Diplomarbeit

A Generalized Approach to Runtime Monitoring for Real-Time Systems

Torvald Riegel

17. Juni 2005

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter:	Dipl.-Inf. Martin Pohlack

Contents

1 **Introduction** 1

2 **Related Work** 5

3 **Design** 7
- 3.1 Reducing the Probe Effect . . . 7
- 3.2 Guaranteed Processing of Sensor Output . . . 11
- 3.3 Additional Requirements . . . 12
- 3.4 Architecture . . . 13
- 3.5 Ease of Use . . . 17

4 **Implementation** 22
- 4.1 Communication between Sensors and Monitors . . . 22
  - 4.1.1 Simple Sensors that are not Invoked Concurrently . . . 24
  - 4.1.2 Concurrently Invocable Sensors . . . 26
  - 4.1.3 Sensors in the Operating-System Kernel . . . 32
- 4.2 Guaranteed Processing of Sensor Output . . . 34
  - 4.2.1 Processing the Output of a Single Sensor . . . 35
  - 4.2.2 Processing the Output of Several Sensors . . . 42
- 4.3 The Monitoring Framework . . . 44

5 **Evaluation** 47
- 5.1 An Example . . . 47
- 5.2 Performance Overhead of Sensor Invocations . . . 50
- 5.3 Performance Overhead of Evaluations . . . 53

6 **Future Work** 54

7 **Conclusion** 56

References 58

Acknowledgments

I thank Martin Pohlack for general advice, for topic-related discussions, and for reading and commenting on drafts of this thesis.

Michael Hohmuth read and commented on drafts of Section 4.1 and of the whole thesis.

Claude-J. Hamann read and commented on a draft of Section 4.2.

1 Introduction

Software systems are getting increasingly complex and thus harder to understand, especially if they interact with each other. To be able to work with them and enhance and control them in spite of the complexity, you must know their properties and be able to determine their state during runtime. Accordingly, runtime monitoring is used to achieve two aims:

- The *determination of previously unknown properties* of the system consists of gathering information about the system and deriving a model of the system or parameters of such a model. This often takes place during development or maintenance and is supposed to help understand the system. Figure 1 shows the basic data flow while monitoring. The evaluation corresponds to the derivation of the model.

- The *verification of already determined properties* of the system requires comparing assumptions about the system to the actual state of the system. This can be used to check constraints at runtime or to control the system based on its current state. So, the evaluation in Figure 1 corresponds to constraint checking.

Nevertheless, tools with which both aims could be achieved are not widely available. However, the situation has improved recently. Especially tracing tools (i. e., tools that interpret traces caused by the execution of a system to determine properties of this system) are getting more attention and are used to find performance bottlenecks or ease administration of production systems.

On the other hand, these tools solely focus on determining properties and are more like high-level debugging tools. They cannot be used to reliably check assumptions about the system at runtime. A guaranteed verification is either not possible or it results in a significant interference with the monitored system leading to false observations. Monitoring tools that can guarantee accurate observations and the evaluation of these observations are often limited to specific systems or kinds of measurements.

Experimentally found information about the monitored system must be verified, for example by checking corresponding constraints at runtime. Even if the verification only happens within the scope of tests, it must be possible to automate the tests, which at least requires soft guarantees regarding successful execution of the tests. Thus, the situation would improve if monitoring tools would support both phases, determination and verification of properties. Additionally, tools should provide similar interfaces for both phases, so that users can reuse, for example, evaluation functions developed during the first phase.

The basic problem that has to be solved by a monitoring tool is simple (see Figure 1): The state or properties of the monitored system have to be evaluated by a monitor.

Figure 1: Basic data flow

The evaluation consists of user-defined functions or automata and is independent of the monitored system.

The observation of properties of the system cannot always be based solely on the system's interaction with other systems because this treatment of the system as a black box leads to the same limitations as in the case of testing—for example, one can only react to changes in state after the system's interaction. However, this basic problem is only the abstract view on the basic data flow. For example, such a problem description does not cover which observations have to take place, how the monitoring tool should transfer the information representing these observations from the monitored system to the monitor, and how the monitor should evaluate this information.

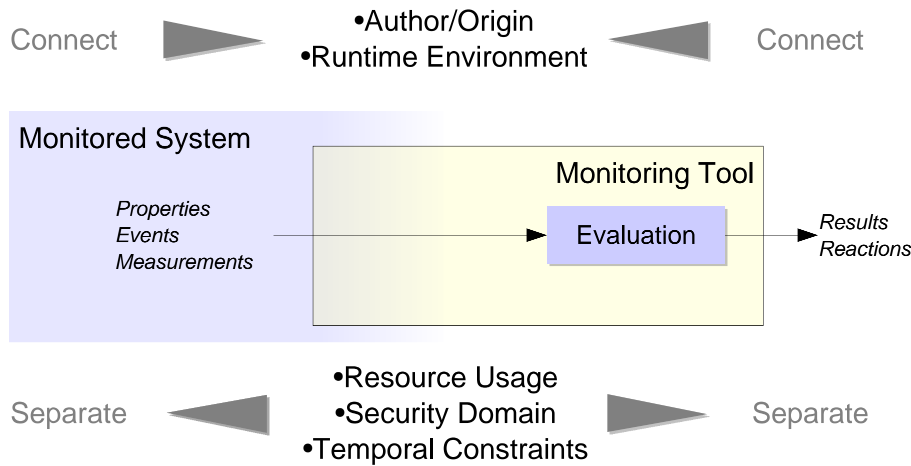

Figure 2: Problem space

Hence, the actual problem is more complex, as Figure 2 shows. Evaluation and monitored system have to be connected to be able to transfer data, nevertheless both are still to be treated as independent parts. As shown in Figure 2, a monitoring tool is the interface between them. The tool has to connect both parts on certain levels and separate them on others. The evaluation is embedded in the tool and receives its input from it. Thus the existence of such a monitoring tool enables users to solve their monitoring tasks in a simpler, solely data-flow–oriented environment. The tool takes care of everything else, for example transferring data, triggering of evaluations, and connecting and separating both parts.

The monitoring tool has to connect monitored system and evaluation on the following levels because both entities are originally separated there:

- Often both entities have a *different origin*; for example, they were created by different authors at different times. Therefore, a proper specification of the data flow is required as well as support for late binding. This must be possible without changing the system and without its active cooperation when establishing such a binding, which often has to happen at runtime.

- Both entities are often contained in *distinct environments* at runtime, for example different tasks, kernel and userland or just dissimilar software environments.

In contrast, the monitoring tool must keep the monitored system and the evaluation separated in the following areas, or it must actively separate them:

- *Resource usage* caused by monitoring must not alter the monitored system's state. Additionally, the system must not be made accountable for this resource usage.
- Evaluation and monitored system are usually located in different *security domains*. This separation must be held up.
- Both are subject to different *temporal constraints*, so the monitoring should not introduce new temporal relations between them; for example, monitoring must not lead to the system missing its deadlines. Furthermore, it should be possible to execute evaluations off-line or delayed.

Achieving these aims is a major part of the problem when designing a monitoring tool. Three very important requirements are not shown in Figure 2:

Small probe effect The overlap between monitoring tool and monitored system that can be seen in Figure 2 represents the area in which the tool interferes with the system. Because this distorts the observations evaluated by the tool, these *probe effects* must be minimized. This can be achieved by separating monitored system and monitoring tool. If they cannot be separated completely, the parts of the tool that overlap with the system must cause as little interference as possible.

Guaranteed evaluation Even if a small probe effect makes sure that observations are correct, the tool must be able to guarantee that every observation is evaluated and no input is missed to produce a correct result. Statistical guarantees are sufficient for a certain kind of evaluations and observed properties. Nevertheless, the monitoring tool must be able to guarantee the evaluation of all observations that have taken place during the tool's runtime because statistical guarantees are insufficient if, for example, monitoring is used to check whether a task met a certain deadline.

Ease of use Monitoring in the previously outlined way is not the sole solution for determining and verifying properties of a system. For example, constraint checks could be a part of the monitored system. To not nullify advantages like late binding or separated security domains, the tool must be easy to use. It must be convenient to experimentally determine properties, and it must be simple to ensure guaranteed evaluation.

The work presented here is closely linked to the need for a monitoring tool that can be used in the scope of the Dresden Real-Time Operating System (DROPS [8]). This operating system consists of a preemptible implementation of an L4 microkernel and a basic set of software services. It can isolate user processes spatially and temporally and can guarantee availability of resources previously reserved by processes. DROPS currently runs on x86 and ARM systems. To be applicable, the monitoring tool must

not be restricted to special hardware configurations and should require only a small set of software services.

Focusing on DROPS as environment for the monitoring tool does not necessarily reduce the applicability of the presented work because the scope of DROPS is rather wide. Real-time support is both a new requirement for the monitoring tool and the means needed to ensure guaranteed evaluation. At the same time, the domain of common non–real-time operating systems is included as well—for example, L^4Linux, an adapted Linux, and Linux userland applications can be executed on top of DROPS. Thus, the tool cannot be limited to smaller and therefore better known system configurations like embedded systems. Consequently, the results of the presented work should be applicable to other operating systems.

Similarly, DROPS shows that monitoring tools should support both easy determination and guaranteed verification of properties of the monitored system. The former is particularly important during the construction of large and complex non–real-time systems, whereas the latter is often required in the case of real-time systems. However, determining properties of a real-time system during its construction is equally important and the resulting evaluations should be reusable for the verification of these properties. Another example would be a video player application consisting of a real-time part showing the video and a complex non–real-time part containing the graphical user interface: To verify that the video is shown at least as long as the user has not pressed the stop–button, information from both the real-time and the non–real-time part must be combined and evaluated.

In this thesis, I will show how to construct a monitoring tool that supports both determination and verification of properties of a monitored system. The tool is called *GRTMon*, fulfills the preceding requirements, can be used in a non–real-time environment if guaranteed evaluations are not required, and is easy to use. It provides a basic monitoring infrastructure that can be used by other tools for more specialized tasks like automated testing or adaption based on the current resource usage of software components. Particular problems, such as how to instrument monitored systems at runtime, will not be covered in this thesis because they have been dealt with in related work and are usually not restricted to a specific environment.

In the following section, I will give an overview of the related work. Thereafter, I will explain the design of the monitoring tool in Section 3. Section 4 contains a detailed description of the chosen solutions for three of the subproblems. I will evaluate the tool in Section 5, mention topics for future work in Section 6, and conclude in Section 7.

2 Related Work

As mentioned previously, most of the related work deals either with the easy determination of a system's properties or with the verification of these properties.

The papers referenced next belong to the first category. They show how systems can be instrumented and how obtained observations can trigger evaluations. However, none of them can guarantee that all observations are evaluated.

DTrace [3] is a tracing facility for the Solaris operating system. It can dynamically instrument the operating-system kernel and user processes during runtime and provides a unified interface for several instrumentation techniques.

Arbitrary evaluations can be defined by the user; however, loops and backward branches are not allowed. To overcome this limitation, DTrace provides several predefined functions that can be called from evaluation algorithms, for example functions for aggregating data or the basic operations required for the supported data types. Evaluation algorithms are specified by the user in a high-level language, compiled by DTrace into an intermediate format based on a small instruction set, and interpreted when probes (i. e., the hooks that are inserted into the monitored system during instrumentation and trigger the execution of evaluations) are fired.

DTrace is designed for monolithic operating-system kernels: the tracing facility is included in the kernel and evaluations are executed in kernel context (however, the interpreter only executes safe operations). When a probe fires, interrupts are disabled on the respective CPU, all evaluations associated with the probe are executed in sequential order, and finally the interrupts are enabled and the monitored system can thus resume work. This enables the user to filter events at the earliest point in time but results in a significant probe effect, which makes DTrace inapplicable for real-time systems. In [3], Cantrill and colleagues consider only the "disabled probe effect": the overhead of instrumentation support when no probe in the monitored system is activated.

K42 [17] is an operating-system kernel with a built-in tracing mechanism. It uses static instrumentation, but the performance overhead caused by disabled probes is small. Measured data (i. e., descriptions of events) can be inserted into per-CPU buffers by concurrent processes without requiring these processes to acquire locks. However, it can happen that consumers evaluate inconsistent data. Although [17] states that inconsistent data can be detected in most cases, it cannot be guaranteed that at least a certain portion of the data is consistent. Therefore, K42's tracing mechanism cannot be used to verify properties of a monitored system. Statistical guarantees cannot be given either. GRTMon does not suffer from this problem.

The Linux Trace Toolkit (LTT) [18] is based on static instrumentation. It is limited to measurements taken in the kernel but can provide information about events caused by user processes. It uses a double-buffering scheme to transfer event descriptions to the consuming user processes. Because only completely filled buffers can be accessed by the consumers, a maximum latency cannot be guaranteed for evaluations. Thus, LTT cannot be used to verify properties, whereas GRTMon can guarantee a maximum latency.

DProbes [14] is a tracing tool that uses dynamic instrumentation to call user-defined handlers that evaluate events. Similar to DTrace, the underlying mechanism that

executes evaluations disables interrupts and thereby causes significant probe effects if evaluations are nontrivial.

In the category of real-time monitoring, most publications only consider the requirements of pure real-time systems. The determination of properties or the convenient instrumentation of systems are usually not covered, at least aspects like ease of use are not addressed in the publications. However, the DROPS context of GRTMon requires both easy determination and verification of properties.

In his thesis [15], Thane gives a good overview over real-time monitoring as the basis for debugging and testing distributed real-time systems. He mentions general requirements that must be realized by monitoring mechanisms, presents the entities that must be observed, but does not elaborate on the implementation of monitoring tools. The results rather apply to embedded real-time systems than to systems like DROPS.

In [4], Chodrow and Gouda present a monitoring mechanism that is based on a special class of logical constraints with which the state of an application can be checked. Monitored system and monitoring tool share buffers with a limited size. The monitored system writes state information into these buffers without waiting for the monitoring tool to consume data. Therefore, the monitoring tool might miss data written by the system. The required special class of logical constraints enables the mechanism to guarantee that every detected violation corresponds to a violation in the system (that is, there are no false alarms) and that a violation that persists in the system will be detected. However, no upper bound is given for the latency of violation detections.

In [13], Mok and Liu show how to efficiently monitor timing constraints by means of event histories and how constraint violations can be detected as early as possible. They give bounds for the number of outstanding constraint checks and the minimum required size of event histories, which depend on the maximum occurrence rate of events. They do not elaborate on how to schedule the task that performs the constraint checks and on whether scheduling constraints change the violation detection latency.

The following two publications do not belong to one of previous categories. Nevertheless, both show what monitoring results can be used for, and thus, which types of applications monitoring tools should support.

Magpie [2] is a toolchain that monitors a system, extracts requests from the observed events based on an event schema, and creates workload models by clustering these requests. Event schemata must be provided by the user and specify which events belong to the same request or start and stop a subrequest.

The Monitoring and Checking with Steering framework [9] (MaCS) uses a requirement specification to create a so-called monitoring script. According to this script, the monitored system is instrumented with operations performing measurements, and a so-called runtime checker verifies assumptions about the system. A steering script specifies actions that are executed when the system's state does not match the assumptions.

The framework is designed for monitored systems written in the Java programming language. Static byte-code instrumentation is used. The monitoring and steering scripts are Java-related as well. MaCS does not give real-time guarantees regarding the maximum latency of steering actions.

3 Design

In this section, I will discuss the primary requirements that drove the design of the monitoring tool GRTMon: A small probe effect and guaranteed evaluation of data acquired from the monitored system. Finally, I present the resulting architecture and explain how the effort needed to use the monitoring tool can be kept small.

3.1 Reducing the Probe Effect

Monitoring or tracing is used to gather information about a system. To get precise observations, the design of a monitoring tool must ensure that the monitored system is protected from effects caused by monitoring, so-called probe effects.

As you can see in Figure 2 on page 2, monitored system and monitoring tool overlap: Measurements have to be taken in the system to observe properties. These measurements require the execution of software because of the following reasons:

- The tool cannot observe only the output the monitored system was built to produce; systems should not be required to export all possibly relevant information on their own. Instead, the monitoring tool should support the systems or the user in doing so by providing the necessary means, for example communication mechanisms or instrumentation techniques.
- Simulating the system to be monitored is not an option because production systems have to be monitored as well. Virtual machines that could be used to transparently monitor the system are usually not being used on devices with limited resources, such as embedded systems.
- Hardware sensors like the MSR registers on x86 hardware are too coarse-grained because they usually return accumulated information. For example, they are able to differentiate between kernel and userland operations but they cannot tell which thread operations belong to. Furthermore, they have to be read by software executed on the same hardware.
- The measured data either have to be evaluated on the same hardware or they must be transfered to external systems. Both requires the execution of additional software and is important in practice. For example, the former applies to isolated production machines, the latter to small and possibly embedded devices that do not have enough resources for expensive data analysis.

The resulting probe effects can be reduced by separating monitoring tool and monitored system. As they cannot be separated completely, the strength of the remaining probe effect has to be reduced.

Separating Monitors and the System to be Monitored When you think of the monitored system as a state machine, a probe effect is a difference in the machine's sequence of state transitions caused by the concurrent execution of a monitoring tool. The machine's state consists of the state of several resources and relations to external entities, for example physical clocks.

The monitoring tool has to be executed concurrently because it must be possible to get measurements of intermediate states. Thus, monitoring results in shared use of resources.

Therefore, the degree of separation between a monitoring tool and a monitored system depends on the amount of resources being shared, how these resources are influenced by the tool and how they in turn influence the monitored system. For example, a computation using the CPU to full capacity will take longer with respect to physical time if intermediate states are to be evaluated by the monitoring tool. However, if the monitored system only cares about the start of the computation, concurrent execution does not result in a probe effect. Consequently, defining the border and content of the monitored system is extremely important because it divides effects related to monitoring into probe effects and the indeterminism inherent to the monitored system.

As the operating system can already separate user processes to a certain degree, the major part of the monitoring tool should be contained in a user process. From now on, such a combination will be called a *monitor*.

Integrating the monitor into the operating system's kernel would result in easier access to information but would make the kernel more complex. Additionally, monitors have less influence on the kernel's behavior if they are ordinary processes. This influence should be even lower in a microkernel architecture because only essential functionality is contained in the kernel. Other parts, like device drivers, are user processes with whom monitors interfere less than with the kernel, at least if they are not using these parts.

It will be a common case that there are several processes running besides the monitored system. They interfere with the system in a way similar to the monitor, which makes the monitor's interference less distinctive. On the other hand, if monitoring is supposed to explain indeterministic behavior of the monitored system, the presence of monitors will make a correct replay of the original behavior less likely.

Reducing the Strength of Probe Effects I have not yet discussed how to extract the information needed by the monitors about the monitored system. Ideally, atomic snapshots of relevant properties of the system must be taken and made accessible to the monitor. The taking of snapshots is triggered by a certain state of the system. Then, the monitor can either process the snapshot synchronously or store the snapshot and evaluate it asynchronously.

Synchronous processing requires stopping the system to be monitored, allowing the monitor to process the snapshot's data, and resuming execution of the system. This has several drawbacks:

- Atomically stopping the whole system to be monitored is not always possible. For example, stopping a system that processes external input might lead to lost data if buffers overrun. Obviously, physical clocks cannot be stopped either.

- If it is nontrivial to detect whether a state should trigger evaluation of a snapshot, the monitored system must initiate the evaluation. Trying to construct

monitors by connecting several simple monitors at different levels of abstraction will make monitoring more similar to simulation with every increase of the complexity of the condition that triggers evaluation.

- Executing monitors requires resources. The execution times of possibly complex monitors that are executed synchronously have a significant influence on the timing of the monitored system with respect to external clocks (for example, real-time applications are related to an external clock). Additionally, the execution is triggered by the monitored system but the amount of resource usage depends on the evaluation algorithms, which are provided by the user.

- Fine-grained control over which data should be accessible to monitors requires the monitored system to mark or copy the accessible parts of the data contained in the system, making synchronous processing similar to the asynchronous case.

- Switching to the monitoring process for every snapshot is expensive.

Asynchronous processing of snapshots requires the monitored system to take the snapshot and ensure that it is available to the monitor for a certain amount of time, for example by copying it to a shared buffer.

The monitored system itself detects when a snapshot has to be taken and triggers the execution of software *sensors*. When invoked by the system, these sensors extract information about the current state of the monitored system and copy the information (from now on called an *output element*) to buffers that can be read from by monitors. However, monitors cannot write to these buffers.

Sensors do not synchronize with monitors. Instead, they provide synchronization information that is used by monitors to determine which output elements have been written by a sensor, in which order these elements were written, and whether elements read by a monitor did change during the read operation. Consequently, sensors are independent from monitors. How the information contained in an output elements is structured and whether it gives a complete or incremental state description is up to the monitored system. Sensors can be located at several places:

- In the monitored system, to inform about the internal state of the system

- At the edges of the monitored system or in components used by the system, to provide information about the system's interactions for reasoning about the state of the system

- In the operating-system kernel, because some information is only visible to the kernel

Invoking sensors requires resources and thus induces a probe effect. However, this effect will be small compared to effects caused by synchronously executed monitors.[1] It is outweighed by the following advantages:

[1] Evaluating data is usually more expensive than just copying it. If the monitor processes just a small subset of each output element, the sensor is either designed for monitors demanding more detailed data, or it produces unnecessary data and should be downsized.

- The amount and type of resource usage caused by sensors is known to the monitored system because it triggers execution of the sensors and it knows what kind of sensor code will be executed.[2] The sensor's interference is similar to that of other code contained in the system and can be treated in the same manner. Copying or storing data are simple operations and require only few resources. This results in a higher degree of separation between monitors and monitored system.

- The monitored system is able to detailedly control the information released by sensors and the states that trigger the release.

- Monitors and monitored system are temporally decoupled. Monitors have no direct influence on the monitored system's timing.

- The overall resource usage per output element is smaller compared to synchronous communication because of less switching between processes.

If synchronous evaluation of output elements is required, the monitored system should explicitly synchronize with monitors. This way, the dependencies between the monitored system and monitors are incorporated into the system instead of being added at runtime.

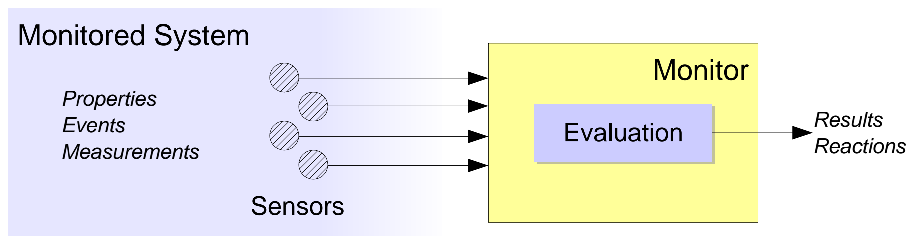

Figure 3: Separation of sensors and monitor

In GRTMon, monitors and monitored system communicate asynchronously because of the aforementioned advantages. Especially the known and small set of dependencies simplifies creation and usage of sensors.

Invocations of sensors producing incremental state descriptions are often inexpensive and show little jitter. Thane [15] states that sensors should be treated as an inherent part of the system to further reduce probe effects. Because the sensors' interference with the system is almost independent of monitors and a part of the system, in this case, sensor invocations will not induce a probe effect as defined previously.

The monitored system has control over the atomicity of sensor invocations with respect to the rest of the system. It can decide whether and how it has to be stopped and resumed to produce a consistent state description.

Figure 3 shows the results of the detachment of sensors. The large and rather blurred overlap between monitoring tool and monitored system has changed to a set of small sensors whose interference is known.

[2] GRTMon's functions for producing sensor output are constructed accordingly (see Section 4.1).

Additionally, the previous discussion shows why aspect-oriented programming (AOP) is useful to instrument a system (see, for example, [11]) but not sufficient for monitoring. Current AOP tools can only separate sensors and monitored system in a software engineering sense but they cannot, for example, separate resource usage. So, a combination of a suitable instrumentation technique, small sensors, and a monitoring tool such as GRTMon is a promising solution.

3.2 Guaranteed Processing of Sensor Output

Monitoring is used to react to the information obtained about a monitored system. Reducing probe effects helps in getting accurate information but it must also be possible to guarantee that the monitor evaluates all the information in a bounded amount of time.

Sensors do not need to produce new output if the state they report about has not changed; if they had to, this would represent a lower bound for the rate at which sensors produce output and would prevent sensors from being used in non–real-time environments. Consequently, a monitor cannot distinguish between the absence of changes in the monitored system and itself missing sensor output. Even if the monitor can detect whether it missed state information (e. g., when new output is processed later on), this detection is not sufficient because a complete picture of the state transitions is usually required for a correct reaction. For example, often it has to be checked whether a property that changes over time always stays under a certain threshold.

Thus, a monitor must be able to guarantee that it evaluates all the sensor output and that the response time (i. e., the delay between the production of an output element and the finished evaluation by the monitor) has an upper bound.

Because of the asynchronous communication between sensors and monitors, it still cannot be guaranteed that the monitor's view of the monitored system is current, as this would require interrupting the execution of the system. On the other hand, in most cases systems cannot be controlled synchronously at every point in time. For example, the way in which packets contained in a data stream are processed usually cannot be changed during the processing of a packet but just before the processing of an individual packet is started. As mentioned in Section 3.1, the monitored system should wait explicitly until the end of evaluation by the monitor if a synchronous reaction is required. For example, the widely used interceptor pattern can be implemented in this way.

Consequently, there will be a trade-off between two different requirements. The sensor is supposed to be independent of the monitor, but it must not too fast—that is, it must not produce output elements at a higher rate than that at which the monitor can consume them.

It is not sufficient to give the monitor a higher priority when it accesses resources. Sensors should be embeddable in real-time systems, too. Furthermore, monitors should be able to guarantee a maximum response time and ensure guaranteed evaluation of sensor output for an unlimited amount of time. For this reason, sensors and monitors must be able to reserve resources prior to their execution. This requires an upper bound for the rate at which sensors produce output elements.

In GRTMon, the production rate is modeled using jitter-constrained streams [6] (JCS). These streams specify arrival times of periodic events that are allowed to arrive earlier or later by a bounded amount of time. Arrival processes can be described more conveniently and more precisely by this means than by, for example, strictly periodic arrival rates. Precise models of the sensors' production rates are extremely important because monitors reserve resources based on these rates. Jitter-constrained streams are used in other DROPS components too, so the use of the same model makes it easier to add sensors to these components.

Monitors are modeled as periodic tasks. The number of output elements that have to be processed by the monitor for a given sensor is calculated based on the monitor's period length, the required maximum latency of evaluations, and the sensor's jitter-constrained stream. The worst-case execution time required to evaluate one output element is then used to determine the resource requirements of monitors. The user provides this worst-case execution time and specifies the monitor's period length and maximum latency of evaluations. I will explained this in detail in Section 4.2.

Even if the monitored system is not a real-time system, a guaranteed evaluation of sensor output is an advantage. GRTMon uses a sensor's jitter-constrained stream only as an upper bound for the rate at which the sensor produces output elements. Non–real-time systems can easily wait to adhere to this constraint.[3] Thus, a monitor giving guarantees can control a non–real-time system because no information about the monitored system is missed and, as a result, the monitor's view of the system is accurate. Additionally, the parameters of the sensors' jitter-constrained streams can be chosen without having to consider possible delays in non–real-time systems.

If the monitored system produces output at a rate that exceeds the upper bound, the monitor will miss output elements. The monitor can still consume a subset of the produced output elements, unless the sensor is so fast that it always preempts the monitor during read operations and thus prevents the monitor from reading consistent data. Nevertheless, the monitor always knows how many output elements it missed. It can either treat such a case as a failure or it can perform evaluations based on a subset of the input data. I will describe the communication mechanism between sensors and monitors in Section 4.1.

3.3 Additional Requirements

There are additional, equally important requirements besides the ones mentioned previously. I will discuss them in the following paragraphs.

n:m Relation of Sensors and Monitors A monitor must be able to combine the output of several sensors because sensors extract information from distinct parts of the system, for example the operating-system kernel and user processes. These parts are,

[3]This results in a probe effect because it changes the system's temporal behavior. However, such an effect is only a probe effect because the one who monitors the system assumes that the temporal behavior of the non–real-time system is more or less deterministic. There is often no guarantee that all deviations from the assumed behavior were caused by the monitoring activities. In practice, increasing the upper bound (and possibly allowing larger peak rates) should often be a sufficient solution.

for example, in different security domains and must be independent from each other. Likewise, it must be possible for several monitors to read one sensor's output because independent monitors are needed in a multi-user environment such as DROPS.

This does not affect sensors because sensors do not synchronize with monitors.[4] If sensors had to synchronize with monitors, a n:m relation would make the implementation a lot more difficult.

Evalution of Sensor Output in Chronological Order Sensors inform about a monitored system by producing either complete state descriptions or just change information (incremental descriptions). Monitors adapt their view of the system by evaluating output elements in the same order as they were produced. This way, monitors can reconstruct the progression of the monitored system.

The output elements produced by a single sensor must be temporally ordered. Monitors have to sort the output elements to restore the chronological order if they are reading from more than one sensor. To accomplish this, sensors tag every output element with a timestamp obtained from the CPU's timestamp counter, which contains the number of elapsed CPU cycles since the CPU has been powered on. Sensors cannot establish such an order themselves because they have to be independent from each other; otherwise, this would lead to significant probe effects.

Arbitrary Sensor Output Sensors must be able to produce arbitrary output—that is, the produced information must not be limited to a certain domain. In the scope of the work presented here, it is not feasible to try to provide a solution or data type for every possible problem. Instead, the one who adds sensors to the monitored system (e. g., the user of the monitoring tool) must be able to select and define the data type of the output produced by a sensor.

Arbitrary Evaluations by Monitors For similar reasons, the power of the algorithms used to evaluate sensor output should not be limited. Although tools like DTrace show that reducing the power (like in this case, no support for loops) can be feasible, there is no actual need in GRTMon or monitoring tools with a similar architecture. The evaluation is decoupled from the invocation of sensors, so the resource usage caused by evaluating sensor output can be solely handled on the monitor side and does not need to be restricted in the first place.

Additionally, determining which restrictions would be useful but not too limiting is a difficult task and beyond the scope of the work presented here. For example, not supporting loops would make the determination of worst-case execution times easier, but currently, there is no indication for loops being not required by future GRTMon monitors.

3.4 Architecture

Section 3.1 explained why I designed the communication between sensors and monitors to be asynchronous and uni-directional. Sensors and monitors are usually separated by

[4]The synchronization information provided by sensors can be read by several monitors.

being contained in different tasks, which will be called *sensor tasks* and *monitor tasks*, respectively. The basic data flow remains unchanged: sensors write output elements to buffers from which monitors read. Each buffer is mapped at the respective sensor task and all associated monitor tasks, but only the sensor task has write permissions. Monitors sort output elements based on their timestamps before evaluation. Figure 4 shows such an arrangement consisting of two sensor tasks and two monitors.

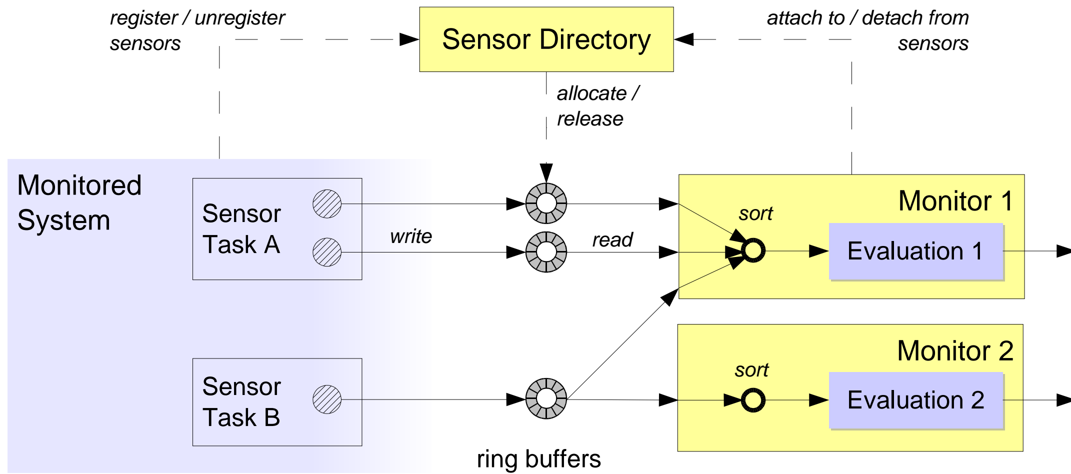

Figure 4: Basic Architecture

A sensor directory is responsible for the management of relations between sensors and monitors. Thus, sensors have no direct relation to monitors, which relieves the monitored system significantly. Sensors in particular do not have to provide an interface for monitors at which these can get access to the sensors' output buffers. This reduces the size and complexity of monitoring-related code in the monitored system. Probe effects become smaller as well because sensors can select the point in time at which they communicate with the sensor directory and do not have to reply to requests by monitors at any time. Instead, sensors can inform the directory about their existence in the initialization phase of an application, when real-time guarantees are usually not required.

Additionally, the sensor directory represents a namespace for sensors and can thus be used as a central name service.

Registering Sensors and Attaching to Sensors Sensor tasks must register their sensors at the sensor directory by informing the directory about the following functional and nonfunctional properties of the interface representing the sensor:

- An arbitrary *sensor identifier* in the form of a character string that must be unique with respect to identifiers of other sensors contained in the same task
- A *type identifier* (character string) specifying the data type of output elements as well as the size of an output element
- Parameters of the jitter-constrained stream used to model the sensor's output production rate

A sensor is addressed by the task it is contained in and its sensor identifier. The task can be used to select an instance of a monitored system or a component of a system, for example a certain application. The identifier addresses a structural part of this instance, for example the name of the property measured by the sensor. Tasks are used instead of threads because the latter are rather implementation-specific and thread identifiers might not be reproducible, for example if threads are started in no particular order. In contrast, software components are often separated by the means of different tasks.

However, this addressing scheme is disadvantageous if sensors are not static parts of the task's structure, that is, if there are several instances of a sensor related to certain entities in the task and sensors are created dynamically. There is no general solution to the resulting problem of addressing a certain instance of a sensor. The solution rather depends on the semantics of the instances, that is, how a monitor can select an instance in a convenient way without having to consider internals of the monitored system. Thus, there does not need to be a complex syntax as long as complex semantics can be expressed. Although not perfect, encoding the required information as a character string and appending it to the sensor identifier is sufficient.

Monitors can query the sensor directory about properties of sensors. They can attach themselves to sensors to get access to the sensors' output buffers. An access control mechanism has not yet been implemented but could be added at any time.

Allocation of communication buffers The sensor directory allocates the ring buffers used for the communication between sensors and monitors. Read or read–write permissions are granted to the monitor tasks and sensor tasks, respectively, but the directory remains the owner of the buffer.[5] This design decision is based on three reasons:

- Monitors are to be allowed to guarantee evaluation of every output element produced by a sensor. Thus, the rate at which monitors would like to consume data, the sensor's production rate, and the size of individual output elements have to be considered when choosing the size of a sensor's output buffer. Ultimately, how to resolve the trade-off between small periods of monitors and large buffers is a configuration decision and cannot be determined by a sensor or a monitor alone.

- Allocated buffers are a form of resource usage. Because the size of a buffer is not determined by the sensor, a sensor task should not be accountable for this resource usage. In fact, neither monitors nor sensors are solely responsible.

- Monitors can read from a sensor's output buffer even if the sensor task has been terminated unexpectedly. Thus they do not need to immediately handle such a failure (i. e., a terminating sensor task does not result in the revocation of the

[5] Memory is allocated at *dm_phys*, the DROPS service that manages main memory. Allocated memory regions have a single task as owner but access rights can be granted to other tasks. If a memory region is released or a region's owner terminates, all access rights to this region are revoked.

monitor's access rights to the buffer). Monitors can evaluate sensor output that they have not processed yet, for example to examine the cause of the termination.

In GRTMon's architecture, it is assumed that there is one central instance of the sensor directory and that it does not terminate as long as any sensors or monitors exist.

Small Set of Dependencies There are just a few dependencies between sensors, monitors, and the sensor directory. This leads to a lower complexity and a smaller probe effect:

- Sensors only depend on the sensor directory. First of all, they use shared memory owned by the directory, which can result in page faults during sensor invocations if failures at the directory lead to the termination of the directory or to the revocation of the sensor's access rights to its output buffer.
  Second, sensors are registered at the directory, so a failed registration prevents monitors from reading from the sensor. However, the sensor can write to a buffer allocated by the sensor task, which makes the failed registration transparent when the sensor is invoked.

- Monitors depend on the sensor directory. Dependencies related to output buffers of sensors are the same as for sensors. However, monitors need the directory to be able to work at all, therefore the dependency is stronger.
  Monitors need the output produced by sensors but they do not depend on sensors (e. g., an unexpectedly terminating sensor task will not lead to page faults in a monitor). Correct sensor output is required to get an accurate evaluation of the monitored system's state, but this always requires that monitors trust their sensors.[6] On the other hand, monitors can protect themselves from obviously wrong output. For example, they can detect basic failures during communication such as timestamps that are not monotonically increasing.

- The sensor directory neither depends on sensors nor on monitors. Its resource usage is affected by the amount of registered sensors and the size of their buffers, but per-task quotas can be used to limit the resource usage caused by sensor tasks. The directory's resource usage is important because it needs to reserve resources when operating in a real-time environment.

The set of used DROPS services is small, too. Basically, services to allocate CPU time (*cpu_reserve*) and memory (*dm_phys*), and the central name service (*names*) are required. Sensors can operate without any of these services if they provide output buffers by themselves (e. g., the operation-system kernel can allocate memory by itself).

[6]Monitors delegate observations to sensors (compare Figure 1 on page 1 and Figure 3 on page 10). Thus, monitors rely on the assumption that observations by sensors are as good as observations performed by the monitor. What this means in terms of, for example, data integrity or timeliness depends on the actual observation. Monitors will usually require that sensor output is correct, complete, and up-to-date.

The small set of mostly weak dependencies between the three parts of the architecture does not significantly decrease the level of safety and security in the whole system. For example, unexpectedly terminating sensor tasks do not lead to page faults in monitor tasks and vice versa, monitors and sensors can remain in different security domains and are independent of each other, and sensors can be located in the operating-system kernel without the need for major changes regarding the communication scheme (see Section 4.1.3).

3.5 Ease of Use

In the introduction, I already explained why a monitoring tool must be easy to use. The central idea is that a proper monitoring tool has advantages (like late binding or separated resource usage) compared to other solutions, and thus can be better regarding aspects such as separation of concerns or code reuse. However, it will not be used if it is too complicated. Users will usually prefer solutions that need less effort in the beginning, even if this decision might lead to disadvantages in the future. For example, it is rather easy to add logging statements that write to the standard output of a system, but this does not scale well.
So, there are three things that a monitoring tool must accomplish:

1. Solving simple tasks must be simple. This attracts users in the first place.

2. When the tool is used for larger tasks or for a longer amount of time, it must exploit its technical advantages and pass the resulting benefits on to the users.

3. Finally, solutions to originally simple tasks must be allowed to grow and evolve into solutions for complex tasks because people work in this way. There should not be a gap between the means used to solve simple and complex tasks.

Similarly, the monitoring tool has to provide means to solve monitoring tasks belonging to the two categories explained in the introduction:

- Determining properties of a monitored system must be simple and fast. In the beginning, it is often not known which measurements have to be performed or which properties are relevant. So, users will experiment with changing sets of sensors and evaluations. Therefore, it must be easy to add sensors to the monitored system, to evaluate the output of these sensors, and to make the results of the evaluation accessible to the user (for example, by visualization of measured values).

- When verifying properties, it must be easy to guarantee the evaluation of sensor output. Basically, this depends on how production rates of sensors are specified and on the determination of the resource requirements of evaluation algorithms. Additionally, it must be simple to trigger reactions such as adaption decisions based on the result of evaluations; for example, a video player application might need to reduce the video's quality if more than ten video frames have been missed.

By integrating both categories, a monitoring tool will be able to decrease its costs compared to custom or specialized solutions, because sensors and evaluations can be reused. For example, the cause of failures or anomalies is often discovered by experimenting with the monitored system. As a result, one gets an evaluation that can be used to detect the failure. The same evaluation can then be reused in the scope of a test or as a runtime constraint to detect failures with a similar cause.

The weak coupling between monitors and small sensors makes it possible to divide the monitoring problem and solve the specific subproblems separately. Additional features of monitors that increase ease of use do not have any negative effect on the monitored system.

Sensors On the sensor task's side, there should be a small and simple interface for the invocation and management of sensors. The production of output elements must not have functional side effects. These requirements do not only allow for a small probe effect as discussed in Section 3.1, they make using sensors easy as well. For example, if the invocation of a sensor only affects a few resources, inserting sensors into the system requires less effort because the user has to consider fewer dependencies to the other parts of the system.

In GRTMon, the interface of sensors consists of only two small functions for the production of sensor output, which I will describe in detail in Section 4.1. A sensor can be registered at the sensor directory at any time by calling a single function, the same is true for removing a registration.

Sensors are not required to produce output, and an output element can be produced at an arbitrary pace—for example, sensors can be preempted. After a sensor has been registered at the sensor directory,[7] producing an output element cannot fail. Thus there is no need for any error handling, which makes it possible to use several methods for instrumenting the monitored system with sensors.

Monitors Because sensor invocations should be inexpensive, they will often cover rather small pieces of the state of the monitored system. To get a complete and meaningful picture of the state of the system, the user needs to combine the output of several sensors. The simple combination and evaluation of the output of several sensors is therefore important to the user.

In GRTMon, evaluation algorithms have to be implemented using C++ or C. Although a scripting language like the one used for DTrace could provide more specialized means, users would be required to learn a new language. Contrary to that, C/C++ is the primary language in the DROPS environment, and the targeted user group is experienced in it. Reacting to results of the evaluation will often consist of calls to existing C/C++ code, so a scripting language would lead to the need for an adaption layer. If a scripting or other language would be beneficial nonetheless, it can still be implemented on top of C/C++ code. However, implementing an interpreter

[7]The sensor task can as well provide an output buffer for the sensor by itself if registering the sensor at the sensor directory fails.

or compiler for such a scripting language is beyond the scope of the work presented here.

Consequently, GRTMon has to provide a convenient software environment for evaluations. Users should only need to write the actual evaluation code and should not need to cope with management tasks and other repetitive jobs. A monitoring framework is the best solution because the amount of evaluation code is in most cases smaller than the code required for other tasks, for example reading and sorting of sensor output or invoking evaluations. Additionally, there is usually no need to customize the code for these tasks.

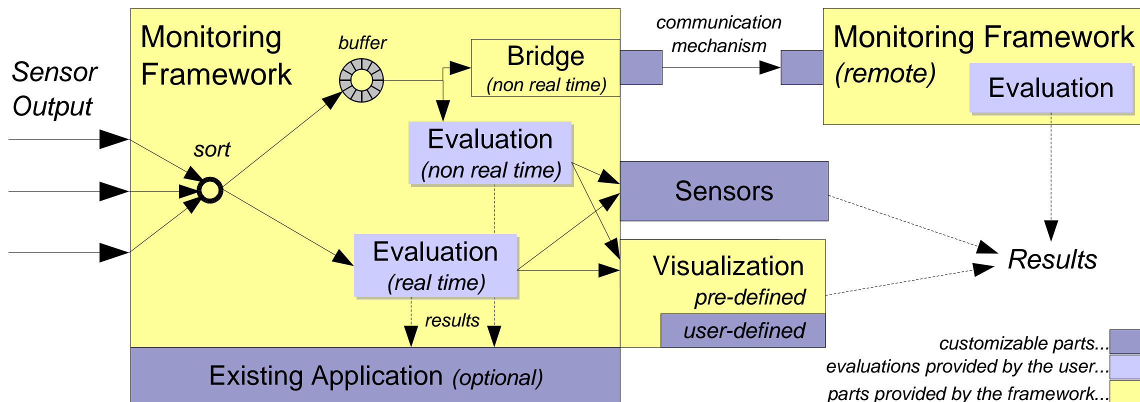

Figure 5: Architecture of monitors

Figure 5 shows the architecture of monitors. The parts of the figure with a dark background can be easily customized, evaluation code is provided by the user, and bright parts are provided by the framework and do not need to be changed. The monitoring framework is the essential element. It usually serves as a stand-alone monitor, that is, it provides all the code required to build a monitoring application. The user just has to embed custom evaluation code. However, the framework can as well be part of an existing application, for example for adaption control. From the application's perspective, the framework is a library in this case, which is easy to include and which encapsulates all the monitoring code.

The data flow is as follows: Sensor output is either evaluated or buffered in its chronological order. Pairs of an output element and its timestamp serve as input to evaluation algorithms. Results can be used in several ways:

- Results can control an existing application.
- Sensors can be used to publish results. This allows for the creation of evaluation hierarchies that can benefit from advantages of the monitoring tool like late binding or the compact specification of sensor interfaces. However, each evaluation step adds latency.
- Results can be visualized. Visualization is an important tool during experiments with the monitored system. There is often a large amount of numerical data, which has to be presented visually to be comprehensible for users. Because monitoring should be easy, related tasks must be easy as well. Therefore, visualization

is supported by GRTMon by providing a library that contains base classes and helper classes for customized visualization methods. Currently, there is support for text–mode and GUI-based (using DOpE [5]) data displays. Implementations for both text–mode and GUI–mode exist for two types of histograms. Creating displays for other data types is simple. The visualization library includes most of the code that is not datatype-specific, so users just have to implement the specific visualization algorithm.

Another possibility is to transfer sensor output to other systems, which is mostly needed for the remote evaluation of data on other computers, for example if the monitored system runs on an embedded system but the evaluation requires resources or software tools that are not available on this system. The used communication mechanism can be easily customized because it only consists of single-threaded, non–real-time converters. Currently, GRTMon provides an implementation that writes Base64-encoded sensor output to the DROPS logging service. This service uses either the serial interface (available on most systems) or a TCP/IP connection (higher throughput but not available everywhere) as output channel. The remote monitoring framework runs on Linux and evaluates the stream of sensor output.

The remote evaluation of sensor output is necessary to monitor distributed systems. However, distributed monitoring is beyond the scope of the presented work.[8]

Ease of use is affected by nonfunctional aspects, too. Although evaluation of sensor output can be guaranteed if the sensor complies with the specification of its production rate, estimates of the worst-case execution times (WCET) of evaluations or other parts of the monitoring framework are often difficult to obtain, or WCET estimates are too high compared to the the actual WCET (which might prevent the monitor from being scheduled even if enough resources are available). GRTMon's monitoring framework treats this problem by two means:

- The monitor can either run as a constant-bandwidth server with a user-provided bandwidth or resource requirements can be determined based on the sensors' jitter-constrained stream specifications. In the former case, as many output elements as possible are processed during each period of the monitor, and the amount of resources reserved does not depend on any estimates of execution times. In the latter case, the number of output elements to process per period is calculated based on the sensors' jitter-constrained streams (see Section 4.2 for details). In turn, resource requirements are determined based on the WCET of evaluations and the number of elements to evaluate per period.

- Furthermore, evaluations can be executed by a best-effort thread. To accomplish this, the original evaluations are replaced by functions merely copying output

[8] There is no support yet for the synchronization of clocks, so the monitoring tool is limited to a system with a single clock (i. e., a single CPU because the CPU's timestamp counter is used as time source).

elements to a ring buffer.[9] The best-effort thread consumes elements from this buffer and executes the original evaluations.

As a result, resource usage policies for the execution of evaluations can be chosen by the monitor. Only the resource usage caused by the copy functions depends on the sizes of the sensors' output buffers and on the production rates of the sensors. However, worst-case execution times are much easier to estimate if the data has just to be copied but not evaluated. In Figure 5 on page 19, evaluations executed by best-effort threads are marked as non–real-time evaluations.

The buffering of sensor output is not required. The best-effort thread can read data directly from a sensor's output buffer, although lengthy evaluations might lead to more data being missed.

There is only one interface that has to be provided by evaluation code, independent of whether the evaluation takes places in a real-time, non–real-time, or remote environment. I will describe this interface in Section 4.3. This allows for using the same evaluation code during both determination and verification of properties, unless services specific to a certain environment, for example special visualization methods on remote systems, are used. The same is true for the small amount of initialization code required for stand-alone monitors.

[9] The size of the ring buffer must be specified by the user. This requires the user to estimate how much space is needed in order not to lose sensor output because of buffer overruns. Nevertheless, this is often the easiest solution if input parameters like the production rate of sensors or worst-case execution times of evaluations are unknown or just roughly estimated values.

4 Implementation

In this section, I will discuss the implementation of the communication between sensors and monitors, I will explain the model based upon which GRTMon guarantees the evaluation of sensor output, and I will present two essential interfaces of GRTMon's monitoring framework.

4.1 Communication between Sensors and Monitors

The following requirements and design decisions regarding the communication between sensors and monitors have been explained in the previous sections:

- Sensors communicate asynchronously with monitors. Sensors do not synchronize with monitors (i. e., they do not pay attention to any monitor's state), but they provide information that can be used by the monitors to synchronize with sensors. Monitors must ensure that only consistent data is evaluated.
- The output of a single sensor consists of several output elements that have to be ordered temporally.
- Monitors preserve the order of a sensor's output elements and merge orders if several sensors are monitored.

If monitors do not read sensor output fast enough, they will miss output elements. A monitor has to be able to handle three situations related to missed output:

- It must be possible to start monitoring after a sensor has been running for a long time. The monitor will miss old output of the sensor because buffer sizes are usually limited.
- If an evaluation algorithm requires that its input is a complete, sequential stream of all the output elements produced by a sensor since a certain point of time, the monitor has to detect whether it missed any output element after monitoring has been started. If output has been missed, the monitor must stop the evaluation and communicate its condition.
- If the evaluation algorithm can cope with incomplete input, it still often needs to know how many output elements have been missed (e. g., for statistical evaluations and error estimates). The monitor must be able to detect the exact number of missed output elements and continue reading from the sensor.

Because sensors and monitors will be embedded in real-time environments, the execution times of the code executed when writing or reading output elements are of interest when designing and implementing the communication scheme.

Probe effects can be caused by the overhead associated with the invocation of a sensor as well as by the jitter of the overhead. There is no general rule which of both, overhead or jitter, has a higher influence, as this depends on the monitored system and on whether sensors are a part of the original system or added on demand. In any

case, the overhead of a single invocation of a sensor must be bounded (i. e., writing an output element must succeed after a finite number of steps) and small.

This requirement applies to monitors as well. When reading an output element, the monitor must be able to determine—after a finite number of steps—whether it read consistent data. If the data is inconsistent, the monitor must be able to find the next available output element in a short, bounded amount of time.

Because of the required temporal order of these elements, the output produced by a sensor is a stream of output elements. Sensors write their output to the ring buffer shared with the monitors and inform them about the current position in the stream. To determine which elements are still available, monitors either use a tail position (i. e., the position of the oldest element still available) provided by the sensor or compute this position based on the sensor's position and the size of the buffer.

In GRTMon, all output elements have a fixed size set by the user during the registration of a sensor. This has the advantage that the memory for new output elements can be allocated easily and no advanced memory management is needed. Additionally, the sensor-specific data of output elements can be embedded directly in the sensor's output stream (i. e., a pointer to another memory region containing the data is not required) if there are no concurrent writes to the same output element, which avoids the indirection needed otherwise and makes producing output faster. The sensor's output buffer can be organized as an array-based ring buffer, which allows for random access to elements based on the sensor's position in the stream. Consequently, a monitor that has just been started or has missed output can resume reading anywhere in the stream. Nevertheless, the sensor can still produce arbitrary output by splitting variably sized data.[10]

The DROPS Streaming Interface [10] (DSI) solves a problem similar to the communication between sensors and monitors but limits a producer to a single consumer. Extending it is not feasible because the tailor-made solution required by the monitoring context does not align well with the rather general problem DSI tries to solve.

Sensor Invocation Interface Invoking a sensor (and thus appending a new element to the output stream) consists of three steps:

1. To prepare for writing sensor-specific data of the element to be produced, the invoker must obtain a pointer to the part of the sensor's output buffer that will hold the data. The function used for this is called `write`, requires the sensor's control structure as argument and returns a `void` pointer. The control structure contains information like buffer addresses and sizes or the current position in the buffer. It is not shared with monitors.

2. Next, the invoker writes the output-element–specific data. During registration of a sensor, the user specifies the data type identifier and size of sensor-specific data. The pointer returned by the `write` function references a data structure of the same type and size.

[10] Variably sized output elements can be supported by adding an appropriate memory allocation mechanism to the implementation of concurrently invocable sensors (presented in Section 4.1.2).

3. Finally, the invoker commits the output element to the monitors by a call to the `commit` function, which reads the CPU's time stamp counter, tags the output element with this timestamp, and informs monitors that a new output element has been produced.

4.1.1 Simple Sensors that are not Invoked Concurrently

In this subsection, I will explain the implementation of *simple sensors* that cannot handle concurrent invocations. Although one might argue that this is a special case, most sensors will not be invoked concurrently by more than one thread. Even if they are, the invoker can often select an efficient mutual exclusion mechanism based on properties of the invoking context.

The reason for providing an implementation optimized for this case is that the absence of concurrency leads to less complexity, which in turn decreases the performance overhead of the implementation. Nevertheless, there is a second sensor implementation in GRTMon that can handle concurrent invocations, which I will describe in the next subsection.

As explained previously, the output of a sensor is a stream of output elements. Sensors provide monitors with the number of elements that have been produced by the sensor. This *element counter* can be used to determine the output element most recently written by the sensor and its position in the ring buffer. Figure 6 shows

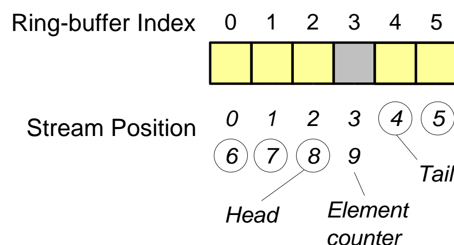

Figure 6: Stream and buffer positions and available elements

an example: nine elements have been produced by the sensor and the ring buffer contains six elements. The element at stream position eight is the most recently produced element and located at index two in the ring buffer. The oldest output element still available could be the element at position three or four, depending on whether the sensor is currently producing a new element. To avoid the performance overhead associated with providing an extra tail position, it is assumed that the sensor is always writing a new element (that is, the element at the position of the current element counter always contains inconsistent data). Because there are no concurrent invocations, only one extra element has to be reserved.[11] Consequently, the tail position (i. e., the least recently produced element still available) can be calculated based on the element counter.

Using the element counter as a means for synchronization requires that updates of the element counter are atomic. On the hardware GRTMon is used on, machine words

[11]The sensor directory considers this fact when it determines the size of sensor buffers.

can be read and written atomically. However, machine words on these platforms have a size of 32 bit, which is not enough for an element counter because ring buffer indices are calculated based on a modulo operation and it is not feasible to generally choose buffer sizes of 2^n.[12] Assuming a CPU frequency of 3GHz and that writing an output element requires at least 30 CPU cycles, a 32bit counter will wrap around after only 10 hours.

Writing and Reading Output Elements As element counters must therefore be wider than 32 bit, a mechanism for atomically incrementing the element counter is required. Newer x86 systems offer a compare–and–exchange instruction that modifies two adjacent machine words. Such an instruction can be used to increment a 64-bit element counter. Because the sensor is not invoked concurrently, the compare–and–exchange instruction will succeed on the first try (unless there are failures in the sensor task).

To read this counter atomically, the monitor has to first read the higher word, then the lower word, and finally read the higher word again and compare it with the value obtained during the first read. As the element counter is strictly monotonically increasing, a consistent 64bit value has been obtained if both reads of the higher word returned equal values. Otherwise, the monitor must retry reading the element counter. Fortunately, the higher word of the counter only changes in case of an overflow of the lower word. This enables the monitor to stop retrying after more than one attempt. The first retry does not indicate an abnormal situation, but further failed read operations mean that each time at least 2^{32} output elements were missed by the monitor. GRTMon monitors treat such a situation as a failure.[13] The number of retries can be customized by the user. Monitors can detect if element counters are not monotonically increasing. After reading or evaluating an output element, monitors check the tail position to determine whether they read consistent data. If the tail position is higher than the position of the read element, then the sensor has reused the element and the element might have been modified.

The disadvantage of the compare-and-exchange–based solution is that this instruction is expensive and not available on ARM hardware. Even if using the compare–and–exchange instruction only when necessary (the higher word of the counter is not modified most of the time), its performance overhead results in a higher jitter and worst-case execution time.

To avoid the performance overhead, GRTMon uses another mechanism to increase the element counter. The compare–and–exchange instruction is replaced by separated modifications of the lower and higher word of the counter. As this will fail if an overflow of the lower word requires an increase of the higher word, the least significant bit of the higher word is also stored in the most significant bit of the lower word. To increase

[12]The element counter wraps around at 2^{32} (i. e., it is set to zero). If the size s of the buffer is not equal to 2^n ($n \in \mathbb{N}$), then the result of $2^{32} \mod s$ will be different to the result of $0 \mod s$, which leads to the calculation of a wrong index after the element counter has wrapped around.

[13]Remember that a 32bit counter will need at least 10 hours to wrap around on current hardware if the sensor increases the element counter by one for each element produced.

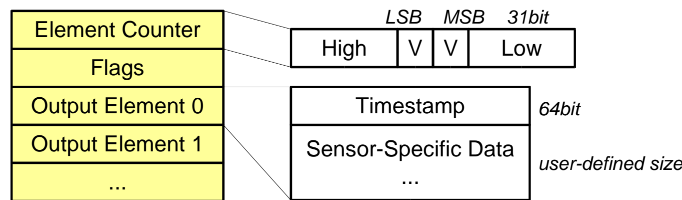

Figure 7: Layout of a sensor's output buffer

the element counter, the sensor first increments the lower word. If the most significant bit of the lower word changed, the sensor increments the higher word as well.

As a result, monitors have to deal with another special case. They still have to read the higher, then the lower, and again the higher word and retry if the higher word changed. If it did not, they now must check whether the most significant bit of the lower word is equal to the least significant bit of the higher word. If it is not equal, the higher word must be incremented because the monitor read the higher word before the sensor was able to increment it. Other possible reasons for this difference of the redundant bit are ruled out because the first and the second read of the higher word returned equal values and the sensor does not decrement the element counter.

The resulting element counter is 63 bit wide, which is sufficient.[14] The number of steps required to read the element counter is still bounded. The performance overhead of a sensor invocation is significantly smaller compared to the compare-and-exchange–based implementation.[15]

Layout of the Output Buffer Figure 7 shows the layout of the output buffer of a simple sensor. The element counter consists of the higher and the lower word; the value of the least significant bit of the higher word is stored in the most significant bit of the lower word as well (marked with a "V"). The sensor flags part is a bit field with basic information about the sensor (e. g., that the sensor cannot handle concurrent invocations). The ring buffer, a sequence of output elements, follows. Each output element is tagged with a timestamp that is obtained from the CPU's timestamp counter during the call to the `commit` function. The sensor-specific data is set by the user after the `write` function returned the address of this structure.

4.1.2 Concurrently Invocable Sensors

To allow simple sensors to be invoked concurrently, the invoking context would have to ensure mutual exclusion, which usually involves locking because the invoker does not know anything about the operations performed by the sensor. As locking has significant disadvantages, GRTMon provides a second sensor implementation that is lock-free and thus nonblocking and can be invoked concurrently by several threads.

The interface for the invocation of sensors described on page 23 contains three steps: (1) obtaining the address of the output element to be produced next, (2) writing user data to the element, and (3) committing the element to the monitors. Because the

[14] On current hardware, the production of 2^{63} elements will need almost 3000 years.
[15] Experiments showed a decrease from 55 down to approximately 30 CPU cycles (see Section 5.2).

second step is performed by the invoking thread and the operations executed in this step are unknown to the sensor implementation, the problem has to be split:

- During the first step, the sensor must allocate an output element that can be used exclusively by the invoking thread.
- In the second step, the invoking thread is free to perform all operations required to write the sensor-specific data to the output element.
- Finally, the sensor must insert the output element in the sensor's stream of output elements.

This requires the decoupling of the sensor's output stream and the output elements. The output stream now has to consist of pointers to output elements. Step one and three therefore become *dequeue* and *enqueue* operations, respectively.

Sensors must produce (enqueue) an output element after the allocation (dequeue) of an element. Although enqueue and dequeue are separate functions, the invocation of a sensor is still considered to be an atomic operation with regard to the executing thread: A single thread must perform the operation and the thread must not mix the three steps of the operation with other operations affecting the same sensor.

Using linked lists to model the stream is disadvantageous because this would prevent random access to elements of the stream. Additionally, element counters are still required to enable monitors to compute the number of missed output elements. For this reason, GRTMon uses an array-based implementation that extends the approach from the previous subsection.

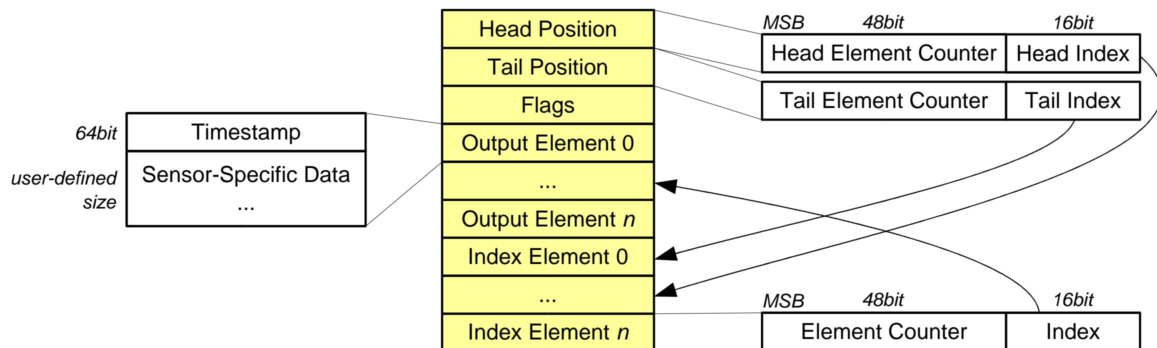

Figure 8: Layout of the output buffer of a concurrently invokable sensor

Layout of the Output Buffer Figure 8 shows the layout of the sensor output buffer. Compared to the layout for simple sensors (see Figure 7), a tail position and a sequence of indices have been added. The array of output elements remains unchanged. The index array has the same number of elements as the output-element array.

To access an output element at a given stream position, first the ring buffer position must be calculated from the stream position (see Figure 6 on page 24). The index stored in the index array at this ring buffer position then points to an output element in the output-element array.

Head and tail position have a similar layout. The head element counter is the number of previously produced elements. The tail element counter is the stream position of the oldest element still available (i. e., the least recently produced element that has not yet been reused by performing a dequeue operation). The index parts of both positions point to the index array element that corresponds to the respective position. This avoids having to recalculate the array index.[16] The tail position has been added because it cannot be derived from the head position anymore; dequeue and enqueue operations are decoupled and not a single atomic operation as in the case of simple sensors.

Elements of the index array consist of tagged indices of output elements. The index is tagged with the value of the element counter at the time of the enqueue operation, which is equal to the position in the stream. Indices have to be tagged because of the ABA problem: a value to be modified by a compare–and–exchange instruction might be changed—if the modifying thread is preempted—by other threads from A to B and again to A, resulting in a successful compare–and–exchange although the modification should have failed. Using the strictly monotonically increasing element counter as a tag avoids this problem. The index part is 16 bits wide and the element counter part 48 bits, which is still sufficient in practice.[17]

The enqueue and dequeue operations that will be described next show similarities to other concurrent queue algorithms, for example the linked-list–based algorithm presented by Michael and Scott [12]. However, the asynchronous producer–consumer relation between sensors and monitors allows for discarding two requirements of general-purpose concurrent queues:

- As a sensor is limited to a single task, the maximum number of threads that can belong to a task is equal to the maximum number of dequeued elements (invoking a sensor is considered to be an atomic operation for the invoking thread).[18] Because monitors only observe produced output and do not consume it by performing a dequeue operation, the queue will never be empty and dequeue operations will never fail if the queue is larger than the maximum number of threads per task and initially filled with dummy elements.

- Because sensors are required to dequeue an element before they produce a new element, a queue will never be full (i. e., an enqueue operation will never fail).

The algorithm can probably be extended to fail if the queue is empty or full, but this has not yet been implemented.

[16] The modulo operation used to compute the index based on a stream position (see Figure 6) is quite expensive as the position is wider than 32 bits. Remember that array sizes are not limited to 2^n values.

[17] A current 3GHz CPU would need almost one year of consecutive enqueue–dequeue operations to cause an overflow of the element counter. Handling of such overflows is possible (but has not been implemented). Monitors and sensors can detect overflows if the maximum timespan between two consecutive operations involving the queue is not larger than the minimum time required for the overflow. Indices will always be correctly calculated from stream positions if the size of the buffer is 2^n or the number at which the counter overflows is a multiple of the buffer size.

[18] GRTMon associates a task with each sensor (see Section 3.4). However, the implementation could be extended so that concurrently invokable sensors can be shared by several sensor tasks. In this case, the user would have to provide the maximum number of dequeued elements.

There is one further restriction. As stream positions are used for tagging indices, at most 2^{48} elements can be produced by such a queue. Monitors rely on stream positions as well, which causes the same restriction. Nevertheless, stream positions provide more information about the current state of the queue than, for example, buffer indices.

There are not many changes required to support concurrently invocable sensors at the side of the monitor. Monitors must use the element counter parts of the head and tail position as real head and tail positions and they must read output elements based on the indices in the index array.

Allocating an Output Element (Dequeue) The implementation of the dequeue operation consists of three steps:

1. The tail position is read. The technique explained in the previous subsection (reading the higher, the lower, and again the higher word) can be reused even if the index part is not monotonically increasing because the element counter part is strictly monotonically increasing and located in the upper 48 bits of the 64 bit wide tail position. If the first and the second read of the higher word returned different values, this step is executed again.

2. The element counter part and the index part of the local copy of the read tail position are advanced.

3. Using a compare–and–exchange instruction, it is tried to exchange the tail position in the output buffer with the modified local copy. If the exchange fails because the value in the buffer has been modified, the algorithm is restarted at step one. If the operation succeeds, the index that was read in step one is returned. It points to the dequeued and thus allocated output element.

As step three will fail whenever another thread completes the dequeue operation before the current thread, an upper bound for the number of retries can only be guaranteed based on assumptions about concurrent sensor invocations. For example, Anderson and colleagues present in [1] upper bounds for the number of retries required by lock-free operations in priority-based systems with periodic tasks. The algorithm is nonblocking because one of the threads that try to dequeue the element will succeed.

When a sensor's output buffer is created, head and tail position and the index array are initialized such that the buffer contains a sequence of dummy elements: Each element of the index array points to the output element with the same index, the tail position is set to zero, and the head position is set to the number of output elements in the buffer. Monitors start reading after these dummy elements (i. e., they start at a stream position equal to the number of output elements in the buffer).

Similar to the single spare element required for simple sensors, the initially determined amount of output elements (e. g., based on the jitter-constrained stream parameters) is increased by the maximum number of concurrent invocations. In the L4 version that GRTMon currently runs on, at most 128 threads can belong to a task, so the number of output elements is increased by 128.

```
do {
  // Step 1
  head = buffer->head_position;
  if (inconsistent data read) continue;
  // Step 2
  index = buffer->index_array[head.index];
  if (inconsistent data read) continue;
  if (index.element_counter > head.element_counter) continue;
  if (index.element_counter != head.element_counter) {
    // Step 3
    buffer->output_element_array
          [index_of_output_element_to_be_enqueued].timestamp = rdtsc();
    // Step 4
    new_index.element_counter = head.element_counter;
    new_index.index = index_of_output_element_to_be_enqueued;
    enqueued = compare_and_exchange(buffer->index_array[head.index],
                   index, new_index);
  }
  // Step 5
  new_head.element_counter = head.element_counter + 1;
  new_head.index = (head.index + 1) % ring_buffer_size;
  compare_and_exchange(buffer->head_position, head, new_head);
  // Step 6
} while (!enqueued);
```

Figure 9: Pseudocode for the enqueue operation. Only `buffer` is shared.

Committing an Output Element (Enqueue) The enqueue operation requires the index previously returned by the dequeue operation as parameter. The enqueue algorithm consists of the following steps (see Figure 9):

1. The head position is read in the same way as the tail position was read during the dequeue operation. Inconsistent data have been read if the first and the second read of the higher word returned different values.

2. The index element that is contained at the read head position is read in the same way.

   If the element counter part of the index element is greater than the element counter part of the head position read during step one, then more than one element has been inserted into the index array by other threads and the current thread has to restart at step one (its local copy of the head position is out-dated).

   If it is equal, then another thread put the element into the index array but did not yet advance the head position. Therefore, the current thread cannot enqueue its element and has to proceed at step five (it joins the group of threads that failed at inserting their element at step four). It must not restart at step one because it must help to advance the head position.

3. The current value of the CPU's timestamp counter is stored in the timestamp field of the output element to be enqueued. This step is performed after reading the head position to keep the timestamps ordered.[19]

4. A compare–and–exchange instruction is used to try to exchange the index element in the output buffer with a new index element whose element counter part has been increased and whose index part points to the output element to be enqueued. The exchange will only take place if the value in the output buffer is the same as the value read at step two (ensured by the compare part of the instruction). The result of this instruction (i. e., whether the exchange succeeded) is stored for later use.

5. A second compare–and–exchange instruction tries to increase the head position if and only if the value in the output buffer is equal to the value read at step one. This step is performed even if the previous compare–and–exchange failed. Consequently, only one of the contending threads will increase the head position, but the position will be increased as soon as and only after one of the threads has successfully inserted an output element into the index array.

6. If the first compare–and–exchange (step four) failed or it has not been tried yet, the algorithm is restarted at step one. If it succeeded, then the current thread was able to enqueue its output element and the enqueue operation is finished.

Similar to the dequeue operation, it cannot be guaranteed that the algorithm terminates after a certain number of tries. Nevertheless, the algorithm is still nonblocking (i. e., progress can be guaranteed for at least one of the contending threads).

[19] The element will only be enqueued if the current thread read the most current version of the head position. If it did, then the obtained timestamp is current, too.

The element counter parts of the head position and the element most recently inserted into the index array will either be equal or differ by one (head position is greater in this case). The elements in the index array do not have larger element counter values because every thread that tries to insert an element will try to increase the head position. The head position never commits elements that have not yet been inserted, because it is only increased by threads that either tried to enqueue an element (step 4) or threads that encountered a state in which the head position lagged behind the most recently inserted element (second condition at step two). In both cases only one of these threads increments the head position.

Threads that execute the steps based on out-dated information will not modify anything because all modifications require an up-to-date local copy of the element counter. Step four uses the element counter part of the index to be replaced as condition for the compare–and–exchange operation, but this element counter is only used if the index is old (i. e., the element counter is smaller than the head position).

The second condition at step two makes sure that all threads that operate based on current information try to advance the state of the queue. The two modifications (steps four and five) will each be performed by exactly one of these threads.

4.1.3 Sensors in the Operating-System Kernel

Some information required by evaluations cannot be observed by userland code. For this reason, the operating-system kernel has to make this information accessible, for example by the means of sensors that are a part of the kernel.

Examples of this kind of information are the points in time at which context switches took place, inter-process communication (IPC) between arbitrary threads (which do not or cannot invoke userland sensors reporting about these IPC operations), or implementation-specific events in the kernel itself such as calls to certain kernel functions.

Context switches become even more important when combined with information obtained from hardware sensors. Such sensors, for example the performance counters of x86 systems, provide statistical and often accumulated information. They can tell whether an event took place while the CPU was in kernel or userland mode, but they cannot associate an userland event with the thread whose execution caused the event. To accomplish this, the values of the performance counters have to be taken at each context switch. A monitor can then use these values and perform per-thread accounting for the properties measured by the performance counters (e. g., important metrics like the number of cache misses for a certain thread during a user-definable time span).

Like every sensor task, the kernel has to be protected from effects caused by the invocation of sensors. Probe effects must be kept small, although they cannot be avoided completely.[20]

[20]For example, microkernels are built to do a small set of things fast. Information like which kernel functions have been called cannot be embedded in other output, so it usually has to be written to a buffer. These memory accesses alter the cache use of the kernel, which will result in probe effects.

Fiasco Trace Buffer Fiasco, the microkernel used in DROPS, already contains a mechanism [16] to report about kernel-level events, called the *trace buffer*.

Entries of this buffer are produced upon several events, for example context switches, IPC, rescheduling, or page faults. Users can select which events trigger a new tracing entry being placed into the trace buffer.

All these entries contain a timestamp taken from the CPU's timestamp counter during the production of this entry, values of up to two performance counters that can be selected by the user, an integer value representing the type of the event, and event-type–specific data (e. g., the threads that were involved in case of an IPC event). Entries have a fixed size of 64 byte.

The communication with the consuming userland process works as follows. The trace buffer is split into two adjacent buffers from which userland processes can read. The kernel treats both buffers like one ring buffer and fills them with tracing entries. When it has completed filling one of the buffers (i. e., it advances to the next buffer), it fires a virtual IRQ. Consumers evaluate the entries in the respective part of the trace buffer upon receiving an IRQ message.

This double-buffering scheme causes a serious problem: If no entries are written to the trace buffer, then consumers cannot evaluate entries already located in the buffer. Furthermore, if the information from the trace buffer has to be combined with output from other sensors, then the trace buffer entries and the sensor output have to be evaluated in chronological order. As a result, evaluation of the output of other sensors will be delayed as well. In the following, a solution is presented that does not show this behavior.

Custom Sensors GRTMon's sensors do not suffer from this problem. Fortunately, the trace buffer is similar to a simple sensor:

- The kernel ensures mutual exclusion when trace buffer entries are produced.
- The trace buffer is a ring buffer with elements of fixed size.
- Each entry in the trace buffer is tagged with a timestamp from the CPU's timestamp counter.

Replacing the existing trace buffer with GRTMon's sensor implementation does not seem feasible at the moment because it would require too many changes to Fiasco's source code. On the other hand, adding an element counter that is incremented by the kernel after each production of a new trace buffer entry and that can be read by monitors results only in a few changes to the source code. I have appended such an counter to the status information already accessible at the so-called *trace buffer status page*. This way, the trace buffer becomes a *custom sensor* in GRTMon's perspective and can be treated like the sensor implementations provided by GRTMon, which do not suffer from the previously mentioned latency problem.

Because of the different layout of sensor output buffers and the trace buffer, an adapter has to be used in the monitoring framework. These adapters are called *custom sensor readers*. They provide the same interface and operations as monitors do for reading from GRTMon's sensors (e. g., checking whether new output elements are

available or reading of an output element) but the implementation is tailored to the custom sensor (e. g., the trace buffer). Custom sensor readers are not shared among monitors. Custom sensors can be referenced by a monitor-local ID.

In the case of the trace buffer, the implementation of the associated custom sensor reader follows the same algorithm that is used for the reading from GRTMon's sensors. Differences are mostly caused by a different layout of the data structures, for example a different position of the timestamp in trace buffer entries and output elements.

As a result, the trace buffer can be used like any other sensor. Trace buffer entries and output elements of other sensors are evaluated in chronological order.

A remaining problem is how to easily specify the rate at which the trace buffer produces information. The average rate of events might be extremely high and depends on the behavior of all tasks and on external input. Currently, users specify the production rate of custom sensors based on a rule of thumb.

4.2 Guaranteed Processing of Sensor Output

To be able to guarantee that monitors evaluate all output elements produced by a sensor, the rate at which the sensor produces output elements must have an upper bound (see Section 3.2).

In GRTMon, this upper bound is modeled using a jitter-constrained stream [6, 7] (JCS), which has the following parameters (the notation from [6] is reused):

- A period T ($T \in \mathbb{R}$, $T > 0$)
- A minimal temporal distance D ($D \in \mathbb{R}$, $0 \leq D \leq T$) between two output elements (in the case of a sensor, elements produced by the sensor replace the otherwise abstract notion of JCS events)
- The maximal earliness τ of an element ($\tau \in \mathbb{R}$, $\tau \geq 0$)
- The maximal lateness τ' of an element ($\tau' \in \mathbb{R}$, $\tau' \geq 0$)

Because there is no temporal reference point, the streams (T, D, τ, τ') and $(T, D, \tau + \tau', 0)$ are equivalent. GRTMon performs this transformation, therefore only τ is used in the following and τ' is assumed to be zero.

Monitors are modeled as periodic tasks with a user-definable period length T_M ($T_M \in \mathbb{R}$, $T_M > 0$). Threads that finish work at a relative time smaller than the relative deadline d ($d \in \mathbb{R}$, $d > 0$, $d \leq T_M$) have not missed the deadline.

The reason for bounding the production rate of sensors is to be able to calculate the maximal number of output elements R ($R \in \mathbb{N}$, $R > 0$) that have to be processed by a monitor in each of its periods. The monitor can then reserve resources based on R and the worst-case execution time of the evaluation of a single output element (provided by the user).

GRTMon's producer–consumer relationship between sensors and monitors is based on asynchronous communication. In [6], the consumer is modeled differently:

- The consumer is assumed to wake up and start working immediately once a new element is inserted into the previously empty queue used as communication mechanism.

- The consumer keeps processing elements from the queue until the queue is empty.

Contrary to that, monitors are not woken up when a sensor has produced an output element, and they only process data if they have not yet completely consumed the CPU time available in the current period.

Similar to sensors, there is usually no temporal reference point associated with a monitor, so the phase shift between sensors and monitors is unknown, which must be considered in the following.

4.2.1 Processing the Output of a Single Sensor

Sensors must not be too fast for monitors if the evaluation of all output elements is to be guaranteed. As a result, a worst-case scenario will involve a fast sensor producing output at a high rate and a slow monitor.

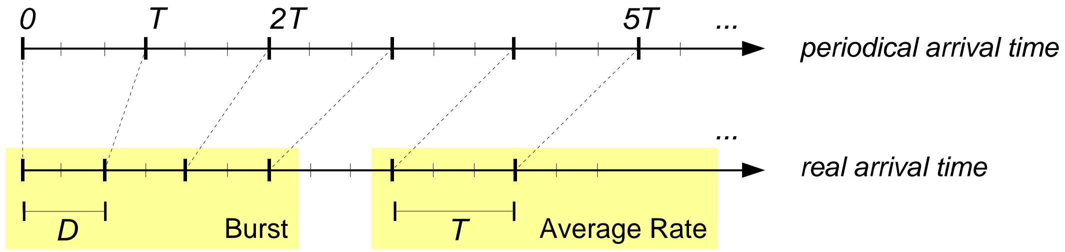

Figure 10: A dense $(3, 2, 3, 0)$-stream

The average production rate ($\frac{1}{T}$) of a sensor is bounded by the period of the associated JCS. However, a sensor can produce output elements earlier than expected (τ specifies the upper bound for this earliness). Consequently, if elements are produced as early as possible after a certain point in time, the production rate in this timespan will seem to have increased.[21] Such a stream or part of a stream is called dense. The first element of the stream is produced at the expected point of time, whereas all the following elements are produced as early as possible (see Figure 10). This leads to a maximum burst at the beginning that is followed by single elements produced at the average rate. The maximum burst length L is determined by[22]

$$L = 1 + \left\lfloor \frac{\tau}{T - D} \right\rfloor.$$

Obviously, the average consumption rate of a monitor must be at least as high as the average production rate of the sensor. However, the timespans at which monitors

[21]Possible lateness (τ') does not need to be considered because it is transformed to be a part of τ.
[22]For $n \in \mathbb{Z}$ and $x \in [0, 1)$: $\lfloor n + x \rfloor = n$, $\lceil n - x \rceil = n$

are executed vary as well because they depend on the scheduling algorithm and the current load of the system (i. e., the amount and priorities of other threads).

Two properties of a sensor–monitor communication are especially important: B, the number of output elements that the sensor's output buffer must contain, and l, the maximum latency of evaluations. For simplicity, l will be referred to as latency from now on and specifies a strong upper bound, that is, the smallest value that is larger than the latency (i. e., the temporal difference between the availability of the element and end of evaluation at the monitor) of all elements.

The model used for monitors is rather simple. Monitors are required to evaluate output elements in chronological order, but there are no restrictions regarding execution times or how a monitor has to determine if new output elements are available for evaluation.

In particular, the comparison between the monitor's position in the stream and the head position of the sensor is allowed to take zero time and to take place only at the start of work in each period. As a result, it is possible that a monitor misses all output elements produced in the respective period. This behavior is a precondition for the worst case (a slow monitor).

When monitors that guarantee the evaluation of all elements produced in the future are started, they discard elements already contained in the sensor's output buffer so as to have a known state to start from. Otherwise additional CPU time would be required during the first periods because the monitor would have to catch up.

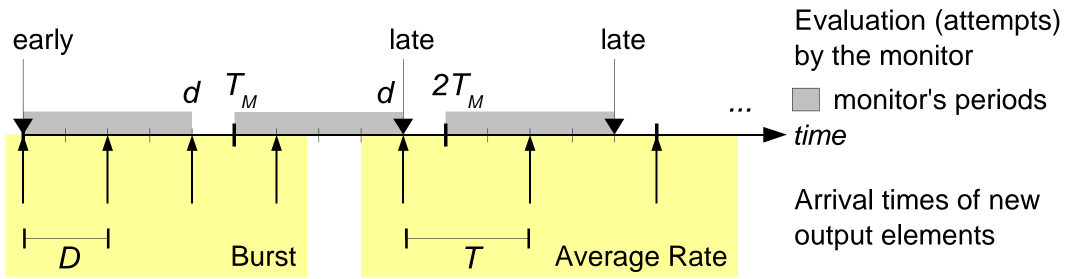

Figure 11: Worst-case for a dense $(3, 2, 3, 0)$-stream and a monitor with $T_M = 5$, $d = 4$

Figure 11 shows the worst case: a combination of a dense stream of output elements and a delayed monitor. In the first period ($[0, 5)$), the monitor checks for new elements exactly at time 0 and thus misses even the first element of the burst (the check is assumed to take zero time). In the following periods, the evaluation finishes as late as possible (but still before the deadline).[23]

In the worst case, monitors will start evaluating elements at the end of the second period. Thus, the latency l has a lower bound l_{min}, which is the minimum latency that can be guaranteed by a monitor:

$$l_{min} = T_M + d \tag{1}$$

[23]This is a pessimistic assumption. Evaluations will have to finish earlier (and thus, will be scheduled to start earlier, too) because of the worst-case execution times reserved. However, taking the execution times for single evaluations into account would make the equations more complex and would still only give slightly better result because the actual order of evaluations is not known and reserved execution times will usually be significantly smaller than the period length.

The latency l determines the required buffer size B because new output elements must be buffered if the monitor is still evaluating an older element. $E(t)$ is the maximum amount of elements produced by a sensor before time t:[24]

$$E(t) = \begin{cases} \left\lceil \frac{t+\tau}{T} \right\rceil & \text{if } t \geq LD, \\ \left\lceil \frac{t}{D} \right\rceil & \text{otherwise.} \end{cases} \tag{2}$$

$$B(l) = E(l) \tag{3}$$

Consequently, the minimum size of the sensor's output buffer depends on the latency's lower bound. As l_{min} in turn depends on T_M, the sensor directory—which is responsible for allocating output buffers—dictates a maximum length for T_M (currently one second, d is assumed to be equal to T_M). When a buffer is allocated, the sensor directory uses the maximum T_M and the parameters of the sensor's JCS to calculate B.

The maximum value for T_M restricts monitors but is the least limiting restriction. Resizing a sensor's output buffer is extremely difficult or even impossible if the sensor's task cannot map the memory pages allocated for the new elements. Additionally, a fixed value for the maximum length of T_M gives monitors the guarantee that their chosen period is not too large with respect to the buffer sizes of future sensors, which is important if monitors need to increase the set of observed sensors during runtime.

Because sizes are calculated based on the maximum value for T_M chosen by the sensor directory, monitors are limited to latencies smaller than or equal to $2T_M$ (Equation 1, $d = T_M$). However, this only bounds the timespan during which monitors must have finished reading an element from the buffer. The latency of evaluations can be higher.

Latency vs. Number of Elements Processed per Period Monitors have to choose between a low latency and a small rate of evaluations ($\frac{R}{T_M}$). Both is not possible because a low latency requires that the monitor is able to catch up faster after a burst, which in turn requires processing elements at a higher rate.

The required latency is usually determined by the purpose of monitoring. Monitors that only gather data for later use (e. g., logging or tracing) can usually cope with a large latency, whereas monitors that control the monitored system based on the results of evaluations will require a low latency. For this reason, GRTMon lets the user select feasible values for latency, T_M and d.[25]

Figure 12 shows the production rates of a $(3, 2, 3, 0)$-stream and compares them to the range of possible consumption rates (the highlighted area).

An obvious lower bound R_{min} for R is the average production rate:

$$R_{min} = \left\lceil \frac{T_M}{T} \right\rceil \tag{4}$$

[24] [7] uses a similar equation that calculates the number of events produced at times smaller or equal to t.

[25] Both T_M and d influence l_{min} and the average latency (which is smaller than the maximum latency l).

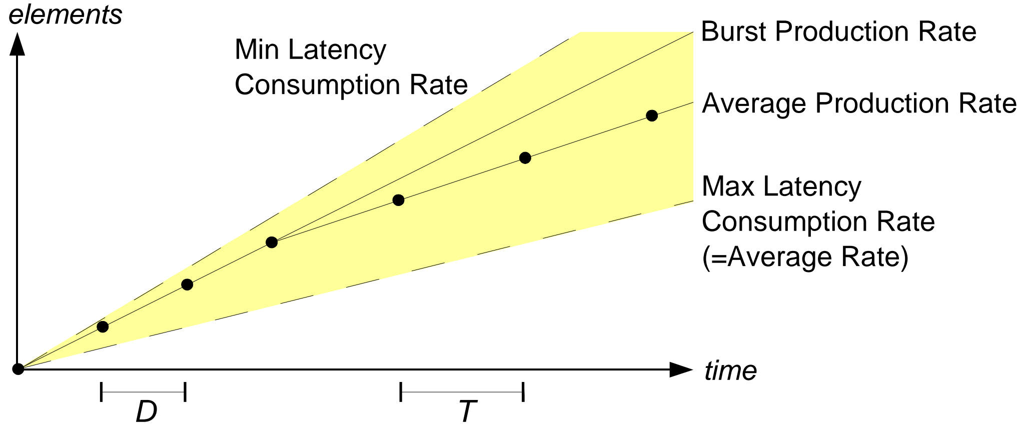

Figure 12: Production Rates vs. Consumption Rates

If a monitor evaluates all elements that were produced in the previous period, it virtually starts again at time 0 (the buffer is initially empty, see Figure 11 on page 36). As a dense stream produces the most elements during intervals that start at time 0, an upper bound R_{max} can be derived from Equation 2:

$$R_{max} = E(T_M) = \begin{cases} \left\lceil \frac{T_M + \tau}{T} \right\rceil & \text{if } T_M \geq LD, \\ \left\lceil \frac{T_M}{D} \right\rceil & \text{otherwise.} \end{cases} \tag{5}$$

In the example given in Figure 11 on page 36, R_{min} is 2 and R_{max} is 3.

Trying to directly calculate $R(l)$ leads to a tricky noncontinuous function. An approach based on calculating $l(R)$ is easier and will be presented next.

$t_S(e)$ states at which time element e ($e \in \mathbb{N}$) is produced by the sensor, and $t_M(e)$ states the earliest point in time at which e could have been completely evaluated by a monitor:

$$t_S(e) = \begin{cases} eD & \text{if } e < L, \\ eT - \tau & \text{otherwise.} \end{cases} \tag{6}$$

$$t_M(e) = T_M(\left\lfloor \frac{e}{R} \right\rfloor + 1) + d - \epsilon \qquad (\epsilon > 0) \tag{7}$$

ϵ is an extremely small value that is used to express that the monitor consumes elements as late as possible but always before the deadline; if we would determine limits, then the expression $\epsilon \rightarrow 0$ could be used.

Both of the previous equations describe the worst case shown in Figure 11. Note that $t_M(0)$ is equal to $l_{min} - \epsilon$, the latest point in time before the deadline of the monitor's second period. Furthermore, $t_M(e)$ does not correctly state the time at which an element will be evaluated ($t_M(e)$ can be smaller than $t_S(e)$).

l bounds the latency of each evaluation, so the following inequality must hold for all e:

$$\begin{aligned}
t_M(e) - t_S(e) &< l && (8)\\
t_M(e) - l &< t_S(e) && (9)\\
T_M(\left\lfloor \frac{e}{R} \right\rfloor + 1) + d - l - \epsilon &< t_S(e) && (10)\\
T_M(\left\lfloor \frac{e}{R} \right\rfloor + 1) + d - l &\leq \begin{cases} eD & \text{if } e < L, \\ eT - \tau & \text{otherwise.} \end{cases} && (11)\\
T_M + d + T_M \left\lfloor \frac{e}{R} \right\rfloor - l &\leq \begin{cases} eD & \text{if } e < L, \\ eT - \tau & \text{otherwise.} \end{cases} && (12)\\
l &\geq T_M + d + T_M \left\lfloor \frac{e}{R} \right\rfloor - \begin{cases} eD & \text{if } e < L, \\ eT - \tau & \text{otherwise.} \end{cases} && (13)
\end{aligned}$$

The smallest value of l is equal to the maximum value of the right-hand side of Inequality 13. Note that $e = 0$ yields l_{min} and smaller values (e. g., $t_M(e)$ is smaller than $t_S(e)$ for large e) are not considered. Inequality 13 consists of monotonically increasing and decreasing terms.

The value of $T_M \left\lfloor \frac{e}{R} \right\rfloor$ increases at each period of the monitor. The involved floor–function represents the fact that all evaluations in a period happen at the same time (see Figure 11). e is contained in the decreasing part ($-eD$ or $-eT$), too. As only the maximum value for the right-hand side of the inequality is of interest, it is sufficient to examine only the first elements kR ($k \in \mathbb{N}$) in each $[kR, kR + R)$-interval (period k of the monitor).

If R is equal to R_{max}, then l is equal to l_{min}. For $R < R_{max}$, Inequality 13 distinguishes between elements that are part of the burst ($e < L$) and elements produced at the average rate:

$e < L$**:** Because R is smaller than R_{max} and R_{max} is equal to $E(T_M)$ (i. e., the elements in the first period, Equation 5), at least one element produced in the first period will not be evaluated by the monitor in the second period.[26] As long as the monitor evaluates elements produced during the burst, it will not be able to catch up. It evaluates at most $R_{max} - 1$ elements per period but the sensor produces R_{max} or $R_{max} - 1$ elements in the same timespan. Fewer elements might be produced in the final period of the burst, but these elements will not be evaluated until the following period.

As a result, the number of elements that are not yet evaluated is monotonically increasing for increasing k. Because an increasing backlog at the monitor increases the latency, the largest k for which $kR < L$ still holds can be used to determine the maximum latency for the first part of the inequality. This element is called e_b:

$$e_b = R \left\lfloor \frac{L - 1}{R} \right\rfloor \tag{14}$$

[26] Remember that there are no evaluations in the first period.

$e \geq L$**:** This part of the inequality covers elements produced at the average rate. Because R is equal to or greater than R_{min} (derived from the average production rate, see Equation 4), the monitor's backlog will decrease with each period or at least stay constant. Thus, the largest latency will be found at the the smallest k for which $kR \geq L$ holds:

$$e_a = e_b + R = R\left\lfloor\frac{L-1}{R}\right\rfloor + R \tag{15}$$

e_b, e_a, and Inequality 13 can be used to derive two conditions regarding l:

$$l \geq T_M + d + T_M\left\lfloor\frac{e_b}{R}\right\rfloor - e_b D \tag{16}$$

$$l \geq T_M + d + T_M\left\lfloor\frac{e_a}{R}\right\rfloor - e_a T + \tau \tag{17}$$

Elements between e_b and e_a do not have to be considered because $t_S(e)$ (see Equation 6) is monotonic (element indices correspond to the temporal order) and decreases the right-hand side of Inequality 13, while the other parts of the expression do not change in $[e_b, e_a)$.

Because none of the two conditions is stronger (i. e., always yields a higher value than the other), l must at least be larger than the maximum of the values obtained by the conditions. However, both conditions already yield the maximum value in their part of the inequality's domain ($e < L$ and $e \geq L$) and are thus the only conditions to be considered. Therefore, l can be assumed to be equal to the maximum value:

$$l = T_M + d + max\Bigg(\left\lfloor\frac{L-1}{R}\right\rfloor(T_M - RD), \left(\left\lfloor\frac{L-1}{R}\right\rfloor + 1\right)(T_M - RT) + \tau\Bigg) \tag{18}$$

l is monotonically decreasing if R increases because evaluating elements at a higher rate will keep the backlog smaller.

Because $l(R)$ is monotonically decreasing and R_{min} and R_{max} specify a range for possible values of R, binary searching can be used to determine the smallest R that results in l being smaller than or equal to the latency specified by the user. Performing a search instead of directly calculating the value is not a problem because R is only determined once before monitoring is started.

Maximum Backlog of Monitors The previous equation for the required size of output buffers ($B(l)$, Equation 3) only returns the amount of output produced by a sensor. However, it does not consider that monitors read from the buffer. As a result, the obtained value is correct if a monitor's l_{min} is equal to the latency used to calculate $B(l)$, but the value will be too high if l_{min} is smaller.

A lower bound for the buffer size can be calculated based on the backlog $b(t)$ of the monitor, which is the difference between the amount of elements produced by a sensor and the amount of elements evaluated by a monitor at a certain point in time:

$$b(t) = -R\left\lfloor\frac{t-d}{T_M}\right\rfloor + E(t+\epsilon) \qquad (\epsilon > 0, \text{ see Equation 7}) \tag{19}$$

The right part of the equation represents the amount of elements produced by a sensor exactly at time t ($E(T)$ returns the amount of elements before time t and is monotonically increasing). The left part is a monitor evaluating elements as late as possible and thus reducing the backlog (expressed by the steps at $kT_M + d$, $k \in \mathbb{N}$). Because we are interested in the greatest difference, we only need to the consider the points in time right before the monitor evaluates elements ($kT_M + d - \epsilon$):

$$b(kT_M + d - \epsilon) = -R\left\lfloor \frac{kT_M - \epsilon}{T_M} \right\rfloor + E(kT_M + d) \tag{20}$$

$$= -R(k-1) + \begin{cases} \left\lceil \frac{kT_M + d}{D} \right\rceil & \text{if } kT_M + d < LD, \\ \left\lceil \frac{kT_M + d + \tau}{T} \right\rceil & \text{otherwise.} \end{cases} \tag{21}$$

The highest value of b can be obtained in a way similar to the solution for the latency. There are two conditions for the points in time during the burst and after the burst, for which the values $k_b = \lceil \frac{LD-d}{T_M} \rceil - 1$ and $k_a = \lceil \frac{LD-d}{T_M} \rceil$, respectively, lead to the maximum values:[27]

$$b_b = -R(k_b - 1) + \left\lceil \frac{k_b T_M + d}{D} \right\rceil \tag{22}$$

$$b_a = -R(k_a - 1) + \left\lceil \frac{k_a T_M + d + \tau}{T} \right\rceil \tag{23}$$

If k_b is smaller than 1, b_b must be discarded because it determines the value for a time at which the monitor has not yet begun reading elements (i. e., before $T_M + d - \epsilon$). This happens when the monitor starts reading after the end of the burst. If k_a is smaller than 1 as well, it must be set to 1, which represents the time at which the monitor starts reading ($T_M + d - \epsilon$). For example, the configuration shown in Figure 11 yields $k_b = 0$, $k_a = 1$, and $b_a = 4$.

The maximum of b_b and b_a is $b_{max}(R)$, the maximum backlog with which a monitor evaluating R elements per period will have to cope. This function can be used to determine:

- Whether the sensor's output buffer provided by the sensor directory is large enough for a certain value of R (probably calculated based on the latency requested by the user)
- The size of a private buffer allocated by the monitor if the sensor's output buffer is not large enough

The second option is only useful if the worst-case execution time required for the evaluation of output elements is significantly higher than the time required for copying elements to the private buffer. Because elements have to be evaluated in chronological order, the backlog of copied but not yet evaluated elements might become larger than

[27] It is assumed that $R \geq R_{min}$ and $R < R_{max}$, so the backlog will have its maximum in the time around the end of the burst, which is represented by the equations for k_a and k_b.

R, which leads to the situation that all further elements are first copied and evaluated later in one of the following periods.

Although this situation might not always be the case, the amount of elements to copy per period is currently determined based on the worst case. It can be obtained by a binary search for the smallest R for which $b_{max}(R)$ is smaller than the provided buffer.

4.2.2 Processing the Output of Several Sensors

Monitors must evaluate sensor output in chronological order. This requirement eases the construction of evaluation algorithms because it ensures that the monitor's view of the monitored system always represents the complete state of the system at the point in time given by the output element's timestamp.

However, this requires the monitor to merge the output elements of all observed sensors according to the elements' timestamps. To accomplish this, the monitor always evaluates the oldest output element still available in one of the sensor output buffers. Because the output elements of a single sensor are already in chronological order, the monitor only has to consider the oldest element of each sensor. Like in the single-sensor case, the monitor's period T_M and the maximum latency of evaluations l are provided by the user. R is calculated separately for each sensor based on the common latency and the sensor's jitter-constrained stream.

The time at which a sensor's output elements are evaluated is influenced by other sensors that produce elements at a rate higher than the average rate. Other sensors producing elements at the respective average rate have no influence because R is assumed to be greater than or equal to R_{min}.

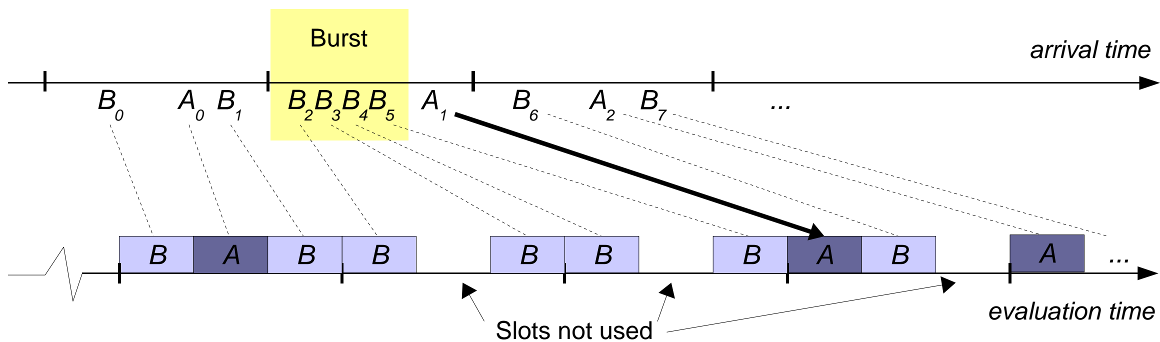

Figure 13: Delayed evaluation because of a burst at another sensor

Figure 13 shows what happens when a burst at sensor B delays the evaluation of sensor A's output elements. R_A and R_B (i. e., the number of output elements that have to be evaluated per period for sensors A and B) have values of 1 and 2, respectively. They are correct in the scope of a single-sensor scenario. Nothing unusual happens in the first period, where A and B produce elements at an average rate. However, B produces elements at a burst rate in the second period. Because of the required chronological order of evaluations, B's burst elements have to be processed first. This leads to two periods where no element of sensor A is evaluated, and the reserved

CPU time is lost.[28] Element A_2 results in an unused time slot for sensor B, too. The evaluation of element A_1 is delayed by two periods and leads to further delays and unused CPU time in the following periods. Thus, the CPU time required to evaluate elements within the bounds given by the latency increases.

This problem can be avoided by allowing the monitor to spend CPU time according to the chronological order of the elements to be evaluated. The CPU time required in each period for a certain sensor's elements is calculated based on this sensor's R and the worst-case execution time for a single evaluation. These sensor-specific CPU times are then added to a pool of CPU time, which will be used by the monitor to evaluate elements in chronological order.

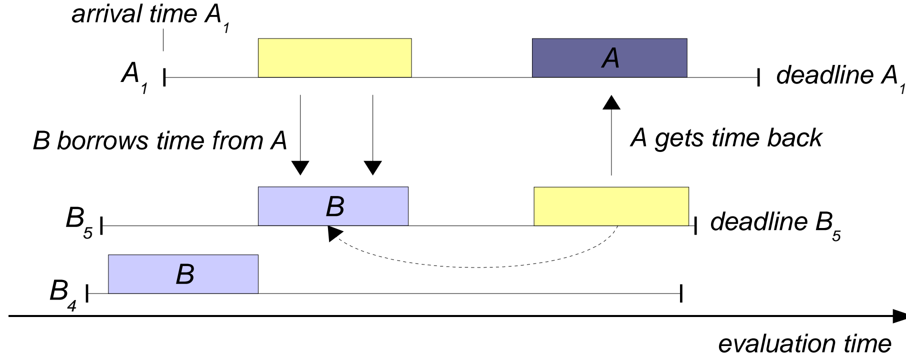

Figure 14: Evaluations borrowing CPU time from each other

The borrowing of CPU time has no influence on the latency that can be achieved. Figure 14 illustrates the reason for this. Elements B_4, B_5 and A_1 from the previous figure have to be evaluated. The amount of CPU time reserved for the elements of sensor B is large enough to guarantee the latency l. Thus, when sensor B's evaluation borrows time from sensor A's evaluation, it uses CPU time that would be available later (e. g., in one of the following periods) for sensor B's elements.

Because the evaluation of B_5 did not use CPU time reserved for sensor B, sensor A's evaluation will be able to borrow the same amount of CPU time from B at a later point in time. The CPU time will be available soon enough because the latency l applies to all sensors. B_5 is older than A_1, so the deadline for its evaluations will be earlier than A_1's deadline and B will be able to lend CPU time back to A.

Thus, this borrowing scheme corresponds to earliest-deadline-first scheduling. Output elements are still evaluated in chronological order because the temporal order of the deadlines corresponds to the temporal order of the arrival times.

Although delayed evaluations do not increase latency, delays have an influence on the required buffer size. Calculating a precise minimum for the required buffer size is difficult and depends on the characteristics of the other sensors. Additionally, the set of sensors observed by a monitor can change during monitoring. This might require the monitor to enlarge buffers during runtime, which is difficult as well.

[28]In the following, only CPU time is considered. Similar resources (i. e., resources that are consumed by the monitor and refreshed at the beginning of each period) can be treated in the same way.

However, the worst case is known: a sensor that always evaluates elements with a maximum latency (l). In this case, the buffer size is equal to $E(l)$, the amount of elements before time l. If $E(l)$ is larger than the number of elements in the sensor's output buffer, the monitor must allocate a private buffer and must reserve additional CPU time for the copying of elements to this buffer. The copy operations are independent of evaluations and other sensors. The amount of required copy operations per period and sensor can be determined based on $b_{max}(R)$ and the size of the sensor's output buffer.

Different approaches based on adding the delay caused by the bursts of other sensors to the delayed sensor's backlog seem to be promising but I have not yet developed them further.

Limitations of the Current Implementation GRTMon's current implementation of the concepts explained in this section is a proof of concept. It shows that evaluations of the output of several sensors can be guaranteed and that the amount of resources required by these evaluations can be calculated and reserved before monitoring is started.

The present model and equations already yield useable values or at least bounds. However, the size of buffers or the amount of reserved resources are sometimes larger than required (e. g., the amount of required copy operations).

The user can select the important parameters T_M, d and l but currently the user has no information about or control over internal trade–offs. For example, it could be better to increase R to avoid the overhead of copy operations. On the other hand, if a sensor exceeds its specified production rate, a fast consumption (i. e., low-latency copy operations) of sensor output can lead to fewer missed elements.

Consequently, further work should try to examine which parameters are the most beneficial for users, lead to the best results (e. g., resource utilization), and are feasible in practice (e. g., decreasing the monitor's relative deadline d decreases l_{min} but also limits scheduling possibilities).

4.3 The Monitoring Framework

GRTMon's monitoring framework is written in C++ to take advantage of the benefits of object-oriented programming. Programs written in C can use wrapper functions. Users mostly work with two essential parts of the framework:

- The interface between evaluation algorithms and the framework
- The interface used to start and stop evaluation of a sensor's output

Both interfaces are each represented by an abstract class and are linked via a third class. The relations between these three classes are shown in Figure 15 on the next page and will be described in the following.[29]

[29]The figure shows simplified versions of these classes. For example, real data types are mostly omitted and only the classes' methods that are relevant for the following description are shown.

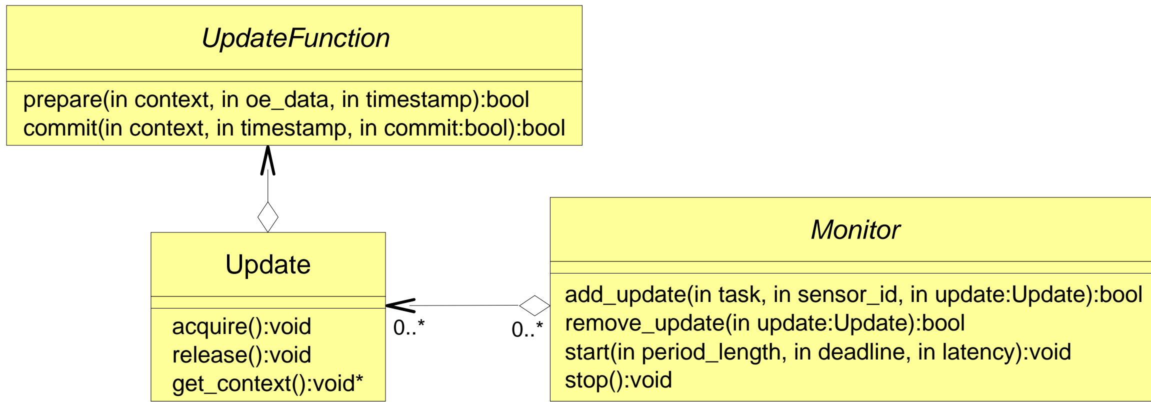

Figure 15: Update functions, updates, and monitors

Update Functions The `UpdateFunction` class represents the interface that evaluation algorithms have to implement: A function that gets called for each output element to be evaluated. Because the class is not required to be stateless, it can as well be an automaton. During an evaluation, the function updates the monitor's view of the state of the monitored system (hence the name "update function"). The function can additionally trigger reactions by the monitor (e. g., it can send control commands back to the monitored system).

However, the function has to be split into `prepare` and `commit` parts because the monitor does not have exclusive access to the evaluated output element. The sensor could reuse and thus change the output element during evaluation, which would lead to the evaluation of inconsistent data.

To avoid this problem, the monitoring framework first instructs the evaluation algorithm to prepare the result based on the sensor-specific data in the output element (`oe_data`) and the output element's timestamp (`timestamp`). The evaluation algorithm returns whether the preparation succeeded, which depends on the current state and the input data. If the `prepare` method succeeds and the output element has not been changed during the call to `prepare` (determined by the monitor by comparing the sensor's tail position with the own stream position), the monitor calls the `commit` method with the `commit`–parameter set to `true`. Otherwise, `commit` is called with the `commit`–parameter set to `false`. The evaluation algorithm then either commits or discards the previously prepared update.

The `context`–parameter passed to both methods is provided by the user and can point to state shared by several update functions, for example an instance of a model of the monitored system, which is updated according to the observations made by the monitor. Shared state is important if the output of several sensors has to be combined.

Because monitors execute evaluations in the chronological order of the processed output elements, every evaluation is atomic with respect to other evaluations and does not need to cope with concurrent modifications of local or shared state. This serialization makes the implementation of evaluation algorithms easier.

The sum of the worst-case execution times of the `prepare` and `commit` methods is the worst-case execution time of an evaluation. Both functions are executed in a real-time context if the monitor gives real-time guarantees.

On the other hand, the constructors and destructors of the `UpdateFunction` class and its subclasses are executed in a non–real-time context. An update function instance is destructed as soon as it is not used by monitors any more (for example, because evaluation of the sensor's output has been stopped). The destructor can then be used to, for example, release resources or visualize the results of the observation without having to give real-time guarantees.

Updates Contrary to update functions, instances of the `Update` class do not implement an evaluation algorithm but associate such an algorithm with a certain context (the `get_context` method returns the context provided by the user for the calls to `prepare` and `commit`). That way, algorithms remain independent of the context and of the management of the context and its relation to the algorithm.

Additionally, the `Update` class counts references to itself (reference holders call the `acquire` and `release` methods) and destructs itself and its update function when it is not being used any more.

Monitors Users only have to interact with monitors to start and stop monitoring and change the set of sensors whose output is evaluated.

Evaluations are started by calling the `add_update` method, which binds an `Update` instance to a certain sensor addressed by the sensor's task and identifier. The `remove_update` method schedules the removal of an update. Evaluations are not stopped immediately because this would require waiting for evaluations with possibly large worst-case execution times. Additionally, evaluations must be released (by a call to `Update`'s `release` method) in a non–real-time context.

Evaluations can nevertheless be started and removed from a running monitor without disturbing other evaluations. This feature is important because pausing the monitor would lead to missed sensor output and transferring state between monitors is difficult (for example, the view of the monitored system's state depends on the amount of sensor output that has already been processed by the other monitor).

GRTMon provides several implementations of the `Monitor` interface. Section 3.5 explained the environments in which evaluations can be executed (e. g., real-time, non–real-time, and remote, see Figure 5), which corresponds to the implementations that are available.

5 Evaluation

The following evaluation consists of two parts. First, GRTMon is evaluated from a user's perspective by presenting a real-world monitoring scenario. Second, the performance overhead of sensors and of the monitoring framework will be discussed.

The CPU cycles that have been used by a thread for the execution of instructions will be called the *thread time*.

The computer used for the following benchmarks and examples is an AMD K7 at 500 Mhz with an AMD 751/756 chipset and 128MBytes of RAM running current versions of DROPS' components and a current L4v2 kernel.

5.1 An Example

The example presented here is based on a monitoring scenario provided by Norman Feske. He wanted to find out how much CPU time is spent in each part of DOpE [5], a window server for real-time and embedded systems, when clients execute a function in the server.

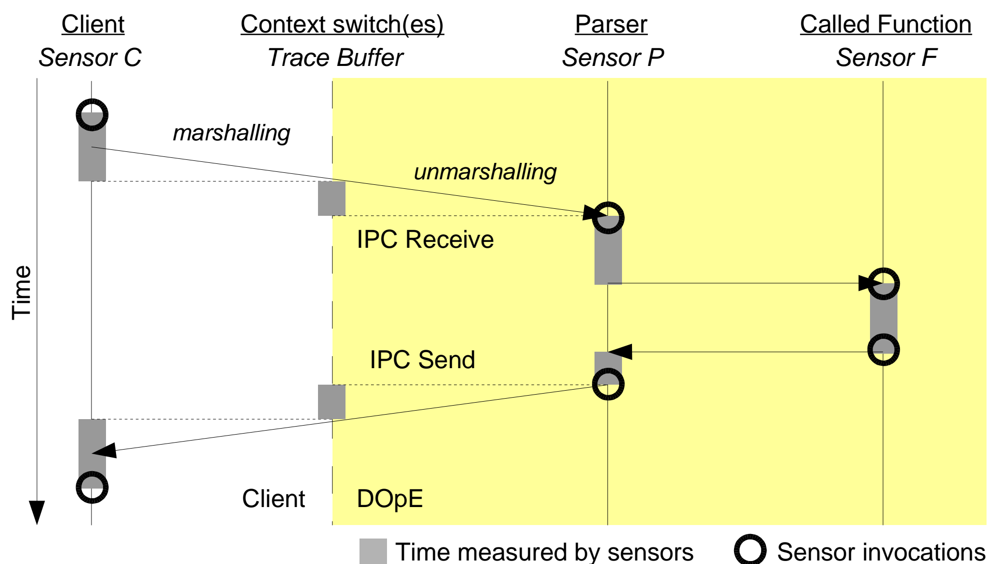

Figure 16: Path taken by calls to functions provided by DOpE's server process

Figure 16 shows the path taken by a call to functions provided by DOpE's server process. On the client side, a character string representing the function to be called and the arguments to this function is sent to the server via an IPC message. The server receives the message, parses the string, calls the function specified by the client, and sends an IPC message containing the function's return value back to the client.

In the figure, time spans are represented by the sum of dark rectangles at a column. Four different time spans have to be measured:

1. The thread time required by the client for calling the function's client-side stub and the time spent for inter-process communication in the client's context, which is equal to the thread time between the two sensor invocations shown in the left column

2. The thread time spent by DOpE for inter-process communication, that is, the thread time between the context switch and the activation of the parser plus the time between the return of the parser and the next context switch back to the client

3. The thread time required for parsing the character string and setting up the IPC message with the function's return value

4. The thread time used by the called function (in the example, the `probemode` function of virtual screens is called)

Required Sensors To measure these time spans, two sensors must be added to the server and one to the client. In the example, the client is a small test application that periodically calls the function to be measured.

The sensor C at the client is invoked immediately before and after the call to the function. On each invocation, the sensor computes the thread time of the invoking thread and copies this value to the output element. An additional flag in the sensor-specific part of each output element is set according to whether the element was produced before or after the call. When the client registers the sensors at the sensor directory, it sets the size of sensor-specific data based on the sizes of the data types used for the thread time and the flag.

Sensor P at the server is invoked before parsing starts and before the function's return value is sent to the client via IPC. Its output elements contain the thread time of the server thread and the ID of the calling thread. This way, the monitor is able to consider only the calls executed by the test application. Sensor F is invoked immediately before and after the function call.

Because DOpE uses a single thread to process the received function calls, simple sensors can be used. The same applies to the client's sensor. Parameters of the jitter-constrained stream used to specify the production of sensors can be chosen based on a rule of thumb because the evaluation of all output elements is not necessary (the monitor can easily discard a measurement if it missed output elements).

Sensors invoked by the server or the client cannot be used to compute the time used by the server thread for receiving and sending the IPC messages associated with a certain call. The time span between two calls can be measured but it consists of values belonging to distinct calls. For this reason, the trace buffer is used to log context switches. The monitor is then able to compute the CPU time used by the server thread between two points in time by adding the time spans during which the server thread has been active.

Because the client can be preempted by the server when the server is processing other requests, the monitor must only consider the most recent context switch from the client to the server before the parsing of the client's character string is begun. Only this context switch is related to the IPC message of the client. The client cannot be activated in between because it is blocked by the IPC call.

It is easy for the user to add the three required sensors because of the small interface for sensor invocations. Only three statements must be added for each invocation: a call to the sensor's `write` function, a call to a function that computes and stores the

CPU time used by the thread (provided by GRTMon), and the final call to the `commit` function. No functional side effects have to be considered.

Implementing the Monitor To calculate the time spans to be measured based on the sensors' output, the evaluation algorithm only has to follow the callpath of the function. Consequently, the user just has to implement a small state machine (remember that GRTMon's monitors execute the evaluations associated with output elements in the chronological order of the elements).

Each output element produced by the first sensor in the client starts a new measurement. For each sensor, the user has to implement a small update function that advances the state shared by the update functions. Thus, the shared state and the update functions form a state machine. The data contained in the trace buffer are processed in the same way because it is treated as a custom sensor by the monitoring framework. The update functions insert successfully measured time spans into histograms. GRTMon's visualization library can be used to display the data contained in the histograms.

The user only has to implement the evaluation algorithm, the remaining functionality required for a monitor is provided by GRTMon's monitoring framework. This single implementation of the evaluation algorithm can be executed in a real-time, non–real-time, or remote context.

Results Running the example yields the following results (the range containing the majority of measured values is given): For each function call the client uses 5,000 to 8,000 cycles, the server needs 4,500 to 6,500 cycles for the handling of IPC messages (receive and send parts totalized), 8,000 to 12,000 cycles are required for the parsing of the character string specifying the function to be called, and the called `probemode` function uses 600 to 1,300 cycles.

The required amount of CPU time depends on the overall system's load. If, for example, a large DOpE window is moved during the measurements, the values are significantly larger (50% to 100% depending on the time span). Moving the window involves copying large volumes of data, which probably increases the rate of cache misses. Additionally, thread times are influenced by the amount of context switches from and to the thread because a portion of the CPU cycles required for a switch is added to the thread time. I have not investigated this in detail.

This small monitor already shows that a significant part of the overhead associated with DOpE's functions is caused by the parsing of the character string.

More importantly, this example shows that utilizing monitoring tools can increase the user's knowledge about the monitored system. GRTMon's monitoring framework and the simple sensor invocation interface keep the costs of monitoring small: In the example, only sensors had to be added to the monitored system and the evaluation algorithm had to be implemented.

The example is rather simple, and so was the solution. Nevertheless, it can be easily extended, for example by adding more sensors to the system and extending the evaluation algorithm accordingly. Thus, GRTMon realizes the requirements for ease of use given in Section 3.5 (at least in the case where properties are determined, the

guaranteed evaluation of output elements is not covered by the example presented here).

Furthermore, the general requirements for a monitoring tool shown in Figure 2 on page 2 are realized by GRTMon as well: Information from different parts of the system (in our example, from different applications and from the operating-system kernel) can be combined easily without making the monitored system dependent on the monitor.

5.2 Performance Overhead of Sensor Invocations

The performance overhead associated with each invocation of a sensor is an important quantity because it gives an indication for the possibility of probe effects, whereas the kind and extent of the resulting probe effects depends on the sensitivity of the monitored system to such overheads and on whether performance aspects are part of the monitored system or at least influence it (see Section 3.1).

In most cases, an invocation of a sensor only results in the modification of the sensor's output buffer and the sensor's control structure because in practice, the sensor-specific data of an output element will mostly be produced by copying data. Thus, the overhead caused by the invocation of a sensor can be roughly characterized by the amount of CPU cycles required for the invocation and by the effects caused by accesses to memory (e. g., influence on caches).[30]

To be able to evaluate the performance overhead, I have measured the CPU cycles required for the invocation of a few typical sensors, have shortly examined the relation to caches, and have added sensor invocations to a ping–pong benchmark. I will present the results in the next paragraphs.

CPU Time For the measurement of CPU overhead, a microbenchmark is used that lets each measured sensor produce 1000 output elements. The production of an element consists of a call to the sensor's `write` function, copying of dummy data to the output element to be produced, and the call to the `commit` function. Elements are produced in a loop with no additional load.

Sensor-specific data	Elements in the output buffer	CPU cycles per element
no data	20	28
4 Bytes	20	32
8 Bytes	20	35
8 Bytes	20000	59
64 Bytes	20	257

Table 1: Overhead of simple sensors

Table 1 shows the average number of CPU cycles required for producing a single output element using simple sensors.

[30]Invoking a sensor does not, for example, trigger any inter-process communication, which would lead to modifications of the state of the kernel. The CPU instructions required to produce an output element without any sensor-specific data consist of accesses to memory, integer arithmetic, conditional jumps, and the `rdtsc` instruction. Every page of a sensor's output buffer is accessed during the registration of the sensor and cannot be swapped out, so page faults will not occur.

The minimal overhead is shown by the first sensor whose output elements do not contain sensor-specific data (as a value for comparison, sensors that update the element counter using a compare–and–exchange instruction require at least 55 cycles per element). If output elements contain only a small payload (second and third row), the overhead is just slightly higher. It increases significantly if a larger amount of sensor-specific data has to be copied (fifth row).

The results of the measurements show that the implementation is efficient. However, the probability of cache misses is small because of the small size of the sensor's output buffer. As a comparison, the invocation of a sensor with a large buffer (fourth row) will require almost twice as much CPU cycles.

Sensor-specific data	Elements in the output buffer	CPU cycles per element
no data	150	252
no data	20000	269
8 Bytes	150	258

Table 2: Overhead of concurrently invocable sensors

The CPU cycles required for concurrently invocable sensors are shown in Table 2. The overhead is significantly higher, so concurrently invocable sensors should only be used if necessary (e. g., if the invocation of sensors must be concurrent and nonblocking or if using separate sensors for each thread is not possible because of the larger amount of memory required in this case).

Nevertheless, the size of the overhead is still satisfying given the additional complexity caused by concurrent invocations (e. g., the need for a dequeue operation or the additional index array required to decouple the enqueue operation and the copying of data to the output element).

Caches Caches have a high influence on the overhead of sensor invocations. Furthermore, memory accesses by sensors can result in cache lines being replaced, too.

Experiments showed that the first invocation of a simple sensor with no sensor-specific data in the output elements takes between 400 and 600 CPU cycles, whereas the second invocation following almost immediately takes only 50 to 70 CPU cycles. Thus, cache misses can cause an overhead that is ten times higher (or even 20 times higher when compared to the results of the microbenchmark in Table 1).

However, all code executed in the monitored system is affected by this problem. Invocations of sensors make the problem worse because they access memory that is usually not accessed by the other parts of the monitored system—but sensors are not the sole cause of this problem. Any other context switch can lead to a much higher probability of cache misses.

The invocation of a sensor results in accesses to at least three different memory addresses, which roughly corresponds to the number of cache lines used. All types of sensors access their control structure, the element counter located at the beginning of the output buffer, and the output element to be produced. Concurrently invocable sensors additionally access the index array. Further memory accesses are required if

sensor-specific data is copied to output elements. In most cases, the number of cache lines affected by the invocation of a sensor cannot be reduced.[31]

Asynchronous communication between sensors and monitors always requires storing information in a buffer. However, synchronous communication would have an even higher influence on caches because the monitored system would have to be preempted and arbitrary monitoring code would have to be executed.

Ping–Pong Benchmark I have added sensor invocations to a ping–pong benchmark[32] so as to examine their overhead in a known environment. Additionally, I will use this benchmark to compare GRTMon's sensors to Fiasco's trace buffer.

In the benchmark, one of the two threads invokes the sensor once before calling the other thread (both threads belong to the same task). The CPU time used by the thread invoking the sensor is measured. Because the sensors produce elements at a high rate, the sensors' output buffers are large (100,000 elements in the case of a simple sensor, 2^{16} elements otherwise). There are 4 Bytes of sensor-specific data in each output element, which are set during each invocation.

Invoked Sensor	without trace buffer	with trace buffer
No sensor invocation	504	610
Simple sensor	547	656
Concurrently invocable sensor	778	890

Table 3: Overhead of sensor invocations in a ping–pong benchmark

Table 3 shows the CPU time required for a single IPC call and one invocation of the respective sensor. The first column contains measurements without context switch logging and the second column shows the measurements taken while context switches were logged to the trace buffer.

The overhead of sensor invocations is not higher than in the microbenchmark (Table 1 and Table 2),[33] even if context switch logging is enabled.

Logging a context switch has an overhead of approximately 50 cycles (two context switches are logged for each IPC call), which is similar to the overhead associated with GRTMon's simple sensors.[34]

Adding sensor invocations to IPC calls would thus result in 10% or 50% overhead depending on the type of sensor being used. However, an absolute overhead of approximately 50 cycles is probably not significant in applications that perform IPC calls at a lower rate than in the benchmark presented here.

[31] For example, placing the sensor's control structure in the output buffer can result in element counter and control structure sharing a cache line. However, the output buffer is allocated by the sensor directory and mapped by the sensor task to a previously unknown address. As a result, the address would first have to be read from another variable, which probably requires another cache line.

[32] This benchmark consists of two threads that send IPC messages back and forth. It is used to measure the performance overhead of IPC operations.

[33] The simple sensor has a larger buffer and the overhead of 43 CPU cycles is between the values in the first and fourth row of Table 1.

[34] Logging a context switch requires writing several bytes of data, so the minimal overhead of a trace buffer entry will be slightly smaller.

5.3 Performance Overhead of Evaluations

The resource usage of monitors is not as important as the resource usage of sensors because monitors are not a part of the monitored system. However, monitors will run on the same hardware as the monitored system, so their resource usage increases the overall system load and limits the amount of sensor output that can be evaluated. To get an indication for a monitor's resource usage, I have measured the CPU time required for the evaluation of a single output element.

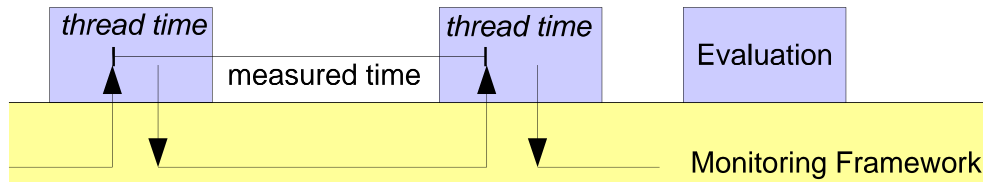

Figure 17: Measured performance overhead of evaluations

Figure 17 illustrates the measurement. A monitor executed by a best-effort thread is used to evaluate output elements of a single sensor. The sensor's output buffer is initially filled with elements so that the monitor does not need to wait for the arrival of new elements. The monitor stops after it has processed the final element in the buffer.

The used evaluation determines the CPU time that has been used so far by the monitoring thread and inserts the difference to the previous time into a histogram. The data contained in the output element is not evaluated.

Because the monitor only evaluates a single sensor's output, the temporal difference between two evaluations is equal to the CPU time spent in the framework for a single output element (see Figure 17, arrows represent the control flow).

Results The sensor output buffer used for the measurement contained 20,000 output elements. 98% of the measured CPU time differences were within the range of 380 to 420 CPU cycles.

A performance overhead of approximately 400 CPU cycles per evaluated output element is promising, especially because GRTMon's monitoring framework has not yet been optimized for speed.

However, only a single sensor is used for the benchmark, so the performance overhead will be slightly higher if more sensors are being accessed because of the required sorting of output elements. More expensive evaluations will lead to a higher footprint as well because of the higher possibility of cache misses.

The evaluation algorithm executed during the measurement is typical for a large class of performance-related evaluations: It computes the difference between two values and inserts the difference into a histogram. Obtaining the CPU time used by a thread is not an inexpensive operation either and should be at least as expensive as copying the CPU time from an output element (sensors used for performance measurements often produce this kind of information). Consequently, the measured average of 400 cycles per output element does not only consist of the framework's overhead but includes a small yet typical evaluation algorithm.

6 Future Work

Although adding sensors to monitored systems is simple, it is a tedious task for users. Instrumentation techniques can make this task much easier, which in turn decreases the costs of monitoring. Both aspect-oriented techniques [11] and dynamic instrumentation of systems during runtime (e. g., the methods used by DTrace [3] or DProbes [14]) should be evaluated and integrated into GRTMon.

Although GRTMon contains a useful library for visualization of evaluation results, there are much more powerful tools available for the post-processing and presentation of tracing data, for example [2] or the various tools used in the high-performance computing area. GRTMon's support for remote execution of evaluations can be used as a bridge from monitored systems to the environments required by these tools. Adapters (i. e., evaluation algorithms converting sensor output into the format required by the tools) should be written for a set of to be selected tools.

The monitoring framework should be further improved. First, evaluation contexts are currently just pointers to shared state. However, they should rather be treated as instances of models of the state of the monitored system. Correspondingly, libraries with basic building blocks for such models should be designed and implemented. For example, it should be possible to create a state machine just by declaring the states and transitions of the machine.

The framework should be able to deal with several clocks so as to support distributed systems and systems with more than one CPU.

Currently, caches have a large influence on the performance overhead of sensor invocations. It should be investigated if the average rate of cache misses and TLB misses can be reduced by changing the layout of the data structures accessed during sensor invocations. For example, several sensors could share memory pages or element counters of several sensors could be placed next to each other.

Choosing the parameters for the jitter-constrained streams used to bound the production rate of sensors is often difficult. Because servers operate based on the requests of clients, the production rate of sensors is usually related to the rate of requests. Clients can donate resources like CPU time to servers but a donation mechanism cannot be easily applied to production rates of sensors because of the limited sizes of sensor output buffers. Additionally, a client increasing the production rate will lead to an increased resource usage by monitors. This problem must be further investigated.

Currently, DROPS lacks a proper testing framework. Building such a framework based on monitoring and instrumentation techniques would have the following advantages:

- The majority of the available testing tools are limited to functional properties. Contrary to that, monitoring can be used to measure and check nonfunctional properties, for example response times. Therefore, functional test and benchmarks can be performed by one tool, in one environment, and with one set of test cases.

- Bugs and performance problems found by using monitoring during the construction of a system can be easily transformed into tests. Similarly, tests can be reused as constraint checks during the system's runtime.

- GRTMon's monitors can combine and evaluate information from userland processes and from the operating-system kernel. The interaction of a system with its environment is important information for black-box tests.

7 Conclusion

In the introduction, I argued that runtime monitoring tools should support both the determination and the verification of properties of the monitored system. I explained that the basic data-flow–oriented monitoring problem consists just of observing properties and evaluating these observations but that the problem that has to be solved by a monitoring tool is more complex. In particular, a monitoring tool must ensure a small probe effect, must be able to guarantee evaluation of all observations obtained after a certain point in time, must separate observations and evaluations with respect to some aspects, and must connect them with respect to other aspects. Furthermore, it must be ease to use so that the user just has to deal with the basic problem—observations and evaluations.

My goal was to construct a monitoring tool that could be used in DROPS. Therefore, the tool, which I called GRTMon, should provide solutions for all the preceding subproblems and it must run on DROPS, a multi-user system consisting of real-time and non–real-time parts.

In Section 3, I presented an architecture for monitoring tools in which the monitoring tool is split into sensors that are located in the monitored system and monitors contained in user processes. Sensors observe properties and monitors evaluate observations. The communication between sensors and monitors is asynchronous, uni-directional, and based on shared memory.

This architecture allows for separating sensors and monitors in terms of resource usage (sensors do not need to allocate communication buffers), security domains (sensors do not need to provide any interfaces and monitors cannot write to communication buffers), and temporal constraints (asynchronous communication). On the other hand, sensors and monitors in different runtime environments can be easily connected because sharing memory is possible in most environments. For example, I adapted DROPS' kernel-level trace buffer to be accessible like any other userland sensor.

Sensors and monitors are independent from each other and have only a few weak dependencies to other parts of DROPS, although monitors of course have to evaluate sensor output. Sensors and monitors are part of an n:m relation, which is a requirement in a multi-user system like DROPS.

GRTMon is not limited to a certain set of observations or evaluations because sensors can produce arbitrary output and evaluation algorithms are not restricted in any way. Monitors can always detect inconsistent data before these data are completely evaluated. At all times, monitors know exactly how much sensor output they have missed, which allows them to either stop evaluations or give precise statistical results.

I explained in Section 3.1 why asynchronous evaluation of sensor output and asynchronous communication between sensors and monitors are required to keep probe effects small when monitoring real-time systems. I implemented efficient wait-free simple sensors and lock-free concurrently invocable sensors. However, most operations performed during sensor invocations access memory and are thus influenced by caches, which results in a significant difference between worst-case and best-case execution time of sensor invocations.

I extended the previous producer–consumer model of jitter-constrained streams by an asynchronous variant in which the producer is modeled by a jitter-constrained stream and the consumer is a periodic task whose jobs have varying release times. I developed equations with which required buffer sizes and the amount of sensor output that the monitor must process in each of its periods can be calculated. The user can choose a suitable maximum latency for evaluations, which in turn determines how fast sensor output burst are worked off. The developed equations can also be used if the monitor processes output of more than one sensor.

GRTMon's communication mechanism supports the determination of worst-case execution times at the monitor side because sensor output can be accessed randomly and the monitor can determine—in a bounded number of steps—whether it read inconsistent data.

I have developed a monitoring framework, which handles everything that is not related to the basic data-flow–oriented monitoring problem, so that the user only has care about the essential tasks—adding sensors to the monitored system and implementing evaluation algorithms.

The interface for the invocation of sensors is simple because of the simple communication mechanism between sensors and monitors. Similarly, the interface that evaluation algorithms have to implement is small as well. An implementation of such an algorithm can be executed by real-time, non–real-time or remote monitors without having to change the implementation.

The evaluation of sensor output in chronological order makes the implementation of evaluation algorithms easy because the algorithm just has to follow the progress of the monitored system.

GRTMon's support for remote evaluation can be used to build adapters to other visualization or evaluation tools.

All of the preceding features are important for a monitoring tool. One can argue that some of them are not required in certain monitoring scenarios. Often such criticism would be justified but it only applies to a specific case.

DROPS shows why a generalized approach to runtime monitoring is useful. DROPS' components often consist of real-time and non–real-time parts, so a monitoring tool limited to pure real-time systems could not be used. Both the determination and verification of properties of the monitored system are important. If two different tools would be required for these two tasks, then there would be a gap in the development process and development costs would be higher because, for example, evaluations could not be reused easily.

DROPS tries to integrate these different environments. Therefore, a monitoring tool has to do the same.

References

[1] James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions of Computer Systems*, 15(2):134–165, May 1997.

[2] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004.

[3] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the USENIX 2004 Annual Technical Conference*, pages 15–28, 2004.

[4] Sarah E. Chodrow and Mohamed G. Gouda. Implementation of the Sentry System. *Software—Practice and Experience*, 25(4), April 1995.

[5] Norman Feske and Hermann Härtig. Demonstration of DOpE — a Window Server for Real-Time and Embedded Systems. In *24th IEEE Real-Time Systems Symposium (RTSS)*, pages 74–77, Cancun, Mexico, December 2003.

[6] Cl.-J. Hamann. On the quantitative specification of jitter constrained periodic streams. In *MASCOTS*, Haifa, Israel, January 1997.

[7] Claude-J. Hamann. Schwankungsbeschränkte Ströme. Technical report, TU Dresden. In German.

[8] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[9] Monjoo Kim, Insup Lee, Usa Sammapun, Jangwoo Shin, and Oleg Sokolsky. Monitoring, Checking, and Steering of Real-Time Systems. In *2nd Int. Workshop Run-Time Verification*, 2002.

[10] Jork Löser, Lars Reuther, and Hermann Härtig. A streaming interface for real-time interprocess communication. Technical Report TUD-FI01-09-August-2001, TU Dresden, August 2001. Available from URL: `http://os.inf.tu-dresden.de/~jork/dsi_tech_200108.ps`.

[11] Daniel Mahrenholz, Olaf Spinczyk, and Wolgang Schröder-Preikschat. Program Instrumentation for Debugging and Monitoring with AspectC++. In *ISORC '02: Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 249, Washington, DC, USA, 2002. IEEE Computer Society.

[12] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual*

ACM Symposium on Principles of Distributed Computing (PODC '96), pages 267–275, New York, USA, May 1996. ACM.

[13] Aloysius K. Mok and Guangtian Liu. Efficient Run-time Monitoring Of Timing Constraints. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, pages 252–262, 1997.

[14] Richard J. Moore. A Universal Dynamic Trace for Linux and Other Operating Systems. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 297–308, Berkeley, CA, USA, 2001. USENIX Association.

[15] Henrik Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. PhD thesis, May 2000.

[16] Andreas Weigand. Tracing unter L4/Fiasco. Term paper, TU Dresden, April 2003. In German.

[17] Robert W. Wisniewski and Bryan Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *SC2003 Conference CD*, November 2003.

[18] Karim Yaghmour and Michel R. Dagenais. Measuring and characterizing system behaviour using kernel-level event logging. In *Proceedings of the USENIX 2000 Annual Technical Conference*, 2000.