Distributed Performance Monitoring: Methods, Tools and Applications

R. Hofmann, R. Klar, B. Mohr, A. Quick, M. Siegle

Universität Erlangen-Nürnberg, IMMD VII
Martensstraße 3, D-91058 Erlangen, Germany
Tel: +49–9131–857411, Fax: +49–9131–39388
email: siegle@informatik.uni-erlangen.de

Abstract

A method for analyzing the functional behavior and the performance of programs in distributed systems is presented. We use hybrid monitoring, a technique which combines advantages of both software monitoring and hardware monitoring. The paper contains a description of a hardware monitor and a software package (ZM4/SIMPLE) which make our concepts available to programmers, assisting them in debugging and tuning of their code. A short survey of related monitor systems highlights the distinguishing features of our implementation. As an application of our monitoring and evaluation system, the analysis of a parallel ray tracing program running on the SUPRENUM multiprocessor is described. It is shown that monitoring and modeling both rely on a common abstraction of a system's dynamic behavior and therefore can be integrated to one comprehensive methodology. This methodology is supported by a set of tools.

Keywords (Index Terms, Key Phrases):
hardware monitoring, hybrid monitoring, event-driven monitoring, instrumentation, modeling, performance evaluation, debugging, tuning, parallel and distributed systems, SUPRENUM.

1. Introduction

Users and operators of parallel and distributed systems often find it very difficult to exploit the immense computing power at their disposal. Writing and debugging parallel programs which use the underlying hardware in an efficient way proves to be a difficult task even for specialists. There is typically not enough insight into the internals of the hardware, the system software and their alternating effect with the user program. Bugs are hard to locate and tuning, which depends on a detailed knowledge of such factors as idle times, race conditions or access conflicts, is often not done systematically but by using ad-hoc methods. To analyze the functional behavior and the performance of a parallel program it is not enough to employ standard methods such as profiling and accounting. Sophisticated methods and tools are needed to handle these issues. Event-driven monitoring is a technique well-suited for analyzing programs running on a parallel or distributed system. It can be done by hardware, software or hybrid monitoring. We prefer hybrid monitoring which combines advantages of both hardware monitoring and software monitoring.

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*" [Fer86]. Therefore we do not only wish to develop methods, but to implement tools that assist users in writing real-world applications. We built a distributed hardware monitor (ZM4) which is scalable and can be adapted to arbitrary object systems (the system on which the program under study is running). It has a high-precision global clock which allows to monitor several nodes of the object system simultaneously, providing globally valid time stamps. Completely independent of the monitor system ZM4 is the event trace analysis system SIMPLE, which allows the evaluation of arbitrarily formatted event traces. SIMPLE is designed as a software package which comprises independent tools that are all based on a new kind of event trace access: the trace format is described in a trace description language (TDL) and evaluation tools access the event trace through a standardized interface (POET).

The analysis of parallel programs is not limited to observing the program actually implemented for an underlying distributed architecture. Many questions can be solved more easily and more efficiently by setting up a model of the program and the machine, varying parameters in the model to predict the performance of the program under various circumstances. There is a great potential to integrate event-based models and event-driven monitoring since both methods are based on the same abstraction of the dynamic behavior of the program under study: *the event*. Integrating modeling and monitoring allows to define events systematically by using a model to prepare the measurement. It allows to validate the model by comparing it to event traces of the actual program, and finally the use of measured parameters makes models more realistic.

The paper is organized as follows: the tools, i.e. the hardware monitor ZM4 and the package SIMPLE are described in section 2. This section also contains a brief survey of other monitor systems which have been described in the literature. We then present a case study (section 3) in which a parallel ray tracing program on the SUPRENUM multiprocessor is analyzed. We show how our tools enabled us to detect unexpected behavior and program bottlenecks, which helped to improve performance considerably. Current work on the integration of monitoring and modeling is presented in section 4, which also includes some concluding remarks.

2. The ZM4/SIMPLE Monitoring Environment

The Relevance of Hybrid Monitoring. Event-driven monitoring represents the dynamic behavior of a program by a sequence of events. Unlike time-driven monitoring, event-driven monitoring is suitable for efficient program analysis, as the aim of monitoring is gaining insight into the dynamic behavior of a parallel program [FSZ83]. Time-driven monitoring (sampling) provides only summary statistical information about program execution and is therefore insufficient for behavior analysis. An *event* is an atomic instantaneous action. The definition of events depends on the monitoring technique used. 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 on a processor bus or in a register. It is detected by the probes and detection circuitry of a hardware monitor. In this case it is difficult to relate the recorded signals to the monitored program, i.e. to find a problem-oriented reference. Using *software* or *hybrid monitoring*, the events are defined by inserting monitoring instructions at certain points in the program under investigation (*program instrumentation*). These instructions write event tokens into a reserved memory area of the monitored system (software monitoring), or to a hardware system interface which is accessible for a hardware monitor (hybrid monitoring). In defining events by program instrumentation, each monitored event token can be clearly assigned to a point in the program; this provides a source-related reference. Thus, the evaluation of the event trace can be done on the program level which is familiar to the program designer. As hybrid monitoring combines source-related event specification with a small interference on the object system's behavior, it is our favorite monitoring technique.

Whenever the monitor device recognizes an event, it stores a data record (a so-called *event record*). An event record contains the information *what* happened *when* and *where* and consists of at least an *event token* and a *time stamp*. The time stamp is generated by the monitor and represents the acquisition time of the event record. Beside these fields, an event record can contain optional fields describing additional aspects of the occurred event. The sequence of events is stored as an *event trace*.

It is strongly recommended to wisely restrict instrumentation to essentials. One reason is that CPU time overhead increases with the number of events issued. The other reason is that a problem should be analyzed on an adequate level of abstraction. Therefore instrumentation should be limited to those events whose tracing is considered essential for an understanding of the problems to be solved. An interesting approach to the limitation of program perturbation is presented in [RAM$^+$92]. It is called adaptive instrumentation control and is used in the Pablo performance analysis environment.

Monitoring is a great help when analyzing programs running in modern parallel or distributed systems, but it is a task too complex to be done intuitively. Therefore monitoring and especially program instrumentation should be carried out in a systematic way. This is discussed in depth in section 4.1.

2.1. ZM4 – a Universal Distributed Monitor System

2.1.1 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

(a) deal with a large number of processors (nodes in the object system),

(b) cope with spatial distribution of the object nodes,

(c) be adaptable to different node architectures,

(d) supply a global view on all interesting events in the object system for showing causal relationships between events in different nodes,

(e) provide a problem-oriented (source-related) view.

We have designed and implemented a universal distributed monitor system, called ZM4 (abbreviation for German "Zählmonitor 4"), which fulfils the demands (a) - (d). Its concepts for meeting these challenges are:

(a) In order to deal with a large number of object nodes the monitor ZM4 has a *distributed architecture*, scalable by allowing an *arbitrary number of monitor agents*.

(b) ZM4 interconnects the monitor agents by a local area network. Therefore, monitor agents need not be spatially concentrated and can also monitor *spatially distributed* object systems.

(c) ZM4 is not dedicated to just one object system but can record events from arbitrary object systems with arbitrary physical event representation.

(d) ZM4 has a *global clock* with an accuracy of 100 ns. This provides sufficient precision for establishing a global view in any of today's parallel and distributed systems.

(e) A problem-oriented view can be achieved by representing measured events and activities by the identifiers familiar to the programmer.

Issues (a) - (d) are dealt with in section 2.1.2, containing the description of the architecture of ZM4 and its major component DPU. Issue (e) is rather a problem of object instrumentation than of monitoring hardware. However, the ZM4 hardware monitor supports (e) by accepting a wide variety of physical event formats.

The following considerations are important for the notion of *global view* in distributed systems: monitoring distributed systems or multiprocessors provides an event stream for each processor. When processors are working on a common task, they have to exchange information, resulting in an interdependence of their event streams. One concept to globally reveal all causal relationships is to order events. It suffices to locally order the events of each processor and to globally order events concerning interprocessor communication. Since local ordering is automatically achieved if the events are recorded in the order of their occurrence, we can restrict the following arguments to global ordering.

In systems communicating via *message passing* a global ordering of the communication events can be achieved by the inherent causality of Send- and Receive-operations [Lam78]. If monitoring shall provide not only a functional sequence of correctly ordered events but also performance, it is necessary to introduce 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. As the read-access to a shared variable can immediately follow the (state-changing) write-access, two consecutive accesses to a shared variable must be ordered correctly. Thus, only a monitor clock with a global resolution less than memory access time allows to globally order communication events in systems with shared variables. As these demands of time resolution exceed those for ordering Send/Receive-events by orders of magnitude, a monitor using a clock with an accuracy less than memory access time can be used universally.

2.1.2 Architecture of the Hardware Monitor System ZM4

The ZM4 monitor system is structured as a master/slave system with a *control and evaluation computer* (CEC) as the master and an arbitrary number of *monitor agents* (MA) as slaves (see fig. 1). The distance between these MAs can be up to 1,000 meters. Conceptually, the CEC is the host of the whole monitor system. It controls the measurement activities of the MAs, stores the measured data and provides the user with a powerful and universal toolset for evaluation of the measured data, which is described in section 2.2.

The MAs are standard PC/AT-compatible machines equipped with up to 4 *dedicated probe units* (DPUs). We use their expandability for configuring ZM4 appropriately to the various object systems. Each MA provides processing power, memory resources, a hard disk and additionally a network interface for access to the data channel. The MAs control the DPUs and buffer the measured event traces on their local disks. The DPUs are printed circuit boards which link the MA and the nodes of the object system. The DPUs are responsible for event recognition, time stamping, event recording and for high-speed buffering of event traces.

A local clock with a resolution of 100ns and a time stamping mechanism are integrated into each DPU. The clock of a DPU gets all information for preparing precise and globally valid time stamps via the tick channel from the *measure tick generator* (MTG). Time stamps in a physically distributed configuration may be adjusted after the measurement according to the known wire length. While the tick channel together with the synchronization mechanism is our

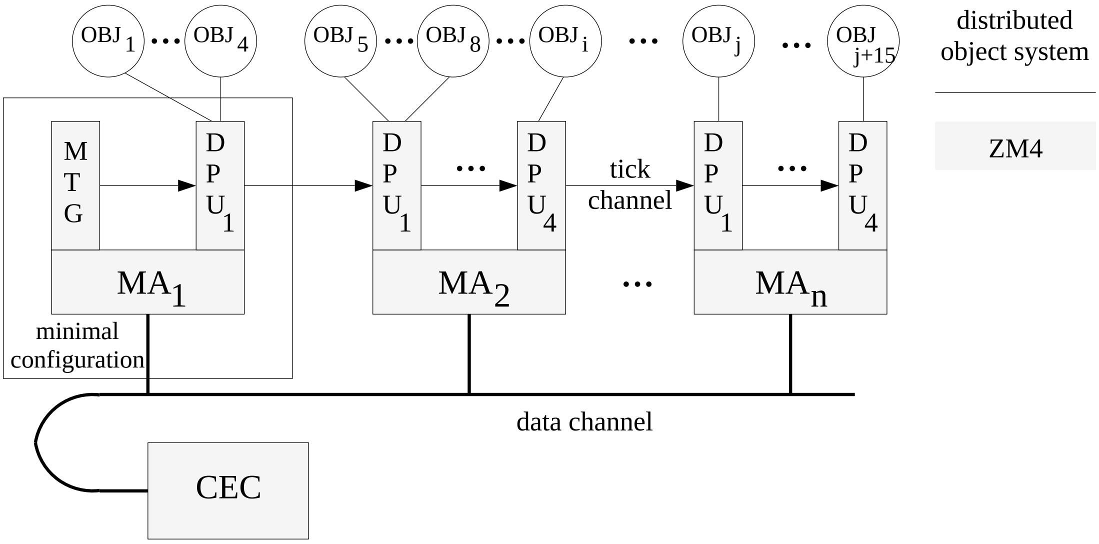

Figure 1: Distributed Architecture of ZM4

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 ZM4 and it is used to distribute control information and measured data.

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. Up to now interfaces have been built for SUN-4, DIRMU, Transputer, IBM PC, SUPRENUM and some embedded systems. ZM4 is fully scalable in terms of MAs and DPUs. The smallest configuration consists of one MA with one DPU (see fig. 1, left), and can monitor up to four object nodes. Larger object systems are matched by more DPUs and MAs, respectively. In the following, the DPU architecture, the event recorder and the globally synchronized clock are discussed in a top-down fashion.

DPU Architecture

The DPUs physically realize a functional separation into the three tasks of event processing: interfacing, event detection and event recording (see general DPU in fig. 2, left).

Dedicated DPU-Parts. The *interface* has a tight connection to the object system, so it is not intended to be universal but dedicated to the object system. The *event detector* investigates the rapidly changing information supplied by the interface in order to recognize the events of interest and to supply the event recorder with appropriate information about each event. The complexity of the event detector largely depends on the type of measurement: to recognize predefined statements in a program running on a processor without instruction cache and memory management unit, a set of comparators or a memory-mapped comparison scheme suffices. If the object system uses a processor with a hardware cache, or if predefined sequences of statements are intended to trigger an event, much more complex recognition circuits will be necessary [KL86]. In some cases of hybrid monitoring, the object system itself presents event tokens in form of parallel bit patterns. In this case no event detector is needed and a bitparallel interface captures the event tokens directly from the object system (see simple DPU in fig. 2, right).

Universal DPU-Part. The universal part of a DPU is the *event recorder*. It is completely independent of the object system. It receives a bit pattern from the event detector or the

Figure 2: Monitor Agent equipped with DPUs

bitparallel interface, triggered by a signal for its occurrence. Its functionality includes event capturing, time stamping, event record definition and event record buffering.

Universal Event Recorder

The event recorder has to fulfil two tasks: assigning globally valid time stamps to the incoming event tokens, thereby building event records, and supplying a first level of high-speed buffering.

The interface between event detector or bitparallel interface and event recorder is a data path transferring the event token itself, and a control path signalling the occurrence of events. The control path mainly consists of four request lines ($\mathrm{Req_i}$) and four grant lines ($\mathrm{Gnt_i}$), each pair $\mathrm{Req_i}/\mathrm{Gnt_i}$ servicing an asynchronous and independent event stream. That means, up to four object nodes can be monitored with only one DPU.

Each of the four event streams can be furnished with an arbitrary fraction of the data field, which in total supplies 48 bits. If at least one of the request lines signals an event, the DPU's capture logic latches the data field into a 48 bit data buffer in order to establish a stable signal condition for further processing. The output of this data buffer together with the flag register (8 bit) and the clock's display register (40 bit) define a 96 bit physical event record. This is written into the FIFO memory within one 100 ns cycle of the globally synchronized clock.

Each event stream is associated with a bit ($\mathrm{E_1}$ to $\mathrm{E_4}$) in the flag register which indicates that its event stream contributed to a valid event. This mechanism allows to recognize the relevant part(s) of the data field and ignore the rest of it. Coincidence of events in different streams is possible. Then more than one bit in the flag register is set, meaning that their corresponding parts in the data field are valid event descriptions. There is an additional bit $\mathrm{E_s}$ which indicates that a fifth event stream — internal to the monitor system — has generated a synchronization event from decoding the information transmitted via the tick channel. The transmitted synchronization

Figure 3: Universal Event Recorder

information supports a sophisticated fault-tolerant protocol which allows to prove the correctness of all time stamps at the synchronization events and confirms a clock skew of less than 5 ns. This is described below.

Providing a bandwidth of 120 Megabytes/s at the input of 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 helps to partly overcome the restricted bandwidth (10,000 events/s) of the monitor agent's local disc: for a short time bursts of events can be recorded and buffered in the FIFO even if the mean event rate of the disc will be exceeded by orders of magnitude. In case of a buffer overflow, a flag is set in the following event record. A second advantage of the FIFO buffer architecture is the ability to read the FIFO buffer while monitoring is going on. This enables continuous monitoring, i.e. there is no restricted maximal length of a trace. So, a high input event rate and arbitrary trace length add to the universality of this event recorder.

Globally Synchronized Clock. In order to achieve the desired global precision of 100 ns at a guaranteed accuracy, a two-level synchronization mechanism was developed, whose main blocks are shown in figure 4. The basic PLL-level [Gar79] ensures that all clocks proceed at exactly the same rate and filters the synchronization signal on the tick channel, whereas the token level is responsible for the global start and stop of all clocks and allows to prove the correctness of all time stamps or to correct them in case of an error.

Coming from the tick channel, the Manchester coded signal MT (100 kHz) is split into two paths. The left one leads to the PLL-level which consists of a clock separator responsible for converting the Manchester coded signal into a coherent square wave signal that can be used as a reference for the following PLL-circuitry (right output). Additionally, the clock separator checks the validity of the input signal and opens the PLL's control loop (via the switch, left output) if an error is detected . In this case the local oscillator VCO which was previously correctly synchronized will not be forced to deviate from this correct frequency.

Figure 4: Synchronization Mechanism for Global Clock

Based on the PLL-level, the token level decodes the Manchester coded MT as follows: the output signal of the frequency divider is fed into the phase shifter SHIFT(pi/2) in order to generate a signal for sampling the MT with a timing suitable for retrieving the binary information in every cycle of MT. This binary data stream is shifted into a shift register within MTDEC and compared to a set of predefined token values. If one of these values is detected, the appropriate action will be executed, e.g. the start token leads to starting the display counter, and the stop token stops it.

Under normal conditions, this hardware part of the clocking scheme ensures a robust global time scale. But in spatially distributed systems, problems can arise from cabling or electromagnetic influences. In order to quickly recognize problems and to prove their absence, this scheme is enhanced by the concept of synchronization events: after fixed time intervals the MTG broadcasts tokens, called *sync_token*, which are recorded by the event recorder in the same manner as regular events from the object system (E_S-bit in the flag field). The time stamp assigned to such a synchronization event is known a priori because of the fixed intervals for generating them. This feature provides the ability to prove the correctness of all time stamps at such synchronization events. Supervising the state of the synchronization and recording this in the flag field for each event, too, allows the extension of the proof to any event between two synchronization events. Additionally, error recovery for corrupted time stamps is possible, because the differences are known at the synchronization events.

2.1.3 ZM4 in Relation to Other Monitor Systems

The following table illustrates how the features of the ZM4 architecture are related to some other interesting monitor concepts.

	Distribution	Object independence	Monitoring technique	Global view
ZM4 University of Erlangen	LAN distance	yes accepts events from arbitrary objects	hardware and hybrid monitoring	yes resolution 100 ns clock error correction
NON-INVASIVE MONITOR University of Ill., Chicago	multiprocessor	no interface dedicated to 680x0	hardware monitoring	no
TRAMS/REMS NIST, Gaithersburg	multiprocessor	partly intended for 80x86, 680x0, 32x32	hybrid monitoring	yes resolution 100 ns
BLACKBOX University of Ill., Urbana	multiprocessor	currently dedicated to CEDAR	hardware and hybrid monitoring	yes resolution 85 ns
NETMON-II University of Karlsruhe	LAN distance	yes	hybrid monitoring	yes resolution 8 μs
ELAN ETH Zuerich	local	no dedicated to M3 (68000)	hybrid monitoring	no
TOPSYS Techn. Univ. of Munich	local	no dedicated to iAPX-80386	hybrid monitoring	yes in a local environment
TMP University of Kaiserslautern	LAN distance	yes implemented for a 680x0 proc. bus	hybrid monitoring	no

NON-INVASIVE MONITOR. Tsai et al. describe a monitor system which is called a Non-Invasive Monitor. It is aimed at monitoring of multi-microprocessors based on the MOTOROLA 68000, which neither use 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.

TRAMS/REMS. The advantages and drawbacks of hardware, software and hybrid monitoring are analyzed by Mink et al. [MCNR90]. The authors state that hybrid monitoring permits 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 collect data. They are interconnected by a VME bus with a central analysis computer. 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.

BLACKBOX. At the CSRD a highly modular and flexible hardware monitor system (BLACKBOX) designed for the CEDAR multiprocessor system was built [Mal89]. It comprises modules for such different tasks as signal conditioning, counting, timing and data logging. The locally concentrated monitor system provides a high resolution global time base. It can record buffered versions of internal signals (hardware monitoring) as well as software controlled signals (hybrid monitoring). The monitor is generally configurable to the type of measurement experiment desired. For example, one module can be configured as an interval timer and another as a counter. In addition to the BLACKBOX monitor, CEDAR has a comprehensive set of other instrumentation and monitoring facilities, including parallel profiling and software tracing. Some basic performance measurements of CEDAR are described in [GJW91].

NETMON-II. 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 board for PCs, which is dedicated to hybrid monitoring, and has an 8 bit wide Centronics printer port as an 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 transmitted via LAN distances, erroneous corrections due to spikes on the time channel can occur.

ELAN. ELAN [BM89] was one of the first monitoring projects to deal with high-level monitoring tools. It was aimed at the experimental multiprocessor M3 which is a locally concentrated machine and needs no global clock.

TOPSYS. The original goal of the TOPSYS monitor system was to gather information for debugging purposes. Later, performance aspects were included. TOPSYS uses a 80386 bond-out chip for implementing the measurement interface. It has been used for monitoring nodes of an iPSC/2 configuration [BLT90].

TMP. TMP [WH90] is a hybrid monitor consisting of many TMP nodes which are connected via the TMP network to a central monitor station. The TMP nodes can monitor object nodes and object communication activities. The instrumentation philosophy "*... to view monitor and measuring as an integral and permanent part of any computer system*" corresponds to our philosophy of restricting measurement to a small set of essential events. TMP has no global clock.

This short summary of recent monitors shows that many approaches tend to prefer event-driven hybrid monitoring and that several authors suggest a monitor architecture with many monitoring nodes and one central computer for data collection.

ZM4 is distinguished from other approaches by the following features:

- The modular design of interfacing, detection and time stamping has proved to be adaptable to arbitrary object systems with small expense
- A global clock mechanism which guarantees high resolution / precise synchronization over large distances
- A global clock transmission code which supports detection of synchronization errors in the DPUs.

Several performance measurements with ZM4 (e.g. [LS90, OQ91]) proved that the ZM4 architecture is flexible and efficient. However, these experiences also show that a well working monitor needs to be accompanied by an evaluation environment which supports program instrumentation (event definition), provides a monitor-independent and problem-oriented event trace description and offers powerful tools for event trace evaluation. The following section addresses the problem of event trace description and analysis. Tools for program instrumentation are discussed in section 4.1.

2.2. SIMPLE — a Performance Evaluation Environment

SIMPLE (*S*ource-related and *I*ntegrated *M*ultiprocessor and -computer *P*erformance evaluation, mode*L*ing and visualization *E*nvironment) is a tool environment designed and implemented for analysis of arbitrarily formatted event traces. It runs on UNIX and MS-DOS systems. SIMPLE has a modular structure and standardized interfaces. Therefore it is easily extendible, and tools which were developed and implemented by others can be integrated into SIMPLE with little effort.

2.2.1 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 one monitor system only. The three following requirements are essential to make the evaluation system capable of efficiently handling measured data produced by event-driven monitoring of parallel and distributed computer systems:

- *Object system independence*: there are many differences in structure and function of the systems to be monitored, e.g. in the node architecture and in the configuration of the interconnection network. There is a variety of operating systems and applications. An evaluation system should be applicable to the measured data coming from all these differently configured computer systems, offering a wide variety of functions.
- *Monitor independence*: the measured data, recorded by different monitor devices, should be accessible in a uniform way, even if it is differently structured, formatted and represented.

- *Source reference*: data recorded by monitor systems is usually encoded and compressed. But during analysis and presentation of data, users want to work with the problem-oriented identifiers of hardware and software objects of the monitored system.

Our approach to fulfilling these requirements is to consider the fundamental structure of the measured data because this is what the evaluation system sees of the monitored system. All requirements mentioned are related to the structure, format, representation and semantics of the measured data. In order to abstract from these properties we developed 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 proposed general logical structure can focus on a *general logical event trace structure*.

This structure of an event trace is given in fig. 5. It relies on the fact that measurements store the physical event records sequentially in a file (*event trace file*), resulting in a sequence of event 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 time interval in which none of the detected events was lost. The knowledge of segment borders is important, especially for validation tools based on event traces. Usually 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. The segment header is followed by an arbitrary number of event records, each consisting of record fields, one of which represents the acquisition time of the event record. With the hierarchy *event trace* / *trace segment* / *event record* / *record field* we have a general logical structure which enables us to abstract from the physical structure and representation of the measured data.

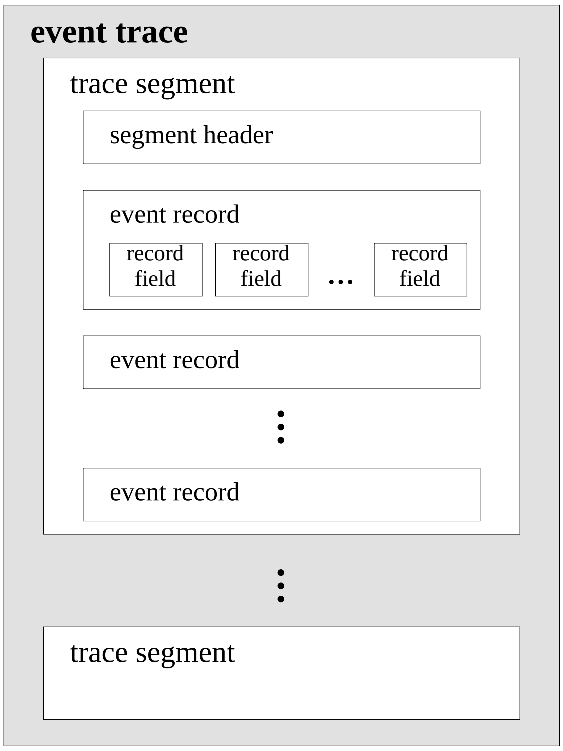

Figure 5: General Event Trace Structure

2.2.2 TDL/POET — a Basic Tool for Accessing Measured Data

Based on the general logical event trace structure, we designed and implemented the access tool TDL/POET in order to meet the mentioned requirements. The basic idea is to consider the measured data a generic abstract data structure. 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 monitored object systems. The tool consists of the three components POET, TDL and FDL [Moh91], see fig. 6:

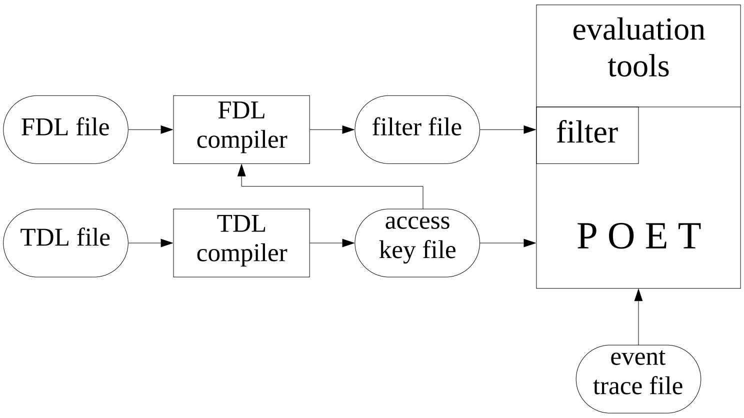

Figure 6: Event Trace Access with TDL/POET/FILTER

- TDL (*T*race *D*escription *L*anguage): the language TDL is designed for a problem-oriented description of event traces. The compilation of a TDL description into a corresponding binary access key file has the advantage that syntactic and semantic correctness is checked once and before evaluation. 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 to enable even users not familiar with all details of the language to read and understand a given TDL description. In addition, a TDL description provides a documentation of the performed measurement.
- POET (*P*roblem *O*riented *E*vent *T*race Interface): the POET library is a 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 differently structured 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, the access key file includes the user-defined (problem-oriented) identifiers for the recorded values, enabling the required source reference.

The third component, FDL, extends the capabilities of TDL/POET by allowing user-defined views on the measured data.

- FDL (*F*ilter *D*escription *L*anguage) is an approach similar to TDL. It is used for specifying rules for filtering event records depending on the values of their record fields. The problem-oriented identifiers of the TDL file are also used for filtering.

A prototype of the tools TDL/POET/FILTER was designed and implemented in 1987. A redesign took place in 1991 (version 5.2). The tools enable us to analyze event traces which were recorded by ZM4 or other monitor systems such as network analyzers, logic analyzers, software monitors, or even traces generated by simulation tools. POET is an open interface. This means that the user can build his own customized evaluation tools using the POET function library.

2.2.3 Rating of the TDL/POET Approach

Configuration files or some sort of data description language are often used in order to make a system independent of the format of its input data. 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. A similar approach to describe and filter monitoring data was used by Miller et al. in the DPM project (*D*istributed *P*rogram *M*onitor) [MMS86]. Their language allows the description of name, number and size of the components in an event record. The description of trace structures such as segments and of the physical representation of data values is not supported. Its main targets are distributed systems with Send/Receive communication.

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 physical event traces for debugging and evaluation systems [Utt90]. We feel that standardization of the physical 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. Furthermore, 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 without requiring standardized physical formats. So, we plead for standardizing the event trace access interface instead of standardizing the trace format.

2.2.4 The Performance Evaluation Tools of SIMPLE

This section gives a short overview of the main components and the flow of data within the SIMPLE environment (see fig. 7). For a more complete discussion and an application example see [Moh91].

Global View. Using a distributed monitor (assertions) for system results in several independent event traces. The first evaluation step is to generate a global event trace (tool MERGE) in order to get a global view of the whole object system. MERGE takes the local event trace files and the corresponding access key files as an input and generates a global event trace and the corresponding access key file. In the global trace, the event records of the local event traces are sorted according to their time stamps.

Trace Validation. The next step is often forgotten but nevertheless necessary. Before doing any trace analysis it should be tested whether the measurement was performed correctly. The tool CHECKTRACE performs some simple standard tests which can be applied to all event traces. The tool VARUS (*VA*lidating *RU*les checking *S*ystem) enables the user to specify some rules in a formal language

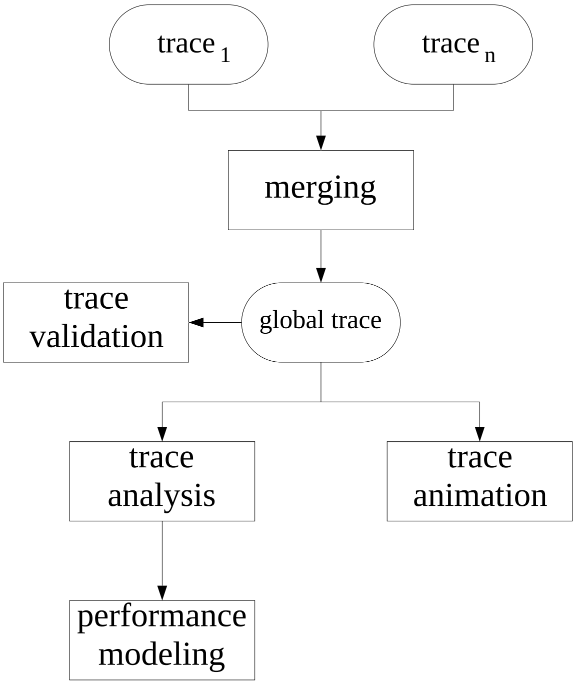

Figure 7: Evaluation with SIMPLE

validating the event trace. Trace validation in the context of model-driven monitoring is discussed in section 4.

Trace Analysis. There are two standard tools for *trace analysis*. The tool LIST for the generation of readable trace protocols and the tool TRCSTAT for computation of frequencies, durations and other performance indices.

For more complex investigations the measured data can be analyzed interactively using the data analysis package S from AT&T [BCW88]. The event trace is stored in a relational data base and analyzed via the TDL/POET interface. Typical results are histograms or pie charts. Time-state diagrams (Gantt-charts) are generated with the tool GANTT.

Trace animation. The dynamic visualization of an event trace presents the monitored dynamic behavior at 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. 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 graphics capabilities and is based on X-Windows, support the layout of the animation in an animation description language.

3. Monitoring Program Behavior on SUPRENUM

We have already gained experience using our tools in several different environments. For example, we have used ZM4/SIMPLE to analyze a communication system for Transputer networks [OQ91] and a high-speed LAN [LS90]. In this section we describe how the ZM4 hardware monitor and the SIMPLE tools were used to analyze the behavior of a parallel program on the SUPRENUM multiprocessor.

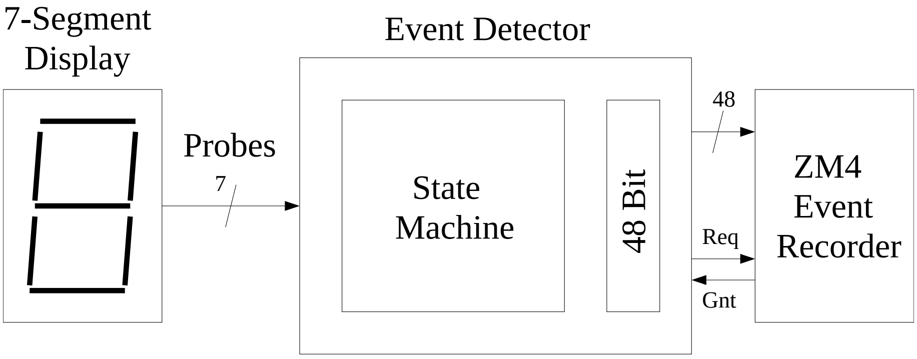

Figure 8: The Interface between SUPRENUM and ZM4

SUPRENUM [ST88, BBF91] is a loosely coupled parallel architecture consisting of up to 256 processing nodes. Each node has a CPU, 8 MBytes of memory and several coprocessors (among them a vector processing unit). Nodes are grouped in clusters of size 16, and the interconnection network is a two-level bus system: processors within the same cluster communicate via a fast parallel cluster bus, whereas inter-cluster communication is done via the torus-shaped serial SUPRENUM bus system. There is a seven segment display on each of SUPRENUM's processing nodes which, under normal operating conditions, displays the internal state of communication firmware. We use the seven segment display to output measurement data. Since the seven segment display can only display 16 different patterns (due to limitations of the driving hardware), we devised a scheme for outputting 48 bits of event data sequentially in portions of size 3 bits. An event detector was built whose probes are plugged into the socket of the seven segment display on one side and which connects to the event recorder of ZM4 on the other side (see fig. 8). The event detector reassembles the original 48-bit events which are then written to the FIFO buffer of the ZM4 event recorder.

The parallel program under study is a ray tracer. Ray tracing [Gla89] is a computer graphics method for generating high-quality images from formal descriptions of a scene. For the scope of this paper, the reader need not be familiar with ray tracing. In the implementation considered here, the algorithm is parallelized in the following way: there is one master who does administrative work, distributes jobs to an arbitrary number of servants, receives the results and writes them to an output file. The master communicates with all of the servants by message passing, but there is no communication between any two servants. Servants receive a job, work on the job (this involves mainly geometric intersection computations) and return the results to the master. One job only constitutes a small fraction of the total work to be done. The basic structure of the master and servant processes is shown in fig. 9.

A window flow control scheme is employed to ensure that each servant always has work to do: initially the master has a fixed number of credits from each servant. The master keeps sending jobs to a servant (thereby decrementing the servant's credit count) as long as there are credits from that servant available. With every result a servant returns one credit to the master. Good load balancing is achieved by this dynamic job assignment and by choosing a small job size.

Evolution of the Parallel Program

Version 1: Mailbox Communication. We started by monitoring a basic version of the ray tracing program in which SUPRENUM's mailbox mechanism was used for the communication

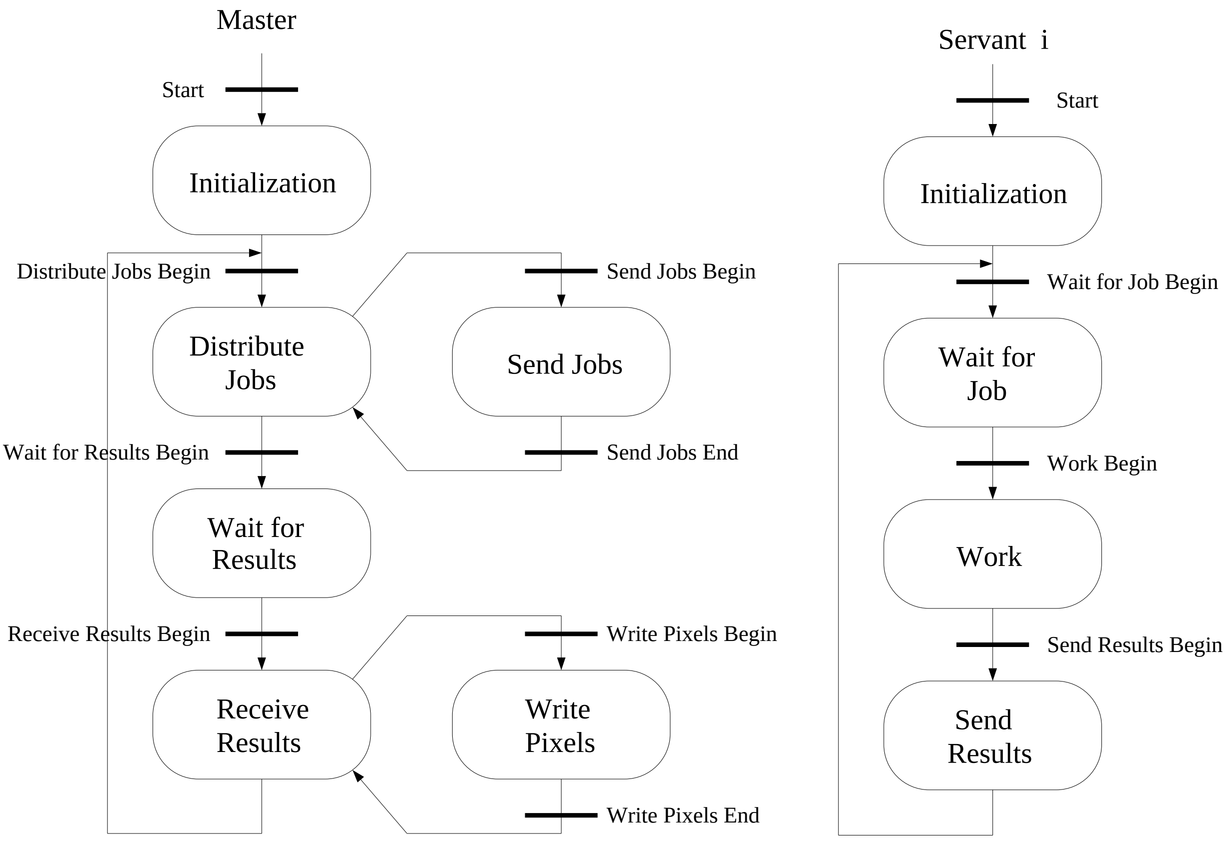

Figure 9: Basic Structure of Master and Servant Processes

between the master and the servants. Mailboxes were used in order to avoid blocking of the sender while the receiver is not ready to receive the message. The sender of a message does not hand over the message to the receiver during a rendezvous as in synchronous communication, but puts the message in the receiver's mailbox from where the receiver can pick up the message at a later time. Monitoring a run of this version of the ray tracing program in which the program was running on 16 processors (i.e. there are 15 servants) revealed that the servant processor utilization was only about 15%, which means that the servant processors spent about 85% of total time waiting for a job to arrive. This lead us to analyze the program's behavior in more detail.

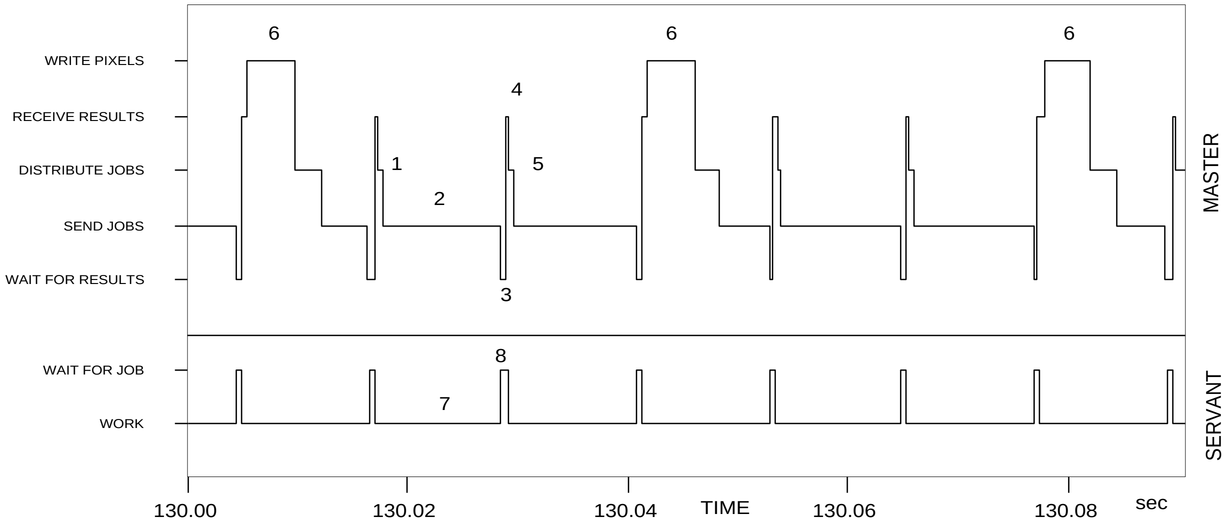

Figure 10: Behavior of the Mailbox Communication (Ray Tracer on two Processors)

Fig. 10 shows a Gantt-chart obtained from evaluating a measurement of the ray tracing program during which the ray tracer was running on two processors only, the master and one servant. In the chart, the activities of the master and the servant are shown over a common time axis.

One can observe in the chart that the master goes through the following activities in a cyclic fashion: the activities "Distribute Jobs"(1) and "Send Jobs" (2) are followed by a "Wait for Results" activity (3). Then results computed by the servant are received ("Receive Results", 4) which is followed by the next "Distribute Jobs" activity (5). Since a window flow control scheme is employed to control the number of outstanding jobs, the results received are not the results for the job just sent but for a previous job. Some of the master's cycles also contain a "Write Pixels" activity (6), during which results are written to the output file. Writing to the output file is not done in each of the master's cycles. This is because pixels have to be written to the output file in correct order. Results may not be received in the order in which the corresponding jobs have been sent, because the time to process a job varies considerably. Writing to the output file takes place whenever a continuous stretch of pixels has been processed.

We can observe from the chart in fig. 10 a major drawback. We see that the transition from "Send Jobs" to "Wait for Results" (2 → 3) on the master processor can only take place synchronously with the transition from "Work" to "Wait for Job" (7 → 8) on the servant processor. Contrary to what we expected, the master becomes blocked during the "Send Jobs" activity, which is exactly what was to be avoided by using mailbox communication. A lot of time is wasted during the "Send Jobs" activity.

The reason for this behavior is as follows: with SUPRENUM, a mailbox is a light-weight process owned by the receiving process and running together with the receiving process on one processor in a time-sharing manner. The scheduling strategy used is round robin. However, instead of using time slices each process is allowed to run until it becomes blocked. The master cannot finish his "Send Jobs" activity because he can put a message in the servant's mailbox only if this mailbox process is actually running. This is not the case until the servant relinquishes the processor because he is waiting for a message. Thus, (asynchronous) mailbox communication behaves like synchronous communication. As a result for the ray tracing application, the master cannot keep 15 servants busy because he is spending too much time being blocked while sending jobs to the servants.

Version 2: Communication Agents. Having observed this behavior, we decided to implement our own asynchronous communication in order to overcome the bottleneck at the master. For the communication from the master to the servants we introduced a pool of light-weight processes which we call communication agents. They are running on the master processor and communicate with the master via shared variables. When the master wishes to send a message (i.e. a job) to one of the servants, he indicates this to an agent who is currently unemployed. If all agents are busy a new agent is created and added to the pool. It is the agent's task to forward the master's message to the servant. Communication between the agent and the servant is synchronous, i.e. no mailboxes are used. It is the advantage of this scheme that the master can proceed with his work, while the agent becomes blocked until the receiving servant is ready to accept the message.

Versions 3 and 4: Reducing Total Communication Volume and Further Tuning. Monitoring program version 2 showed that there were no significant waiting times at the master, i.e. the master is busy all the time. However, monitoring the servants over a longer period of time showed that the servants were busy only about 29% of total time. The servants spend the rest of the time waiting for a message (another job) from the master. We decided to reduce the total volume of communication by increasing the job size. So far a job consisted of only a single pixel, which is certainly not a good choice. By sending jobs containing 50 pixels, the total number of messages is reduced by the factor 50. A job consisting of 50 pixels is still only a very small fraction of the total work to be done (256K pixels have to be processed for a 512 by 512 image), and therefore load balancing is not impaired by this step. Measurements showed that this improved the utilization of the servants to about 46%.

After further tuning which was suggested by the measurements and included the adjustment of some internal program constants, the program achieved a 60% servant utilization. Fig. 11 shows the improvement of the servant utilization. Servant utilization is a key performance measure for this application. It is directly correlated with execution time which was reduced by a factor of about 4 from version 1 to version 4. It should be mentioned that the scene used for all the above measurements is of moderate complexity only, which means that there is relatively little work to be done per pixel, resulting in a high communication overhead. Therefore this test scene constitutes a difficult test candidate for the ray tracing program. However, we needed a test scene like this since we wanted to locate all the possible bottlenecks in the program. We verified that the program performs significantly better when processing more complex scenes (over 90% servant utilization was achieved with a complex scene) which is a more realistic workload.

4. Current Work and Conclusion

We have presented a modern and powerful tool environment which has been successfully used in numerous applications, among them the study described above. We will now discuss our current research on the integration of monitoring and modeling.

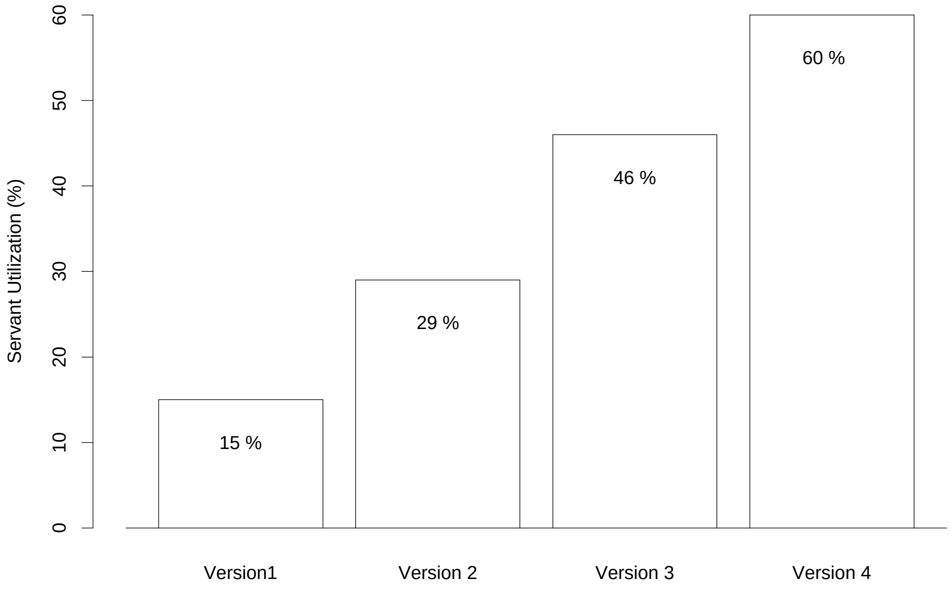

Figure 11: Improvement of Servant Utilization

4.1. Model-driven Monitoring

In order to define events systematically one needs knowledge about the aim of measurement and about the functional behavior of the system to be analyzed. We agree with Nutt's statement [Nut75]: "*The most important questions to be answered before attempting to monitor a machine are what to measure and why the measurement should be taken.*" Already in 1981, Kobayashi postulated [Kob81]: "*The system tuning requires a clear understanding of the complex interactions among the individual system components. A systematic procedure of performing this task is yet to be developed.*" In [Rei90] there is a whole chapter titled "The Event Collection Problem". These citations show that it is difficult for a designer or programmer to do monitoring systematically and efficiently.

Integration of Monitoring and Modeling. Our idea of performance evaluation, tuning and debugging is to integrate modeling and monitoring to a comprehensive methodology. When using event-based models (e.g. graph models [ST87], Petri nets [ABC84], queueing network models [Kle75] or formal specifications like CSP [Hoa85] or EXL [Her91]) which explicitly model the functional interdependence of activities, and event-driven monitoring, there is a common abstraction — the event — which enables us to integrate both methods. Both in modeling and monitoring, important points are represented as events of interest, and the overall dynamic behavior as an event trace. Due to this very close connection between the flow description in modeling and monitoring it is desirable to use the same set of events for modeling and monitoring.

Since not every detail of the functional model (e.g. a formal specification), which is used for implementing the program, is of interest in terms of performance evaluation and program analysis, the analysis of the program is carried out on a different level of abstraction. For monitoring purposes, the so-called *monitoring model* is created. This model is derived from the functional model considering the aim of measurement. It is a subset of the functional model, i.e. it covers some but not all details of the functional model. In the monitoring model the functional dependencies of the implemented program are described on the level of abstraction on which the program should be monitored.

Model-driven Instrumentation. Model-driven instrumentation guarantees by construction the same set of events in the instrumented program and in the model. The monitoring model forms the basis for the instrumentation (see fig. 12). There is a one-to-one mapping between the model events and the monitoring events. After specifying instrumentation points in the model, a command file for an automatic instrumentation tool is generated. All activities represented in the model will be instrumented in the program at their entry and all exit points [KQS92]. The difficult questions "Which events should be defined?" and "Where should the program be instrumented?" are implicitly answered by building the monitoring model.

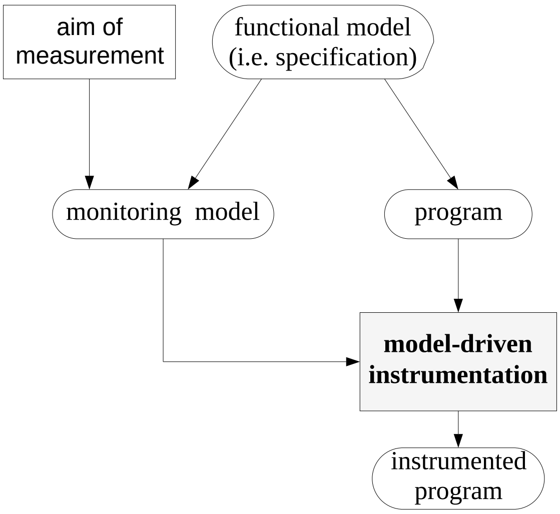
aim of measurement
functional model (i.e. specification)
monitoring model
program
model-driven instrumentation
instrumented program

Figure 12: Model-driven Instrumentation

Model-driven program instrumentation offers the following advantages over intuitive instrumentation :

- Instrumentation need no longer be an intuitive action, but may be executed systematically. Therefore, instrumentation can be carried out automatically with the support of tools.
- Instead of a fault-prone manual re-instrumentation of the program in each design phase, it suffices to modify the model. Then an automatic instrumentation of all model activities is carried out.
- The monitored event trace can be validated in a systematic way. This leads to automatic, model-driven event trace validation.
- The interpretations of the instrumented event tokens are known. This knowledge of the event semantics can be used to generate an event trace description in TDL (cf. section 2.2) automatically, allowing automatic event trace evaluation and program animation.
- The monitoring model can be transformed into a performance model by adding timing and frequency attributes (e.g. runtime distributions and transition probabilities). These attributes can be derived from a measured event trace.

Measurement, Validation and Evaluation. Model-driven instrumentation (fig. 12) is the foundation for model-driven monitoring. After instrumentation, the instrumented program can be executed and monitored which produces an event trace as a result. As the sets of events used in modeling and monitoring are the same, the monitoring model can be used for validating the dynamic behavior of the program. Validation is necessary in order to draw correct conclusions. A simple form, such as checking whether time stamps are increasing or whether the set of monitored event tokens is correct, is addressed in section 2.2.

Based on the monitoring model, the monitored behavior represented in the event trace is checked against the functional behavior of the model (model-driven event trace validation). The set of possible event sequences is defined by the model. During validation, it is checked systematically whether the behavior represented in the monitored event trace is a possible occurrence sequence in the model. This kind of validation provides hints for finding program errors (see fig. 13, arc from *validation* to *program*). If the recorded event trace matches the monitoring model the trace can be used as a base for evaluation. For tool-supported automatic event trace validation the existing modeling tools PEPP (graph models) [DHK^{+}92] and GreatSPN (Petri nets) [Chi92] were extended.

There are some other debugging tools for checking a specification against an event trace: the debugger ESCP [BDV86] can only check specified communication behavior and hierarchical fork-join parallelism, and with the TSL system [HL85] a specification is automatically checked against events generated by an Ada tasking program. As with our approach of model-driven validation, TSL also requires a linearly ordered event trace.

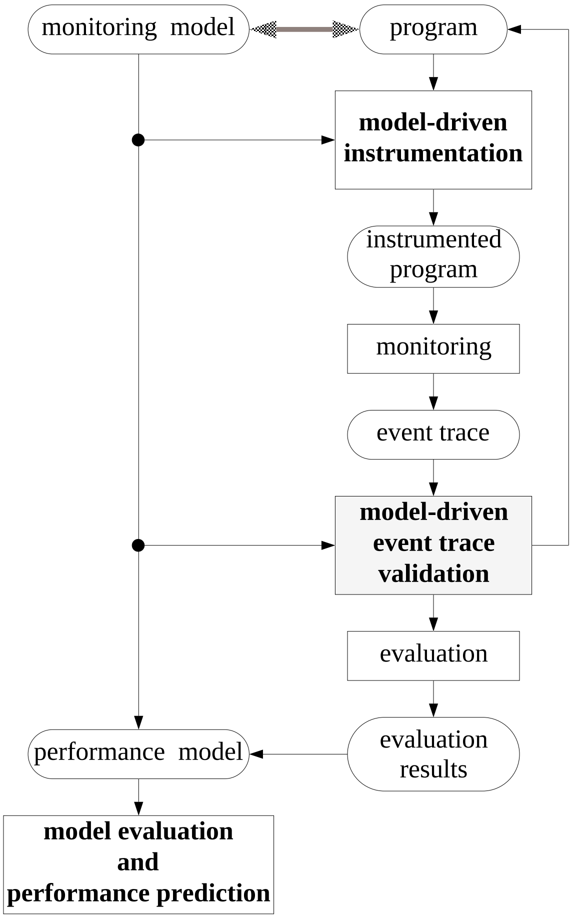

Figure 13: Model-driven Monitoring

The results of the trace evaluation allow the computation of performance indices of the measured program. They can be used for creating a *performance model*. Runtime distributions and branching probabilities will be assigned to the activities of the monitoring model. The performance model is a prerequisite for predicting the performance of not yet implemented program versions, process mappings or other computer configurations. The use of measured data means model evaluation with realistic parameters and therefore relevant results.

Summary. In model-driven monitoring the monitoring model is used for program instrumentation, event trace validation, and for creating a performance model. 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 inter-process communication as it is done in [HC89, JLSU87]. In addition to debugging and tuning (critical path analysis [MCH^{+}90]), the integration of modeling and monitoring enables us to carry out automatic event trace validation and performance prediction of not yet available systems and implementations.

The following references illustrate that model-driven monitoring is independent of the modeling paradigm. In [KQS92] the automatic, model-driven instrumentation of a parallel multigrid algorithm based on a stochastic graph model is described. The graph model can be generated and evaluated by the tool PEPP (*P*erformance *E*valuation of *P*arallel *P*rograms) which provides a command file for model-driven instrumentation with AICOS (*A*utomatic *I*nstrumentation of *C* *O*bject *S*oftware) [DHK+92]. Furthermore we have experiences applying the method to queueing network models: the dynamic behavior of a communication system for Transputer networks was modeled and monitored [Sch91, OQ91].

4.2. Conclusion

In this paper we discussed event-driven monitoring, a technique which is very well suited for analyzing the functional behavior and the performance of parallel and distributed systems. We presented the scalable distributed hardware monitor ZM4 and the SIMPLE package. Practical use of ZM4 confirmed that it can easily be adapted to arbitrary object systems and that the global time stamps are extremely helpful when analyzing concurrent activities on more than one node of a distributed system. SIMPLE is a highly flexible and comfortable tool with which all kinds of event traces (not only traces recorded by ZM4) can be evaluated. The concepts of object system independence and of integrating tools for modeling, monitoring, and trace evaluation proved to be a big step forward.

As an application of our monitoring and evaluation system, the paper includes a case study in which a parallel ray tracing program running on the SUPRENUM multiprocessor is analyzed. Monitoring helped to detect unexpected program behavior and to locate bottlenecks whose removal resulted in a dramatic improvement of the ray tracer's performance. Methods such as ZM4/SIMPLE are a valuable aid to designers and users of parallel and distributed systems.

Finally, we described how event-driven monitoring and event-based modeling can be integrated into one methodology because they both rely on the same abstraction of the dynamic behavior: the event. Integration makes it possible to carry out program instrumentation systematically and with the help of tools, thereby helping to avoid errors which can easily occur in manual instrumentation. We also explained how a model can be validated by comparing it to the sequence of events in a monitored event trace.

We wish to further test and improve our concepts and tools for integrating monitoring and modeling. Validating not only the functional aspects of a model but also considering time information is a challenging topic. Our current research also includes building a performance model of the parallel ray tracing program described in section 3. It will be possible to validate the model by comparing it to event traces of the program. The model will enable us to predict the runtime of the program on a larger number of processors.

References

[ABC84] M. Ajmone Marsan, G. Balbo, and G. Conte. A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2(2):93–122, May 1984.

[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.

[BBF91] A. Böhm, J. Brehm, and H. Finnemann. Parallel Conjugate Gradient Algorithms for Solving the Neutron Diffusion Equation. In *International Conference on Supercomputing*, pages 163–172, Cologne, June 1991 1991. ACM Press.

[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.

[BDV86] F. Baiardi, N. DeFranco, and G. Vaglini. Development of a Debugger for a Concurrent Language. *IEEE Transactions on Software Engineering*, SE-12(4):547–553, April 1986.

[BLT90] T. Bemmerl, R. Lindhof, and T. Treml. The Distributed Monitor System of TOPSYS. In H. Burkhart, editor, *CONPAR 90–VAPP IV, Joint International Conference on Vector and Parallel Processing. Proceedings*, pages 756–764, Zürich, Switzerland, September 1990. Springer, Berlin, LNCS 457.

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

[Chi92] G. Chiola. GreatSPN 1.5 Software Architecture. In G. Balbo and G. Serazzi, editors, *Proceedings of the 5th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Torino, Feburary 1991*, pages 117–132. Elsevier Science Publisher B.V., 1992.

[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.

[DHK$^+$92] P. Dauphin, F. Hartleb, M. Kienow, V. Mertsiotakis, and A. Quick. PEPP: Performance Evaluation of Parallel Programs — User's Guide – Version 3.1. Technical Report 5/92, Universität Erlangen–Nürnberg, IMMD VII, April 1992.

[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.

[GJW91] K. Gallivan, W. Jalby, and H. Wijshoff. Some Basic Performance Measurements of the 16x16 CEDAR Configuration. Technical Report 1146, Center for Supercomputing Research and Development, Urbana, Illinois, August 1991.

[Gla89] A.S. Glassner. *An Introduction to Ray Tracing*. Academic Press Limited, London, San Diego, New York, Boston, Sydney, Tokyo, Toronto, 1989.

[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.

[Her91] U. Herzog. Performance Evaluation and Formal Description. In V.A. Monaco and R. Negrini, editors, *Advanced Computer Technology, Reliable Systems and Applications, Proceedings*, pages 750–755, Bologna, May 1991. IEEE CompEuro 91, IEEE Computer Society Press.

[HL85] D. Helmbold and D. Luckham. Debugging Ada Tasking Programs. *IEEE Software*, 2(2):47–57, 1985.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[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.

[Kle75] L. Kleinrock. *Queueing Systems*, volume 1: Theory. John Wiley & Sons, 1975.

[Kob81] H. Kobayashi. *Modeling and Analysis — An Introduction to System Performance Evaluation Methodology*. Addison–Wesley, reprinted and corrected edition, October 1981.

[KQS92] 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 165–180. Elsevier Science Publisher B.V., 1992.

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

[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.

[Mal89] A.D. Malony. Multiprocessor Instrumentation: Approaches for CEDAR. In M. Simmons, R. Koskela, and I. Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, chapter 1, pages 1–33. ACM Press, Frontier Series, Addison–Wesley Publishing Company, New York, 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*, 23(9):63–75, September 1990.

[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.

[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*, 8(11):51–61, November 1975.

[OQ91] C.-W. Oehlrich and A. Quick. Performance Evaluation of a Communication System for Transputer–Networks Based on Monitored Event Traces. *ACM SIGARCH*, 19(3):202–211, May 1991. Proc. of the 18th Int. Symp. on Computer Architecture, Toronto, May 27-30, 1991.

[RAM$^+$92] D.A. Reed, R.A. Aydt, T.M. Madhyastha, R.J. Noe, K.A. Shields, and B.W. Schwartz. An Overview of the Pablo Performance Analysis Environment. Technical report, University of Illinois, Urbana, November 1992.

[Rei90] M.H. Reilly. *A Performance Monitor for Parallel Programs*. Academic Press, San Diego, CA, 1990.

[Sch91] K. Schimek. Modellierung eines Kommunikationssystems für Transputernetzwerke. Master's thesis, Universität Erlangen–Nürnberg, IMMD VII, Oktober 1991.

[ST87] R. Sahner and K. Trivedi. Performance Analysis and Reliability Analysis Using Directed Acyclic Graphs. *IEEE Transactions on Software Engineering*, SE-13(10), October 1987.

[ST88] K. Solchenbach and U. Trottenberg. SUPRENUM: System essentials and grid applications. *Parallel Computing, North–Holland*, 1988(7):265–281, 1988.

[TFC90] J.J.P. Tsai, K. Fang, and H. Chen. A Noninvasive Architecture to Monitor Real–Time

Distributed Systems. *Computer*, 23(3):11–23, March 1990.

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

[WH90] D. Wybranietz and D. Haban. Monitoring and Measuring Distributed Systems. In M. Simmons and R. Koskela, editors, *Performance Instrumentation and Visualization*, chapter 2, pages 27–45. ACM Press, Frontier Series, Addison-Wesley Publishing Company, New York, 1990.