scheme. Thus, monitoring gave valuable hints for improving the throughput.

Monitoring a communication system for a Transputer network [OQM91]

In another recent project, the behavior of a packet-oriented communication system, TRACOS, developed at the University of Erlangen, was observed. On Transputers, without using a communication system, process communication is implemented following the rendezvous concept, and only communication between neighboring Transputers is supported. With TRACOS,

[6] BERKOM stands for *BER*liner *KOM*munikationssystem

packets are buffered so that the sender does not have to wait for a packet to be received by the receiver. The other major task of TRACOS is the routing of packets between any pair of Transputers in the network. For our measurements we used a basic testbed consisting of three Transputers (T1, T2, and T3). All properties of interest could be monitored in this configuration. The hardware monitor ZM4 was adapted to each Transputer via a link and the INMOS link adapter. The application running on the Transputer network was a data transfer between Transputers T1 and T3 via T2, and an application process on Transputer T2. The effect of the communication on the application process was to be studied. For a given packet size, the CPU availability for the workload process decreased linearly with increasing packet rate. This behavior can be explained by the fact that the time for packet management and memory allocation is constant for each packet. It was found that only 65% of the theoretically possible transfer rate on a Transputer link were achieved for 1 kbyte packets. A reimplementation was suggested by the results of the measurements, which improved the performance of TRACOS by about 30 percent, so that the overall performance was close to optimal (about 85% of the link bandwidth).

Experiences gained during these and other projects showed that the design principles of ZM4 and SIMPLE are sound. Practical use of ZM4/SIMPLE confirmed that the hardware monitor ZM4 can easily be adapted to arbitrary object systems and that SIMPLE is a highly flexible and comfortable tool with which all kinds of event traces can be evaluated. Using the TDL/POET interface makes it possible to access event traces of any format and origin by simply giving a TDL description of the trace. The concept of object system independence and of integrating monitoring and evaluation tools proved to be a big step forward. Methods like ZM4/SIMPLE provide a valuable aid to designers and users of parallel and distributed systems.

References

[AL89] T.E. Anderson and E.D. Lazowska. Quartz: A Tool for Tuning Parallel Program Performance. Technical Report TR # 89-10-05, Dept. of Computer Science, Univ. of Washington, Seattle, WA 98195 USA, September 1989.

[Bat89] P. Bates. Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior. *ACM Sigplan Notices, Workshop on Parallel and Distributed Debugging*, 24(1):11–22, Januar 1989.

[BCW88] R.A. Becker, J.M. Chambers, and A.R. Wilks. *The New S Language, a Programming Environment for Data Analysis and Graphics*. Wadsworth & Brooks/Cole Advanced Books & Software, Pacific Grove, California, 1988.

[BM89] H. Burkhart and R. Millen. Performance Measurement Tools in a Multiprocessor Environment. *IEEE Transactions on Computers*, 38(5):725–737, May 1989.

[BW82] P. Bates and J.C. Wileden, editors. *A Basis for Distributed System Debugging Tools*, Hawaii, 1982. Hawaii International Conference on System Sciences 15.

[DHHB87] A. Duda, G. Harrus, Y. Haddad, and G. Bernard. Estimating Global Time in Distributed Systems. In *Distributed Systems, Proceedings of 7th Int. Conf.*, Berlin, September 1987.

[ESZ90] O. Endriss, M. Steinbrunn, and M. Zitterbart. NETMON–II a monitoring tool for distributed and multiprocessor systems. In *Proceedings of the 4th International Conference on Data Communication and their Performance, Barcelona*, June 1990.

[Fer86] D. Ferrari. Considerations on the Insularity of Performance Evaluation. *IEEE Transactions on Software Engineering*, SE–12(6):678–683, June 1986.

[FSZ83] D. Ferrari, G. Serazzi, and A. Zeigner. *Measurement and Tuning of Computer Systems*. Prentice Hall, Inc., Englewood Cliffs, 1983.

[Gar79] F.M. Gardner. *Phaselock Techniques*. John Wiley & Sons, New York, 2nd edition, 1979.

[HC89] A.A. Hough and J.E. Cuny. Initial Experiences with a Pattern–oriented Parallel Debugger. *ACM Sigplan Notices, Workshop on Parallel and Distributed Debugging*, 24(1):195–205, Januar 1989.

[HKL+90] R. Hofmann, R. Klar, N. Luttenberger, B. Mohr, A. Quick, and F. Sötz. Integrating Monitoring and Modeling to a Performance Evaluation Methodology. In T. Härder, H. Wedekind, and G. Zimmermann, editors, *Entwurf und Betrieb verteilter Systeme*, pages 122–149. Springer-Verlag, Berlin, IFB 264, 1990.

[JLSU87] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring Distributed Systems. *ACM Transactions on Computer Systems*, 5(2):121–150, 1987.

[KL86] R. Klar and N. Luttenberger. VLSI–based Monitoring of the Inter–Process–Communication of Multi–Microcomputer Systems with Shared Memory. In *Proceedings EUROMICRO '86, Microprocessing and Microprogramming, vol. 18, no. 1-5*, pages 195–204, Venice, Italy, December 1986.

[Kle82] W. Kleinöder. *Stochastic Analysis of Parallel Programs for Hierarchical Multiprocessor Systems (in German)*. Dissertation, Universität Erlangen–Nürnberg, 1982.

[KQS91] R. Klar, A. Quick, and F. Sötz. Tools for a Model–driven Instrumentation for Monitoring. In G. Balbo, editor, *Proceedings of the 5th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 159–173. Elsevier Science Publisher B.V., 1991.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[LMF90] T.J. LeBlanc, J.M. Mellor-Crummey, and R.J. Fowler. Analyzing Parallel Program Executions Using Multiple Views. *Journal of Parallel and Distributed Computing*, 9:203–217, June 1990.

[LS90] N. Luttenberger and R.v. Stieglitz. Performance Evaluation of a Communication Subsystem Prototype for Broadband–ISDN. In *Proceedings of the 2nd Workshop on Future Trends of Distributed Computing Systems in the 1990's*, Kairo, 1990.

[Lut89] N. Luttenberger. *Monitoring multiprocessor and multicomputer systems (in German)*. Dissertation, Universität Erlangen–Nürnberg, März 1989.

[MCH+90] B.P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski. IPS–2: The Second Generation of a Parallel Program Measurement System. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206–217, April 1990.

[MCNR90] A. Mink, R. Carpenter, G. Nacht, and J. Roberts. Multiprocessor Performance–Measurement Instrumentation. *Computer*, pages 63–75, September 1990.

[MH89] C.E. McDowell and D.P. Helmbold. Debugging Concurrent Programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[MMS86] B.P. Miller, C. Macrander, and S. Sechrest. A Distributed Programs Monitor for Berkeley UNIX. *Software – Practice and Experience*, 16(2):183–200, February 1986.

[Moh89] B. Mohr. TDL / POET — Version 5.1. Technical Report 7/89, Universität Erlangen–Nürnberg, IMMD VII, Juli 1989.

[Moh90] B. Mohr. Performance Evaluation of Parallel Programs in Parallel and Distributed Systems. In H. Burkhart, editor, *CONPAR 90–VAPP IV, Joint International Conference on Vector and Parallel Processing. Proceedings*, pages 176–187, Zurich, Switzerland, September 1990. Springer, Berlin, LNCS 457.

[Moh91] B. Mohr. SIMPLE: a Performance Evaluation Tool Environment for Parallel and Distributed Systems. In A. Bode, editor, *Distributed Memory Computing, 2nd European Conference, EDMCC2*, pages 80–89, Munich, Germany, April 1991. Springer, Berlin, LNCS 487.

[Nut75] G.J. Nutt. Tutorial: Computer System Monitors. *IEEE Computer*, pages 51–61, November 1975.

[OQM91] C.-W. Oehlrich, A. Quick, and P. Metzger. Monitor-Supported Analysis of a Communication System for Transputer–Networks. In A. Bode, editor, *Proceedings of the Second European Distributed Memory Computer Conference*, pages 120–129, München, 1991. Springer–Verlag, LNCS 487, Berlin.

[Pin88] H. Pingel. Stochastic Analysis of seriesparallel programs (in German). Internal study, Universität Erlangen–Nürnberg, 1988.

[Pla84] Bernhard Plattner. Real-Time Execution Monitoring. *IEEE Transactions on Software Engineering*, SE–10(6):756–764, November 1984.

[Qui89] A. Quick. Synchronized Software Measurements for Performance Evaluation of a Parallel UNIX Operating System (in German). In G. Stiege and J.S. Lie, editors, *Messung, Modellierung und Bewertung von Rechensystemen und Netzen*, pages 142–159. 5. GI/ITG–Fachtagung, Springer–Verlag, IFB 218, September 1989.

[Sah86] R.A. Sahner. *A Hybrid, Combinatorial Method of Solving Performance and Reliability Models*. PhD thesis, Dep. Comput. Sci., Duke Univ., 1986.

[Sch87] L. Schmickler. Approximation of Empirical Distribution Functions with Branching Erlang and Cox Distributions (in German). In U. Herzog and M. Paterok, editors, *Messung, Modellierung und Bewertung von Rechensystemen*, pages 265–278, Erlangen, 1987. GI/ITG, Springer, IFB 154.

[Sch89] L. Schmickler. Extension of MEDA for Describing Empirical Distribution Functions with Analytical Distribution Functions (in German). In G. Stiege and J.S. Lie, editors, *Messung, Modellierung und Bewertung von Rechensystemen und Netzen*, pages 175–189, Braunschweig, 26.–28. September 1989. GI/ITG, Springer, IFB 218.

[Söt90] F. Sötz. A Method for Performance Prediction of Parallel Programs. In H. Burkhart, editor, *CONPAR 90–VAPP IV, Joint International Conference on Vector and Parallel Processing. Proceedings*, pages 98–107, Zurich, Switzerland, September 1990. Springer–Verlag, Berlin, LNCS 457.

[Svo76] L. Svobodova. *Computer Performance Measurement and Evaluation Methods: Analysis and Applications*. Elsevier, New York, 1976.

[SW90] F. Sötz and G. Werner. Load Modeling with Stochastic Graphs for Improving Parallel Programs on Multiprocessors (in German). In *11. ITG/GI–Fachtagung Architektur von Rechensystemen*, 1990.

[TFC90] J.J.P. Tsai, K. Fang, and H. Chen. A Noninvasive Architecture to Monitor Real–Time Distributed Systems. *Computer*, pages 11–23, March 1990.

[Utt90] S. Utter. Birds-of-a-Feather session on standardizing parallel trace formats at Supercomputing '90. private communication, 1990.