Technical Report

POSIX Trace Based Behavioural Reflection

Filipe Valpereiro

Miguel Pinho

TR-060203

Version: 1.0

Date: February 2006

POSIX Trace Based Behavioural Reflection

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

Traditional Real-Time Operating Systems (RTOS) are not designed to accommodate application specific requirements. They address a general case and the application must co-exist with any limitations imposed by such design. For modern real-time applications this limits the quality of services offered to the end-user. Research in this field has shown that it is possible to develop dynamic systems where adaptation is the key for success. However, adaptation requires full knowledge of the system state. To overcome this we propose a framework to gather data, and interact with the operating system, extending the traditional POSIX trace model with a partial reflective model. Such combination still preserves the trace mechanism semantics while creating a powerful platform to develop new dynamic systems, with little impact in the system and avoiding complex changes in the kernel source code.

POSIX Trace Based Behavioural Reflection

Filipe Valpereiro, Luís Miguel Pinho

Polytechnic Institute of Porto, Porto, Portugal
{fvalpereiro, lpinho}@dei.isep.ipp.pt

Abstract. Traditional Real-Time Operating Systems (RTOS) are not designed to accommodate application specific requirements. They address a general case and the application must co-exist with any limitations imposed by such design. For modern real-time applications this limits the quality of services offered to the end-user. Research in this field has shown that it is possible to develop dynamic systems where adaptation is the key for success. However, adaptation requires full knowledge of the system state. To overcome this we propose a framework to gather data, and interact with the operating system, extending the traditional POSIX trace model with a partial reflective model. Such combination still preserves the trace mechanism semantics while creating a powerful platform to develop new dynamic systems, with little impact in the system and avoiding complex changes in the kernel source code.

1 Introduction

Traditional Real-Time Operating Systems (RTOS) are designed to support a generic real-time environment. In this scenario, *a priori* assumptions are made on the tasks characteristics, resource utilization requirements and platform. Consequently, the decisions made in the RTOS design narrow the range of possible applications. However, the need to support a new rich set of applications, maybe running on embedded devices, such as multimedia and real-time telecommunication, introduce more stringent requirements on the dynamicity of the underlying operating system.

Although these applications still present real-time requirements, the characteristics of tasks and resource utilisation patterns vary considerably. Typically, multimedia applications demand resources in a non-deterministic way. Under such scenario, the application should deliver the best possible service while respecting the real-time requirements. To achieve such functionality, the application may need to change its own behaviour, for which it is important to be perceptive of the system's current state.

One particular strategy that fits well with dynamic behaviour is Reflection [1], a well know technique in the object-oriented world. Nevertheless, the use of the reflection paradigm to acquire (and control) the state of the system is hindered by the lack of support for reflection in current RTOS. In this scenario we present a flexible framework to reify operating system data using the POSIX trace [2] as a meta-object protocol. Research in the field has already addressed the problem of adapting a reflective approach to an RTOS kernel. Systems like ApertOS [3]; the Spring kernel [4] and more recently DAMROS [5] are attempts to provide reflective capabilities to

operating systems. Our approach differs from previous works, since it is intended to be used in general purpose RTOS.

We consider the use of a partial reflection model [6] to establish behavioural reflection, integrating this model with the POSIX trace mechanism to achieve an efficient reflective framework. It is our belief that such combination can create a powerful tool on non-reflective RTOS, giving the developer freedom to implement new dynamic support on current and well know systems. This will allow providing feedback from the operating system to applications running in parallel with the system application. By providing such feedback, it will then be possible to support quality of service requirements evaluation [7] using real data from the system and to collect valuable metrics on the overall system behaviour.

In this paper, we focus on the reification of data through the use of the POSIX trace mechanism [2] and on its implementation in the MarteOS operating system [8], to validate its usefulness and analyse its impact in the latency and determinism of the system. The paper is structured as follows. Section 2 presents a brief notation on computational reflection and previous approaches on using this paradigm in RTOS. Section 3 presents a brief discussion of the POSIX tracing mechanism, and on the benefits of its use, whilst Section 4 presents the proposed framework and discusses some of the strategies used to reify data using the POSIX trace mechanism. Finally, Section 5 presents some conclusions and future work.

2 Computational Reflection

Reflection can be described as the ability of a program to become 'self-aware'. Self-aware programs can inspect themselves and possibly modify their behaviour, using a representation of themselves [1] (the meta-model). The meta-model is said to be causally connected with the real system in such a way that any changes in the meta-model will be reflected in the system behaviour. In the same way, any changes in the system are reflected in the meta-model. This “inter-model connection” is performed through the use of a meta-interface (often termed as meta-object protocol: MOP).

A reflective system is thus composed by the meta-interface and two levels: a base level where normal computation takes place and a meta-level where abstract aspects of the system are being computed. Through the use of a meta-interface the meta-level can gather information from the base-level (a process termed *reification*) and compute the non-functional aspects of the system, eventually interfering in the system and changing the behaviour (a process termed *reflection*). This principle clearly separates the normal system computation from non-functional aspects of the system.

There are mainly two models of computational reflection [1]. The structural reflection model is focused on the structural aspects of a program (e.g. data types and classes). In contrast, behavioural reflection exposes the behaviour and state of the system (e.g. methods call execution). These models can also be classified as being partial if any form of selection can be performed on the entities being reflected. Partial behavioural reflection [6] is an efficient approach that balances flexibility vs. efficiency by allowing a tight control over the spatial and/or temporal selection of entities that need reification. While spatial control can be applied at compile time by

selecting the objects and methods to be reflected, temporal selection requires an efficient runtime support which goes beyond the scope of our framework.

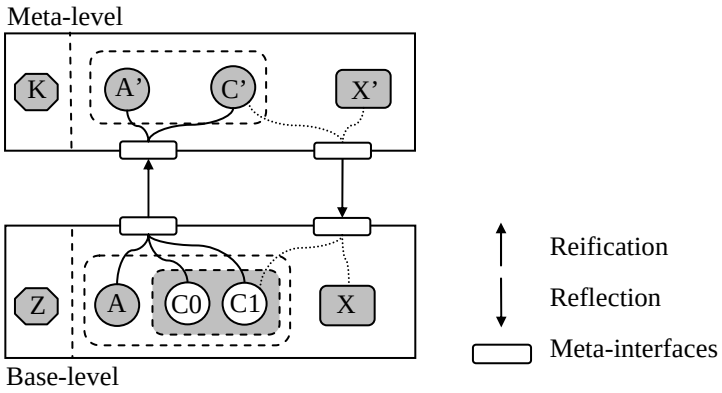

Fig. 1. A reflective system

Figure 1 illustrates a partial behavioural reflective system. Entities can be reflective as A, C, and X and non-reflective as Z. Not all entities may need to reflect into a unique meta-object. C0 and C1 have a common base class and thus in this example are reflected by a single meta-object C' which represents a generalization of class C. Entities A, C0 and C1 belong to a group with related functionality (or behaviour) and thus the meta-model may explore that relation.

2.1 Reflection in Real-Time Operating Systems

RTOS systems are usually designed to support a wide range of applications. It is common for such design to assume no specific knowledge on the target application. However, this approach is not suitable for some class of applications. Some applications may require a real-time response yet they present factors of non-deterministic behaviour. Thus, the dynamic behaviour must adapt to new system (or functional) constraints. It is clear that an interface between the OS and the application needs to exist. The system must allow the application to be aware of system constraints and resources, eventually it may even allow valuable data to be accessed (read only) by the application. In return, the application is responsible to determine the best strategy for its behaviour, and ask the system to incorporate this new strategy. It is also clear that this interface should allow different strategies to be available simultaneously in the target OS.

Computational reflection is a promising solution since it allows us to expose key OS data and computational behaviour into the meta-level where the application non-functional concerns can be expressed and evaluated [9]. Early works on reflective RTOS (such as ApertOS [3] and Spring [4]) have addressed these concerns, incorporating the reflection mechanism in the design and programming language. In Spring, reflection has been used for task management and scheduling, and to support

on-line timing requirements analysis, exposing tasks requirements data. The ApertOS approach relies heavily in the object-oriented model and proposed a complete reflective kernel. While these approaches certainly offer some advantages, they rely on the development of completely new operating systems.

A more recent approach has been done in DAMROS [5] which augments a µ-kernel with a reflection mechanism. This approach allows the application to install user-defined policies in the form of executable source code under certain restrictions. Applications, for example, may not access certain data from the kernel. This limits the implementation of some functionality exclusively in the application space.

3 POSIX Trace

The POSIX trace [2] is a mechanism to collect information on a running system and related process through the use of events. The standard defines a portable set of interfaces whose purpose is to collect and present trace logs over selected functionality in the OS such as: internal kernel activities or faults, system calls, I/O activity and user defined events. A major advantage on the POSIX trace is the ability to monitor (or debug) the kernel and applications during execution.

Another important feature in the trace mechanism is the ability to record events as a stream, allowing the OS to store the traced data on a file system or upload it to a remote server via a network link. The ability to read this stream back again gives the developer a powerful tool to monitor, analyse and understand the application behaviour in a post-mortem analysis. There are few restrictions on standard usage; applications are free to use the trace streams for any particular purposes. Trace streams can be shared across the operating system, with each traced system call placing trace events in one or more streams. It is up to the developer to choose the event calls/stream configuration used in the operating system. For example, several trace streams can be used simultaneously and shared by the operating system and running applications. It is easy to think of an application that can take simultaneous advantage of this architecture to log data into a server while performing system metrics and do some self-monitoring using the trace mechanism.

3.1 The Trace Mechanism

The trace mechanism is composed by two main data types: the trace event and the trace stream. The trace activity is defined as the period between stream activation and deactivation where events are recorded/processed from the trace stream. Traces events are a convenient way to encapsulate data with meta-attributes that refer to the actual instance, conditions and event record status within the trace stream. This information defines (up to some time resolution) the exact moment where the trace event has occurred in the traced process. During this activity, the standard identifies three different roles [2]: the trace controller process, the traced process and the analyser process (also called monitor process). The trace stream establish a link (eventually controlled) that connects the traced process and the analyser process.

There are no restrictions in the standard forbidding a merge between the trace controller process and the analyser process and thus we can view the traced system as being composed by two levels: the observed level where the trace occurs, and the observer level where the streams are controlled and data is analysed. It is also clear that no auto-feedback should occur in the observer level which could influence the actual observation.

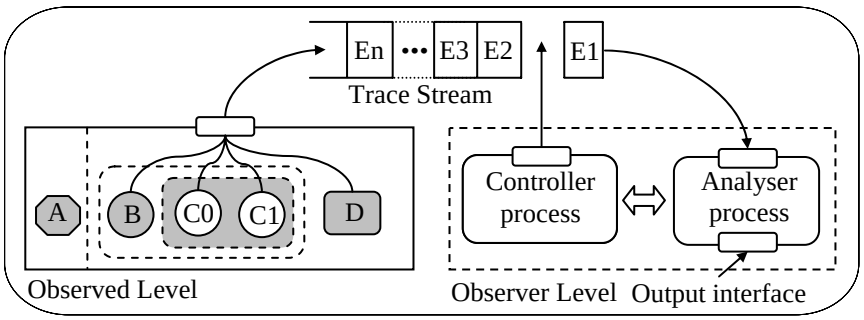

Fig. 2. A system with the trace mechanism

Figure 2 exemplifies the different roles that take part in the traced system. Not all objects in the system may be traced (or needed to be traced). The analyser process presents an output interface which can be used by the application (or system) to obtain information derived from the traced data. As an example, the quality of server managers of [7] requires access to information concerning the actual resource utilization of the system. This information can be provided by analysing the events generated by the operating system.

3.2 Flexibility

The POSIX standard defines the trace mechanism as a monolithic component. There is no room for customization, and thus, this component can not be used with the RTOS targeted for the Minimum Real-time System Profile (MRSP) [10]. Nevertheless, features required by the standard such as filesystem and process inheritance are of no use in this profile and do not compromise the trace functionality. Our work intends to supply a flexible, customizable trace implementation with a small memory footprint, toward the application requirements, in a way that only the necessary trace functionality will be present in the final application.

4 Framework Design

Several techniques have emerged in the RTOS research to address the lack of proper support for dynamic applications behaviour. Our main motivation for the development of this framework is the need of a common, portable platform for data collection and system actuation where these and future techniques can be evaluated.

The goal is to support reflection on static application-oriented RTOS, allowing soft real-time applications to change behaviour in response to the system's state, therefore becoming more adaptive. Moreover, the framework will allow to separate the application development from the development of system state analysis mechanisms, and to minimize the system interference.

The need of a portable interface to collect and reify system data led us to consider the POSIX trace mechanism [2] as the basis component of our framework. However, the standard requires the operating system to support functionality which is not required for the role played by the trace mechanism in our framework. The standard rationale defines a monolithic trace mechanism, creating dependencies between the individual trace components. To overcome this limitation we consider a trace mechanism based on modular components, avoiding unnecessary code dependencies while preserving the functional semantics.

We do not consider the use of computational reflection as a whole. Instead a partial reflection model [6] is used; where data is reified without direct transfer of control to the meta-level (an asynchronous reflection model). Traditionally, computational reflection belongs to the language domain, usually implemented into the language run-time environment. This approach does not explore the advantages of concurrent systems and thus reflection occurs as a linear transition between the base-level to the meta-level and vice-versa. The framework takes direct advantage of our problem domain, redefining the transition between the base and meta-level. This is supported by extending the controller and analyser process roles within the POSIX trace mechanism. We also introduce a third process to reify data from the trace streams and create/modify the meta-objects (see figure 3). Under this extended model, the meta-objects act as a "consciousness memory" of the system state, while the analyser process performs some "consciousness" analysis. Eventually, the analyser process may intercede asynchronously in the system, introducing non-functional aspects.

In this paper we focus on the data reification and meta-objects construction; the analyser interface and intercede mechanism (necessary to complete the framework model) will be the focus of further work.

4.1 Modular POSIX trace

The main reason why the trace standard is not available in the MRSP profile is the lack of filesystem support which is a required feature for the implementation of the POSIX trace interface. The organization of the tracing rationale text for the trace interface defines the trace as a monolithic component, thus leaving no flexibility in the usage. Yet, there are distinct individual components composing the trace mechanism.

A detailed examination of the standard and the trace use cases show us that filesystem support is only useful for offline analysis, a feature used by the trace logs to record data into a permanent storage (a use case not addressed in this paper). On contrast, online analysis is a useful tool to reason on the current system state and does not require filesystem support. This component works as an extension to the main trace functionality, adding new features that support other trace scenarios.

We can explore the inter-component relations to avoid non-functional and unnecessary code in the final binary image, thus reducing the application memory footprint and minimizing the impact on the traced system. Modularity can be achieved if each component is implemented as a separate package in such a way that the main tracer component does not require linking against other component packages. All the remaining trace components must depend strictly on the main package which contains the base definitions, unless a dependency exists between different modules.

The modularity goal is to preserve the functional semantics while eliminating inter-component dependencies. To break these dependencies we need to work on the trace implementation. The application may not use some of the trace components; however the existence of these code dependencies will create a link between the application code and the implementation code. An example of a usage scenario for the trace mechanism is the ability to perform system metrics. Such example may not require the filtering or trace log features. Consider the steps performed on every call to the `posix_trace_event` function to successfully trace an event:

- Find an available trace stream or return.
- Discards the event and return if the event does not pass the filter.
- If it is a user event and data is larger than maximum value, truncate the data.
- Store the event
- Adjust the stream properties (trace policy).
- Flush the stream into the trace log if required.

The function semantics will require the filter component due to the existence of a function call, even if it the function result is irrelevant for the usage scenario, thus a link to this code will be established at compile time. To implement the desired modularity we are currently using a dispatch table that invokes the requested function if the table entry is not null. This solution minimizes the amount of compiler work, since it only needs to recompile the main component. It introduces a new indirection level, but that does not generate any measurable delay in the trace execution.

4.2 The extended trace model

In this extended model, we introduce some principles of partial reflection using the POSIX trace. In the model, the trace streams are used as the meta-interface that allows the meta-level to reify information from the base-level. We also introduced a new process in the trace observer level (the meta-level), the reify process, that acts upon instructions from the controller process. Its main role is to read events from the trace stream and to create and modify the corresponding meta-objects. These objects will be used by the analyser process to perform some “consciousness” analysis, thus the analyser process works exclusively with reified data. The amount of data and analysis type depends only on the developer purpose. For this reason the framework defines the task type but not the task body and properties, giving the developer the freedom to manage the meta-objects.

To avoid possible data inconsistencies any meta-object access is performed using a protected type and the associated interface (see figure 5). The base interface for meta-objects is a procedure to replace a meta-object, a procedure to commit changes in the meta-object and a function to obtain a full copy of the meta-object. However it might

be useful to extend this interface with specialized read functions to access some data in the meta-object, improving the access time by avoiding a full object copy. Figure 3 illustrates the extended trace model and the relation between the various entities.

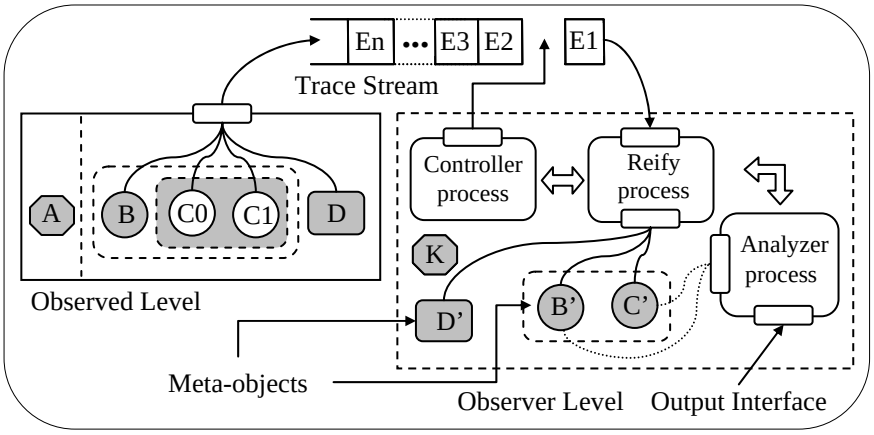

Fig. 3. The extended trace model

To use the framework the developer must define the analyser task body, completing the meta-model. This activates the framework, triggering the code inclusion in the application; otherwise the compiled code will be trace free. Note that from the traced call point of view no transfer of control from the base-level to the meta-level takes place during the traced call execution; hence the reflection model behaves asynchronously. This is a powerful property, since the meta-level will behave as a central state of the operating system that can be queried by higher abstract concerns that can not be expressed with traditional reflection. Note that at present we do not define the output interface which is beyond the scope of this paper.

4.3 Implementation details

The first step to implement the extended model is to determine which functionality offers interesting data to be reified. Examples of possible information to reify are: memory usage per process (or task), mutex operations including some internal details (accessing task ID, blocked task list, mutex policy …) and CPU bandwidth used, but different types of information can be also be gathered. Figure 4 presents some simplified examples of event definitions used to reify data from the mutex functions. The `Data_Envelop` type as been defined to ensure that trace events with size larger than the maximum data size allowed in the trace stream can flow without loosing information. However, this option pays heavily and must be avoided whenever possible. Note that we omitted some type info just for clarity's sake.

Some of the reified information exhibits patterns of similarity and thus we can group it, creating a convenient way of express the application "reflective" requirements. With this purpose, we define four sets of functionality that can also be expressed in terms of sub-sets for convenience and further fine-grain control over the

data. The sets of functionality are: internal kernel structures; scheduler, locking mechanism and signals; system calls; and I/O triggering (data transmitted/received).

```
package Trace_Events_Data is

   type Data_Envelop is record
      Info : Data_Info;
      Data : Data_Buffer;
   end record;

   type Mutex_Init_Event is record
      Op              : Op_Code;
      Mutex_Id        : Integer;
      Policy          : Locking_Policy;
      Prio            : Task_Priority;
      Preemption_Level : Task_Preemption_Level;
   end record;

   type Mutex_Event is record
      Op          : Op_Code;
      Mutex_Id    : Integer;
      Task_Id     : Integer;
      Task_Status : Task_Status;
      Prio        : Task_Priority;
   end record;

   -- ...
end Trace_Events_Data;
```

Fig. 4. Events definition

Each set of functionality is also connected to a unique trace stream, leaving the remaining streams free to other purposes. This simplifies the data reification process, avoiding intersection of data events from different functionality sets which would result in a larger and complex decoding task. We also define the events used for each functional unit in each traced set and the corresponding meta-objects and the event(s) associated, creating a map that links the reified units and the meta-objects. This information is vital to the “reify process”, to create the meta-objects whenever a related event is received. A second map is created dynamically that binds the created meta-objects to the meta-object ID and type. This step ensures that any update information arriving in further events is committed to the corresponding meta-object.

Figure 5 present a simplified meta-object definition and the protected object used to ensure that no data inconsistency occurs whenever an update operation is performed in the meta-object. The protected interface was kept simple, but can be extended to support faster access to individual fields on the meta-object to improve the access time. This option might be useful for testing some properties without requiring access to the whole object.

Figure 6 presents some maps definitions and task types. This package also defines the controller, the reify and the analyser tasks. Bodies for the first two tasks will be defined within the framework. The third task must be defined by the application developer to perform the desired analysis using the meta-objects.

```
with Trace_Events_Data;
package Meta_Objects is

   type Meta_Mutex is record
      Owner            : Integer;
      Mutex_ID         : Integer;
      Policy           : Locking_Policy;
      Preemption_Level : Task_Preemption_Level;
      Blocked_Tasks    : Tasks_Lists;
      Status           : Boolean;
   end record;

   procedure Init_Meta_Object (Event : in Mutex_Init_Event);

   protected type Meta_Mutex_Access is
      procedure Store_Object (Mutex : in Meta_Mutex);
      procedure Commit_Changes (Event : in Mutex_Event);
      function  Get_Copy return Meta_Mutex;

   private
      Mutex : Meta_Mutex;
   end Meta_Mutex_Access;

   -- ...
end Meta_Objects;
```

Fig. 5. Meta-objects definition

```
with Trace_Events_Data;
with Meta_Objects;
package Meta_Level is

   type Mutex_ID is Integer;
   type Mutex_List_Access is
      new Map (Mutex_ID, Meta_Mutex_Access);

   Mutex_List : Mutex_List_Access;

  task type Controller_Task (Prio : Task_Priority);
  task type Reify_Task (Prio : Task_Priority);
  task type Analyser_Task (Prio : Task_Priority);

   -- ...
end Meta_Level;
```

Fig. 6. Meta-level definition

4.4 Performance metrics and results

We have done some experiments in order to find the impact of the framework both on the size of the code and on the execution times of the traced functions. Table 1 presents the overhead on the code size of a traced system. The results allow determining that, depending in the number and type of trace events embedded in the traced unit, an overhead of approximately 10% is created in the overall code size.

This presents a considerable impact, but it is an expected side effect of the increased functionality.

Table 1. Comparison of code size

Description	Size in Bytes
Simple procedure (sum of one integer)	480
Simple procedure with a single trace event	780
Mutex unit without trace events	15884
Mutex unit with eight trace events	17692
Scheduler unit without trace events	13032
Scheduler unit with eleven trace events	15088
Trace implementation with all dependable units	38056
Hello World without trace	341936
Hello World with trace unit	379088

Tables 2 and 3 show the execution times for some of the traced functions, with and without the trace functionality. The tests were performed on a Pentium-III at 930 MHz. The time values are measured by the time-stamp counter (TSC), and mean values were obtained after 5000 measures. The test application sets up a trace stream with sufficient space for all the events generated during the simulation.

Table 2 presents the results of a test setup, where events related to the mutex unit were generated by a loop performing several calls to obtain a lock on the mutex. The results show an increase of execution time by a factor of approximately 0.8 µs for each traced function. The last test also shows the average trace time for regular events versus events using the data envelop capability. As expected they are heavier but offer a more flexible solution to trace large amounts of data.

Table 2. Execution times for the mutex unit

				Mean	
Function	**Trace**	**Min**	**Max**	cycles	µs
Pthread_Mutex_Lock	No	137	221	179	0.19
	Yes	840	1031	900	0.97
Pthread_Mutex_Unlock	No	251	362	296	0.32
	Yes	931	1133	1024	1.1
Event Trace		699	962	740	0.8
Event Trace with envelop		1317	1640	1396	1.5

Table 3 shows the execution times for the scheduler unit. They were performed with the same configuration, except that the events generated by the scheduler unit were obtained using four simultaneous tasks with different periods, execution time and priority, to create some scheduler activity.

In this case, the experiments showed an increase of approximately 0.7 µs for each traced function, which is in the same order of magnitude of other kernel to user mechanisms available in the MarteOS kernel [11,12]. Considering the gained functionality, this overhead is more than acceptable, since it allows applications to have access to “fresh” kernel data.

Table 3. Execution times for the scheduler unit

Function	Trace	Min	Max	Mean	
				cycles	µs
Ready_Task_Reduces_Active_Priority	No	124	286	156	0.17
	Yes	741	983	833	0.90
Running_Task_Gets_Blocked	No	92	167	118	0.13
	Yes	702	1242	774	0.83
Running_Task_Gets_Suspended	No	174	494	270	0.29
	Yes	612	1624	821	0.88
Task_Gets_Ready	No	100	305	130	0.20
	Yes	739	2108	825	0.89
Do_Scheduling	No	116	573	202	0.22
	Yes	744	1286	853	0.92
Event Trace		495	958	650	0.7

5 Conclusion

Soft real-time multimedia applications tend to present factors of non-deterministic behaviour. Developing applications in this domain requires the study and development of dynamic strategies which allow the system and application to adapt, improving the quality of the output generated by the application. This requires, however, applications to have access to the current state of the system, particularly in what resource availability (CPU included) is concerned.

In this paper we present a framework, which uses the POSIX trace mechanism as a Meta-Object Protocol, to implement a partial asynchronous reflection model. Using this framework, applications can query the system state by accessing a meta-level which presents reified information of the system. The design requirement for the framework is the use of standard functionality available (or easily incorporated) in current real-time operating systems. The framework is not tied to any particular operating system, thus making further ports straightforward. We hope that this work can open new perspectives into the use of reflection in real-time operating systems.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments and suggestions. 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. P. Maes. “Concepts and Experiments in Computational Reflection”, in. Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA’87), Orlando USA, 1987, pp. 147–155.

2. IEEE Std. 1003.1, Information technology – Portable Operating System Interface (POSIX), Section 4.17 – Tracing, 2003.
3. Y. Yokote, “The ApertOS Reflective Operating System: The concept and its implementation”, in Proceedings of the 7th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA’92). ACM Press, 1992, pp. 414–434.
4. J. A. Stankovic, “Reflective Real-Time Systems”, University of Massachusetts, Technical Report 93-56, June 28, 1993.
5. A. Patil, N. Audsley, “Implementing Application Specific RTOS Policies using Reflection”, Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium, San Francisco, USA, 2005, pp. 438–447.
6. E. Tanter, J. Noye, D. Caromel, and P. Cointe, “Partial behavioural reflection: Spatial and temporal selection of reification” Proceedings of the 18th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003), October 26-30, 2003, Anaheim, USA, pp. 27–46.
7. Luís M. Pinho, Luís Nogueira and Ricardo Barbosa, “An Ada Framework for QoS-Aware Applications”, Proceedings of the 10th International Conference on Reliable Software Technologies (Ada-Europe 2005), York, UK, June 2005, pp. 25–38.
8. M. Aldea and M. González. “MaRTE OS: An Ada Kernel for Real-Time Embedded Applications”. Proceedings of the 6th International Conference on Reliable Software Technologies (Ada-Europe-2001), Leuven, Belgium, May, 2001, pp. 305–316.
9. S. Mitchell, A. Wellings, A. Burns, "Developing a Real-Time Metaobject Protocol", Proc. of the 3rd IEEE Workshop on Object-Oriented Real-Time Dependable Systems, Newport Beach, USA, February 1997, pp. 323–330.
10. IEEE Std. 1003.13, Standardized Application Environment Profile – POSIX Realtime and Embedded Application Support, 2003
11. M. Aldea and M. González, “Evaluation of New POSIX Real-Time Operating Systems Services for Small Embedded Platforms”, Proc. of the 15th Euromicro Conference on Real-Time Systems, ECRTS 2003, Porto, Portugal, July, 2003, pp. 161–168.
12. M. Aldea and J. Miranda and M. González , “Integrating Application-Defined Scheduling with the New Dispatching Policies”, Proceedings of the 10th International Conference on Reliable Software Technologies (Ada-Europe 2005), York, UK, June 2005, pp. 220–235.