IPP HURRAY!

www.hurray.isep.ipp.pt

Technical Report

Run-time Monitoring Approach for the Shark Kernel

Filipe Valpereiro

Miguel Pinho

TR-060104

Version: 1.0

Date: January 2006

Run-time Monitoring Approach for the Shark Kernel

Filipe VALPEREIRO, Miguel PINHO

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: {fvalpereiro, lpinho}@dei.isep.ipp.pt

http://www.hurray.isep.ipp.pt

Abstract

Typically common embedded systems are designed with high resource constraints. Static designs are often chosen to address very specific use cases. On contrast, a dynamic design must be used if the system must supply a real-time service where the input may contain factors of indeterminism. Thus, adding new functionality on these systems is often accomplished by higher development time, tests and costs, since new functionality push the system complexity and dynamics to a higher level. Usually, these systems have to adapt themselves to evolving requirements and changing service requests. In this perspective, run-time monitoring of the system behaviour becomes an important requirement, allowing to dynamically capturing the actual scheduling progress and resource utilization. For this to succeed, operating systems need to expose their internal behaviour and state, making it available to the external applications, usually using a run-time monitoring mechanism. However, such mechanism can impose a burden in the system itself if not wisely used. In this paper we explore this problem and propose a framework, which is intended to provide this run-time mechanism whilst achieving code separation, run-time efficiency and flexibility for the final developer.

Run-time Monitoring Approach for the Shark Kernel

Filipe Valpereiro, Luís M. Pinho
Polytechnic Institute of Porto, Porto, Portugal
{fvalpereiro, lpinho}@dei.isep.ipp.pt

Abstract

Typically common embedded systems are designed with high resource constraints. Static designs are often chosen to address very specific use cases. On contrast, a dynamic design must be used if the system must supply a real-time service where the input may contain factors of indeterminism. Thus, adding new functionality on these systems is often accomplished by higher development time, tests and costs, since new functionality push the system complexity and dynamics to a higher level. Usually, these systems have to adapt themselves to evolving requirements and changing service requests. In this perspective, run-time monitoring of the system behaviour becomes an important requirement, allowing to dynamically capturing the actual scheduling progress and resource utilization. For this to succeed, operating systems need to expose their internal behaviour and state, making it available to the external applications, usually using a run-time monitoring mechanism. However, such mechanism can impose a burden in the system itself if not wisely used. In this paper we explore this problem and propose a framework, which is intended to provide this run-time mechanism whilst achieving code separation, run-time efficiency and flexibility for the final developer.

1. Introduction

Modern real-time applications are no longer exclusively dedicated to complex and expensive systems. A rich set of applications are commonly used in everyday life. With the emerging of inexpensive hardware and new telecommunications technology, new applications are emerging using Real-Time Operating Systems (RTOS) theory. Common use cases where user can directly benefit from this approach are mostly related to the multimedia domain, but not limited to it.

This need for reliable but yet adaptable systems is now a constant concern. With current and future demands for real-time embedded applications, developers and system engineers are faced with complex design problems [1]. Whether the OS should be designed to support a generic application or a specific one depends heavily on the final use case. OS reuse is very often the best way to minimize development costs.

Efforts where made to create new tools and theories that approach this problem in a straightforward way. From all these research fields, one that is particularly important, and that is still much unexploited, is monitoring [1]. The Monitoring and Checking paradigm (MaC) allows us to perform testing for verification, validation of critical applications [2] and, importantly in the context of this work, to observe the run-time behaviour of the system after deployment. Nevertheless, in order to monitor we must acquire sufficient information about the state of the system [3], and avoid any interference. Therefore, the monitoring mechanism must allow the monitored information to be arbitrarily chosen and a clear separation between monitoring code and system code must exist.

In this paper we present a flexible framework for information acquisition and monitoring, tailored to the system and application requirements. Our goal for the framework is to allow developers to create applications where the monitoring mechanism is automatically generated and merged with the system and application code, leading to efficient and flexible applications. This work is part of an ongoing project that intends to provide feedback from the operating system to monitoring applications running in parallel with the system application. By providing such feedback, it will then be possible to support quality of service requirement evaluation [4] using real data from the system himself. The practical benefits are obvious if we consider the impact that such a tool has in developing modern embedded systems.

This framework is currently being targeted for the S.Ha.R.K. [5] operating system. The availability of its source code, its modular structure, and the existence of a tracing mechanism make him a good candidate for experimentation. Nevertheless, the current trace mechanism implementation does not allow much room for freedom and it does not follow the POSIX trace standard [6]. The standard defines this mechanism as a monolithic component that can be embedded in all POSIX RTOS profiles expect the Minimum Real-Time System Profile (MRSP) [7]. We believe that this imposition should not limit any developer from using the trace standard as a regular tool on system development.

By using a customization scheme at compile time it is possible to integrate (or not) specific components of code responsible for acquiring the necessary information and support any needed functionality. Therefore, our secondary goal is to implement a POSIX trace mechanism [6] in the S.Ha.R.K. kernel [5] suing a modular approach while respecting the standard semantics.

This paper is structured as follows. Section 2 presents our motivation and the advantages of run-time monitoring. Section 3 presents the POSIX tracer while in section 4 we present the proposed framework for monitoring, and we describe the basic mechanisms and strategies that can be used for implementing this framework. Sections 5 provide some conclusions and further work.

2. Advantages of run-time monitoring

Monitoring should be considered a desired feature for development and deployment phases. For soft real-time applications, monitoring can be used successfully, avoiding the typical *state space explosion* associated with formal verification methodologies [2]. Run-time monitoring gives to the system the necessary degree of freedom in order to dynamically change, adapt and evolve. With a system under monitoring a developer can validate a set of constraints, ensure a quality of service policy working on real data and to observe the internal state of the system.

Thus, it ensures the system overall response and can account for unexpected situations. Furthermore, it is possible to stress the application, supplying unpredictable inputs and test the application response time and resource utilization. In [8], the motivation for the separation of the monitoring mechanisms from the application is provided. From the development process to the actual design and implementation of both the real-time application and the monitoring mechanisms, the advantages are considerable and must be taken into account.

2.1 How to Monitor

In order to monitor we must acquire sufficient information about the state of the system [3], particularly the internal behaviour and state of the operating system. However, such task must be carefully planned. Providing information which is not used decreases the system response time, leaving pieces of non functional code (and possible bugs). On the other hand, providing a reduced amount of information may not allow guaranteeing valid assumptions. Other important aspect to keep in mind when we look into monitoring is the non deterministic effect of observing a system. Through the addition of code lines, we may expect to see the *Heisenberg uncertainty principle* or *probe effect* [1] appearing into the observed system.

We can however minimize this impact and turn interference into a deterministic behaviour. Such task can be accomplished if we provide a clear separation between the real-time application and the existing mechanism for information acquisition. Therefore, a clear separation between monitoring code and application code [9] must exist. As consequence, any monitoring mechanism must be flexible enough to be tailored to specific application needs and must avoid any system interference.

2.2 Collecting information

To efficiently generate system information it is important to clearly identify which type of information is needed to monitor the system. In order to easily manage all the information that can be monitored, we can group it according to its origins [1]: *Data Flow* (internal or external), *Control Flow* (execution and timing) and *Resources*. Furthermore, we can have sub-groups that reflect the logical nature of this information. For example, a *Network Driver* and *Semaphore* are sub-groups under *Resources*, while network and console I/O are subgroups of Data Flow. This partition scheme allows us (for example) to select sets of related functionality at once, and latter at run-time apply a filter to select only the desired information. Still, individual selection can be performed to achieve a fine-grained tuning of the monitored information. When developing a real-time application, the developer selects the data groups that best reflect the requirements and then apply a higher control over each individual part.

There are several ways to gather this data. Our framework is implemented using the POSIX trace mechanism [6] as the base mechanism for monitoring and thus, trace events are the natural choice to collect monitoring data.

2.3 When to monitor

The monitor system must not interfere with the system being monitored, and thus, it is important to determine when and where to monitor. Monitoring code acts as a test probe in the system, generating information whenever it is necessary or relevant. The strategies to place this code depend solely on the origin of the information being monitored and on the language used in the system. The information to be monitored can be divided regarding the origin. There are different problems to address whether we plan to monitor a function call invocation or some kernel attributes. Thus, monitored information divided in three groups: function calls, module private attributes and functions and the kernel attributes.

Furthermore, we need to consider the system time evolution. New functionality can be added and some code may be changed. Thus, it is not feasible to generate code for all possible monitored information. However, it is possible to maintain a set of information on function calls. API's do not change often, and thus we can reuse the same monitoring information in the next system version. The monitoring mechanism must support probe placement in a way that minor code changes do not break the probe context or validity. The remaining system aspects must be monitored as needed, and thus, the developer may need to place probes in some module or on the kernel primitives to data. However, monitoring a module has become easier given the proper support to manage probes during the system development.

3. The POSIX Trace Standard

The POSIX trace standard [6] defines a portable set of interfaces whose main purpose is to collect data over selected functionality in the traced OS. For this purpose, the standard defines two main data types and three different roles that take part during the trace activity. The trace activity is the period of time between a trace stream activation and shutdown where events are recorded. Trace events are a way of encapsulating data with meta-attributes that capture the exact moment and conditions where the data has occurred. Trace streams are a convenient way of recording trace events. The trace streams data type also supports the log functionality, a feature useful to record data for post-mortem analysis.

The standard does not impose any restrictions on the information type that can be collected by events, except that event size is implementation dependent. It is even possible for a user application to use the events to monitor application code.

3.1 Flexibility

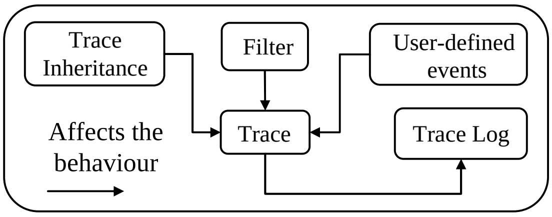

Figure 1 Trace components

Figure 1 presents the POSIX trace [6] components and their relation internal relation. The standard defines the trace mechanism as a monolithic component. Therefore, in order to implement the trace mechanism a target OS must support all the required functionality. The lake of filesystem support is the main reason why the trace standard does not figure in the optional components in MRSP systems [7], since filesystem support is required for trace log operations. However, the absence of filesystem does not compromise the trace operations. An application may only require the trace for online analysis thus; it is possible to incorporate only the required trace functionality in order to support a monitoring mechanism.

However, the trace operating semantics ties the components in a way that makes them to be required even if they are not used. We are currently implementing a POSIX trace mechanism [6] using a modular approach that breaks these functional dependencies through the use of dispatch tables, which may point to the required functionality. Through the use of one indirection level code dependencies are broken. During compile time, the monitoring framework will determine which trace functionality is required for the application, creating the dispatch table and compiling the final trace code.

4. Run-Time Monitoring Framework

The purpose of this framework (Figure2) is to allow developers to choose which parts of the information acquiring mechanism are needed in order to fully support the desired monitoring scheme.

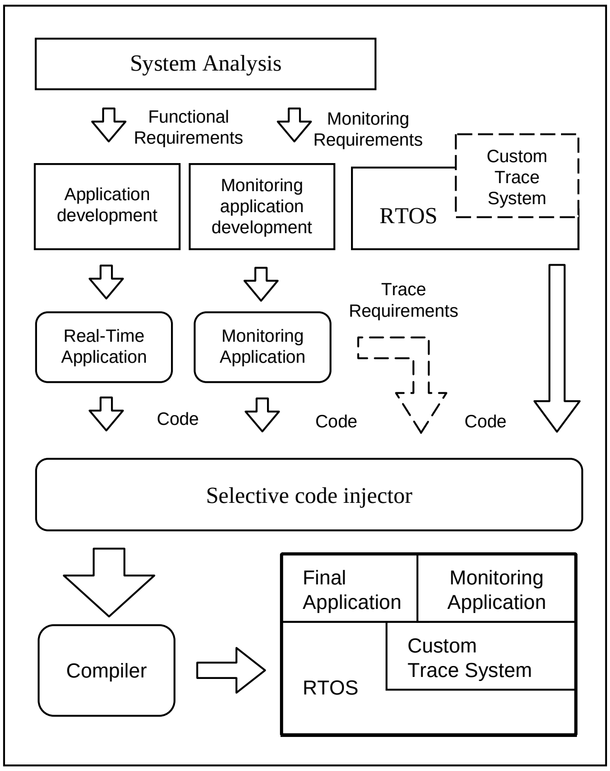

Figure 2 Run-Time monitoring framework

During the development stage the developer must specify which functionality should be monitored, selecting any subset from the information groups or performing an individual selection. It is possible to obtain a fine-grained selection trough the use of event filtering at run-time. The trace functionality requirements can be automatically deduced from the monitoring application. Based on this knowledge our framework will automatically generate a customized trace implementation and place the necessary probe code in the system.

Thus, the developer only needs to focus on the application development, increasing the productivity, shortening the developing phase and giving more time to test and deploy the final application. Another advantage comes from the fact that all communication issues are removed from the real time system context and pass directly to the monitoring application, making the system even more versatile and clean. This separation is clearly an advantage, minimizing intrusive behaviour and approaching the *intrusiveness* principle that should be the motivation for every monitoring solution, eliminating the existence of non functional code, which could be potentially hazardous [10].

The generated code only supplies the base mechanism. To complete the process the developer has to define the monitoring task body. The framework purpose is to handle the monitoring mechanism. The developer still needs to define the body of the monitoring task. A code skeleton that handles the mechanism initialisation is generated; however it is the developer responsibility to tune any particular settings for the POSIX trace mechanism [6] and to define the final monitoring purpose.

4.1 Strategies for customization

The trace customization is achieved using a tool to analyze which components from the POSIX trace [6] are used by our application based on the used API. An example of this are calls to the tracer filter functionality. If such a call is detected then, this component must be incorporated in the trace implementation. Similar analyses are performed for the remaining components. If some component functionality is requested in other component, a link is established using a dispatch table. This solution is elegant given the language used for the target OS [5]. Trough the use of an indirection level we manage to break the implementation into modules, yet retaining the implementation semantics and improving our implementation portability. On contrast, probe generation code and placement is a different problem. Figure 3 illustrates a code injection point.

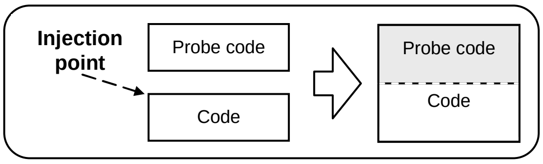

Figure 3 Code Injection

In order to perform these operations the tool must gain some knowledge about the existing trace implementation. Such knowledge can be represented as meta-information (or meta-tags), over the system source code, allowing the tool to instrument the system, incorporating the probes code. To perform this we first analyze the previously specify functionality to be monitored, crossing this knowledge with the meta-information presented in the system source code. For every match the tool defines all the appropriated events, related constants and header files. Finally the tool injects the event generating code in the injection points. The resulting instrumented system code can now be compiled with the custom trace mechanism and the application code to create the final monitoring-aware application.

4.2 Minimizing the probe effect

Probes must be carefully placed to avoid any probe effect [1]. Our strategy depends on the type of information to be monitored. We have proceeded to a systematic identification of information for most of the S.Ha.R.K. kernel [5] primitives grouped using the scheme presented in section 2.2. Currently only the *Control Flow* and *Resources* group where taken into account. Figure 4 illustrates some of the resource access policies implemented in the S.Ha.R.K. kernel.

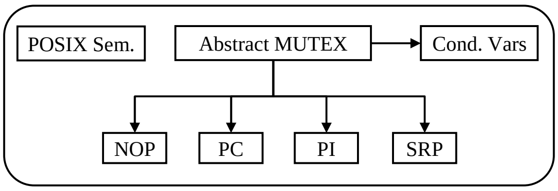

Figure 4 S.Ha.R.K. Resource Access Policies

The kernel implements the usual mutex operation trough the use of an interface, allowing the developer to choose the access policy for each mutex at creation time, and thus creating a flexible approach to resource synchronization. For this particular group we monitor the function invocation and the internal module attributes. This is particular relevant for the QoS strategy [4] that we plan to develop on top of our monitoring framework. This need has leaded us to develop different monitoring strategies based on the type of information to be collected.

If a function has to be monitored and it works atomically (i.e. does not invoke other functions where monitoring probes occur) then we can delay the monitoring of these changes until the next return point in the monitored function. This is efficient, since we avoid unnecessary calls to the monitoring mechanism. Examples of this strategy can be found on the kernel memory allocators and mutex initialization and destroy functions. On contrast, if a function invokes other functions where probes occur then any monitored information should be traced before the function invocation. Our purpose is to guarantee that events are traced in the same succession as they are generated.

For this to succeed we also need to define the streams the events can be traced. Thus, every probe places the trace event into a specific trace stream. The POSIX trace standard [6] defines eight trace streams, from which we reserve four streams to be used in our framework. Three streams will be used for specialized trace purposes (ex. mutexes policies and scheduler information) acting as rotating buffers. The remaining one will be used for general trace purposes (i.e: non-blocking functions). It is up to the developer to specify the trace policy to avoid lost of events in the trace streams.

4.3 Event definition

Events are the base unit to collect information in our target system. We define a hierarchy of event classes whose purpose is to diminish the number of events that must be define for the monitoring framework. Figure 5 shows some event definitions for the mutex operations in the S.Ha.R.K kernel [5].

```
// All structures are packaged, for simplicity
// the gcc macro is not show here.

typedef struct {
    short int func_id; // Function identifier
    short int class;   // Event class type
} event_class;

// Base mutex class
typedef struct {
    event_class class;
    PID         pid;   // Current process pid
} mutex_class;

// Base type for all mutex operations
typedef struct {
    mutex_class class;
    int         mutex_id; // Mutex ID
    int         data;     // Some data
    int         flag;     // Some flag data
} mutex_func_event;

// Base type for large amount of data
typedef struct {
    mutex_class class;
    int         data[12]; // 12 bytes of data
    struct timespec time; // Time instant
} mutex_large_event;

/* ... */
```

Figure 5 Event definitions

The above structures are packed to minimize the amount of unused bytes. By default the GCC compiler performs byte alignment to some multiple of 2. Whenever we mix C types with odd and even sizes we may get extra pad bytes. While this is normal on common C structures we may increase the structure size up to a point where we can not take advantage of fast memory copies in just a few instructions. We take advantage of a dedicated **`memcpy`** implementation that takes direct advantages of special CPU instructions that allow us to copy some blocks of memory using fewer instructions than the block size. Memory copies operations can impose a heavy weight during kernel execution.

Some events are very specialized, due to the way they collect information for the kernel functions. The disadvantage of this approach is the growing number of events to be generated for each traced function. A good solution is to define a single event for function activation if this is the only relevant information to be monitored.

While the meta-information recorded with the events already possesses a time stamp it might the useful to include this information in the event definition. This happens frequently in kernel modules. Monitoring information regarding scheduler decisions is very often dependent on the exact moment where the decision was made. Thus, for this case a timestamp is always placed in the event body. In [10] we define events and trace points for most of the schedulers implemented in the target OS.

4.4 Trace points placement

Our work in [10] has identified the trace points for most of the functionality offered by our target OS. Figure 6 illustrates the placement of trace code.

```
int mutex_lock (mutex_t *mutex) {
   int val;
   mutex_resource_des *m;

   mutex_func_event e =
       MUTEX_EVENT_LOCK_START (mutex);

   // Check for init errors ...

   // Get the module for this mutex policy
   m = resource_table[mutex->mutexlevel];

   // Start lock
   TRACE_EVENT(e, sizeof(mutex_func_event));
   val = m->lock (mutex->mutexlevel, mutex);

   e = MUTEX_EVENT_LOCK_END (mutex);
   // End lock
   TRACE_EVENT(e, sizeof(mutex_func_event));

   return val;
}
```

Figure 6 Trace placement

In parallel with the framework implementation we are also defining a language to allow some instrumentation of C code inside the function body. Most probes need to be placed inside functions and thus we need and simple and efficient way of instrument this code. Currently we use the pre-processing facility from the C language to switch blocks of code. While this solution is straightforward to implement and use, future work in the system code become more difficult to manage and error prone.

4.5 Task definition

The code generated by the framework includes a skeleton for the monitoring task and the necessary changes in the kernel to initialize the monitoring framework. These steps involves the initialization of the trace mechanism and he insertion of the monitoring task in the task descriptor table. Both steps are performed by calls to special functions generated by the monitoring framework. The developer can choose to configure the trace policy used to manipulate the trace stream, and to insert any filtering options if desired.

The other function is used to activate the monitoring task. The developer is responsible for writing the code, choose the scheduling options and perform any necessary initialization. To avoid trace feedback generated by the monitoring application the macros used to trace the events check for the monitoring task PID value in the current executing process, and thus no monitoring occurs. This solution also avoids the placement of event filters to filter any events regarding the monitoring process.

5. Conclusions

In this paper we elaborate on the need for run-time monitoring of operating systems. We propose a framework for run-time monitoring of real-time embedded systems, which considers systems that have to adapt themselves to evolving requirements and changing service requests. Our perspective is that operating systems must expose their internal behaviour and state, making it available to external applications. The proposed framework intends to provide such a mechanism whilst achieving code separation, run-time efficiency and flexibility for the application developer. With this framework we pretend to create a tool to allow a complete customization of monitoring mechanisms, based on a customizable implementation of the POSIX tracing standard.

Further work in this framework includes the development of a meta-information language to model the injection points and related events, dropping the actual solution based on the C pre-processor facility and automatic recreation of the monitored (exposed) information. This aspect as raise some interesting problems to solve. Nevertheless, we fell that automatic recreation of monitored data is a must feature for every monitoring framework to become fully flexible.

Acknowledgements

This work was partially supported by FCT, through the CISTER Research Unit (FCT UI 608) and the Reflect project (POSI/EIA/60797/2004).

References

[1] H. Thane, *Monitoring, Testing and Debugging of Distributed Real Time Systems*, Ph.D. Thesis, MRTC Report 00/15, 2000.

[2] I. Lee, H. Ben-Abdallah, S. Kannan, M. Kim, O. Sokolsky, M. Viswanat, *A Monitoring and Checking Framework for Run-time Correctness Assurance*, Proc. 1998 Korea-U.S. Technical Conference on Strategic Technologies, Vienna, VA, Oct 22-24, 1998

[3] S. Chodrow, F. Jahanian, M. Donner, *Run Time Monitoring of Real Time Systems*, Proc. International Real-Time Systems Symposium, 1991, pp. 74-83.

[4] L. Nogueira, L. M. Pinho, *Dynamic QoS-Aware Coalition Formation*, Proceedings of the 19th IEEE International Parallel & Distributed Processing Symposium, Workshop on Parallel and Distributed Real-Time Systems, Denver, USA, 2005

[5] Soft and Hard Real-Time Kernel (S.Ha.R.K.), http://shark.sssup.it/

[6] IEEE Std. 1003.1, Information technology – Portable Operating System Interface (POSIX), Section 4.17 – Tracing, 2003

[7] IEEE Std. 1003.13, Standardized Application Environment Profile – POSIX Realtime and Embedded Application Support, 2003

[8] R. Barbosa, L. M. Pinho, *Monitoring of Real time Systems: a case for Reflection?* Polytechnic Institute of Porto Technical Report HURRAY-TR-0413, April 2004. Available online at: http://www.hurray.isep.ipp.pt

[9] R. Barbosa, L. M. Pinho, *Mechanisms for Reflection-based Monitoring of Real-Time Systems*, WIP Session of the 16th Euromicro Conference on Real-Time Systems, Catania, Italy, 2004, pp. 21-24.

[10] Leveson N. and Turner C. *An investigation of the Therac-25 accidents.* IEEE Computer, 26(7):18-41, July 1993