# Probe Framework - A Generic Approach for System Instrumentation

Markku Pollari
Technical Research Centre of Finland, VTT
Oulu, Finland
Email: markku.pollari@vtt.fi

Teemu Kanstrén*
Technical Research Centre of Finland, VTT
Oulu, Finland
Email: teemu.kanstren@vtt.fi

***Abstract*—This paper introduces a general framework directed for system instrumentation. The introduced framework provides support for a system instrumentation approach that enables designing information capture, monitoring and analysis features into a software-intensive system. We describe the general concept, architecture and implementation of the framework and two case studies in its application. As a prototyping platform, we dealt with collecting information from Linux systems by probes created with the building blocks and interfaces provided by the framework. We also discuss the effects of building support for the framework in an implementation from the viewpoint of different constraints, especially focusing on real-time embedded systems, where especially strict constraints are present. Overall, we demonstrate the feasibility of a more uniform instrumentation approach through this concept and its application in two case studies.**

***Index Terms*—system instrumentation; monitoring; analysis; testing.**

## I. Introduction

This paper extends the work presented in [1], by offering a broader view on the background concepts, and the implementation of the framework and a more detailed account on the experiences and lessons learned during the work.

Understanding and analysing the behaviour of complex, software-intensive systems is important in many phases of their life cycle, including testing, debugging, diagnosis and optimization. In addition to these, many systems themselves are built for the sole purpose of monitoring their environmental data and reacting to relevant changes, such as detecting patterns in internet traffic or adapting to available resources. All these activities require the ability to collect information from the different parts of the system.

These basic activities and requirements in software engineering have existed since the first days of writing software. However, despite this there has been little research and activity to build support for systematic monitoring and information capture into software platforms. Instead, what is most common is the use of ad-hoc solutions to capture data where needed, as needed. In these cases, the instrumentation required to capture the information is added momentarily into the system and removed after the short-term need has passed. Recent studies have still emphasized this problem, describing large-scale systems where these types of features are important but support for them is lacking [2].

In this paper we present a design concept, and its implementation and validation, for a platform to support the systematic capture and analysis of information related to the behaviour of a system and its environment. This platform is termed as the Probe Framework (PF). The PF provides support for building monitoring functionality for collecting information on the behaviour of software intensive systems and using this information for purposes such as built-in features in the software itself (as product features) and testing analysis of the systems during their development (testing and debugging) and deployment (diagnostics). The prototype implementation of PF is available as open source[3]. We also discuss the effects of building support for this type of a framework into a system from the point of view of the system constraints on available resources and effects on its behaviour and performance, especially focusing on real-time embedded systems, where especially strict constraints are present.

This paper is structured as follows. Section 2 discusses the background and motivation for the work. Section 3 describes the main concepts of the PF at a higher level. Section 4 covers the implementation of the PF and Section 5 presents two case studies of utilizing the PF and experiences gained from this work. Section 6 discusses the related work, and finally the conclusions in Section 7 end the paper.

## II. Background and Motivation

The concept of capturing information from a system and its environment is often described as tracing the system. Similarly, in this paper we use the term tracing to describe the activity of capturing information from a running system, either with external monitoring or internal instrumentation features. The data captured is described as a trace of program execution. Many different domains make use of tracing information, such are: system security analysis, internet monitoring and protection [4], run-time adaptation and diagnosis[5], optimization [6], testing and debugging [5], [7], [8], [9].

There are various tools around for specific instrumentation and tracing on different platforms, such as DTrace for Solaris [10] and OSX [11], Linux Trace Toolkit Next Generation

*Also affiliated at Delft University of Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Mekelweg 4, 2628 CD Delft, The Netherlands.

(LTTNG) [12] and SystemTap [13] on Linux. These tools all share a common goal to observe and store traces of system behaviour and resource use, such as CPU load, network traffic and filesystem activity. They are typically intended to provide a trace facility for the low-level resources and related behaviour of the system kernel, using solutions such as their own programming/scripting language to define where to exactly insert trace code into the operating system kernel [10][13]. For example, in our implementation of PF we make use of SystemTap, which allows one to add trace code into the Linux kernel without the need to recompile or reboot the running system [14].

These low-level frameworks provide an excellent basis for capturing low-level information from a system when ad-hoc instrumentation needs arise. However, while these tools are useful for many purposes, they alone are not sufficient for efficient observation of complex system behaviour. Additional useful information from this low-level information can be gained by using advanced analysis methods such as multivariate analysis to infer additional information such as relations and similar properties from the low-level data [15]. However, what is also needed is information on a higher level, including the events, messages and interactions of different parts of the system. Also, information about the environments of these parts and their relation to the lower level details are needed for a complete view of system behaviour.

This type of information is a part of the higher level design of a system, and it is implemented as higher-level abstractions inside the components. Thus, it is not possible to build generic components that would capture this information from all the components from the OS kernel or any custom application. When solutions such as component based middleware are used, it is possible to build part of this support into the middleware itself to capture the data [2]. However, for an effective and descriptive trace, application specific tracing is also needed. For this level of tracing several frameworks exist, such as Log4J [16] and syslog [17]. Additionally, when the availability of such features and information is highly valued, customized support for these have been built into the system as first-class features [2].

The above descriptions show how effective analysis of software intensive systems requires many different types of traces to be supported, collected and analysed together. Different tools need to be used effectively in different steps and finally combined as one for both built-in features and external analysis. Only in this way is it possible to provide the required support to get a definite view of the behaviour of a complex system.

From this viewpoint of complete system analysis and its support through the life cycle, the described trace tools and frameworks suffer from a set of issues. The tools use their own interfaces, custom data formats and storage mechanisms. Additionally, often the storage is only considered in the form of a local filesystem with the intention of being manually exported to external analysis tools or read as such by humans. This can be problematic in different systems and environemnts. For example, simply accessing this information from an embedded system can be very difficult as these systems are often limited in their external interfaces. Even where this is possible, in the case of a deployed system, it is not always cost-effective for someone to go to the field site to examine the trace file. Additionally, the trend for relying on ad-hoc temporary tracing solutions makes it very difficult to capture a meaningful trace of a system as there is no built-in support to be used when needed. The lack of design support for proper tracing from the beginning further brings problems such as probe effects, where addition of temporary trace mechanisms changes the timings and resource usage of the actual running system that is to be analysed [18]. The probe effect is illustrated in Figure 1, by showing how the timing of two tasks is changed by the tracing.

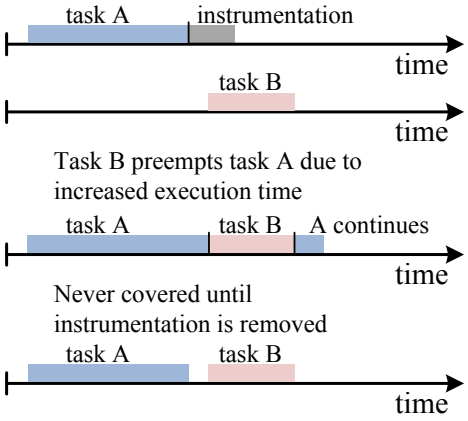


Fig. 1. Probe effect

To address these issues, to build a basis for effective system level tracing, analysis and related program functions, we have developed a trace platform called Probe Framework (PF). Our prototype implementation is created on Linux and enables the collection of trace information both from kernel and user space probes, through a single unified component in the system. By starting with the goal of building these features into the system as first-class features we make it possible to address properties such as probe effects, information access, limited resources and real-time requirements. With a commonly shared and customizable format for the collected trace, it is possible to store and export this information to different analysis tools. With unified interfaces inside the platform it is also possible to easily design built-in features that make use of information from all the various tools. As the main intent is to build a higher-level abstraction mechanism, we use existing tools such as SystemTap and integrate these to the Probe Framework. The PF and its main concepts are described in more detail in the following sections.

## III. General concept

On a higher level, the Probe Framework is a part of a larger concept, which includes three main components; trace capture, trace storage database and trace analysis. The PF provides the needed support as a platform to capture the trace information

from the system under test. An information database server is used to collect the trace information and provide the means to query, filter and export the trace to analysis tools. Various trace analysis tools can be used to analyse the information provided. This includes tools specifically for trace analysis and also tools more generally intended for analysis of data, such as multivariate analysis. For example, experiences on using a multivariate analysis tool to analyse the network functionality and behaviour of a system from information that can be captured with the help of PF have been studied in [15]. In addition to making use of the captured information in external analysis tools, it is also possible to make use of it as part of built-in product features for processes such as adaptation, testing and analysis. This overall architecture is described in Figure 2.

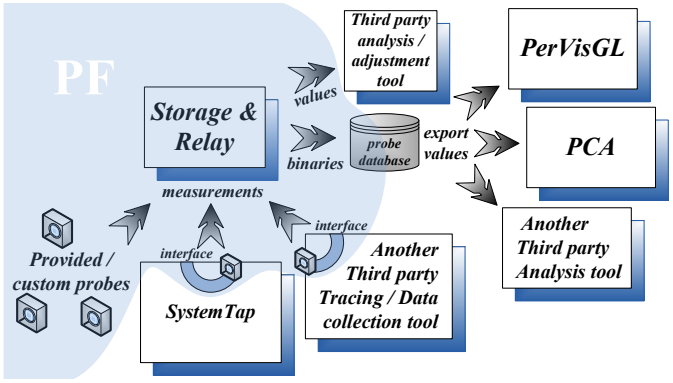


Fig. 2. High level overview of PF and external components.

The Probe Framework itself has a layered architecture as presented by Buschmann et al. [19]. The PF's architecture is divided in three main layers: basic services, monitoring services and test services. Each of these layers builds on the functionality of the layers below it, as described in Figure 3.

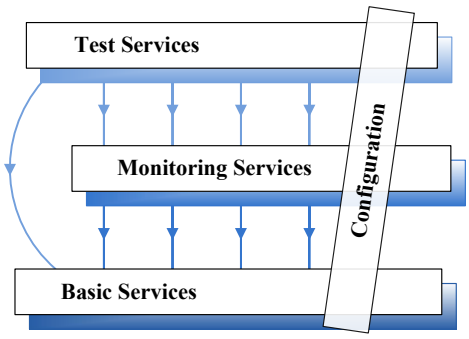


Fig. 3. PF internal architecture.

The basic services contain services deemed necessary for information handling, such as data buffering, storage and relaying to external database. The basic services are general for all the probes, and offer the support for fast implementation of the upper level services. The term probe, in the context of this paper, means the entity that is formed by utilizing the different service layers to create the functionality for collecting and handling the monitoring of some aspect of the target system. The basic services comprises of three parts; the first part is the probe interfaces, the second is the binary formatter and the third is the communication handler. These three parts are illustrated in Figure 4, the figure also describes the internal division of the basic services. It can be seen that the binary formatter and the communication handler are encapsulated as a storage and relay (module), but without going into the details here the upcoming Section IV-B offers more on this subject.

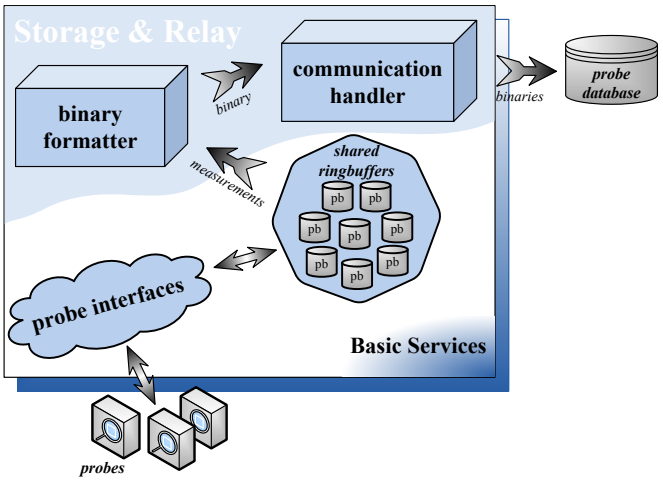


Fig. 4. Structure of basic services

Together these parts take care of all the data management of the tracing as described in Figure 2.

The monitoring services offers a set of readily provided interfaces and probes to attach to the basic services. The actual services at this layer are used to capture and monitor different values, such as memory consumption and CPU usage, and their evolution in the system. Many of these basic monitoring services are provided as ready probe components in the implementation of the PF, including CPU load, memory consumption and network traffic monitoring. Further, they provide simple interfaces for building new monitoring services on top of them without the need to concern with the complex internal details of the data management.

The top layer, test services, is the most implementation dependent and is where the system specific functionality can be build. For example, functionality can be built to inject test data into the system, use a provided set of monitors to see how the system behaves and store the test results using the basic services. Similarly, in a running system the same monitors could be used from a test service (or more accurately, built-in functionality) that adapts the system's runtime behaviour and use of components based on thresholds set for monitored values such as memory consumption, CPU load and network traffic patterns.

An important concept for this type of analysis is that of testability. Although generally associated with testing, it is also relevant in many ways to any requirement to capture

information about system execution for analysis. Testability is commonly divided to two main properties: controllability and observability [20]. Controllability is the ability to control the system execution, for example, in order to make it take a chosen path. Observability is the ability to capture information about the different properties of the system (such as events and messages). Although a monitoring framework such as PF is mainly concerned with observability, also controllability is important in order to build support for dynamic and effective tracing as well as services that make use of this trace functionality. For example, in order to insert ”proxy” style probes to capture communication messages between system components, the design must support the reconfiguration required, possibly even during runtime. Solutions to address this have been discussed in more detail in [2].

## IV. Implementation

A prototype of the PF has been implemented for embedded real-time Linux systems. This platform was chosen as it provides an interesting and realistic platform for the implementation of this type of software, with both possible issues and available options. These issues include the strict timing requirements and limited resources inherent to the embedded real-time systems. Yet, even as we are dealing with embedded software where we know all the running software beforehand it needs to be possible to access the whole platform including the kernel. With Linux as the operating system, this is particularly easy as the whole operating system is open source software.

### A. Setting for the implementation

The setting for the implementation is considered to contain the following elements: run-time monitoring, Linux environment and embedded real-time environment. The setting for the framework entails various interesting challenges. The next sections depict some of those challenges.

*1) Run-time monitoring:* The goal of run-time monitoring creates challenges because is makes the execution somewhat unpredictable. Generally, the system has limited resources and many consumers ”competing” for them. As such, any kind of extra activity in the system can have severe consequences. The extra activity refers to the act of monitoring, which in a software-focused case entails some code being run in order to capture data. The monitoring can be understood as data collection, and instrumentation is a way to realize it through software. The monitoring process in whole changes the targeted system, software monitoring always consumes resources and inflicts an overhead to the underlying system. Depending on the context this might pose a problem.

The run-time monitoring states that monitoring, tests, data collection, etc., are conducted while the target system performs its appointed functionality. In other words, everything happens while the target system is executing normally. However, the concept of ”normally” can be bent a little as certain liberties do affect cases where some specific property is being monitored. For those cases, it is only viable to focus on the necessary items while the rest can be ignored. In practice: a specific part of the target, in this case it could be a sub-program of some sort, is isolated and only it is run in the system, while the rest is substituted with ”stubs”. Still, the aim of run-time monitoring is to stay true to the actual real-life end deployment, and limit the overhead of monitoring, or at least control how the overhead occurs in the target. Because of this, the control aspect of testability is important. For a highly controllable system, the choices of instrumentation leave room to better manage the overhead in the context of the services built to make use of the provided tracing possibilities, as well as to configure more dynamically the used trace services.

The importance of run-time monitoring is diverse; the implementation is seen in its natural environment and several aspects of it can be measured and analyzed. Testing can be done against the ”real thing”, therefore, unexpected issues have a possibility to surface. Wegener et al. [21] mention a few important issues that can be obtained through run-time monitoring. These are the dynamic aspects such as the duration of computations, the actual memory usages of the program during execution, and the synchronization of parallel processes. These aspects are of special interest for real-time systems as resource allocations might be hard to determine beforehand, and estimates for resource consumptions are only suggestive, and therefore need to be verified. As such, run-time monitoring also provides support for a wider part of the system lifecycle, including in-field diagnosis, where the only option available is to use the available run-time services or otherwise lose the information of interest (such as failure state and its cause in deployed systems). Where information is needed for a run-time adaption of system behaviour, run-time monitoring is again the only possible solution.

*2) Linux environment:* The Linux environment indicates that the system has either a limited (customized & embedded) or a full Linux operating system (OS), with support for network access, filesystem, scheduling, etc.

The Linux OS is open source, and so is the Linux kernel, with several distributions readily available depending on what flavour one likes. The distributions aside, the core of Linux is the monolithic kernel. This core is quite versatile and it is autonomous from the rest of the OS. The monolithic means that the entire kernel is run in a privileged kernel space in a hypervisor mode, as opposed to user space that has relatively restricted execution possibilities. In a nutshell the difference is that a program in supervisor (kernel) mode is trusted never to fail, and this is not questioned or accumulated for, where as in the user space the failure is an option and there are measures for managing those failures without a total system crash. Because of this, a general guideline for when operating in Linux is to perform actions in the user space if possible, as kernel space is strictly reserved for running the kernel, kernel extensions, and some device drivers [22].

*a) Modularity:* The modular nature of the Linux kernel is worth mentioning, because it is possible to dynamically load and unload executable modules to the kernel at run-time. This modularity of the kernel is at the binary image level of the kernel. What this means is that after the code

comprising the Linux kernel is loaded from a binary image at boot up, it is possible to dynamically load modules to expand the functionality of the current image at run-time – as opposed to rebooting with a different kernel image. The modules allow easy extension of the kernel's capabilities as required. However, dynamically loadable modules incur a small overhead compared to building the module directly into the kernel image. On the other hand, dynamic loading of modules helps to keep the amount of code running in the kernel space to a minimum, useful for example to minimize the kernel's memory footprint for embedded devices with limited hardware resources. Modularity can be used to address the run-time inclusion of PF services, when there is a need to consider general resource constraints and there is need to disable external trace components used with PF for general operation, while maintaining the option to enable these during runtime.

*b) Linux kernel:* On the implementation level of the Linux kernel the used programming language is C, and as a big part of the kernel deals with device drivers and other hardware-oriented functions, the C language is a fitting choice for all this hardware oriented programming. Due to various constraints, it is generally considered that any code expected to run in kernel space has to be C code. Additionally, the different programs running in the kernel are often separate processes, and user space contains various different processes, and programs implemented with numerous programming languages and environments.

As mentioned the core of a Linux system, kernel, is not dependent on the rest of the OS, therefore it remains relatively the same between different hardware platforms. Also, what is handy about the kernel is that most of the functionality is highly abstracted. Because of these, the gap between host-target development common in embedded systems is not so wide in Linux environment. This gives the option to implement generic services to support capturing information for any system and application running on top of a Linux based system. Thus, some services for OS level information capture can be provided as generic, while providing support for application specific trace implementation.

*3) Embedded real-time environment:* Another factor in the environment is the embedded real-time aspect. Embedded real-time systems possess unique attributes, and requirements. These two attributes, embeddedness and real-timeness, contribute to the complexity of the environment.

The real-time aspect of the system dictates that there are constraints placed on the timing of the actions the system performs. This can be defined as having performance deadlines on computations and actions[23]. Knowing the state of the system has conventionally been a great help in testing the system and in verifying its behaviour. The system being real-time might cloud this knowledge, the knowledge of the internal functionality of the system for a given point in time, as the possible variations in the system's execution paths quickly approach infinity when time is taken into consideration.

The other side of the equation, embeddedness, refers to systems that interact with the real physical world, controlling some specific hardware, such as a cell phone [24].

In combination, embedded real-time systems are systems built with a specific purpose in mind. These systems specialize in fulfilling that purpose, often with the minimal possible cost. The environment where these systems operate is dynamic; computational loads are unpredictable yet responses have to be provided according to precise timing constraints [25]. The minimization of the cost often results in scarce operating resources and having just enough resources is quite common in embedded systems. The resource constraints common to embedded systems are due to limited physical space, weight and energy usage, and cost constraints. The strict constraints associated to embedded real-time, along with the environments these systems operate in, offer challenging design and implementation choices. Schulz et al. [26] list characteristics of the current generation of these systems: continuing increase in system complexity, diminishing design cycles, tightly integrated mixed hardware and software components, and the growing use of reconfigurable devices.

The features of the embedded real-time environment that are considered important in the context of the Probe Framework are resource constraints, scheduling frequencies and adaptability to data extraction, timing alterations and relaying possibilities inside the target system.

*4) Restrictions forced by the environments:* The issues common to embedded systems and real-time systems, such as low memory, limited CPU power, timing requirement, etc., are valid restrictions for the implementation. Also, the Linux environment and the run-time monitoring have their share of challenges for the implementation.

However, the biggest challenge is most likely the indeterminism brought forth by the nature of environment. The reason for this is that the execution has dynamic elements that can be estimated to some extent but not to the extent to be irrelevant. Another factor is that each embedded real-time system is unique in context, and preparing for a wide variety of embedded systems is challenging. Because of this, several generalizations and abstractions are needed for adaptability to be able to cover every system. Deciding on the generalization and abstractions, i.e., the common factors, for embedded real-time systems need a set of restriction to be meaningful. Here the Linux environment offers the needed focus point for that purpose. The possibilities offered by the Linux kernel and OS outline a set of functionality for managing the general required operations for system instrumentation. This set of functionality is the basis for our prototype implementation Probe Framework.

A big restriction concerning the run-time monitoring is the issue of a potentially huge amount of data, as the system can execute at the speeds of several megaticks per second, and keep running for a long time, it is clear that the amount of collected information can accumulate fast. When combined with the low amount of memory and limited disk space of typical embedded systems, along with the timing requirements of real-time systems, the combination requires special atten-

tion. Another restriction is the used programming language. Considering the C dependency of the kernel, it is easy to see why most of the software in Linux is done either in C or C++. Through this factor, it is natural that the choice for implementation language of the Probe Framework prototype is C, and C++ for parts where it is fitting. Also, the partition of the user/kernel space in Linux causes the need to manage each side's monitoring needs separately and as conveniently as possible.

### B. Main parts of the implementation

The focus of the prototype implementation is on the basic services layer, because the basic services are general for all the probes, and offer the support for fast implementation of the upper level services. As mentioned, the basic services comprises of three parts: the probe interfaces, the binary formatter and the communication handler, see Figure 4. Here the implementation of these three parts is covered, starting with the probe interfaces.

Although conceptually one, the actual implementation of the probe interfaces in basic service layer is divided in two. The major reason for this is the way execution in operating systems typically takes place, in either user space or kernel space, refer to Section IV-A2. This separation also acts as a divisor for the probe types, resulting in a split between kernel probes and user probes. Additionally, in Linux as well as most modern OS's each user space process runs in its own virtual memory space, and thus cannot normally access the memory of other processes nor can other processes access its memory [22]. However, for effective implementation of the PF, all the trace data for a single system needs to be centrally managed. This requires that there needs to be a single component that takes care of the data management for all the probes deployed, either in kernel or user space. This division of implementations, probes, probe interfaces and storage and relay module, is described in Figure 5.

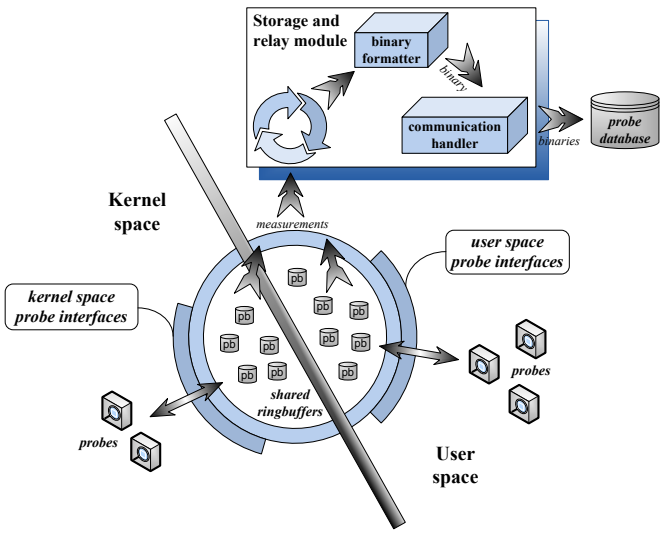


Fig. 5. Division of the implementation

The storage and relay module resides in the user space, conforming to a general guideline for operating systems [22]; perform actions in the user space if possible, as kernel space should be reserved for parts that absolutely must be there as they require special privileges. Kernel code can also crash the whole system with its privileges and thus these parts need to be absolutely secure and reliable. Since we do not need to perform actions with special privileges it makes sense to locate most of the code in the user space. This is also one of the main reasons for why the shared memory regions are used between kernel space and user space, and also inside user space, see Section IV-C5 for details on the shared memory utilization. This was all in order to separate the trace handling functionality, see Section IV-C4 for trace handling information, from the probes and to centralize the trace collected by the probes. This enables the storage and relay module to access the trace, format it and provide it for higher layer functionality or simply relay it to the end storage as requested. All this reduces the interference induced to the target by the monitoring activity conducted by the probe as all the "extra" processing can be done separate to the probe in its own process. Another benefit for having the storage and relay module, i.e., the basic services, in its own process in user space is that it enables easier configuration of the provided services.

The binary formatter part of the said module is the simplest part of the component; it is as the name suggests a formatter used in changing the collected data to a more manageable form. The reason for the use of binary format is to provide an effective, single format to share the data between different tools, layers and databases, refer to Section IV-C6 for detail on the binary format. The intent is to support probes created in different programming languages, running on different platforms and with strict constraints on memory and real-time requirements typical to embedded systems. Implementing this effectively is not trivial; however, the user is completely shielded from the details by the provided abstraction interfaces. The communication handler is the second part of the storage and relay module. This part handles the data transfer to end storage locations, takes care of the configuration of the storage and relay module, see Section IV-C1 for the configuration, and manages the data extraction from the probe buffers.

In practice the basic services are implemented as a shared library component, meaning that the implementation code needs only a single instance (code segment) to reside in the memory during runtime. This makes synchronizing all the trace data for a system overall much simpler due to only having a single instance of basic services for a system at any time. The library is implemented as a dynamically linked library, which is linked to the components during execution, meaning it is shared also between different processes in the user space. For the kernel space there is a similar component.

The following sections shed more light on the internal functionality of the Probe Framework.

### C. Properties of the implementation

The most important properties of the Probe Framework are covered here. The included sections deal with the Probe Framework's configuration possibilities, instrumentation possibilities and the way the data collected by the probes is managed in the framework. Also, the message forming and the used binary protocol is covered here.

*1) Configuration of the implementation:* In order to cope with a variety of different devices, the configuration possibilities of the Probe Framework are substantial. Each probe can be configured separately, as can be the storage and relay module. Various possibilities for accumulating for the different capabilities of the target system are offered by the Probe Framework. The output possibilities and replacement strategies, etc., used by the storage and relay module are all configurable to suit the system's capabilities.

The major control features of the Probe Framework reside inside a configuration file that is read during the activation of the storage and relay module. This allows configuring the basic parameters such as overall buffers, general policies and storage mechanisms externally. Another layer of control is embedded in the creation of the probes, during the implementation of a probe the creator can use custom settings to define probe specific values or leave them out in which case the default generic values will be used. The probe specific attributes include buffer size, preferred storage location, priority, timing accuracy and presumed output type. Additionally, the creation of output types used by the probe introduces control as it is possible to use prioritized data types for increasing the probability that the collected trace reaches its storage location. In order to address restrictions such as keeping the monitoring overhead low, several policy parameters can be defined. One is the possibility of discarding parts of the collected trace if the basic services cannot run fast enough to relay it to a storage destination. This is further influenced by the priority set to the trace through the configuration. More advanced policies can be implemented inside custom probes, such as sampling or time-interval captures.

The most important part in configuration is when, how and on what condition the data is collected by the probe. This is left open on purpose so that the implementor can choose the most suitable collecting method. Choices need to be made about the frequency of the data collection and preconditioning of the collection events. As an example, the collection could be sampling-based so that every Nth data value is collected. Similarly, some exceptions, values larger or differing from the expected, could work as a trigger for the collection. The choice of how to collect the data and the implementation of this selection is hence forth referred to as the front end of the probe.

*a) Configuration file:* The main features of our PF implementation configuration are controlled by a configuration file. It allows one to define the storage locations for the collected data and the relative importances between the storage options when multiple options are activated.

The available storage types are memory, file, folder, TCP and UDP. Depending on the type, additional parameters are given to define, for instance, the ip address of a trace server for the TCP and UDP options, or the path for the storage file. The trace server for TCP and UDP in this context refer to (an external) server component that stores the provided data into the probe database mentioned earlier. Continuing on the main types, two of them are quite similar mass storage types, namely the file and folder. The difference between these two are the used replacement strategies; more on these strategies in next paragraph. Another important part of the configuration is the capability to define if a given location is dynamically or statically allocated. There are two options for each storage type. First is the relative priority of each instance compared to the other storage type instances. The second is the allocation strategy, which can be either static or dynamic. The static allocation implies that resources for utilizing that particular location are reserved immediately when the storage and relay module is instantiated. For example, the memory type might specify that 2 MB of main memory can be utilized for storage, which would be immediately allocated and made writeable. In the case of dynamic allocation, the memory would be allocated as required as more data is written.

In the case of the Linux implementation, a notable property is that the Linux filesystem does not discriminate between mass storage devices, or physical location for that matter. Because of this, it is even possible to use a nonlocal file or folder as a storage destination, given that the location is mounted to the local filesystem. This way, it is possible to use external hard drives, networked hard drives, usb-sticks, mmc cards and even memory-mapped files for storage locations, as long as it is possible to refer to it trough a path in the main configuration file. This offers a lot more versatility to configuration.

*b) Replacement strategies:* The replacement strategy for the used storage is defined in the configuartion file. Depending on the type of the storage, there are several strategies available. For the TCP and UDP types, the strategies are not used because they typically refer to an external trace server and controlling an external entity is not in the scope of a PF functionality embedded inside a system implementation. Buffering strategies would be meaningful but they are different from replacement strategies and we have currently focused on strategies meaningful to embedded functionality. The strategies are therefor relevant to the memory, file and folder types.

For the memory type, the replacement options are *stop*, *restart*, *priority* and *wrapping*. *Stop* means that when the amount of memory allocated has been filled with data, the memory is no longer available as a storage location. With *restart*, when the allocated memory is full, all the previous data is discarded to make room for the new trace. Using *priority*, the data with the lowest priority are removed first to make room when the allocated memory is full. Each data element is stored with a user defined priority as will be described in following sections. The last case, *wrapping*, treats the storage as a circular buffer with varying slot sizes. The next suitable

slot is chosen from the buffer for writing.

For the file and folder types, the basic replacement options are *stop* and restart. Folder has additionally *singleflip* and *multipleflipN* (where $N \in \{\mathbb{Z}^+ \cap \{1,2\}\}$). These two new options are variations of the restart replacement strategy. Here N denotes the number of used files. These flips utilize N number of files to divide the allocated space in N parts. The result obtained is the maximum size of a single file inside the folder. When the current file where the trace is stored to reaches this limit, another file is selected and used as storage for following data. After the total allocated space runs out, the first file is removed and a new empty one takes its place similar to the restart case. The benefit of the flips is that it is possible to save more of the already traced data. For instance, if a simple restart is used, all previous trace is lost when space runs out; but with flips, the trace persists longer in the storage. Therefore, when the monitoring ends, provided that enough trace was collected, the folder will contain N-1 files and have at least

$$\frac{(N-1) * total\ allocated\ space}{N} \quad ,(where\ N \geq 2)$$

bytes of traced data stored.

Lastly before moving on, a short note about utilizing several instances of the storage and relay module. The Probe Framework's architecture places no limitations on this. It is possible to have several of them, each with a specific configuration. This way, the probes can choose the storage and relay module instance to use; the id of the major shared memory region the probe registered to during creation dictates the division. This results in a higher level configuration need that is no longer on the scope of the implemented framework. For the interested, Linux offers several ways for achieving this type of control. One method is called resource groups [27]. It could be used in encapsulating and running the storage and relay modules. This way the modules could be differentiated and priority, resource usage, etc., could be set for each allowing for a more managed solution.

*2) Instrumentation possibilities:* As described earlier, instrumentation is divided into two main types of probes: kernel and user space probes. A distinction can also be made between internal and external instrumentation. Internal refers to embedding the instrumentation code to the software object that is part of the monitored system. In this case the probe is an integral part of the program code. External instrumentation refers to the probes where no modifications are made to the system software itself. Instead a stand-alone process handles the monitoring from outside of the target software. The PF provides support for all these different types of instrumentation. Custom kernel and user space probes and built-in functionality can be created using the services provided at the different layers of the PF. A set of external instrumentation components are provided as kernel probes and processes to collect and analyse generic properties such as task-switches, CPU load and network traffic. More such custom components can also be easily created. All these instrumentation possibilities share the set of basic services that remain unchanged between different implementation possibilities. Therefore, it is simple to analyse the collected trace data, build additional functionality or make other use of the instrumentation data from all different probes and monitoring tools through the provided interfaces.

In order not to limit the instrumentation possibilities, the choice of what to collect, how to collect it and a part of the implementation of this functionality is left open by the Probe Framework. What the Probe Framework provides for implementing the instrumentation, are the basic services that remain unchanged between different implementation possibilities. Therefore, after the choice of what to collect from the SUT and how or if to use sampling or conditioning on the collection, the Probe Framework's basic services are used by passing the trace to it. In short, what is needed is a simple front end, and then the framework's services can handle the rest. Of course, this is overly simplified of what happens on the detailed level, such as calling the probe interfaces and setting the configuration parameters, etc., Still, it gives the bigger picture of creating the probes.

*a) Instrumentation types:* The types of probes can be divided to kernel- and user-space probes. These include the possibility to use third-party frameworks to capture the trace and link these to the basic services in order to provide a uniform trace over different parts of the system. One such framework for kernel space is SystemTap [14]. SystemTap enables one to add probe code in nearly any location of the Linux kernel, without the need to modify the kernel source code. This is done by code injection during runtime execution, meaning that it can be done while the kernel is running. Use of SystemTap is discussed in more detail in Section IV-C3.

A second possibility is to modify the system source code by adding the needed instrumentation code directly to specific locations. However, the downside is that this method requires that the software is recompiled.

For external monitoring in the case of the kernel is is possible to create a kernel module dedicated solely to monitoring some activity. In this case, as well as the other cases, the basic services of Probe Framework provide the means of passing the collected data to the end deployment location.

These different instrumentation options are supported for both kernel-space and user-space as the services provided by the PF can be linked to any monitoring code or service.

*b) Categorizing the probes:* Section III showed the layered structure of the Probe Framework. Here the two upmost layers, the monitoring services and test services, are of interest.

The monitoring and test service layers contain what can be understood as probes. These probes are realizations of certain types of instrumentations. Depending on the implementation, probe front end, etc., they are either context dependent or more general ones. The monitoring services contain the more general probes and focus on instrumentations that collect general information from the target system. These services are reusable, and include for instance the possibility to measure the memory usage of different programs or the CPU load generated by the programs. The complexity of these services is

quite simple as they usually contain just the collection of some data values. Hence the name, general monitoring services.

The higher level test services are more context dependent, and their functionality is more complex compared to monitoring services. An example of a test service might be, for instance, the collection or generation of inputs to some part of the target, along with the collection of the resulting outputs. Some processing could also take place in the service before passing the results to the basic services for end storage. The implementations are not restricted to just testing related functionality. The service could, for instance, provide useful functionality for adjusting some quality parameters in the deployed system. Although our implementation is focused on the basic services, for different domains, more generic higher level components could also be provided.

Finally, as was presented in Figure 3, it is possible to construct probes that utilize all of the layers. This means that it is also possible, for instance, to use previously made probes to create a new more complex one. This case is mostly relevant for the test services layer, where it is possible to use the services from all the layers beneath it.

*3) Linking with external tools:* As an example of linking the PF with external monitoring tools, this section describes how we have linked it with SystemTap. SystemTap is a framework characterized to simplify the gathering of information about the running Linux system. It is also said to eliminate the need for the developer to go through the tedious and disruptive instrument, recompile, install, and reboot sequence that may be otherwise required to collect data.[14]

With SystemTap, it is possible to monitor the variables used by the kernel, even altering the variables and performing extra operations inside the placed probes is possible. For controlling this functionality, SystemTap provides a simple command line interface and a scripting language for writing instrumentations. SystemTap can be utilized in most Linux installations as the required components are commonly included in the distributions. For other system, similar framework, such as DTrace, exist as described before.

How the SystemTap works and how the Probe Framework works in conjunction with the SystemTap is depicted in Figure 6. The figure is adapted from [28], and has been modified to include the addition of the Probe Framework.

*4) Trace handling in the implementation:* This section describes the path the trace data takes through the Probe Framework. The trace data is first stored in an internal ringbuffer inside the PF. To assure correct functionality of the system, the different operations are prioritized so that write operations have a higher priority. This is due to the writes often taking place in kernel-space and reads always taking place in user-space. Since kernel-space is more critical, this is given a higher priority.

After storage in this internal buffer, the trace becomes the responsibility of the storage and relay module. This module runs inside user-space and manages the end storage locations for the trace data. It accesses the shared memory regions and uses a table to access the ringbuffer data values in correct

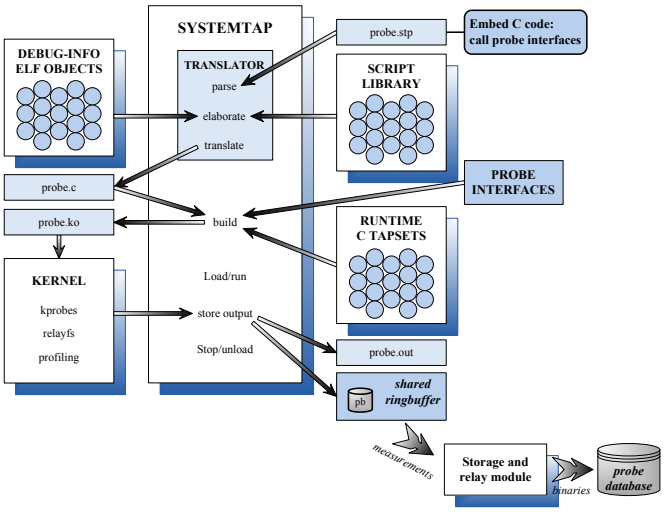


Fig. 6. Probe Framework to SystemTap conjunction.

order. Each probe instance has a specific identifier used to map the correct storage and relay module to the correct probe data, allowing dynamic creation and destruction of probe elements during system runtime. The storage and relay module formats the data into correct format and passes it to the registered storage mechanisms in the order of preference. Failure handling is also included in order to manage cases where a chosen storage becomes unavailable.

For handling special messages including those that denote the start and end information of the provided trace data, special regions in the shared memory are used. These are given higher priority than other parts, as without providing this information first it is not possible to reconstruct the trace data at the end storage location.

*5) Shared memory utilization:* Due to the large volumes of data that can be produced by tracing specific properties of a system, such as the process scheduler, the performance of the trace data relay mechanism is important. In practice, this requires using shared memory regions between kernel- and user-space. There are basically two fast enough ways to address this requirement. One is that the processes can request the kernel to map a part of another process's memory space to its own, and the other is that a process can request a shared memory region with another process. These shared memory regions are also useable between kernel space and user space processes. The choice made when developing the Probe Framework was in favour of the latter option as it works both in kernel space and user space.

The Portable Operating System Interface (POSIX) is a name used to refer to a group of standards specified by the IEEE. The POSIX shared memory falls under the real-time extensions of the POSIX Kernel APIs [29].

In the Probe Framework, the POSIX shared memory is used for inter-process communication (IPC) between the probes in the user space and the storage and relay module. The choice for using the POSIX shared memory is because it

provides a standardized API for using the shared memory, and the implementation is by a set of common C libraries. The API in question makes it possible to use the shared memory regions with reduced effort. The fact that the memory can be easily created, attached to, detached and removed by separate programs in the user space makes it, not only a fast, but also a versatile way of relaying the trace between the probes and the storage and relay module. The shared memory objects can even persist after the creator, the probe of the Probe Framework, no longer exists in the system, thus facilitating versatile collection possibilities for the storage and relay module.

The usage of these shared memory objects in Probe Framework works in the following way: the user space probe calls the probe interfaces of basic services layer to create the internal ringbuffer for itself. This buffer is implemented as a POSIX shared memory object. During the creation operation, the identifier used to refer to the shared memory region, the ringbuffer, is written to a master shared memory region. The identifier of the master shared memory region is known by all parties in the IPC, as such the storage and relay module can then find the ringbuffers and attach to them. This allows the dynamic addition of the probes to the system.

However, the POSIX shared memory is only for sharing memory regions between programs in the user space of the Linux. As such, the RTAI extension was required to obtain a similar solution for sharing memory between the kernel and user spaces.

The Real-Time Application Interface (RTAI) is an extension to the Linux kernel and a collection of a variety of services for real-time programming [30]. In the Probe Framework a specific part of the RTAI is utilized, namely the RTAI shared memory module. This module provides nearly identical functions to those of the POSIX shared memory available in the user space. The difference is that now the shared memory regions are between the kernel probes and the user space storage and relay module, instead of just two user space entities.

Due to the similarity of the RTAI shared memory and the POSIX shared memory, it is possible to use the same memory management method as was used in the user space side. Similar to the POSIX shared memory, a master shared memory region is used in conveying the identifiers of the kernel space ringbuffers to the user space storage and relay module.

*6) Trace communication protocol:* The Probe Framework uses a predefined set of messages for encapsulating the gathered trace. As such, each single data value is associated with a testcase, data type and other relevant information such as time and order stamps. This is important as the collected data needs to be definable and identifiable for it to offer any real value, by value it is meant that the data can be used for instance in detecting anomalous behaviour and in telling when and where it occurred. The used encapsulation enables the data to be related to a specific context and thus gives it meaning. It is also possible to define various properties of the trace, such as the granularity of the timestamps used for stored values. For optimization purposes, it is also possible to leave out all the added properties (such as timestamps) to focus only on the data where high performance is needed.

In order to support different types of analysis scenarios for the captured trace data, PF provides possibilities to defined customized data types. In its basic format, the data can be defined as either input- or output-data for the system. Further from this the data can be described as either a primitive value (such as a number or a boolean) or a text string. This allows for more advanced analysis of the data that is exported. This type information is only given at the beginning of the trace, and different formats are later supported to compress the trace data where the type is the same for a large number of elements. This again supports high-performace tracing.

## V. Experiments and Experiences

To evaluate the PF we carried out two case studies. Both of these are in the domain of monitoring embedded software-intensive systems. We focused on using the monitoring services layer of the PF, and indirectly the basic services through the monitoring layer. We start with describing each experiment and the PF overhead cost on the analysed system. We show what we discovered from the collected trace and how we found the trace could prove useful in a larger sense. In the end we describe our experiences in using PF as a platform for implementing overall instrumentation for system monitoring.

The two case studies we performed were monitoring kernel task switching and the memory usage of different processes. The cases are divided by the type of instrumentation, see Section IV-C2 for the types. The task switching case was done via an internal kernel space probe and the memory usage case via an external user space probe. The memory usage case was conducted on an embedded system that was provided by Espotel[1]. This platform, called Jive[2], is a battery-powered, touch screen equipped PDA type of a device with broad connection interfaces. For the task switching case a typical desktop PC was used.

### A. Task switching case study

The task switching case study focused on the scheduling of processes (tasks). In a typical modern OS there are numerous processes running at the same time [22], and the scheduler handles the execution of tasks by dividing the CPU resources to slots and distributing these slots to the tasks. Our goal was to build a monitoring probe to capture the information on how the task switching is performed with the given usage scenarios. The visibility of the scheduling activity in this scenario is strictly for the kernel space only, and as such, the monitoring had to be implemented as a kernel probe. For this case study, we collected three types of events:

- Task activate
- Schedule
- Task deactivate

[1] http://www.espotel.fi

[2] http://www.espotel.fi/ratkaisut_jive.htm

The schedule event means that the running task is switched to another, the meanings of activate and deactivate are a bit more complex. The activate and deactivate denote that the task is moved to or away from the run queue. For simplicity it can be tought that these two events tell when the task can be run, i.e., scheduled.

*1) Internal, kernel space probe:* In order to keep the case simple, the processor was set to utilize only one core during the instrumentation. A multi-core task switching instrumentation is possible, but it would overly complicate the case as it requires that additional data is collected to identify the core.

The instrumentation used in this case is an inline probe implemented via SystemTap. The code that uses the framework's probe interfaces is added to the SystemTap probe script as embedded C. For details, refer to Section IV-C3. The targets for the instrumentation are the three events, therefore, the first step is to match them directly to the kernel code. The matches for the task activate, schedule and task deactivate for kernel 2.6.24 are found from /kernel/sched.c under the kernel source tree, as show in Table I.

TABLE I
MATCHES FOR THE SCHEDULING EVENTS

| Event | Line no. | Function |
|---|---|---|
| Task activate | 1004 | static void activate_task() |
| Schedule | 3636 | asmlinkage void __sched schedule() |
| Task deactivate | 1016 | static void deactivate_task() |

With the locations of the matches known, the SystemTap is instructed to collect the desired data and insert the code that invokes the kernel probe interfaces of PF to these locations. Also the creation of the probe instance, initialization, etc., are placed to the appropriate locations in the SystemTap probe script.

In each of the locations targeted by the probe, the collected data are the involved task ids, states of the tasks and the current CPU cycle count. Along with those, the time and order stamps issued by the Probe Framework are also stored.

As the instrumentation is done using external instrumentation it also serves to provide a generic reusable kernel monitoring probe for future use when task switching needs to be analysed. Figure 7 illustrates the instrumentation conducted in this case.

*2) Intrusiveness of the instrumentation:* In this case, as task scheduling happens numerous times each second, the used instrumentation is extremely intrusive. There are bound to be consequences due to the instrumentation code. As we want an accurate picture of the task scheduling, all the events need to be collected and no sampling can be used. Therefore, the overhead is so high that the probe is only useable temporarily for purposes such as diagnostics or to provide basis for performance analysis. In this case we do not have any hard real-time requirements so the temporary inclusion of the probe and the temporal effect it poses on the execution is acceptable. Due to the use of SystemTap this probe can also be enabled (included) temporarily in a running system and disabled (removed) when the required diagnostics

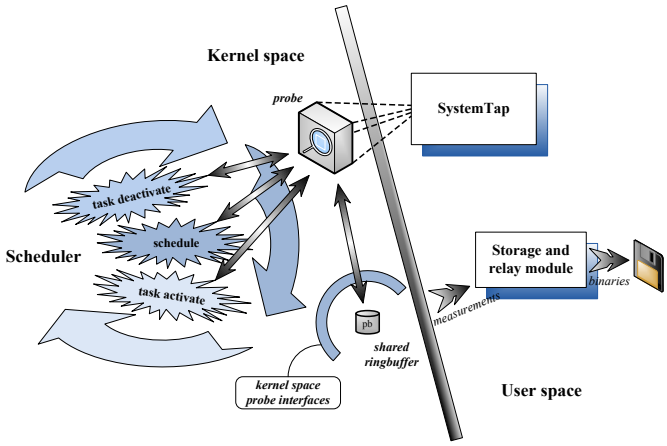


Fig. 7. Task switch instrumentation.

data is collected. Thus, it shows how it is possible to create PF probes that can still be included in the system probe set also in deployed systems while they are only used in ad-hoc style during system execution. Concerning the analysis results, it needs to be taken into consideration that the probe code will consume part of the CPU time, causing skew in the time interval trace, and that the storage and relay module that runs in the user space as a normal task will appear on the obtained task switch trace.

*a) Processing overhead:* The overhead of the PF was measured by capturing system timestamps as jiffies, these jiffies describe system time/clock ticks as 4ms intervals. The jiffies were obtained from the /proc/stat pseudo-file, see Section V-B for more information on the pseudo filesystem. The stamps were collected at the beginning of the instrumentation and at its end. To obtain a reference point, the duration of the instrumentation was measured, and then the same captures were done in a system without the instrumentation, using the measured duration as a time interval between the captures. To give a better picture of the effect the instrumentation had, the overhead is given as the reduction caused to the true idle time of the target system. The overhead is calculated with the following formula:

$$\frac{\left[\frac{DI}{J} - (CJE - CJB)\right] - \left[\frac{DI}{J} - (CJEI - CJBI)\right]}{\frac{DI}{J} - (CJE - CJB)} * 100\%$$

$DI$ = Duration of the instrumentation,
$J$ = Duration of a jiffy,
$CJEI$ = Captured jiffies at the end of instrumentation,
$CJBI$ = Captured jiffies at the beginning of instrumentation,
$CJE$ = Captured jiffies at the end of idle,
$CJB$ = Captured jiffies at the beginning of idle.

Figure 8 clarifies the used formula, note however that the overhead does not accumulate linearily as the figure might suggest. What matters here is the total accumulated overhead, not the momentary overhead values, even though those values might pose an interesting topic for further study.

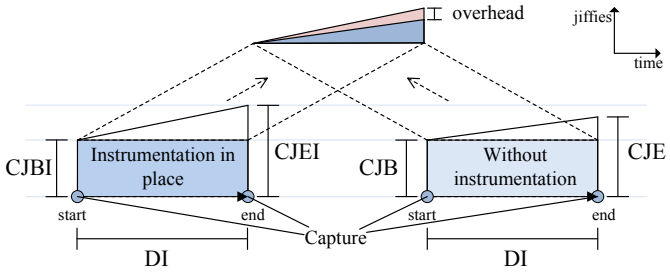


Fig. 8. Processing overhead

The calculated overhead for this case was 16%. Overall, this could be considered a high cost for instrumentation. However, for an analysis case where the monitoring instrumentation is very intrusive, i.e., in one of the most frequently executed function of the kernel, we do not consider this to be a bad result at all. This probe is only intended to be used as a temporary analysis aid and not as a fully included production class feature.

*b) Experiences:* The Probe Framework manages to obtain the targeted attributes. The three events are captured, and the associated cycle counts, task ids and states are collected. The internal probe is successfully injected to the running kernel via SystemTap. The probe is placed into the correct locations in the running kernel, and correct values are extracted. On the user space side, the storage and relay module manages to handle the data collected by the probe, but not without some initial difficulties. We had to optimize our initial code to address the large amount of data captured. However, in the end, the obtained trace is successfully stored according to the configuration on the mass storage.

In our first try, the internal ringbuffer overflowed, causing corruption in the final stored trace data. This was due to the huge amount of data from capturing all task switches in the scheduler. By modifying the monitoring system to use a larger ringbuffer and with the use of the shared memory regions, the experiment was finally carried out successfully and both the trace capture and the storage and relay module were able to perform their duties without error, producing a correct set of trace data.

*c) Utilization possibilities for the trace:* In our case study, we used the obtained trace for different purposes, including the characterization of the process load running on the system as described in [31], for analysing task blocking and scheduler performance.

The used instrumentation provides a fingerprint of the combination of processes run of the system during instrumentation, a load characteristic. Based on this data the internal dependencies between tasks are revealed, such as task execution order dependencies. A practical case conducted by Jaakola [31] showed how this kind of data can be used in simulating how the load characteristic could be executed on a different amount of processing units. With his method, it is possible to simulate how the number of CPUs affects the time required for running the same task set as was run on the instrumented system. For testing purposes, the obtained trace could be used in detecting if priority inversion is taking place. A lower priority task blocks a higher priority task due to the higher priority task waiting for a resource the lower priority task is utilizing. The trace can also be used in estimating how well the scheduling performs and if the time slots dished out by the scheduler are of the adequate length, i.e., the scheduling itself is not restricting the performance.

### *B. Memory usage case study*

The memory usage instrumentation case is defined more specifically. The focus is on user space instrumentation, and the instrumentation target is the procfs, process information pseudo filesystem, which is an interface to the kernel data structures and provides information about the processes on the system. It is located under path /proc in a typical Linux distribution. Actually, the procfs is not really a file system because it consumes no storage space. It needs only a limited amount of memory, as all of /proc resides in the main memory, not on disk. It offers an easy access to information about the processes on the system. Originally, as the name suggests, procfs was meant for kernel and kernel modules to send information about processes' to user space. Nowadays, it is used by all of kernel to report anything interesting to the user space. As such, it is a popular method for user-level to obtain information on the system internals such as the current memory consumption.

The targets of interest here are two memory usage illustrating attributes:

- Total amount of used physical memory
- Total amount of used (memory) swap space

The swap space in Linux context is no different from any other modern OS; it stands for the chunks of memory, normally referred to as pages, that are temporarily stored on the hard disk to cope with the limited amount of available physical memory. Swapping, the process of utilizing the swap space, works by copying a page to or from the preconfigured space on the hard disk to free up or populate a page on the physical memory. With the combination of the physical memory and the swap space usages it is possible to derive the amount of used virtual memory. The used virtual memory is the sum of the used physical memory and the used swap space. Before proceeding further, it is good mention that pages in swap can have duplicates also in the physical memory, thus making the actual values a bit inaccurate.

*1) External, user space probe:* Per specified, the probe used in this case is an external, user space probe. It is an executable that contains the probe front end and the code that calls the user space probe interfaces. This executable is run in the user space with all the other tasks.

As the procfs, target of the instrumentation, houses the wanted data, the probe needs to read the file that contains the total amount of used physical memory and the total amount of used swap. The pseudo-files in procfs reside in memory and have no content until they are read.

Unfortunately, in procfs there are no direct equivalents to the wanted values, but there are four attributes in a pseudo-file meminfo that can be used to derive the target attributes.

The four values, targets of interest, inside the /proc/meminfo are:

```
MemTotal:       X   kB
MemFree:        X   kB
⋮
SwapTotal:      X   kB
SwapFree:       X   kB
⋮
```

In order to capture these values the external probe opens the /proc/meminfo file and reads the content; a snapshot of the values taken when the meminfo was opened. After the probe parses the four values and performs simple subtraction operations to obtain the used physical memory and the used swap:

$$MemTotal - MemFree = MemUsed$$
$$SwapTotal - SwapFree = SwapUsed$$

The probe proceeds by storing the $MemUsed$ and $SwapUsed$ values to its ringbuffer. Along with the two attributes, the time and order stamps issued by the Probe Framework are also stored. The capturing of the four values and the processing is nested inside a loop that iterates every three seconds until the probe executable is terminated. The creation of the probe instance, initialization, etc., are also done inside the probe executable.

Figure 9 displays the instrumentation setup used in this case study.

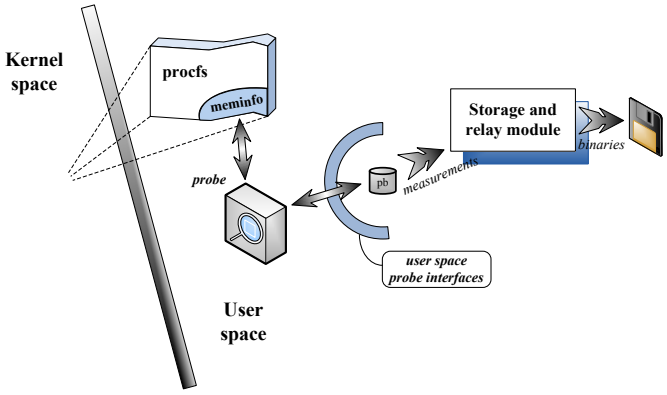


Fig. 9. Memory usage instrumentation.

*2) Intrusiveness of the instrumentation:* The instrumentation in this case has plenty of leeway, as the intrusiveness can be better controlled. The external probe is a normal task in the system. To minimize its effect, the capture frequency for the target attributes was considered. As reading the meminfo file causes extra operations for the kernel, it needs to populate the pseudo-file with values from the memory; the overhead of the read of meminfo is proportional to the read frequency. Similarly, the parsing the probe performs along with the normal operations of using the framework's probe interfaces causes overhead that is proportional to the capture frequency. Also, the fact that the nature of the wanted information does not specify how often it needs to be captured, as opposed to the previous case where all the events had to be captured, leads a result that time sampling is used to limit the probe's impact on the system. The external probe is set to sleep at least three seconds between captures. The three seconds is an approximation because the processing in the system and the processing of the probe itself induces indeterminism.

Considering the frequency of the capture and the small extra work created by the data collection, the overhead of the probe should remain fairly low. Still, some periodic delays to the execution of the processes running in the system are expected. For this case, the overhead caused by the used instrumentation was measured using the same method as in the previous case. The induced overhead was 2%, which is not overly much and could still be further improved with more efficient integration with kernel functionality.

*3) Experiences:* The Probe Framework succeeds in obtaining the targeted attributes, total physical memory usage and total swap usage. The external probe processes the /proc/meminfo pseudo-file successfully and calculates the targeted attributes. The storage and relay module manages to handle the data collected by the probe. The obtained trace is also successfully stored according to the configuration on the mass storage. Overall, the framework performs as expected, no errors or exceptions rose, and the targeted values were collected.

*a) Discoveries made from the trace:* A closer inspection of the obtained trace, after feeding the binary file containing the trace to the probe database and reviewing the actual values of MemUsed, SwapUsed and the accompanying time and order stamps, revealed several issues. First of all, the order stamps indicate that the external probe's internal ringbuffer did not overflow, i.e., all the values captured by the probe made it to the probe database. Second, the time stamps of each target value are all neatly three seconds apart, as specified by the configuration. And Third, an interesting discovery was made by inspecting the SwapUsed values – they were all zero. There are three possible explanations for this: either the external probe failed to process the target values correctly or the storage and relay module misbehaved silently and stored incorrect values or the Linux running in the Jive simply is not configured to utilize swap space. To ensure that the trace was not erroneous, the Jive was inspected, and it turned out that swap space was indeed not in use. The fourth observation was that the MemUsed values in turn all remained under the 64 MB limit, the total amount of physical memory present, and appeared consistent in magnitude.

For effective compression of the trace data values, the storage of two different types of data values at each timestamp proved problematic. The PF implementation only provides options for compressing the data with an expectation that

single data elements are present. This works in a case like task switching, where only the id value of the next task is of interest. However, in a case where several values (MemUsed and SwapUsed) are stored at each timestap, this does not work. Instead, more powerful support would be needed for formats where several data elements are stored in always the same order but there are more than one type of element. This was a design choice in the prototype implementation and could be addressed with an update version.

Additionally, the fact that the swap is not in used by the Jive raises question: Would it not be better to only collect the physical memory statistics? Indeed, it would. But the idea here was to create a generic external monitoring probe that can serve as a monitoring service in other devices too, where the swap could be utilized.

*b) Utilization possibilities for the trace:* Our intent was to use this information for observing memory leaks and growth of consumption over time, similar to audit tests inside a running system as described in [2]. This type of memory problems are considered to be among the most common and difficult to debug due to the long time they take to develop [32]. Thus this type of information is useful to have available to describe the development of the symptoms and to analyse their cause over time.

Worth mentioning is also that the trace can be used for estimating how a new program added to the system or a modification to an existing one changes the memory usage. The trace could help in answering if the available physical memory of the system should be increased to meet the requirement placed by the tasks being run in the system.

### C. Experiences in PF development and use

Few of the hardships faced in developing the PF and experiences gained from those are discussed next.

*1) Testing and debugging of the framework:* A major challenge proved to be the debugging and testing of the implementation itself. As the Probe Framework deals with a large number of error-prone functionality, such as concurrent accesses, shared memory and a great deal of pointer operations in the memory, and contains several internal as well as external interfaces, the testing and verification of it was considered quite important. The testing and debugging had to be done not only in the host environment but also in a system that represents the assumed target environment. The Jive served as an example of such a system.

It took a lot of effort to prune the Probe Framework of errors and to assure its correct behaviour. For testing the tool, C++ unit tests [33] were utilized in testing the user space part, and lots of debugging by hand was carried out. Even with all the effort, the possibility that the tool contains errors still remains. This is the fundamental problem when it comes to testing any complex system. By exaggerating a little it can be said that in every software application there are bound to be lingering errors, because if that was not the case there would be no need for testing to exist in the first place.

As such, this highlights the usefulness of providing a component such as PF, which can be highly intrusive in a system, and needs to work in critical parts such as the kernel-space functionality. Thus re-implementing all the required services in different projects is not feasible but providing a well tested and optimized version would be of great benefit, such as our prototype implementation in the used case studies.

*2) Host-target separation:* The first hand experience on developing on the Jive platform made the complications associated to host-target separation abundantly clear. The reason was that this separation created large obstacles for the development and trials of the Probe Framework, as the many dependencies in the used compilers, capabilities of the host and target systems, and the used C libraries resulted in many complications.

If there is a lesson to be learned from this, it is to ensure that the system where the development takes place resembles the target system as closely as possible. Of course, this does not mean that the host system should be hard to access and contain the challenging features of embeddedness and real-time. Instead, it means that there ought to be easy connectivity and fast trial possibilities available, either on the actual target or in a very close resembling simulator or emulator solution. This also supports the usefulness of having a readily provided trace component available for use in different environments. Even with the common factor of the Linux kernel, and an overall similar OS, between the host and the target, the development was challenging.

*3) Trace handling:* Two particularly challenging issues in the development and trials of the tool were both faced in relaying to collected trace. The first on was the crossing the of the kernel/user space boundary. Even though there are a few methods for achieving it, there is not much information available on the matter, due to this and the unfortunate lack of time the finding of the most effective method had to be left for another time. As mentioned, the Probe Framework had to result to an outside solution for communicating over the kernel/user space barrier, several issues contributed to this choice but the most notable one was the lack of time to implement a custom solution for it.

The other hardship tied to the trace relaying was encountered in the task switch instrumentation. The problem encountered was two-fold; the first part was that the overwhelming amount of collected data required special measures to be taken. For Probe Framework this meant optimizations to certain functionality, but it also increased the intrusiveness of the data relaying as more complex functionality had to be implemented. Overall this kind of instrumentation that collects ”too much” trace is not very feasible to begin with, and certainly cannot be left active in the system all the time.

The latter part of the encountered problem was that because of the nature of the task switch instrumentation, some instability was present. The reason for this is that the scheduling activity cannot be allowed to be blocked. In other words, the system will hang if the probe cannot access its internal ringbuffer and associated pointers, due to the storage and relay

module holding the synchronization lock. As a relieving measure, the probe had to be set to ignore all used synchronization methods and to behave as if it was the only one operating on its shared ringbuffer. Initially, this caused the instability, as the data pointed to by the read and write pointers did not stay consistent with the synchronization gone. To reduce the instability to unnoticeable, the storage and relay module had to take precautions not to alter any of the data while the probe was operating on it. This was achieved with a manner of a safe zone that was utilized in the storage and relay module to keep the read operation from reaching the write location, and thus interfering with it.

*4) Inherent indeterminism:* On a more general level, one insight gained is that preparing for the unexpected is not always feasible. For instance, the case might be to perform an instrumentation or testability related activity in a predefined time slot to minimize the effect it has. As this can be extremely hard, rather than trying to hit a specific time instance, it might be more beneficial to aim for a certain event slot in the execution sequence of the target system. The reason for this is the indeterminism present in the system, the performance and usability improving features, such as the buffering of the write operations, the preemption of tasks, etc., change the time instance when the instrumentation's resource consumption takes place. The indeterminism also contributes to the hardship of estimating the consequences of testability functionality because it induces variations to the observations.

For dealing with the indeterminism, it is beneficial to have several possibilities for implementing the instrumentation. Focusing on the software approaches is not the only way. For a more versatile solution, the hardware-based approaches and hybrid approaches are worth considering. However, this creates another trade-off for testability, by multitude of instrumentation choices the factors that need to be considered also multiply. Thus one goal for future work could be to bring also these types of tracing methods into the framework to support providing a single unified trace also in a combined hardware and software tracing domain.

*5) A general instrumentation approach:* After our trials with the two case studies, we can say that the framework provides a reliable and efficient instrumentation interface with high potential for reuse. We implemented two highly reusable and generic probes. Due to their very generic nature, they can be reused as is or with minor modifications in other contexts. As a generic framework is also bound to be used more frequently and by more people and projects, the code will also become more reliable and optimized over time than separate custom solutions. That is, the more the PF is used the better it becomes. This makes it more likely that found problems are in the system itself and not in the instrumentation code.

The best side of a general, reusable solution is that the more it gets used, the better it becomes. This is true for the Probe Framework, as new probes are created, monitoring services and test services alike, the usability of it increases. For instance, even though the two case studies used in this work were simple, now that the probes have been created they can be reused with very little effort or modification. The ready monitoring services of the Probe Framework also lower the required understanding of the target system. However, a downside related to the usability of the Probe Framework is that the current interfaces of utilizing it might be a bit complex and should be considered for improvement.

As noted earlier, the basic services of the PF have also been implemented on the Java platform. As the PF's implementations both share the same file formats and protocols, we have also been able to successfully use them together. In this sense, through the shared information database storage and export facilities it is possible to get a view of systems with varying component implementations. It is our experience also from these implementations that the simplicity of the provided interfaces is a key to their easy adoption. They must be simple and easy to use and not get in the way of the developer. By hiding all the complexity of trace storage, processing and access behind simple interfaces the PF becomes also more convenient to use. And, that is what the PF aims for, to be a general reuseable approach for instrumentation that lets the developers better focus their efforts on implementing the actual product rather than spend overly much time on creating ad-hoc instrumentation solutions.

Regarding the usability of the Probe Framework, it is not limited to the context of embedded real-time systems. Those attributes are merely something that create a challenge for the PF, i.e., limit the available resource, etc., and in no way limit the environment that the PF concept is suitable to. Similarly, the prototype implemetation being Linux specific doesn't indicate that the PF concept couldn't be used in a different OS. The PF's mentioned Java platform implementation for instance is not limited to the Linux environment.

## VI. Related Work

There are no direct equivalents for the Probe Framework but it does have similar aspects with many other approaches for instrumentation. In general, the instrumentation solutions remain as specific and customized as the systems they target, and as the Probe Framework is just that, a framework for building a more meaningful functionality, it is hard to compare it to other specific solutions. The Probe Framework is considered as a support provider, a building block to be used with other tools and custom handmade solutions, not a replacer for the other instrumentation solutions.

One particular instance, quite similar in the used methods to Probe Framework, is introduced by Chodrow et al. [34]. Their tool is for specifying and monitoring the properties of real-time systems during runtime. Their tool uses an "external" monitor that collects data from events. These events are somewhat similar to the probes used in the Probe Framework but not as versatile, as the events only count the occurrence of some "event" inside a given task. The monitor they use is similar in functionality to the storage part of the storage and relay module offered by the Probe Framework, but not much is said about the storage possibilities. Another similarity is the use

of shared memory. Their tool uses a file-mapped memory, a shared memory region, to pass data from the events to the monitor. This usage of the shared memory is similar to that in the Probe Framework, but it is much simpler and lacks the timing and other associated data that is stored by the Probe Framework. Also, the methods they use for data collection, inline or external entity, are much like the ones used in the Probe Framework. An interesting point they make is to use a tool such as theirs for other functionality than just testing. They give an example: "One can envision an environment where an application task specifies a user-defined function that is invoked by the run-time monitor in response to an event occurrence."[34]

It can be argued that any tool meant for monitoring, testing, data collection, etc., is similar to the Probe Framework, and yes it is a valid deduction. The whole concept of system instrumentation offers a wide field for different tools, and as the Probe Framework can be customized and used for specific purposes on that field then there are bound to be similar existing tools to that specific instance. Yet it is not meaningful to compare the possible implementations realizable with the framework to existing ones, and the sheer number of possible implementations and existing tools makes it unfeasible in this context.

## VII. Conclusion and Future Work

The Probe Framework described in this paper provides the means to build and later on reuse system instrumentation approaches effectively and reliably. It provides support for the basic requirements of storing, relaying and accessing data. More advanced needs such as processing, monitoring and building new functionality to use the traced information are supported by the PF's higher layers. Two cases studies where the PF was used were carried out to validate the different uses of the framework. These cases used the provided building blocks and interfaces to build generic, reusable probes for gathering important system information.

The nature of embedded systems is that there is little consistency between different devices, having led to creation of customized solutions for information access. Here, we have shown that for a system where the PF is available it provides a basis for a uniform instrumentation solution. Generic probes can be reused across systems and new ones implemented by using the provided building blocks and interfaces. The reuse of the framework and probes thus leads to reduction of the implementation effort and also to increased reliability as the found problems are more likely to be in the system itself than the instrumentation code.

For easing the lifespan testing, diagnostics and management of the target system the Probe Framework can be very useful. Given that the Probe Framework is intended to remain in the target system after deployment, it can provide its services during the targets lifespan. Therefore, it can hasten the detection of the possible problems and offer the testing and monitoring services during the targets lifespan. In practice, the Probe Framework requires that the shared library, storage and relay module and the various probes remain in the target, to provide its services after the target has been deployed. Overall, the space requirement of the Probe Framework is minimal, but naturally its presence will affect the rest of the system and should be carefully considered.

As always, there is room for improvement and the list of potential improvements and new features contains several higher-level properties, as well as implementation level optimizations. One interesting high-level improvement is a way to extract internal errors and exceptions from the Probe Framework itself. Voas and Miller [35] mentioned that it is naive to think one can get the monitoring correct when the software under analysis is lacking (no software is seen to be bug-free), as such being able to distinguish between errors of the target and the framework is beneficial. The implementation of this feature could be via predefined "macros", premade messages, for quick and simple reporting on the internal errors and exceptions.

Another desired property for the Probe Framework is the ability to control and configure it remotely. Systems, embedded in particular, are often physically situated in hard-to-access locations. For instance, mobile base stations are scattered around the country, cities, rural towns, etc., and if one happens to malfunction, sending an engineer without any prior knowledge about the problem to do maintenance and repairs will be costly. With remote configuration, the Probe Framework could reside in the system and be adapted to pinpoint the problem. This way, the problem could even be fixed without sending the engineer over, or at least the engineer would have a basic understanding on what the problem is and what to take with him onsite.

However, not all of the development has to be directly in improving and optimizing the framework. Recalling the layered structure of the tool, see Section III, the monitoring services and test services for certain parts are reusable in different implementations. Hence, the utilization of the Probe Framework and the creation of the monitoring and test services can be perceived as development. As more services are created to the monitoring and test layers, the easier it gets to use the tool. The result is that the effort of deploying the Probe Framework in other systems is lower. For future work, in addition to improving the PF and creating new services, the analysis of the data collected by the instrumentations and the ways it could dynamically guide further instrumentations and testing are of great interest.

For further details on the Probe Framework tool and instrumentation methods [36] offers an in depth view.

## Acknowledgment

The authors would like to thank Juha Vitikka from VTT for his collaboration in designing the binary format used in storing the collected traces.

## References

[1] M. Pollari and T. Kanstrén (2009) *A Probe Framework for Monitoring Embedded Real-time Systems*. In Proceedings of the Fourth International

Conference on Internet Monitoring and Protection, Venice, Italy, pp. 109–115.

[2] T. Kanstrén (2008) *A Study on Design for Testability in Component Based Embedded Software*. In Proceedings of the Sixth International Conference on Software Engineering Research, Management and Applications, IEEE Computer Society, Prague, Czech Republic, pp. 31–38.

[3] Framework for Dynamic Analysis and Test, Website, [accessed 23.5.2010], http://noen.sf.net

[4] A. Hussain, G. Bartlett, Y. Pryadkin, J. Heidemann, C. Papadopoulos, and J. Bannister (2005). *Experiences with a continuous network tracing infrastructure*. In Proceedings of the ACM SIGCOMM Workshop on Mining Network Data, Philadelphia, Pennsylvania, USA, August 26 - 26.

[5] S. Elbaum and M. Diep (2005) *Profiling Deployed Software: Assessing Strategies and Testing Opportunities*. IEEE Transactions on Software Engineering 31, pp. 312-327.

[6] Y. Bao, M. Chen, Y. Ruan, L. Liu, J. Fan, Q. Yuan, B. Song, and J. Xu (2008) *HMTT: a platform independent full-system memory trace monitoring system*. In Proceedings of the ACM SIGMETRICS international Conference on Measurement and Modeling of Computer Systems, Annapolis, MD, USA, June 02 - 06.

[7] K. Yamanishi and Y. Maruyama (2005), *Dynamic syslog mining for network failure monitoring*. In Proceedings of the Eleventh ACM SIGKDD international Conference on Knowledge Discovery in Data Mining, Chicago, Illinois, USA, August 21 - 24.

[8] M. Diep, M. Cohen, and S. Elbaum (2006) *Probe Distribution Techniques to Profile Events in Deployed Software*. In Proceedings of the 17th International Symposium on Software Reliability Engineering, IEEE Computer Society, Raleigh, NC, USA, pp. 331-342.

[9] H. Giese and S. Henkler (2006) *Architecture-Driven Platform Independent Deterministic Replay for Distributed Hard Real-Time Systems*. In Proceedings of the ISSTA workshop on role of software architecture for testing and analysis, ACM, Portland, Maine, USA, pp. 28-38.

[10] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal (2004), *Dynamic Instrumentation of Production Systems*. USENIX annual technical conference, Boston, MA.

[11] Mac OS X. Website, [accessed 22.6.2009] http://www.apple.com/macosx

[12] M. Desnoyers and M. Dagenais (2008), *LTTng: Tracing across execution layers, from the Hypervisor to user-space*. Ottawa Linux Symposium.

[13] F. Eigler (2006), *Problem solving with systemtap*. Ottawa Linux Symposium.

[14] SystemTap. Website, [accessed 22.6.2009] http://sourceware.org/systemtap

[15] P. Tuuttila and T. Kanstrén (2008), *Experiences in Using Principal Component Analysis for Testing and Analysing Complex System Behaviour*. In Proceedings of the 21st International Conference on Software & Systems Engineering and their Applications, Paris, France.

[16] Apache log4j. Website, [accessed 22.6.2009] http://logging.apache.org/log4j

[17] C. Lonvick (2001), *The BSD Syslog Protocol*. RFC Editor.

[18] H. Thane and H. Hansson (1999) *Handling Interrupts in Testing of Distributed Real-Time Systems*. In Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications, IEEE Computer Society, Washington, DC, USA, p. 450.

[19] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal (1996) *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, New York, NY, USA, 476 p.

[20] R.V. Binder (1994) *Design for testability in object-oriented systems*, In Communications of the ACM, Vol. 37, No. 9, Sept. 1994, pp. 87-101.

[21] J. Wegener,H. Sthamer. B. F. Jones, and D. E. Eyres (1997) *Testing real-time systems using genetic algorithms*. Software Quality Control 6, pp. 127–135.

[22] A.S. Tanenbaum (2008) *Modern Operating Systems*. 3rd edition, Prentice Hall, 1104 p.

[23] B. P. Douglass (1999) *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley, 749 p.

[24] B. Broekman and E. Notenboom (2002) *Testing Embedded Software*. Addison-Wesley, 348 p.

[25] G. Buttazzo (2006) *Research Trends in Real-Time Computing for Embedded Systems*. ACM SIGBED Review 3, pp. 1–10.

[26] S. Schulz, K. J. Buchenrieder, and J. W. Rozenblit (2002) *Multilevel Testing for Design Verification of Embedded Systems*. IEEE Design & Test of Computers 19, pp. 60–69.

[27] P. B. Menage (2007) *Adding Generic Process Containers to the Linux Kernel*. Ottawa Linux Symposium.

[28] W. Cohen (2005) *Figure 1. Flow of data in SystemTap*. Website, [accessed 22.6.2009] http://www.redhat.com/magazine/011sep05/features/systemtap

[29] IEEE std. 1003.1b-1993 (1993) *POSIX .1b: Real-time Extensions*.

[30] RTAI - the RealTime Application Interface for Linux. Website, [accessed 22.6.2009] https://www.rtai.org

[31] M. Jaakola (2008) *Performance Simulation of Multi-processor Systems based on Load Reallocation*. Masters thesis, Oulu University, Department of Electrical and Information Engineering, Oulu.

[32] J. Vincent, G. King, P. Lay, and J. Kinghorn (2002) *Principles of built-in-test for run-time-testability in component-based software systems*. Software Quality Control 10, pp. 115-133.

[33] UnitTest++. Website, [accessed 22.6.2009] http://sourceforge.net/projects/unittest-cpp

[34] S. E. Chodrow, F. Jahanian, and M. Donner (1991) *Run-Time Monitoring of Real-Time Systems*. In Proceedings of the Real-Time Systems Symposium, IEEE Computer Society Press, San Antonio, TX, USA, pp. 74–83.

[35] J. Voas and K. Miller (1996) *Inspecting and ASSERTing Intelligently*. In Proceedings of the Fourth European Conference on Software Testing, Analysis & Review.

[36] M. Pollari (2009) *A Software Framework for Improving the Testability of Embedded Real-time Systems*. Masters thesis, Oulu University, Department of Electrical and Information Engineering, Oulu.