An Application Adaptive Generic Module-based Reflective Framework for Real-time Operating Systems

Ameet Patil and Neil Audsley
Real-Time Systems Group,
Department of Computer Science, University of York, York, UK - YO10 5DD.
Email:{appatil,neil}@cs.york.ac.uk

***Abstract*— In low-resource real-time embedded systems, dynamically changing demands of applications in the form of resource requests, strict deadlines, fault recovery, etc. are non-deterministic in nature. It is difficult for the underlying real-time operating system (RTOS) to satisfy such demands of each and every application. There is no single existing solution to overcome this problem.**

This paper proposes a generic module-based reflective framework for an RTOS that self modifies and adapts itself to the dynamically changing demands of applications.

I. Introduction

Complex applications running on low-resource Real-time embedded systems face severe constraints, including the need to meet strict deadlines, cope with system faults, low resource availability, etc. The demands of such applications are mainly non-deterministic in nature and are often poorly met by the underlying Operating system (OS). For example: real-time multimedia applications often demand resources non-deterministically and have to compete for them along with other applications [1]. For a few applications the runtime changes in the events and the data contents of these events lead to significant changes in the resource requirements (eg. RADAR systems, Robot control, etc.). Operating systems on the other hand are built without the knowledge of the applications and the constraints involved (eg. no knowledge of the amount of memory, needs of applications that would run on it, availability of various resources, etc.). The OS is built for the general case, rather than the specific requirements of an application.

Applications may want to impose certain changes in the OS inorder to increase efficiency in functioning predictably and attain their deadline. For example: a complex real-time security database application would want as much of its data as possible to be present on the physical memory at all times. It would do no good if the OS implements a virtual paging memory management algorithm that would swap the physically present data for some other pages required by other applications. The application would hence require to impose a change in the memory management policy.

Most real-time OSs implement a fixed static model having a fixed memory management algorithm, scheduling algorithm, etc. However, several solutions have also been proposed and implemented to solve the above issues individually. There has been no all-in-one single solution. For example: components called ‘configurator’ have been implemented to take care of the demands of multimedia applications and provide better Quality of Service (QoS) [1]; Adaptive Resource Allocation (ARA) mechanisms have been developed to predictively and efficiently handle resource management for the dynamic needs of complex real-time applications [2]. Having implemented the solutions before hand would be of no use. Also, it is impossible to build an OS consisting of all the possible solutions.

The real-time OS needs to be flexible enough to dynamically adapt to changes in the internal(eg. application needs) and external environments. We employ reflection mechanism to design a generic reflective framework for such an RTOS. Although there has already been much work done on reflective OSs (eg. ApertOS [3], Spring [4]) but as we pointed out earlier they concentrate on particular problems alone. For example: ApertOS was mainly built for mobile computing environment where object needs to migrate from one environment to other. Spring was developed to be a distributed OS to support fast and efficient cross address space calls.

This paper proposes a generic module-based reflective framework for an RTOS which caters to all the application needs (eg. memory management, resource management, etc). The Applications would thus be able to bring about change in the OS thereby, making the OS adapt itself dynamically to the changing demands of applications. The framework focuses on building each individual system module to be reflective with support for reflection built into the kernel. We also introduce as an example, the design of two reflective system modules: a Reflective Scheduler and a Reflective Virtual Memory Manager (VMM).

II. Background On Reflection

The mechanism of Reflection is not new to the world of programming languages and also OSs. Languages like Java, Smalltalk, CLOS have support for reflection; OSs like ApertOS [3], Spring [4] are reflective OSs [5]. Reflection essentially is a mechanism by which a program code or application becomes ‘self-aware’, checks its progress and can change itself or its behaviour [6]. This change can occur by changing data structures, the program code itself, or sometimes even the

semantics of the language its written in. To facilitate this, the application or program code has to have knowledge about the data structures, language semantics, etc. The process by which this information is provided to it is called ‘Reification’.

Often we find cases where the application needs to change its course of execution depending on some unpredictable/unknown run-time conditions. Reflection comes to rescue here. It allows to make this change dynamically(there are some exceptions to this however) at runtime with/without the knowledge of the application program.

The Reflection model consists of a base-level (which is the application code), and one or more meta-level forming a structure called Reflective tower. The code in the meta-level is responsible to intercept the necessary calls from or to the base level, analyse the reified information and affect any change if required. A protocol defined so as to establish a mechanism by which the meta-level entities introspect (analyse), intercede (eg. by intercepting calls to or from base-level) and affect change to the base-level is called the ‘Meta-Object Protocol (MOP)’ [7].

III. Our Reflective Framework

As shown in figure 1, the generic module-based reflective framework is based on the micro-kernel architecture with all system modules running as separate individual processes outside the base kernel.

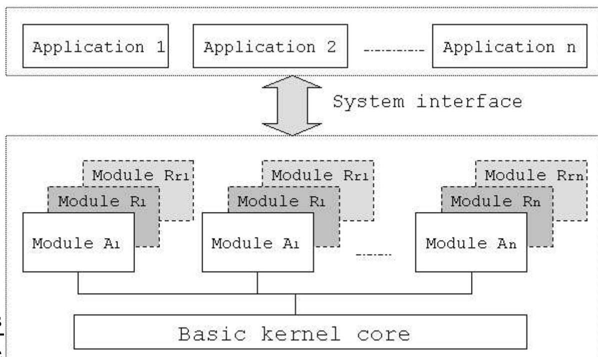

Fig. 1. Generic Reflective OS framework

The kernel provides basic support for reflection such as interface for system modules/applications to reify information, interface for reflective module to introspect and intercept. One or more or all system modules can be individually designed to be completely reflective. A reflective system module (eg. a reflective scheduler) is responsible to analyse reified information and take intuitive steps to intercept and change its behaviour.

A. MOP for the Reflective Modules

The MOP provides with the basic set of rules that decide how the reflective framework will work. Following are various rules defined:

exactly what and how much is reified?: the only factor that affects this decision is the available memory resource. Obviously we cannot fix memory size during design or implementation time (because systems change and so does memory size vary for different systems). Hence, our approach would be to make an initial decision on what information to reify and later dynamically reify as much of this information as possible in the order of importance so as to minimise the memory utilisation. This way the model would suit all systems with different memory sizes.

should there be implicit MOPs?: Implicit MOPs (i.e. the base-level is intercepted and control transfered to the meta-level without the knowledge of the base-level entity) have their advantages over explicit MOPs (i.e. base-level explicitly transfers control to meta-level) in that they do not require any change in the module code. We provide implicit MOPs by a ‘caller-registration’ mechanism that requires the meta-level entities to register initially what calls need to be intercepted and reflected upon. This information is registered in the kernel which invokes the respective meta-level reflective modules whenever it detects the registered calls.

should there be explicit MOPs?: Although implicit MOPs have the advantage of providing transparency, there is no harm in having explicit MOPs. Hence, the system designer is free to make use of explicit MOPs in order to directly invoke the Reflective modules in the meta-level.

how to define interface for introspection?: The interface for introspection (i.e. the mechanism by which the meta-level objects inspect/observe/analyse the base-level activities) would be with a set of kernel shared functions or system calls that help in retrieving the reified information from the kernel.

how many meta-levels should be allowed?: if a reflective system module at one meta-level is sufficiently well designed then there would be no need for more meta-levels above it. But there might be cases where after the implementation of the meta-level module there arises an extra requirement to be met that has not been implemented. In such cases we could have another meta-level above the existing one to provide the required functionality. Thus, practically there would be no limitation on the number of meta-levels a system module could have at any time. It is also possible to share a meta-level module among one or more system modules if need be. This considerably reduces the accounting space required to keep information about the modules inside the kernel, thereby resulting in efficient use of memory resource.

B. Kernel support for Reflection

We have designed a minimal micro-kernel that implements a fast inter-process communication (IPC) mechanism, keeps all the process accounting information and provides support for reflection. All other system modules (eg. scheduler, memory manager, etc.) run as separate individual system processes. The kernel keeps track of only the most important information reified as indicated in MOP above and supplies this information to the system modules on request through the defined interface.

The interface consists of a set of common shared functions like *requestInfo()*: used to read the reified information from the kernel; *linkData()*: used to create a direct link with the underlying information (data) in the kernel to form a causal connection (two objects are said to be causally connected

to each other when a change initiated by one affects the other). Such a connection helps the system modules to directly inspect/analyse/modify the information without incurring any extra overhead on the system; *interceptCall()*: allows the meta-level code to register the details about the function call to be intercepted; *installCode()*: used by applications to install user-defined policies. The following two subsections describe example design of reflective scheduler and reflective VMM in brief.

C. Example 1: Reflective Scheduler

Figure 2 shows the structure of Reflective Scheduler. The base-level implements a default scheduling policy (eg. Round Robin, FIFO, etc.). The Reflective scheduling module (code in the meta-level) analyses the reified information and accordingly installs new scheduling policies as and when required. An Application can request to install a user-defined scheduling policy to schedule its child processes using the *installCode()* interface.

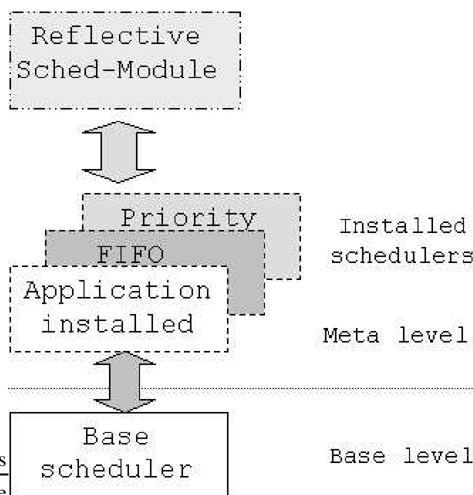

Fig. 2. Reflective Scheduler

D. Example 2: Reflective VMM

Figure 3 shows the structure of the Reflective Virtual Memory Manager (VMM). The design adheres to our generic reflective framework and hence is very similar to the reflective scheduler design described above. The base-level VMM would implement a default policy, while the reflective code in the meta-level affects any change in this default policy if required. Again, an application may request to install its user-defined VMM policy.

IV. Summary and Future Work

This paper outlines a generic module-based reflective OS framework to build a small real-time embedded OS that changes and adapts itself to the non-deterministic, dynamically changing demands of applications. We also present the design of a Reflective Scheduler and Reflective VMM. To evaluate our approach, further work has to be done to implement a

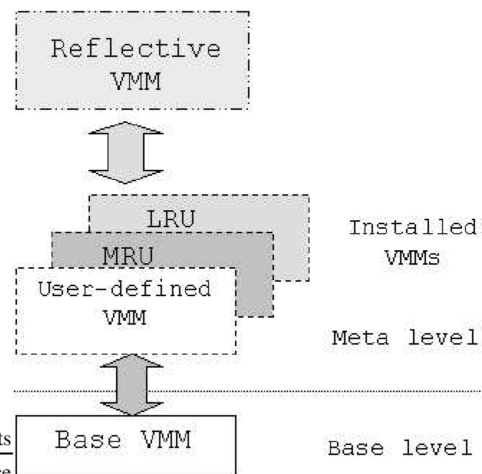

Fig. 3. Reflective Virtual Memory Manager

small reflective RTOS that has kernel support for reflection and has one or more reflective system modules. There is also a need to determine the effect trade-offs involved in providing such flexibility.

References

[1] B. Li and K. Nahrstedt, “Dynamic Reconfiguration for Complex Multimedia Applications,” in *ICMCS, Vol. 1*, 1999, pp. 165–170. [Online]. Available: citeseer.ist.psu.edu/article/li99dynamic.html

[2] D. Rosu, K. Schwan, S. Yalamanchili, and R. Jha, “On Adaptive Resource Allocation for Complex Real-Time Applications,” Georgia Institute of Technology, Tech. Rep. GIT-CC-97-26, 1997. [Online]. Available: citeseer.ist.psu.edu/article/rosu97adaptive.html

[3] Y. Yokote, “The Apertos Reflective Operating System: The Concept and Its Implementation,” in *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 1992, pp. 414–434.

[4] G. Hamilton and P. Kougiouris, “The Spring Nucleus: A Microkernel for Objects,” in *In Proceedings of the USENIX Summer Conference*, 1993, pp. 147–159.

[5] S. Son, *Advances in Real-Time Systems*. Prentice-Hall, 1994.

[6] Patrick Rogers, “Software Fault Tolerance, Reflection and the Ada Programming Language,” Ph.D. dissertation, University of York, UK, October 2003.

[7] J. Malenfant, M. Jaques, and F.-N. Demers, “A Tutorial on Behavioral Reflection and its Implementation,” in *Proceedings of the Reflection 96 Conference, Gregor Kiczales, editor, pp. 1-20, San Francisco, California, USA*, April 1996.