Multi-Source Performance Analysis of Distributed Software*

Christoph Steigner, Jürgen Wilke
{steigner,wilke}@uni-koblenz.de
University of Koblenz-Landau, Rheinau 1, 56075 Koblenz, Germany

Abstract

In this paper, we argue that a combination of various monitoring techniques is essential for identifying performance bottlenecks in distributed applications that execute on locally networked computer systems. We propose an integrated tool that is comprised of an application monitor, an operating system monitor, and a network and host monitor. Each of these monitoring techniques is well established on its own. However, we show that it is the integration of these techniques that is often crucial to finding the real causes of performance problems. We understand integration as the seamless merging, synchronisation, and presentation of performance data from all monitoring sources. The proposed tool allows both automatic and interactively controlled measurements, whereby automatic measurement mode is intended as a first monitoring step in order to discover possible performance bottlenecks with minimal effort.

Keywords: performance analysis, distributed programs, integrated monitoring, software monitoring, application monitoring, network monitoring, operating system monitoring

1 Introduction

The performance of distributed software executing on loosely coupled systems such as workstation clusters is affected by a variety of factors. The efficiency of the application code is only one of them. Sub-optimal performance may just as well result from shortcomings in the underlying hardware or inadequate operating system behavior. Hence, a performance monitoring tool for distributed programs should be able to gather measurement data from each of these subsystems. In practice, a performance problem will often be the result of a combination of performance degrading factors. Therefore, integrated presentation of the data from the various sources is desirable because it helps reveal data interdependencies.

Usually, when starting out with the performance analysis of an application, little or nothing is known about the performance characteristics of the application. Hence, a performance monitoring tool should provide an automatic monitoring mode to give a first survey of possible deficiencies (see figure 1). However, automatic monitoring is a rather coarse-grained technique that tends to collect huge amounts of measurement data, most of which is often not relevant to the analysis. Thus, while being a valuable means for indicating starting points for closer examination, it most often needs to be followed by a more fine-grained analysis which can only be carried out in an interactive, user-controlled fashion.

A distributed computation is generally characterized by a set of co-operating processes executing on a set of computers connected by a network. The co-operation aspect implies that events occurring in one process may trigger events in other processes, thus establishing causal relationships between events from different processes. Clearly, a monitor for distributed software must be able to capture these causal relationships. A vital prerequisite for doing so is the ability to determine the temporal sequence of events in all involved processes [1]. Unfortunately, the absence of a globally synchronous clock in loosely coupled systems makes this a hard task to accomplish. This creates a further requirement for a monitor for distributed programs. Namely that it has the ability to correctly synchronize the trace data gathered on different machines.

*The work reported in this paper was funded in part by the *Stiftung Rheinland-Pfalz für Innovation*

Monitoring Mode	Monitoring Sources		
	Application Software	Operating System	Network and Hosts
Automatic	All functions	All processes, system calls, interrupts, ...	All host and network resources
Interactive	Selected functions, constructs, statements	Selected processes	Selected host and network resources

Figure 1: Monitoring Modes

Performance bottlenecks in distributed programs, especially those stemming from a combination of performance degrading factors, are usually not trivial to detect. Most often it requires several iterations of analysis and measurement refinement to track down a bottleneck. It is therefore a desirable feature for a performance monitor to provide a means to correlate the displayed measurement data with the appropriate locations in the application source code where the trace data originated. The user can thus be guided to the hotspots where to refine the measurement specification.

Based on these insights, five key requirements can be derived, that should be met by a performance monitoring tool for distributed programs:

1. capturing of performance data on the application level, the network and host level, and the operating system level,
2. facilities for automatic and interactively controlled measurements,
3. synchronization of the performance data gathered on different machines.
4. integrated visualization of the performance data, and
5. provision of a means to correlate performance data with application source code.

The monitoring systems developed so far have been primarily designed for the performance analysis of high performance parallel programs running on multiprocessors (see section 5). They implicitly assume that the execution environment exclusively serves the monitored application and thus disregard contention for resources with other processes. As a consequence, these monitoring systems focus on gathering application level data, but fail to provide facilites for recording performance data of the operating system and the network and host resources. Performance bottlenecks of distributed programs that are due to a combination of factors on different system levels will remain undiscovered by these tools, because they lack support for an integrated multi-source analysis.

In contrast, the *CoSMoS*[1] performance monitoring tool presented in this paper pursues an approach of integrated and coherent evaluation of performance data from all relevant sources, as depicted in figure 2. Both automatic and interactive monitoring are supported. By visualizing the interdependencies between the data from the different monitoring sources, the system effectively aids in revealing the causes of time critical or faulty behavior of programs. These features render the system particularly well suited for the tuning of network applications such as client/server software [2].

1.1 Outline of this Paper

Section 2 describes the design of the *CoSMoS* system, which basically consists of a set of monitoring components for collecting trace data, a post-mortem analysis program for evaluating and synchronizing the recorded data, and a set of visualization toolkits for displaying the evaluated data. Section 3 shows in detail how the synchronization of the trace data is accomplished by the analysis program. Section 4 demonstrates the abilities of the monitoring system by presenting three examples illustrating the integrated performance tuning process using *CoSMoS*. Section 5 discusses related work in the area of distributed program monitoring, before section 6 draws conclusions and indicates directions for future work.

[1] Coblenz Software Monitoring System

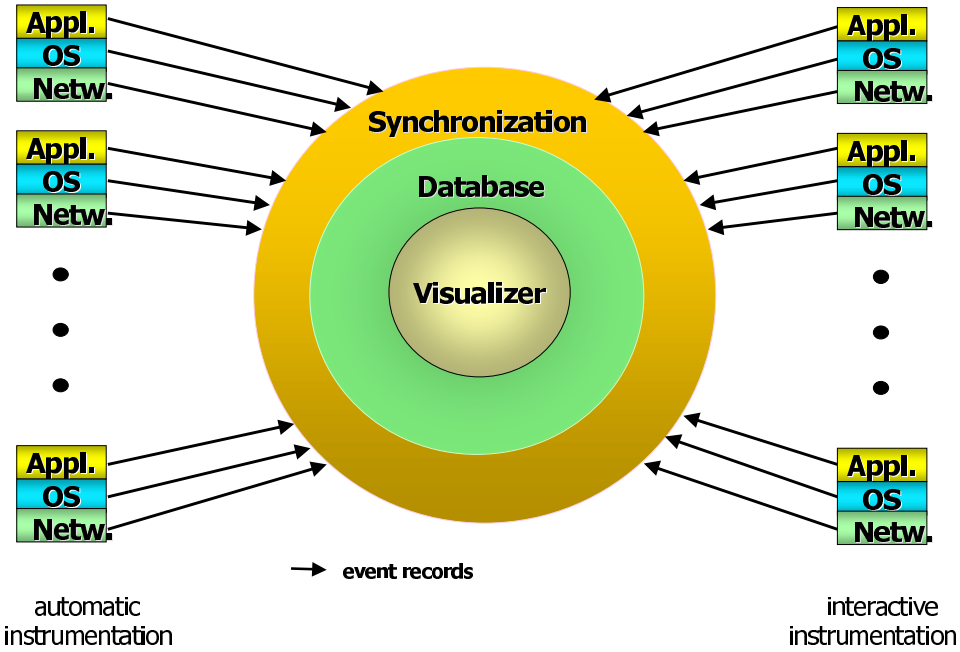

Figure 2: Capturing and Visualization of Performance Data

2 The *CoSMoS* Architecture

The performance of distributed programs executing on workstation cluster systems is affected by a multitude of factors. The efficiency of the application code, scheduling policies of the underlying operating systems, and host and network resource availabilities all contribute to the overall performance. Consequently, the *CoSMoS* monitoring system is comprised of an application monitor, a network and host monitor, and an operating system monitor, as shown in figure 3. Each of the monitors can be operated in automatic or in interactive mode. The trace data collected by the monitoring components is written to disk files. A post-mortem utility program merges, synchronizes, and analyzes the trace data files and stores the evaluated data in an SQL database, thus achieving the integration of the data from the various monitoring sources. The visualization toolkits finally present the data from the SQL database in integrated displays.

A key feature of the *CoSMoS* system is its ability to correlate performance data with application source code. By providing a click-back functionality, the visualization toolkits allow the user to jump by mouse-click from a performance data display to the appropriate point in the application source code. The user can then refine the measurement specification and start over with a new measurement session. The *CoSMoS* system thus supports a cyclic tuning process with multiple iterations of analysis and refinement (see figure 3). This technique enables the user to track down the causes of poor application performance in a well-directed, organized manner.

2.1 The Application Monitor

The application monitoring component of the *CoSMoS* system allows the capturing of application level runtime events by inserting measurement sensors into the source code. This technique is called instrumentation [3]. A measurement sensor is an additional piece of code which serves as an event recording operation. During runtime of the application, the event records are sent to a central log server on the network which stores them in log files for post-mortem analysis and visualization (see section 2.4). The architecture of the *CoSMoS* application monitor is shown in figure 4.

The *CoSMoS* application monitor consists of four major components:

1. An *instrumentation GUI* (graphical user interface) for interactive specification of measurements. A source code browser allows the user to select code portions for measurement. The system then annotates the selected source code with measurement instructions.

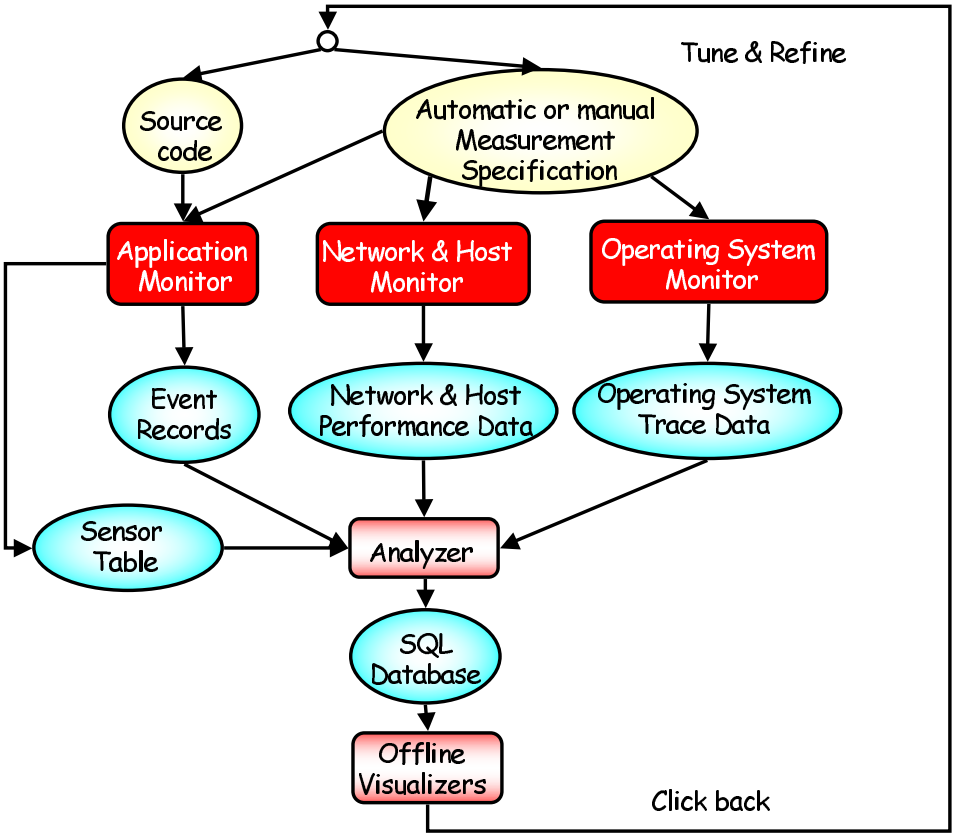

Figure 3: Tuning Cycle with *CoSMoS*

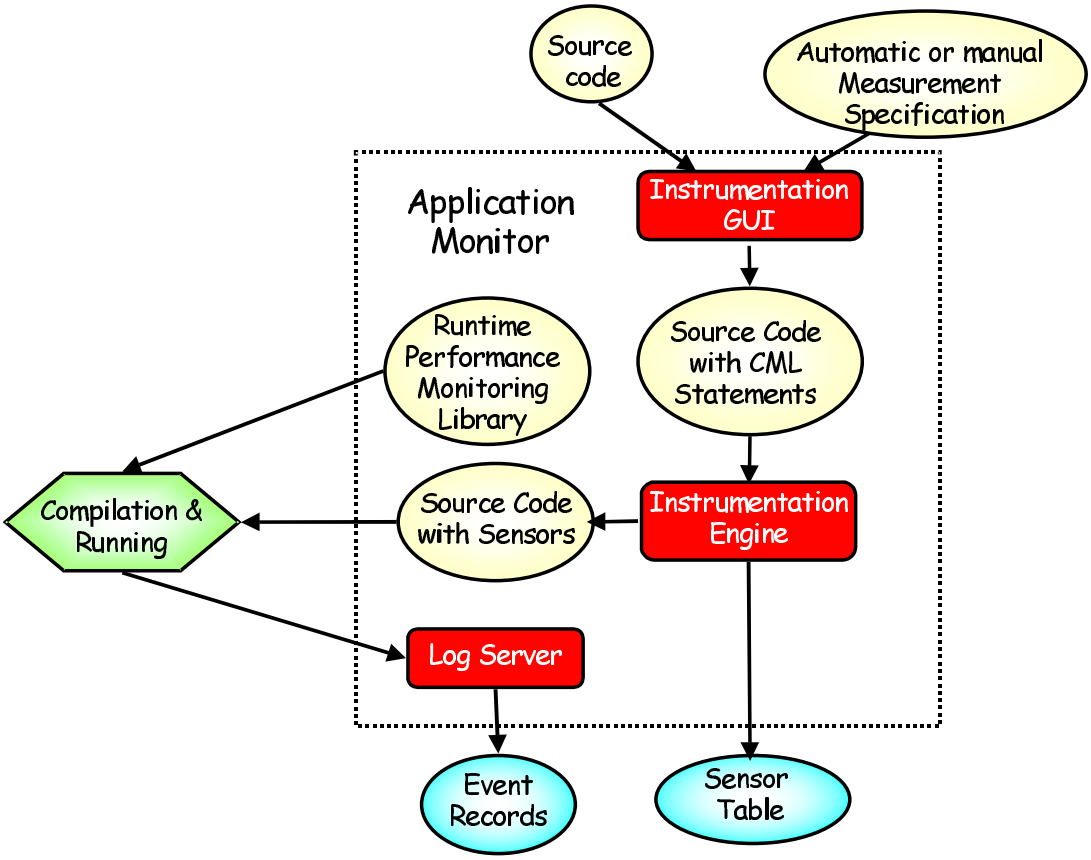

Figure 4: Architecture of the Application Monitor

2. An *instrumentation engine* consisting of a source code parser and a code generator. The instrumentation engine parses the annotated source code and generates the instrumented source code, which consists of the original source code enhanced by calls to measurement sensors. Alternatively, the instrumentation engines allow to automatically instrument the source code. Depending on the configuration, all function calls and/or the bodies of all function definitions can be instrumented automatically. The *CoSMoS* system provides instrumentation engines for the currently most popular programming languages C, C++, and Java.

3. A *runtime performance monitoring library* which is linked to the instrumented application. The library contains code for the measurement sensors, for establishing a connection to a measurement log server, and for sending the recorded data to the log server. Each buffer of measurement data that is transferred to the log server is tagged with a timestamp indicating the time for each recorded event elapsed since the start of the process. Alternatively to sending the measurement data over the network, the library can be configured to store the data in a local disk file in order to preserve network bandwidth.

4. A *measurement log server* which collects the recorded measurement data of all executing processes storing it in log files. Based on the timestamps transmitted along with the measurement data, the log server computes and records the differences between its own relative clock and those of the application processes. Based on these clock differences, the trace data can later be synchronized (see section 3).

Since monitoring by means of source code instrumentation inevitably perturbs the behavior of the program under observation [4], it is desirable to keep the perturbation induced by the monitoring system to a minimum. *CoSMoS* therefore encourages a stepwise refinement approach. The analysis typically starts out with coarse-grained automatic instrumentation, yielding a survey of possible deficiencies. Due to the huge amounts of measurement data generated by automatic instrumentation, a significant degree of perturbation is usually introduced by this monitoring mode. Therefore, the results obtained by automatic instrumentation are used as starting points for a detailed analysis based on manual instrumentation. In successive iterations, more and more fine-grained instrumentation is applied to selected parts of the application source code, until a performance bottleneck has been identified. As a consequence of this well-directed technique, only the hotspots of the application source code get instrumented, leading to least possible perturbation of the application behavior.

2.2 The Network and Host Monitor

Application performance may be significantly impaired by resource contention with other programs or by hardware limitations [5]. The *CoSMoS* network and host monitor is able to reveal such situations by capturing resource utilization data about both the network and the hosts executing the monitored application. Recordable data include CPU usage, memory usage, paging activity, in- and outgoing network packets, etc. In automatic monitoring mode, all available data are recorded. In interactive mode, the data items to be recorded are freely configurable.

The monitor uses an agent-based approach, running one agent on each host involved. Agents can communicate with other agents. Using this ability, a tree-like topology with a unique top level agent can be configured. The top level agent takes the role of a management station, that can be used to control the other agents. Instead of having to configure each agent individually, the whole set of agents can thus be steered via the top level agent. All event records get timestamped, allowing synchonization with the event records from other sources of the integrated CoSMoS monitor.

2.3 The Operating System Monitor

Besides the efficiency of the application software and the capabilities of the underlying hardware, a third major subsystem significantly affects application performance, which is the operating system. In-depth analysis of the observed process behavior is often not possible without knowing precisely, when a process was scheduled for the CPU and when it was displaced from it. In order to obtain this information, *CoSMoS* provides an interface to the FKT toolkit (Fast Kernel Tracing [6]), which is a kernel tracing facility that allows the construction of a precise, timestamped trace of the dynamic activities of the operating system kernel. FKT is currently implemented on Linux only, thus the full functionality of *CoSMoS* is limited to this platform. However, the operating system monitor is layed

out as an optional component of *CoSMoS*, leaving the system fully operational without it, albeit with possibly reduced quality of analysis.

When the monitored application starts, the initialization code of the runtime performance monitoring library of the first application process on each host triggers the recording of trace data in the kernel. Likewise, at termination of the last application process on each host, the kernel trace data recording session is stopped. The resulting trace files of all hosts are then collected and centrally stored for later analysis and display. The visualization toolkits present this information in synchronized correlation with the application level and network and host level trace data, as shown in figure 5.

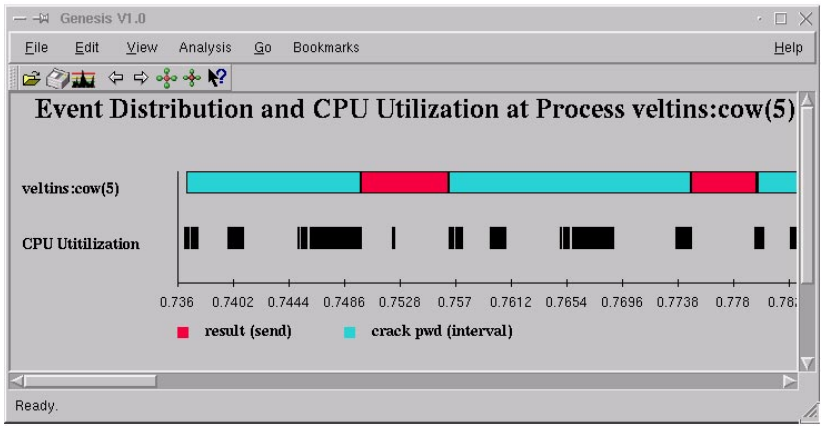

Figure 5: Integrated display of CPU scheduling information

2.4 The Analysis and Visualization Subsystem

The raw measurement data recorded at runtime of a monitored application is in a format not yet suitable for visualization. First, it has to be pre-processed by an analysis program. The analysis program merges the trace data files written by the log server and applies a synchronization algorithm to the data (see section 3). The synchronized event records are then stored in an SQL database (see figure 2). Any operating system trace data and network and host monitoring data that is available for the period of execution is merged into the database as well.

The *CoSMoS* visualization subsystem offers complex views of the measurement data. It consists of an animation toolkit for display of dynamic aspects and a static visualization toolkit for the presentation of statistical data. Both toolkits operate on the evaluated trace data stored in the SQL database.

The animated visualization toolkit allows replay of the application execution dynamics. It presents the performance data from the application level, the operating system level, and the network and host level in integrated displays. The main diagram form employed by the animation toolkit is the Gantt chart, which allows the visualization of state changes along a timeline. Gantt charts are most useful for displaying the succession of execution states of application processes. Furthermore, they are well suited for visualization of hardware resource utilization. During animation, a time cursor runs across the diagrams. Using a control panel, the animation speed can be increased or decreased at the user's will. Moreover, the animation can be stopped anytime, allowing the user to manually move the time cursor back and forth. Zoom views are provided for closer examination of execution states.

The static visualizer is mainly used for presentation of statistical diagrams. The user can select from a multitude of diagram types. They include bar charts, pie charts, Gantt charts, Kiviat diagrams, and others. As shown in figure 5, the static visualizer also provides a Gantt chart for visualizing application process execution states along with the appropriate CPU scheduling information. This diagram form enables in-depth analysis of the impact of operating system behavior on application performance.

3 Synchronization of Trace Data

A basic requirement for a monitor of distributed programs is its ability to capture the causal relationships between events of different processes. A prerequisite for identifying these relationships is to have knowledge about the relative ordering of all events on a global time scale. However, there is usually no globally synchronous clock available in loosely coupled systems. Thus the monitor has to provide a means to determine the relative event ordering on the basis of the local timestamps taken on the individual machines.

The synchronization method applied by the *CoSMoS* trace data analysis program operates solely on relative time specifications. Absolute timestamps as obtained from the local processor clocks are only used by the runtime performance monitoring library for computation of the relative times. Since only differences between absolute timestamps are recorded, the global time maintained by the individual processor clocks is immaterial. *CoSMoS* thus does not need to rely on the implementation of an external clock synchronization mechanism on the machines involved in a measurement session.

When an instrumented application process starts, the initialization code of the measurement library records a timestamp of the local system clock. Each time a sensor is visited during execution of the process, another timestamp is taken. The difference between the timestamp of the sensor and the timestamp taken at process start yields a relative time specification indicating the time elapsed since process start. This relative time is recorded by the runtime performance monitoring library for each sensor invocation.

Just before the runtime performance monitoring library of a process p transfers a buffer of measurement data to the central log server, it computes the relative time t_p since process start and attaches that time specification to the data buffer. Upon reception of a buffer, the log server s computes the relative time t_s since its own start. It then records the difference $t_s - t_p$ which consists of the time offset $O_{s,p}$ by which p started later than s plus a message delay d_b spent for network processing and transmission of the data buffer b, i.e. $t_s - t_p = O_{s,p} + d_b$ (see figure 6). Obviously, the log server s has to be started before a measurement session begins, so $t_s - t_p > 0$ holds.

Let an execution of a distributed program comprise n processes $p_1, p_2, \ldots, p_n$, indexed by their real time start-up order as depicted in figure 6. Knowing $O_{s,p_1} + d_{b_1}$ and $O_{s,p_2} + d_{b_2}$ from the transmissions of two buffers b_1 and b_2 by processes p_1 and p_2 to the log server s, and assuming message processing and transmission times on a local network segment being roughly equal ($d_{b_1} \approx d_{b_2}$), the difference $(O_{s,p_2} + d_{b_2}) - (O_{s,p_1} + d_{b_1}) \approx O_{s,p_2} - O_{s,p_1}$ indicates the time offset O_{p_1,p_2} by which process p_2 started later than process p_1. Adding the offset O_{p_1,p_2} to the relative timestamps recorded by process p_2 adapts them to the time scale of p_1.

Using the method described, the *CoSMoS* analysis program transposes the time scales of all application processes p_j to that of p_1. Thus, synchronization of the trace data of all processes is achieved solely on the basis of relative time specifications which are by nature independent of a processor's concept of global real time. Absolute timestamps are only used by the runtime performance monitoring library to compute the relative times which are used thereafter.

Note that the algorithm assumes approximately constant transmission times over the network. While this assumption is valid on local network segments, it is not applicable to larger networks or even the internet. However, *CoSMoS* is intended for monitoring distributed programs executing on workstation clusters. The algorithm has proven to work well in these environments.

4 Application Examples

This section presents three examples that illustrate how *CoSMoS* can help to gain insight into application behavior. The first example deals with the analysis of cascaded client/server interactions. The second example demonstrates the usefulness of automatic source code instrumentation for providing starting points for further investigation. As the example shows, automatic instrumentation may sometimes already provide sufficient information to identify a problem, especially if combined with additional monitoring techniques like kernel tracing or network and host monitoring. The third example presents a case study describing the optimization of a distributed password cracking application with *CoSMoS* using integrated monitoring of various data sources.

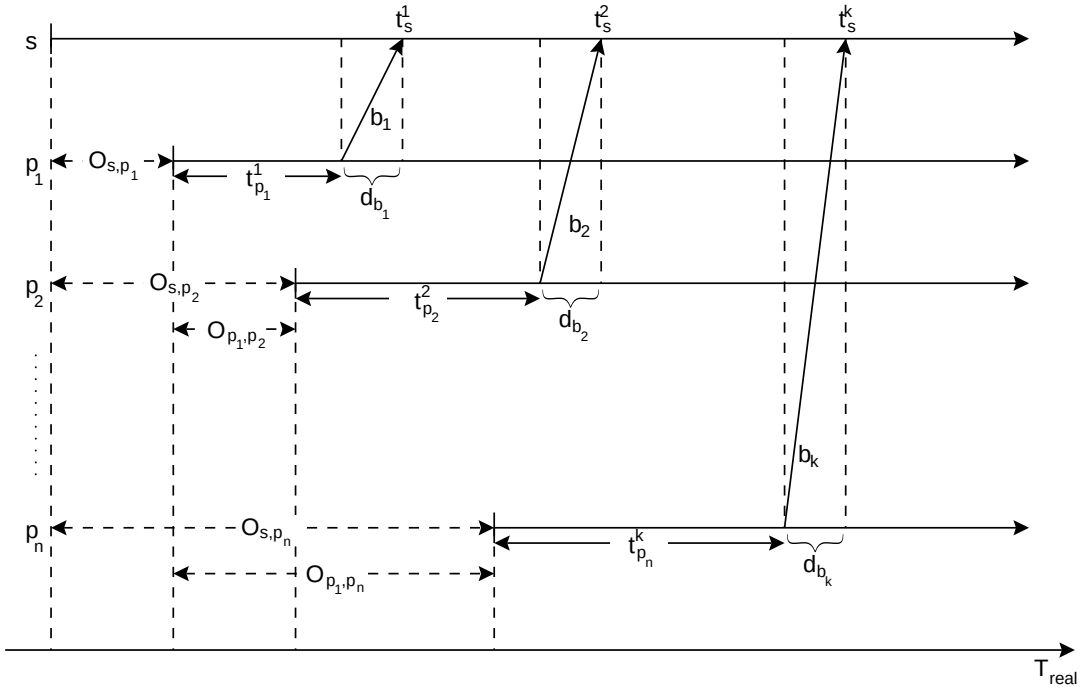

Figure 6: Trace Data Synchronization

4.1 Analysis of Client/Server Interactions

The example presented in this section demonstrates the analysis of cascaded client/server interactions with *CoSMoS* using automatic monitoring. In the client/server model, a client program requests a service from a server program which provides that service through a programming interface. In order to perform the requested function, the server may play the role of a client to other servers by issuing requests to them [7]. For example, a web client may request a price table from the web server of an online shop, which may in turn retrieve the current pricing information from a database server.

The Gantt chart depicted in figure 7 was produced by the animated visualizer of *CoSMoS*. The data presented were obtained by automatic monitoring. The chart points out communication interactions by connecting the corresponding send and receive events to each other and thus aids in the identification of situations where one process is significantly delayed by another.

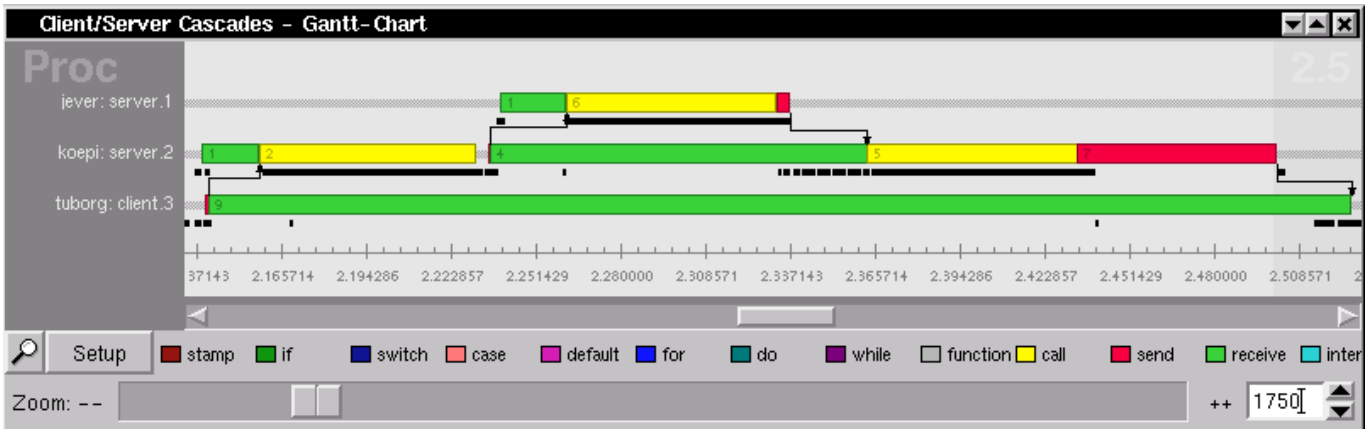

Figure 7: Gantt Chart Showing Communication Interactions

The chart contains one horizontal row for each process that was monitored. Each row is preceded by the name of the host that executed the process, the process name, and a consecutive number. The rows represent the succession of execution states of the corresponding processes along the timeline applied on the *x* axis. The chart in figure 7 displays

three process states: sending state, indicated by red (dark) bar segments, receiving state, indicated by green (medium grey) bar segments, and calculating state, indicated by yellow (light) bar segments. Correlated with this process state information obtained from the application level monitoring data, the chart also shows scheduling information gained from the operating system monitor. Each of the black line segments shown below the process bars indicates a scheduling interval during which the corresponding process was assigned the CPU. Thus, the chart shown in figure 7 presents an integrated view of trace data gained from application monitoring and operating system monitoring. Note that trace data synchronization is a vital prerequisite for the construction of this chart, because the data displayed stem from different machines.

The bottom row in figure 7 represents the client process. The middle row represents the server process that the client connects to. This server process again plays the role of a client to the server process represented by the top bar. The chart clearly reveals how much time is spent in the nested client/server interaction between the middle and the top server. Also, it precisely breaks down the server response times into the portions spent for communication(red, green) and calculation (yellow). If the client experiences waiting times beyond an acceptable limit, the chart uncovers which of the nested interactions consume most of the time. It further reveals whether the time is mostly spent for communication or for calculation. The host and network monitoring data may then be used additionally to clarify if resource shortages such as CPU, memory, or network overload are responsible for the perceived performance problems. If resource shortages are identified, the network and host monitoring data can further help in determining if the shortages are due to inadequate hardware or resource contention with other processes.

4.2 Automatic Monitoring of a Client/Server Application

This example shows the results of automatic integrated monitoring of a sample client/server application. In figure 8 the client sends a stream of small TCP messages which receive no replies. For example, imagine a window system where mouse motions trigger a stream of mouse events that is sent to an application with a graphical user interface. Since users of graphical applications expect quick responsiveness, significant transmission delays of mouse events are not acceptable.

In the test setup, the client and the server process executed on different machines, both running Linux with kernel release 2.4.7. The Gantt chart in figure 8 resulted from automatic instrumentation of the client and the server software. As already known from figure 7, the horizontal bars show the execution states of the monitored processes. However, instead of visualizing scheduling information below the process bars as shown in figure 7, occurrences of hardware interrupts of the network interface cards have been chosen for display.

Figure 8: Communication Events and NIC Interrupts with standard TCP

In figure 8, function calls are depicted by yellow (light) bar segments. The client and the server process bars consist of a multitude of extremely short yellow segments, reflecting the fast succession of calls to the send and receive function, respectively. Most of the function calls took so short that, in fact, the darker border color of the yellow segments gets dominant. The many individual function calls that make up the darker areas of the process bars get distinguishable when the zoom factor is increased. Figure 9 shows a view of the rectangular region marked in figure 8, zoomed by a factor of 20.

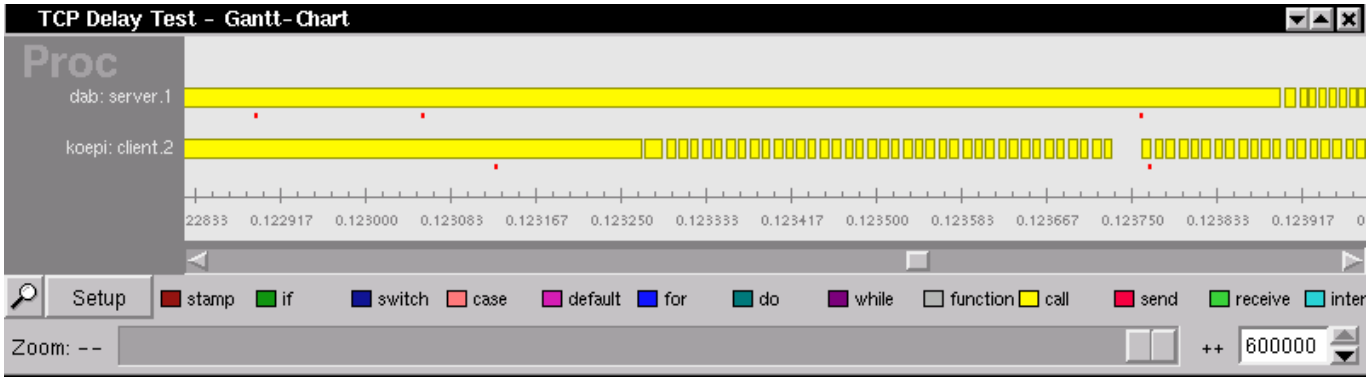

Figure 9: Zoomed View of Figure 8

Besides the many very short send and receive events that are predominant, figure 8 shows extended segments of light yellow hue on the server side, obviously indicating phases in which no message was received. Given the fact of a constantly sending client, as proven by figure 9, one would not expect such long periods of inactivity on the server. One possible explanation could be CPU contention on the server machine, preventing the server process from processing messages because it currently does not have the CPU. However, the measurements were carried out in a controlled environment where no processes were active except for the measured client and server processes.

Closer examination of figure 8 reveals a hint to the real cause of the observed behavior. Below the process bars, the occurrences of hardware interrupts of the network interface cards (NIC) are displayed as small dots. As each transmission of a network packet triggers an interrupt on the interface card, the number of interrupts directly corresponds to the amount of network traffic generated. In figure 9, it is clearly visible that the number of NIC interrupts is insignificant as opposed to the number of messages transmitted. This observation suggests that some component of the network protocol stack of the client machine buffers data scheduled for transmission in order to reduce network traffic.

In fact, the Nagle algorithm commonly found in TCP implementations does exactly this. The algorithm, defined in RFC 896 [8], strives to increase the efficiency of a network by delaying partial packets briefly in the hope that more data will soon become available so that a fuller packet can be sent. Fortunately, the Nagle algorithm can be disabled on a per-connection basis by setting the socket option `TCP_NODELAY`. Figure 10 shows the effect of disabling the algorithm.

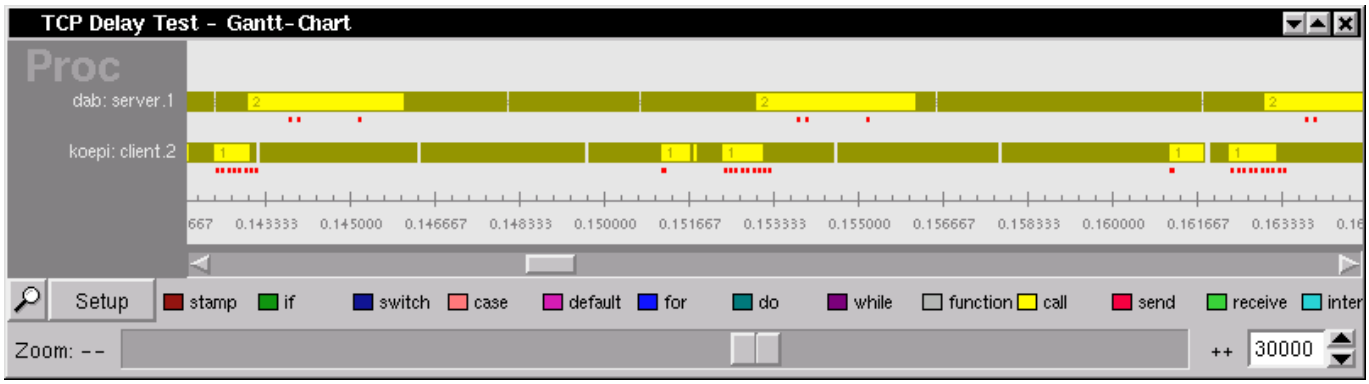

Figure 10: Interrupt Occurrences with Nagle Algorithm Disabled

Surprisingly, the situation has not much improved. Although the number of NIC interrupts has increased, there are still long periods with intense activity on the client, but no interrupts generated. Interestingly, interrupts tend to happen in bursts on the client side. During these burst periods, the system shows the desired behavior of rapidly sending out small packets, but after a while, it falls back to qeueing data. Further improvement can only be hoped for, if a way is found to prevent the TCP software from building up large data packets. Such a way actually exists by reducing the maximum segment size for outgoing TCP packets. On Linux, this can be done using the socket option `TCP_MAXSEG` [2]. Figure 11 shows the effect of reducing the maximum segment size to 20 bytes.

[2]Similarly, Solaris provides the socket option `SO_RCVBUF`, which can be used to control the window that TCP advertises to the peer.

Figure 11: Interrupt Occurrences with Nagle Algorithm Disabled and TCP Maximum Segment Size Set to 20 Bytes

It is clearly visible that the number of NIC interrupts has vastly increased. Messages get almost immediately transmitted over the network and processed by the server. In the context of the graphical application processing mouse events, this means that each movement of the mouse will instantly become visible and that mouse motions will appear much more smoothly.

This example demonstrates vividly the power of the integrated approach pursued by *CoSMoS*. The combination of automatic instrumentation and other monitoring techniques allow reasoning about application behavior that would not be possible with either monitoring technique alone. Note that no user interaction was required to obtain the performance data presented in the diagrams of this section.

4.3 Tuning of a Distributed Password Cracking Application

This example presents a case study describing the optimization of a distributed password cracking application with *CoSMoS*. Figure 12 shows an integrated view of the application level and network and host level monitoring data gathered during execution of the application. The application follows the master/worker scheme. The master process *cpw* (**c**rack **p**ass **w**ords) initially spawns a set of worker processes *cow* (**c**heck **o**ne **w**ord), which it then feeds with words from a dictionary. Each worker process reads one word at a time, checks the word against the accounts in a given UNIX password file, and returns any accounts that can be cracked with that word. When a worker has checked all accounts of the password file, it asks for the next word to try.

Figure 12 was produced by the animation toolkit of *CoSMoS*. In the chart depicting the communication behavior of the processes, the bottom bar represents the *cpw* process while the upper four bars represent the *cow* processes. The chart reveals a significant amount of synchronization overhead. While the *cow* process indicated by the topmost process bar was in computing state almost all of the time, each of the other three *cow* processes exhibit long phases of inactivity waiting for a new job from the master process. This observation is confirmed by the CPU load data of the hosts executing the application processes, shown in the bottom chart. The CPU load data was gathered by the network and host monitor during the corresponding period of time.

Obviously the *cow* processes on the faster hosts were often blocked because of the *cpw* master process waiting for a result from a slower machine. Based on this observation, an asynchronous communication scheme was implemented, having the master process listen for results from whatever process is ready to send instead of iterating through the list of worker processes and communicating with them in a fixed order. Figure 13 shows the behavior of the modified application.

The Gantt chart in figure 13 proves that the modified communication scheme lead to a drastic reduction of synchronization overhead. The hosts executing the *cow* processes exhibit a consistently high CPU load, as can be seen from the host performance data diagram below the Gantt chart. Only the host running the master process (bottommost bars of Gantt chart and CPU load diagram) spends most of its time idling. This is the expected behavior, since *cpw*'s only job is to distribute words from the dictionary and listen for results.

The implementation of an asynchronous communication scheme lead to a remarkable 50% reduction of execution time. However, the statistical visualization toolkit of *CoSMoS* was able to reveal yet another performance bottleneck.

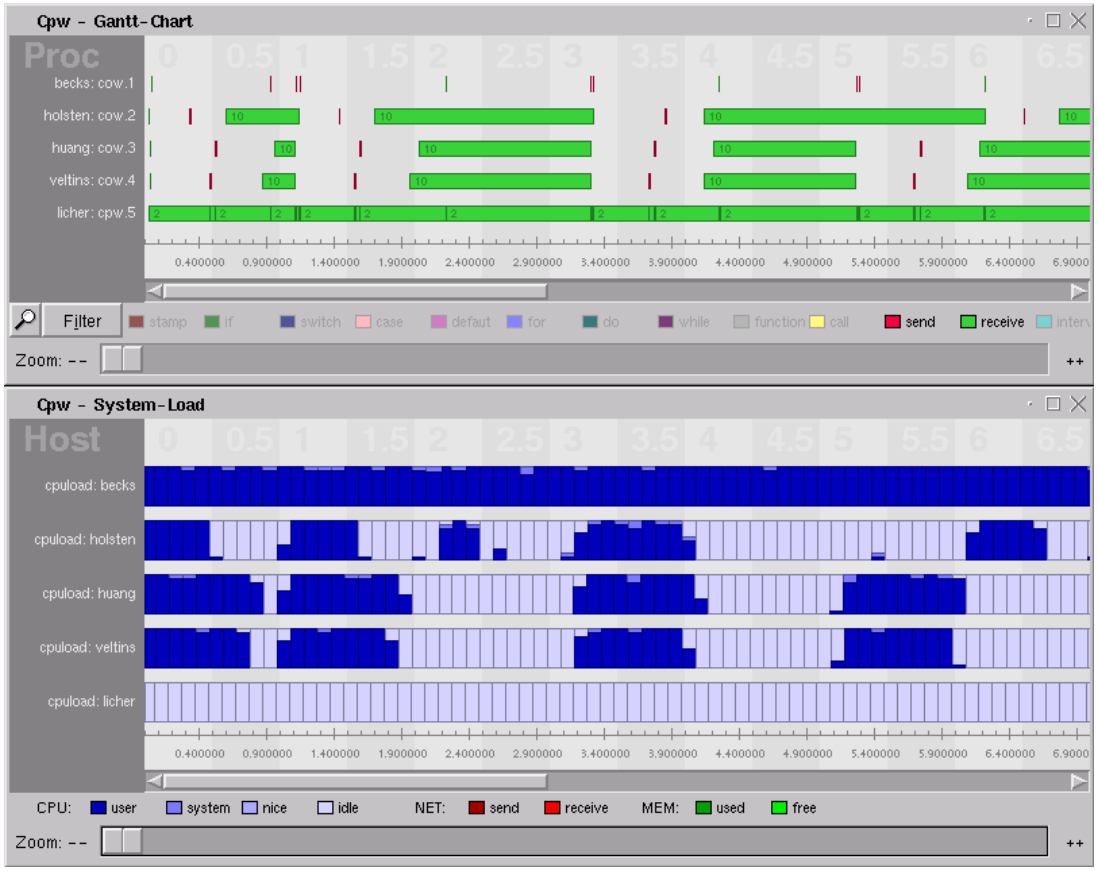

Figure 12: Gantt Charts Showing Process Behavior (Above) and CPU Load Data (Below) of Non-Optimized Password Cracker

Figure 14 shows performance statistics for function *try_word*. The tree view on the left hand side shows the nesting structure of the measurement points that were inserted into the processes *cow* and *cpw*. The bar chart on the right hand side breaks down the total execution time measured for function *try_word* into the execution times of the nested encryption operations.

Function *try_word* is the core function of *cow* doing the actual cracking work. It first checks if the account name or any of the words found in the gecos field[3] of the account information have been used as the password. If these checks fail, the dictionary word supplied by the *cpw* master process is tried. Figure 14 clearly shows that the encryption of the account name is far more time consuming than the following encryptions of the gecos field and the dictionary word.

A key feature of the *CoSMoS* visualization toolkits is their ability to correlate performance data with application source code. Clicking on an item in a performance data display causes the corresponding source code portion to appear highlighted in a source code window. Figure 15 illustrates the result of clicking on the left bar of figure 14 labeled “encrypt account name”. The system, however, is not limited to simply displaying source code. It also allows the user to directly switch back to the instrumentation environment for refining the instrumentation of the respective source code portion, thus closing the tuning cycle.

As can be seen from the source code in figure 15, the encryption of the account name takes place immediately after a new salt has been set[4]. The gecos field and the dictionary word are then encrypted using the same salt. The measurements suggest that the *crypt* function performs some sort of initialization each time a new salt has been set. This observation motivated a change of the algorithm with the aim of reducing the total number of salt reset operations. Instead of reading words from a dictionary and checking them against a password file, the algorithm was modified to read entries from the password file and check them against the dictionary. Since the salt is derived from the encrypted password, the *cow* process has to only set a new salt once a new password record is received from the master. The whole dictionary file can then be encrypted using the same salt.

[3]The gecos field is usually used to store the real name of the user.

[4]The salt is a two-character string that is used to perturb the DES algorithm.

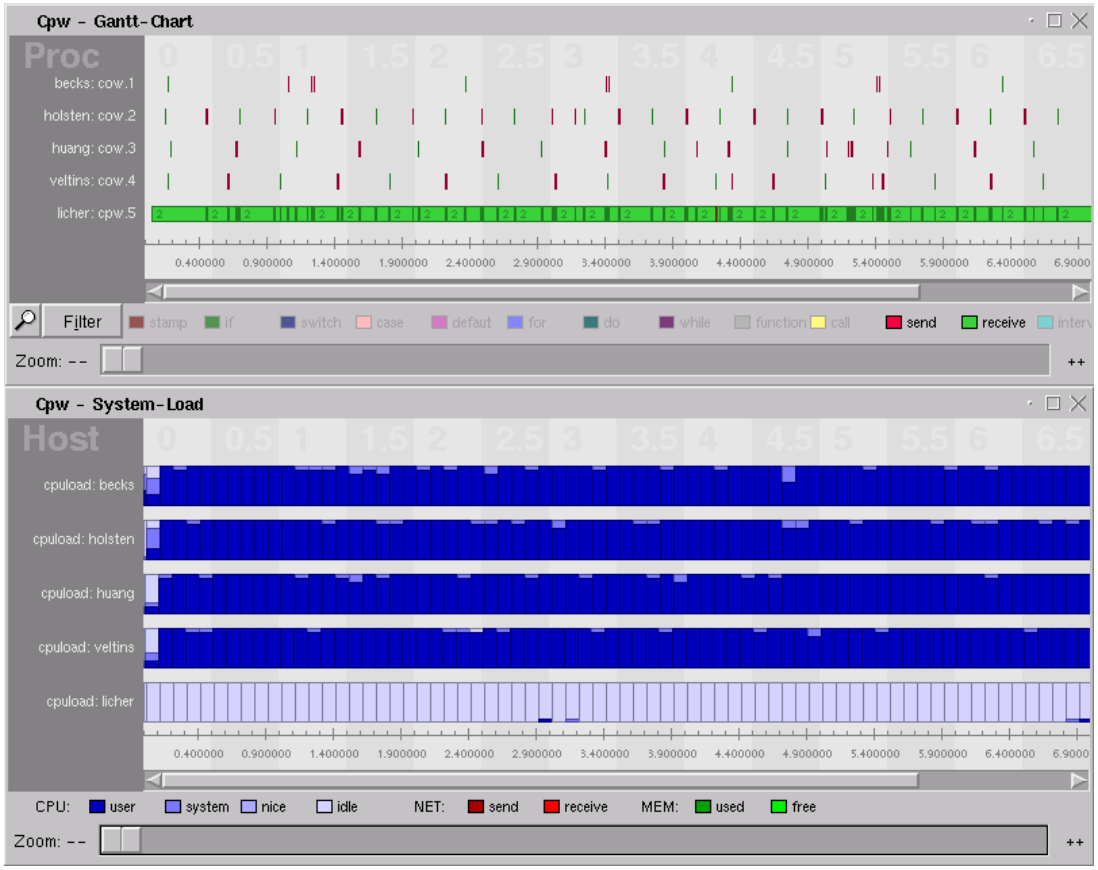

Figure 13: Gantt Charts Showing Process Behavior (Above) and CPU Load Data (Below) of Password Cracker with Optimized Communication Scheme

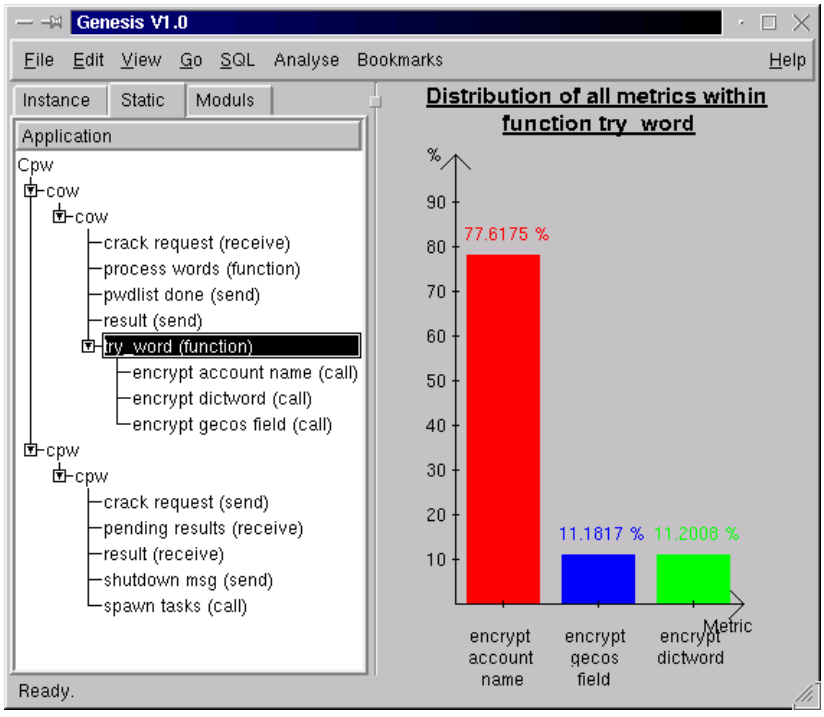

Figure 14: Performance Data for Function *try_word*

The reduction of salt changes resulted in an overwhelming performance gain of another 80%. Thus, using the *CoSMoS* system, the overall execution time of the application could be cut down to a tenth of the original execution time.

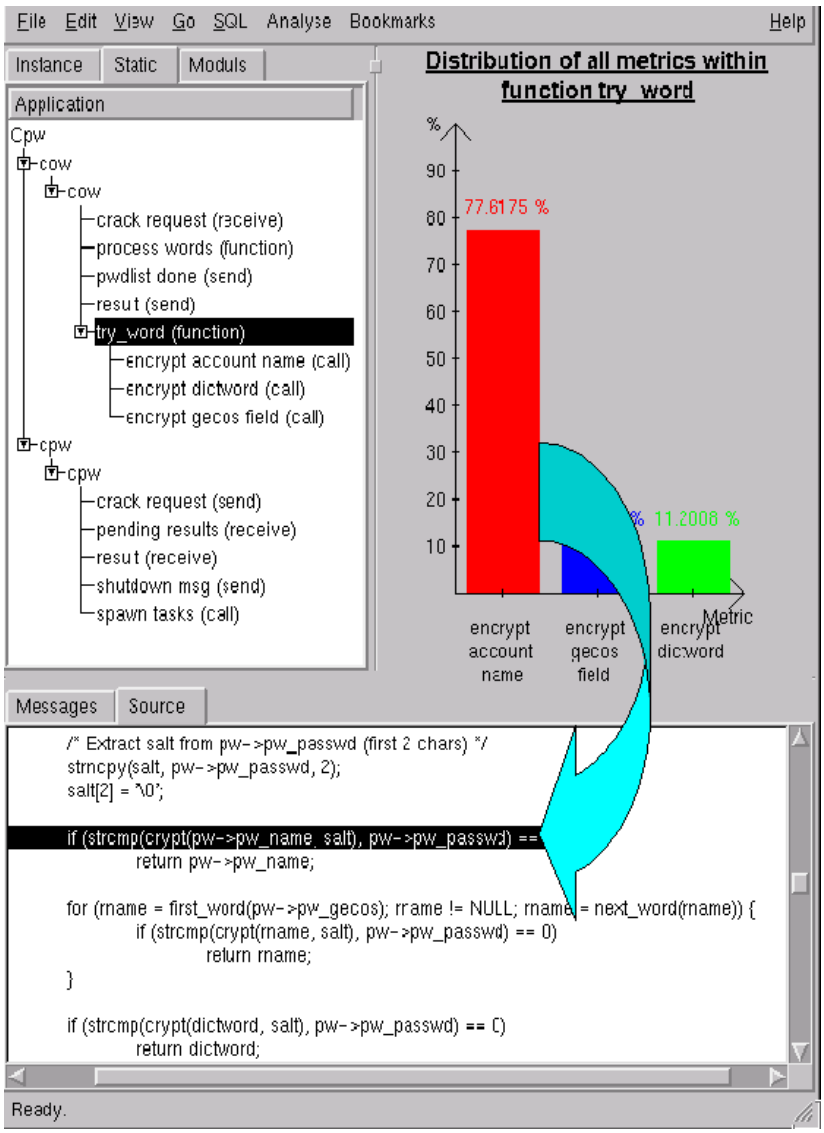

Figure 15: Performance Data Correlated with Source Code of Function *try_word*

5 Related Work

Quite a number of performance monitoring tools for parallel and distributed programs have been developed. Among the most prominent are AIMS [9], Paradyn [10] and Pablo [11], especially its derivative SvPablo [12]. Similar to *CoSMoS*, each of them features source code instrumentation, analysis of the resulting trace data, and visualization of the results. Furthermore, both AIMS and SvPablo provide mechanisms to correlate measurement data with application source code. Unlike all other tools, Paradyn dynamically instruments the application and automatically controls the instrumentation in search of performance problems. Paradyn's Performance Consultant module directs the placement of instrumentation, using a knowledge base of performance bottlenecks and program structure so that it can associate bottlenecks with specific causes and with specific parts of a program.

However, these tools have been primarily designed for the monitoring of parallel programs running on multiprocessors. Hence, they do not address the specific needs of distributed program monitoring, as mentioned earlier. Most of the tools support the programming languages HPF (High Performance Fortran) and C. Very few support C++, and there seems to be no tool currently available supporting Java. As a further consequence, these tools focus on source code analysis and neglect the monitoring of the execution environment. The latter, however, is of particular importance in the area of distributed program monitoring, where resource contention with other processes and hardware limitations like network bandwidth significantly affect application performance.

6 Conclusion

In this paper we have argued, that in order to obtain revealing information about performance bottlenecks, an integrated monitoring approach should be employed. The *CoSMoS* monitor was designed to collect trace data on the application level, the operating system level, and the network and host level, because any of these subsystems can significantly contribute to the overall application performance. The monitor supports both automatic and interactive monitoring. It was particularly shown that the integrated approach adds powerful potential to the abilities of automatic monitoring.

We have further stated that the ability to determine the relative ordering of events across different machines is crucial to capturing their causal relationships. It was shown that the analysis program of the *CoSMoS* monitor is able to map the trace data to a common time base without requiring the local processor clocks to be synchronized. The synchronization algorithm applied by the analysis program was explained.

Finally, we have highlighted the system's ability to correlate performance data with application source code. The visualization toolkits allow the user to jump by mouse click from a performance data display to the source code location where the displayed data originated. This click-back functionality encourages a cyclic tuning process with multiple iterations of analysis and refinement.

Future work will be directed towards development of an online monitoring and visualization component to enable measurement of applications that execute for hours or days. Another aim is to further reduce the measurement perturbation that is inevitably introduced when employing a software monitoring approach. A promising way to achieve this goal is to make use of the performance monitoring counters and cycle counter registers found in most modern microprocessors. Finally, work is underway to extend the trace data synchronization algorithm for large scale local networks.

References

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Communications of the ACM, 21(7): 558-565, July 1978.

[2] Ch. Steigner, J. Wilke and I. Wulff, "Integrated Performance Monitoring of Client/Server Software," in *Proceedings of the 1st IEEE European Conference on Universal Multiservice Networks (ECUMN'2000)*, Colmar, France, October 2000.

[3] D.A. Reed, "Performance Instrumentation Techniques for Parallel Systems," L. Donatiello and R. Nelson (eds), *Models and Techniques for Performance Evaluation of Computer and Communications Systems*, Springer-Verlag Lecture Notes in Computer Science, 1993, pp. 463-490.

[4] A. Malony and D. Reed, "Performance Measurement Intrusion and Perturbation Analysis," *IEEE Transactions on Parallel and Distributed Systems*, 3(4), July 1992.

[5] J.K. Hollingsworth, R.B. Irvin and B.P. Miller, "The Integration of Application and System Based Metrics in a Parallel Program Performance Tool," in *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pp. 189–200, Williamsburg, VA, USA, April 1991.

[6] R.D. Russell and M. Chavan, "Fast Kernel Tracing: A Performance Evaluation Tool for Linux," in *Proceedings of the 19th IASTED International Conference on Applied Informatics (AI 2001)*, Innsbruck, Austria, February 2001.

[7] M.W. Johnson, "The Application Response Measurement (ARM) API, Version 2," in *Proceedings of the 23rd International Conference on Technology Management and Performance Evaluation of Enterprise-Wide Information Systems, CMG97*, Orlando, Florida, 1997.

[8] J. Nagle, "Congestion Control in IP/TCP Internetworks," RFC 896, `http://www.ietf.org/rfc/rfc896.txt`, January 1984.

[9] J.C. Yan, “Performance Tuning with AIMS – An Automated Instrumentation and Monitoring System for Multicomputers,” in *Proceedings of the 27th Hawaii International Conference on System Sciences*, Wailea, Hawaii, 1994.

[10] B.P. Miller et al., “The Paradyn Parallel Performance Measurement Tools,” *IEEE Computer 28*, vol. 11, pp. 37–46, November 1995.

[11] D.A. Reed et al., “Scalable Performance Analysis: The Pablo Performance Analysis Environment,” in *Proceedings of the Scalable Parallel Libraries Conference*, pp. 104–113, IEEE Computer Society, 1993.

[12] L. DeRose and D.A. Reed, “SvPablo: A Multi-Language Architecture-Independent Performance Analysis System,” in *Proceedings of the International Conference on Parallel Processing (ICPP’99)*, Fukushima, Japan, September 1999.