POSIX Trace Based Behavioural Reflection

Filipe Valpereiro
L. Miguel Pinho

Ada-Europe 2006
Porto, Portugal
06 July 2006

www.hurray.isep.ipp.pt

Summary

- Motivation
- Proposal
- Reflection
- POSIX trace
- Architecture details
- Implementation details
- Experiments and evaluation
- Conclusions

Motivation

- Traditional RTOS
  - Designed to support a generic real-time environment
  - Assumptions are made on the tasks characteristics, resource utilization requirements and platform
  - Traditional RTOS limits the quality of services offered to the end-user
- Modern real-time applications
  - Resource utilisation patterns vary considerably
  - User interaction is usually unpredictable
  - Soft real-time applications usually presents indeterminist behaviour caused by unpredictable inputs

Motivation

- Soft real-time applications
  - Inherently dynamics, requiring adaptive resource strategies to maximize system throughput
  - Typically running on hardware and energy constraints
  - Global user experience and quality perception is affected by
    - Service quality
    - Response time
    - Usability

Motivation

- Developing new applications in traditional RTOS
  - Reusing legacy application code
  - Increasing system throughput
  - To validate the application behaviour
  - To test and debug the application
    - Under unpredictable inputs
    - Under deployment
  - To decouple the adaptive behaviour from the application

Proposal

- Reuse existing RTOS
  - Taking advantage of well know systems
  - Reusing legacy application code
  - To address the lack of adaptable behaviour

- Use the POSIX trace mechanism
  - Can be use to test and debug an application
  - Can be used after deploy (low overhead)
  - Can be used remotely to monitor an application

Proposal

- To support reflection on “static” RTOS
  - Allowing soft real-time applications to change behaviour
  - To separate the application development from the development of system state analysis mechanisms
  - To capture the system state
    - Creating a “conscious memory” of the system state
    - Capturing the resource utilization history
    - To be able to query the system state
  - To be able to apply spatial vs. temporal reflection
    - Choosing when to reflect
    - Choosing what to reflect

Proposal

- Allow applications to change their own behaviour
  - To become perceptive of the system's current state
  - To specify a QoS policy using a specification interface
  - Or using their own QoS manager
- Test different adaptive strategies
  - Decouple adaptive code from application
  - Several applications can reuse the same strategy
  - We can test a particular strategy under some input patterns

Behavioural Reflection

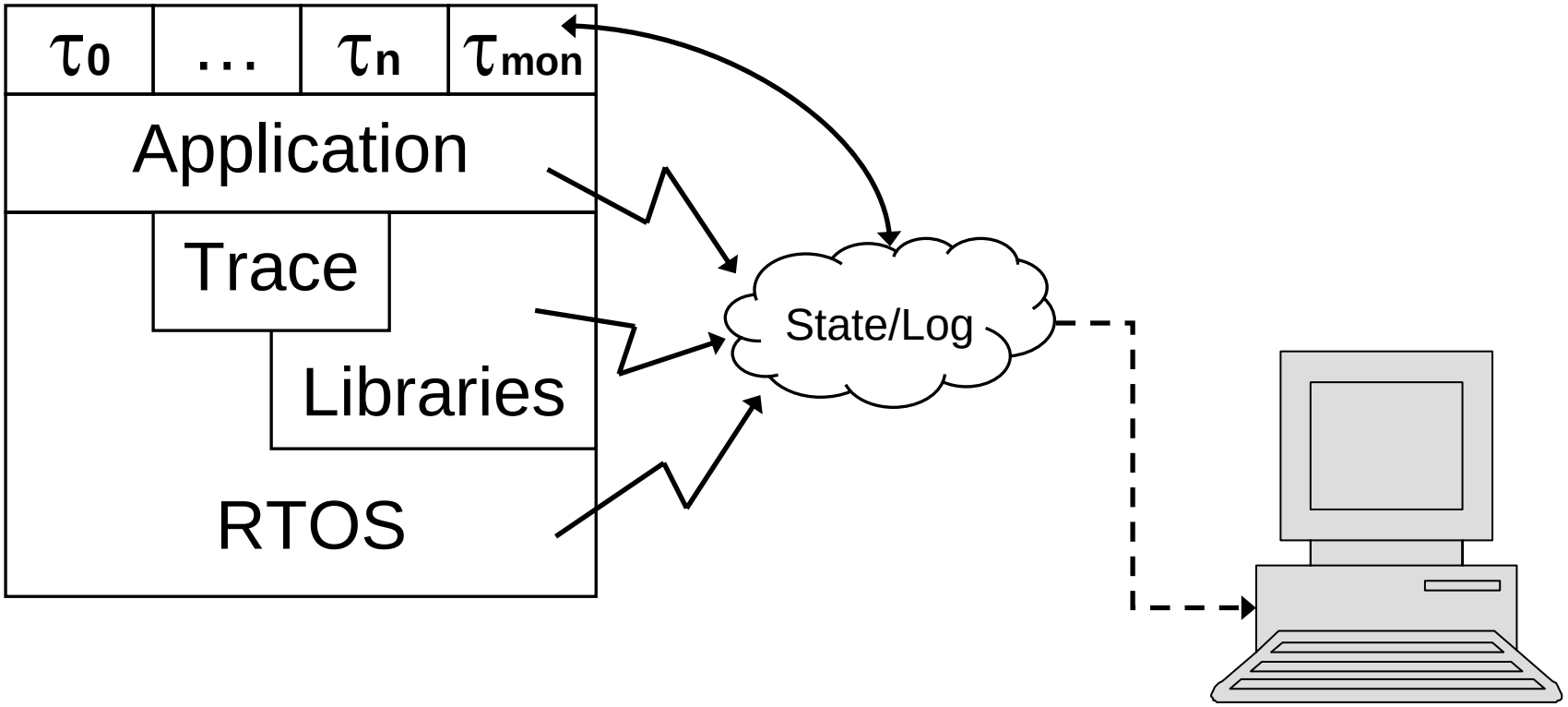
- Reflection allow us to
  - Become aware of our own behaviour
  - Modify system behaviour at run-time
  - Decouple behavioural code from application code

- Behavioural reflection allow us to
  - Capture the system state as a whole
  - To observe a particular behaviour
  - To enforce a specific behaviour

POSIX Trace Mechanism

- Compatible with any POSIX RTOS
  - Can be implemented in the MRSP profile with a bounded computation time
  - Low overhead (code size and execution time)
  - Can carry any amount of data
  - Flexibility and versatility, can be used in
    - Online debug
    - Post-mortem analysis
    - System metrics
    - Tests and validation

Architecture overview



Architecture details

- Modular POSIX trace implementation
  - Bounded execution time allows WCET analysis
  - Preserves the trace semantics
  - Components can be selected at compile time
    - Based on the application requirements
    - Individually selected by the programmer
  - Low code footprint and execution time
  - Can be used in the MRSP profile for embedded systems
  - Allows applications to send trace events

Architecture details

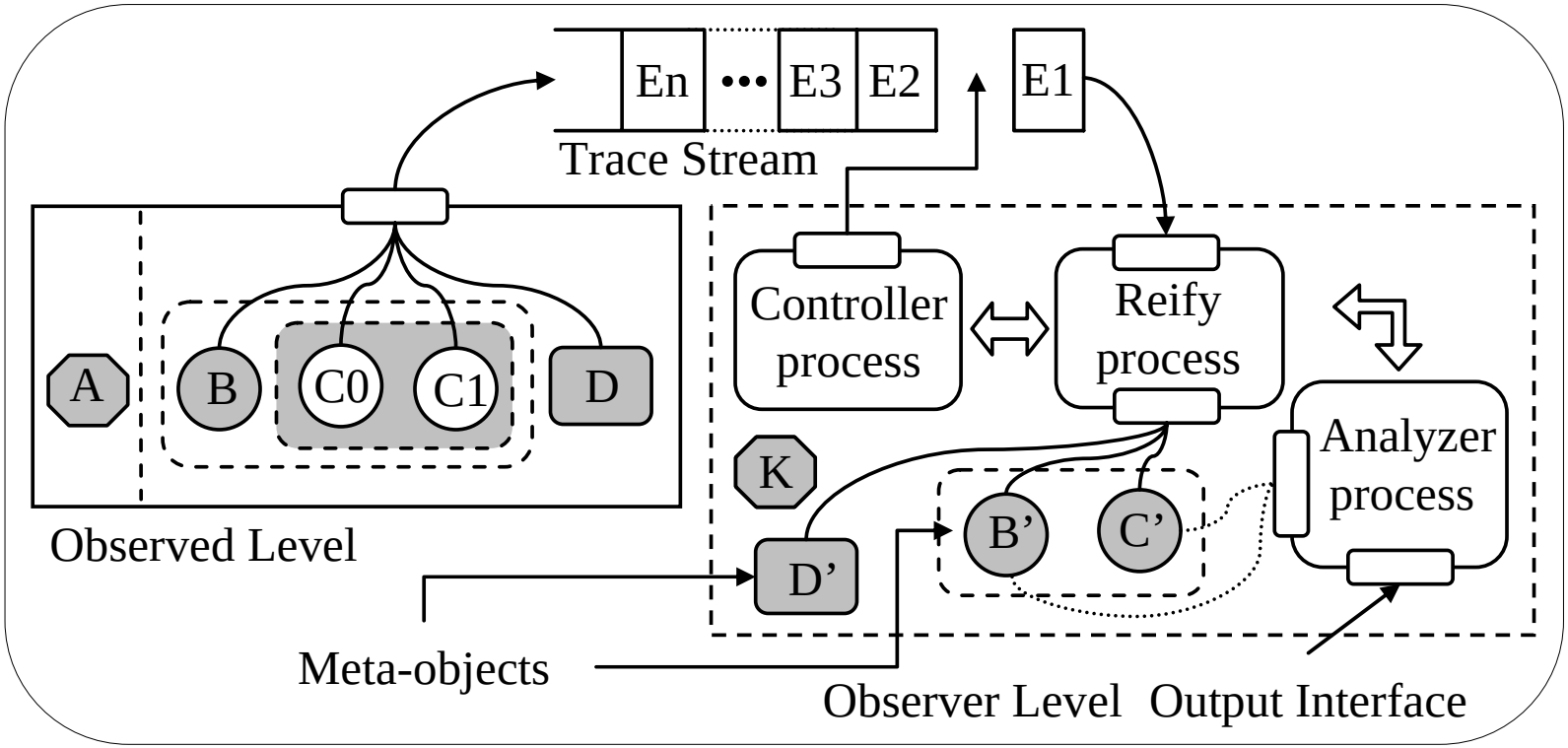
- Reflection framework
  - Build on top of the POSIX trace mechanism
  - Extending the roles of the Controller and Analyser process
  - Introducing a new process to Reify data
  - Defining two main data types to carry reflected data
    - A fixed size envelop for small data sizes
    - A multiple-size envelop for larger data sizes
  - Traced events are grouped into sets of functionality
    - Selection of trace events can be determined at compile time
    - Events can be filtered, offering a fine-grained control

Architecture details

- Events are mapped into meta-objects
  - A map links sets of events into meta-objects definitions
  - A dynamic map ensures that new instances of a meta-object are updated with the matching object data
- Access is performed through a protected interface
  - Meta-objects access must be consistent
    - Writing data must be an atomic operation
    - Reading an object can be done by getting a copy
    - The protected interface for an object can be extended

Architecture details

- Merging the POSIX trace with a reflective architecture



Implementation details

- Mutex event definition

```
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;
```

Implementation details

- Meta-objects definition

```
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;
```

Implementation details

- Meta-objects protected interface

```
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;
    -- Interface can be extended

private
        Mutex : Meta_Mutex;
end Meta_Mutex_Access;
```

Size overhead

- Evaluation of code size overhead
  - Tests where made on a Pentium-III at 930Mhz
  - Time values where taken directly from the TSC
  - Operations where repeated for 5000 iterations
  - The trace mechanism was configured with sufficient space for all the events generated during the experiment
  - All data was written to a simple structure in memory
  - Data was read from a serial line

Size overhead

- Size overhead for the trace mechanism
  - Approximately 10% with the trace unit (all components)
  - Less then 300 bytes for each trace event

Description	Size in Bytes
Simple procedure (sum of one integer)	480
Simple procedure with a single trace event	780
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

Execution time

- Measuring the performance impact
  - Tests where made on a Pentium-III at 930Mhz
  - Time values where taken directly from the TSC
  - Used four simultaneous tasks with different characteristics to create enough scheduler activity
  - Operations where repeated for 5000 iterations
  - The trace mechanism was configured with sufficient space for all the events generated during the experiment
  - All data was written to a simple structure in memory
  - Data was read from a serial line

Execution time

- Execution overhead for the trace mechanism
  - Approximately 0,7 -- 1 µs for each trace event
  - Using envelops can add extra 0,5 µs

				Mean	
Function	**Trace**	**Min**	**Max**	**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

Conclusions

- POSIX trace based reflection framework
  - Low overhead, approximately **10%** in size
  - Lightweight event traces, less then **300** bytes
  - Fast trace time, approximately **0,7 – 1 µs**
  - Exposing the system state without compromise
  - Allows adaptive behaviours to be modelled on top of it.

The End

Thank You