IPP HURRAY!

www.hurray.isep.ipp.pt

Technical Report

Run-time Monitoring in Real-Time Operating Systems

Filipe Valpereiro

Luis Miguel Pinho

TR-050501

Version: 1.0

Date: May 2005

Run-time Monitoring in Real-Time Operating Systems

Filipe VALPEREIRO, Luis 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

Embedded systems are increasingly complex and dynamic, imposing progressively higher developing time and costs. Tuning a particular system for deployment is thus becoming more demanding. Furthermore when considering systems which 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 external applications, and a runtime monitoring mechanism must be available. 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 in Real-Time Operating Systems

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

Abstract

Embedded systems are increasingly complex and dynamic, imposing progressively higher developing time and costs. Tuning a particular system for deployment is thus becoming more demanding. Furthermore when considering systems which 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 external applications, and a run-time monitoring mechanism must be available. 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

With current and future demands for real-time embedded applications, developers and system engineers are faced with complex design problems [1]. The need for fault tolerant, reliable, but yet adaptable systems is a constant concern every time a new system or application is build from scratch. Efforts where made to create new tools and theories that approach these problems in a straightforward way.

From all these research fields, one that is particularly important, and that is still much unexploited, is monitoring [1]. Such mechanism allows us to perform testing for verification, validation of critical applications 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 [2], particularly the internal behaviour and state of the operating system.

However, such task must be carefully planned. Providing information which is not used may leave pieces of non functional code and the system may not behave as expected, eventually failing. On the other hand, providing a reduced amount of information may not allow guaranteeing valid assumptions. This implies that any mechanism must be flexible enough to be tailored to specific applications. Furthermore, a delicate balance between the monitoring requirements and the system performance must be considered in order to avoid interference. Therefore, a clear separation between monitoring code and application code must exist.

To overcome this delicate equilibrium we propose a framework for information acquisition, tailored according to the monitoring requirements. The key to achieve this is the customization of the underlying information acquisition mechanism. 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. Such customization will be made in accordance to the monitoring requirements.

Our goal for the proposed framework is to separate the application development from the development of the monitoring mechanisms and to minimize the system interference. 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 [3] 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.

One approach to implement the information acquiring mechanism could be the use of reflection [4]. Using such feature a clear code separation can be made, and run-time customization would also be possible. However, for current operating systems which do not directly support reflection, the alternative is to provide a customizable tracing mechanism, which can be selectively applied during compilation. For compatibility reasons, the goal is to make this tracing mechanism as compliant with the POSIX trace standard [5] as possible.

This framework is currently being targeted for the S.Ha.R.K. [6] operating system. The availability of its source code, its modular structure, and the existence of a tracing mechanism make it a good alternative for experimentation. Nevertheless, the current trace mechanism implementation does not allow much room for freedom and it does not follow the POSIX trace standard.

The paper is structured as follows. Section 2 presents the motivation to the proposed framework. In section 3 we present the proposed framework for monitoring, while section 4 briefly describes the basic mechanisms and strategies that can be used for implementing this framework. Sections 5 provides some conclusions.

2. Run-time Monitoring

Monitoring should be considered a desired feature for development and deployment phases. In [7], 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.

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 ensure a quality of service policy and to observe the internal state of the system working on real data. Thus, it ensures the system overall response and can account for unexpected situations. Furthermore, system requirements can change. Under deployment it may be necessary to change a particular quality of service. Monitoring can play the judge rule, enforcing that such policy will go as expected.

2.1 How to Monitor

One 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. The monitoring can then be implemented on top of it.

To efficiently generate system information it is important to clearly identify which type of information is needed to monitor the system. Latter, we can benefit from this approach, simply because we do not pay extra run-time to generate or check if that particular information is needed. Excess of information to monitor may impose an extra burden in the system and intrusive issues may arise.

In order to easily manage all the information that can be monitored, we can group it according to its origins: *Data Flow* (internal or external), *Control Flow* (execution and timing) and *Resources* [1]. Furthermore, we can have sub-groups that reflect the logical nature of this information. For example, a *Network* and *Sensor Data* sub-groups under *Resources*. With this scheme we provide room for flexibility. When developing a real-time application, the developer selects data groups that best reflect the requirements and then apply a higher control over each individual part.

Finally, we must consider how to determine when the system should produce information. There are several approaches to deal with this issue, but the one that best reflects the nature of current operating systems is driven by events. Events can then be used to trigger the data collection on the internal system state. The tracing mechanism described by the POSIX standard [5] is one of the possible approaches to perform this task. It is oriented by events and doesn't impose any limitations on the type and quantity of information to be collected.

2.2 The POSIX Trace Standard

The POSIX trace standard is based on two main data types and three different roles that take part during the trace activity. The data types are the trace event and the trace stream. The three different roles that take part during the trace activity process are the trace controller process, the traced process and the analyzer process (also called monitor process). For now it is sufficient to state that using the two main data types supplied by the trace standard we have plenty of room for flexibility. The standard does not impose any restrictions on the information type that can be collected by events and allows an application to supply its own events and data.

The only concerns when applying the POSIX standard to real-time systems are the Real-Time Systems Profiles [8]. Which different roles that take part during a trace activity and which level of trace functionality can be supported in a target system? The standard does not state what level of trace functionality should be available in each profile at runtime. Thus, it is possible to incorporate only the required trace functionality in order to support a monitoring mechanism. Later, when the system has all the necessary support, the other levels could be implemented, thus ensuring a full compliance with the POSIX standard. Once again, customization is the key to achieve this. There can be a complete implementation for a full fledged real-time POSIX system, yet we may adjust such implementation to each system needs.

Since our target operating system [6] currently only supports the MRSP profile we need to merge the three different roles into a single (or even multiple) task. From the POSIX standard point of view, no restriction exists that limit this approach. However, work is still needed to determine which configuration for the three different roles should be used in order to achieve a higher efficiency.

Using the current S.Ha.R.K. implementation, it is possible to implement the base trace level, the event filter, and the trace log[1]. Trace inheritance will not be considered since S.Ha.R.K. does not provide support for multiple process. The trace filter is an important feature since it allows a developer to include monitoring over a group. Finally, the log option can be useful if the embedded system has a file system, a fast network link or a flash memory device [9]. Such devices allow a developer to store the internal state changes occurred in the system for a post-mortem analysis and fine-tuning.

3. Run-time Monitoring Framework

The purpose of this framework (Figure 1) is to allow developers to choose which parts of the information acquiring mechanism are needed in order to fully support the desired monitoring scheme. Achieving this goal is only possible if we pay attention to the system analysis and functional requirements.

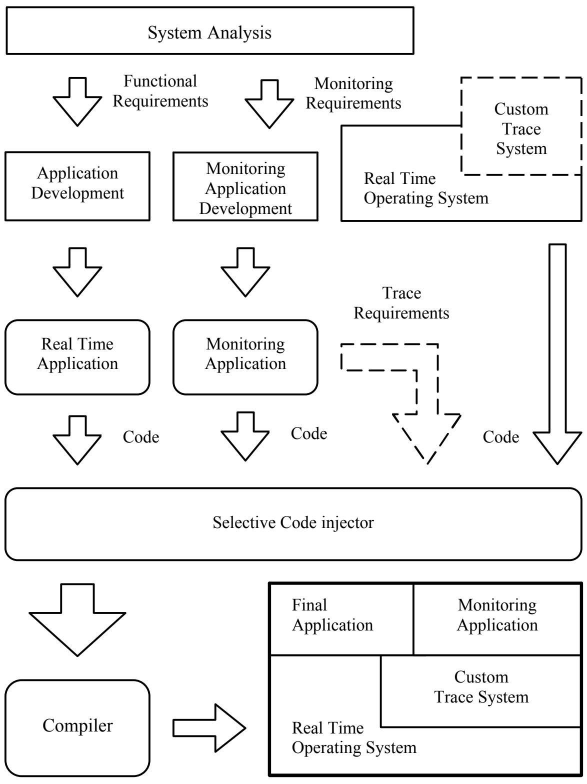

Figure 1. Run-time Monitoring Framework

With a careful identification of such features, we can take full advantage over the underlying trace mechanism. Knowing in advance which parts take a role into the application offers us the possibility to impose some control over the tracer. With such control we achieve a lower interference on the system, eliminating the existence of non functional code, which could be potentially hazardous [10], and avoiding the overall impact that such mechanism produces.

At compile time, a tool will select all the relevant features to be inserted according to the monitoring requirements, injecting the required trace code with the application. During this stage we can apply group control over the information or individually select which part of the system should generate trace information. This leaves space to achieve a greater flexibility, if we choose to apply a trace filter over the tracing data, thus achieving a higher control over the monitoring.

In order to take full advantage of the customization stage over the POSIX trace standard, we must ensure that the target operating system supports all the required functionality. Thus, prior to the creation of this tool the standard (or parts of it) must be implemented over the target system.

Virtually it becomes possible to describe almost any internal state in the system. With such powerful tool it will be possible to achieve a higher degree of flexibility, customizing the system according to specific needs. 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.

An indirect consequence of this approach is the portability of embedded real-time applications. Using a POSIX operating system and having the chance of customize the tracing/monitoring mechanism, a developer does not need any longer to create or adapt previous schemes. 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.

4. Strategies for Customization

We can view the code customization as a process where a developer can select individually features and transform the target system. Such manipulation can be made by disallowing code sections and if necessary, to inject code into specific locations. 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, thus indirect reflection over the implementation source can be used to change the actual target implementation.

When executed, the tool will play two roles in the source code generation. First, it begins to analyze the

[1] Since S.Ha.R.K. has some file system support (FAT16) we can implement the trace log.

monitoring requirements. Then, crossing this knowledge with the meta-information on the actual implementation produces the final source code. The result is a carefully selection of features that should be activated in the operating system. Those features are the minimum necessary subset of the standard trace in order to fully support the monitoring application. Finally, when the tool starts the code generation, it defines the selected events and injects the code to generate them. Then, the resulting source will be ready to feed into the compiler.

Code injection is the key to achieve the customization of the tracing/monitoring mechanism. This is done using a three step approach (Figure 2): identification of event generation code, identification of the injection points in the source files and injecting the desired event generation code.

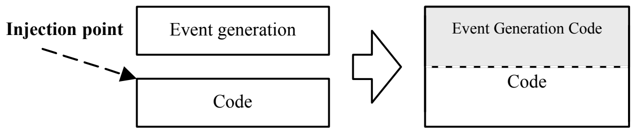

Figure 2. Code Injection

Therefore it is possible to select which parts of the standard trace mechanism should be compiled. We limit the trace mechanism that can be available at run-time, removing any unnecessary pieces of code. It is also at this stage that the tool generates a header defining the selected kernel events in accordance with the POSIX standard rules. For the application specific events the POSIX standard states that an explicit definition should occur at run-time. If application event definitions are not required, then the event register mechanism will not be needed.

The monitoring requirements also provide meta-information specifying which event groups or individual events will be used by the application. The tool transforms this piece of meta-information into injection points in the source code, since prior to this stage we have defined the groups and identified the specific source code injection points.

After the careful identification of the injection points, we can proceed with code inoculation. The selected injection points will receive the necessary code to generate the selected trace events. This process concludes the source code manipulation, allowing the developer to finally compile and test the application.

We are currently evaluating the use of Aspect-Oriented Programming [11] techniques for the basis of the framework customization. Tracing is a well-identified crosscutting aspect, and by using this approach, the event generation code can be seen as advices which must be applied to the system code at specific injection points (join points). Then the tool becomes a weaver for this particular aspect.

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. 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.

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] S. Chodrow, F. Jahanian, M. Donner, *Run Time Monitoring of Real Time Systems*, Proc. International Real-Time Systems Symposium, 1991, pp. 74-83.
[3] 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
[4] 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.
[5] IEEE Std. 1003.1, Information technology – Portable Operating System Interface (POSIX), Section 4.17 – Tracing, 2003
[6] Soft and Hard Real-Time Kernel (S.Ha.R.K.), http://shark.sssup.it/
[7] 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
[8] IEEE Std. 1003.13, Standardized Application Environment Profile – POSIX Realtime and Embedded Application Support, 2003
[9] IEEE Std. 1003.13, Section 8: Dedicated Realtime System Profile (PSE53), 2003
[10] Leveson N. and Turner C. *An investigation of the Therac-25 accidents.* IEEE Computer, 26(7):18-41, July 1993;
[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V.Lopes, J. Loingtier and J. Irwin, *Aspect-Oriented Programming*, In Proceedings of the European Conference on Object-Oriented Programming, 1997.