On-Chip Monitoring of Single- and Multiprocessor Hardware Real-Time Operating Systems

Mohammed El Shobaki
Department of Computer Engineering
Mälardalen University, Västeras, Sweden
mei@mdh.se

Abstract

This paper presents a novel hardware monitoring system that gives non-intrusive observability into the execution of hardware-accelerated Real-Time Operating Systems.

Monitoring is a necessity for testing, debugging and performance evaluations of real-time computer systems. Most research into monitoring of real-time systems have been devoted to minimising the execution interference imposed by the monitor. One approach to this has been the use of hardware support to extract software execution traces by probing the external processor (or system) busses.

However, the use of cache memories on various levels, and the increased integration of system components on-chip (SoCs) in addition to limited chip-package pins, severely obstructs traditional hardware monitors from probing processor signals and busses. For real-time systems built on these premises there is a need to access execution information residing on-chip, as well as to avoid interference with the system's execution behaviour.

In this paper we present an integrated solution to on-chip monitoring of system-level events in a real-time system. The monitor, called MAMon[1]*, probes a hardware-based Real-Time Kernel using a Probe Unit integrated as an IP-block. This component detects and collects events regarding process' execution, communication, synchronisation, and I/O interrupt activities. Collected events are timestamped and transferred to a separate computer system hosting an event database and a set of monitoring application tools. We describe the monitor architecture, the implementation of a prototype, and an evaluation of its use.*

1 Introduction

Run-time observability in embedded system architectures is a requirement for testing, debugging, and for validating design assumptions made about the behaviour of the system and its environment. The classical approach to run-time observability is to apply *monitoring*, i.e. the process of detecting, collecting, and interpreting run-time information regarding the system's execution behaviour. In monitoring real-time systems an important aspect is to minimise, or completely avoid, the intrusiveness of the monitor on the system's timing and exeecution properties. Failing to handle monitor intrusivity may lead to *probe effects* which cause non-deterministic behaviour in programs with race conditions and poor synchronisation [6, 13].

The research efforts on real-time monitoring has over the past decade been mostly devoted to dealing with probe effects and timing interference in various applications of monitoring [17, 18, 2, 8, 7]. Hence, a wide spectrum of monitoring approaches have been proposed, ranging from pure software techniques [17, 8] to the use of special hardware support [12, 18, 7]. *Software monitoring systems* offer the cheapest and most flexible solution where a common technique is to insert instrumentation code at interesting points in the target software. When the instrumentation code is executed the monitoring process is triggered and information of interest is captured into trace buffers in target system memory. The drawbacks of instrumentation is the utilisation of target resources such as memory space and processor execution time. Moreover, to avoid probe effects, the instrumentation code must be kept in the deployed software or be compensated for in the real-time schedulability analysis [17] - with both alternatives resulting in performance penalties. *Hardware monitoring systems* on the other hand use special hardware to passively probe the target's physical busses, such as the processor and system busses, and collect information of interest without interfering with the target's execution. The main advantage with hardware monitoring is that probe effects can be completely avoided. The disadvantages are the dependancy on the target architecture and its related costs. *Hybrid monitoring* uses a combination of software and hardware monitoring and is typically used to reduce the impact of software instrumentation alone [7].

[1] Multiprocess Application Monitor

With today's highly integrated hardware, encapsulating complete systems on a chip (SoC), the traditional hardware monitors are facing severe difficulties. Processor cores, I/O components, cache memories, and even standard memory, are all integrated on the same chip. Given also that chip packages can be obstructive (as in Ball-Grid Array packages) and have limited pins, it has become almost impossible for external hardware to probe internal signals. For real-time systems built on these premises there is a need to access execution information residing on-chip, as well as to avoid interference with the system's execution behaviour.

In this paper we present an architecture for on-chip monitoring of single- and multiprocessor real-time systems that are based on hardware-accelerated operating systems [1, 10, 14, 15]. The monitor, called MAMon, probes a hardware-implemented Real-Time Kernel (RTK) using a Probe Unit integrated as an IP-block at the VHDL-level. A hardware RTK implements traditional (software) RTOS functions, such as scheduling algorithms, process management and communication, in hardware [1, 11]. Operating at the system-level the Integrated Probe Unit detects and collects events regarding process' execution, communication, synchronisation, and I/O interrupt activities. The collected events are timestamped with the resolution of the system clock frequency (10 MHz = 100ns) and then transferred, via a high-speed parallel port link, to a separate host computer system. At the host the events are stored in a database which constitutes the heart of a monitoring application framework featuring event analysis and debugging (searching, filtering, and graphing), performance evaluations, and more. Monitoring occur mainly at the system-level, but lower abstraction-levels are supported too by allowing instrumentation code to write to dedicated *probe registers* in the monitor hardware. This opportunity would, however, classify the monitor as a hybrid system, and thus requires a perturbation analysis of the software instrumentation.

The main contributions of this work are the ideas on system-level monitoring of hardware RTKs, on-chip rather than by probing external processor busses. We believe that on-chip monitoring support will be required in future development of real-time systems, especially those based on SoCs.

The paper is organised as follows. Section 2 describes a multiprocessor system concept based on a hardware-accelerated RTOS. This system will be the target platform in further discussions on our proposed monitor. Section 3 describes the monitor architecture for a generic target RTOS that utilise hardware RTKs. An overview of the system and a detailed description is given for the Integrated Probe Unit, the host-based monitoring application framework, and the communication interface in between. Section 4 describes an FPGA prototype implementation of the monitor for a multiprocessor system with 3 PowerPC-750 processors. An evaluation of the prototype is given in Section 5, and finally, Section 6 summarises the paper with some concluding remarks and directions on future work.

2 A Real-Time Multiprocessor Architecture - SARA

The Scalable Architecture for Real-Time Applications (SARA [10, 9]) is a research platform for real-time multiprocessor computing systems. The two main research objectives with SARA are: 1) to provide a hardware architecture that behaves predictably to the real-time application, and 2) to provide a flexible system architecture that simplifies processor (performance) scalability. In attaining these design goals, a SARA architecture is based on a *hardware-accelerated* RTOS. The hardware support comes from a co-processor called RTU (Real-Time Unit [1, 11]) which provides the RTOS with kernel-level services such as process/task scheduling, synchronisation and communication, see Section 2.1 for more details.

Figure 1 shows the hardware view of a SARA system which includes one or more processor nodes, a communication network (bus), and the RTU as a shared software process scheduler. This view is the same whether the hardware is implemented on a multi-board computer system, such as VME or CompactPCI-based [16] systems, or as a SoC. A SARA implementation on a CompactPCI system is described in Section 2.2, and in [3] a SoC implementation is proposed.

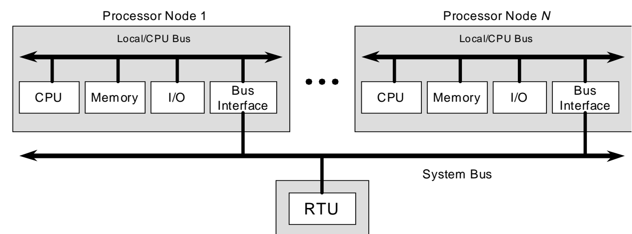

Figure 1. Hardware view of a SARA system

The software, which is partitioned onto each node in SARA, includes a *minimal* RTOS which mainly interfaces to the RTU, and a collection of processes which are scheduled to execute on one or more processor node(s). To simplify the programming model, hardware is abstracted to the software so that processes need not be bound to a certain processor, and process migration is allowed.

Communication between processes takes part over a virtual bus (VCB) which spans over all processor nodes. The

VCB programming model, shown in Figure 2, uses the concept of *virtual slots* which processes must attach to in order to send and receive messages. Moreover, synchronised sending, broadcasting and multicasting of messages is supported.

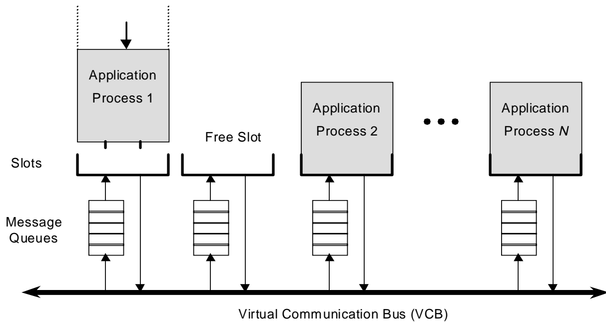

Figure 2. Process communication model in SARA

2.1 RTU - Real-Time Kernel in Hardware

Hardware support to increase performance and predictability in real-time operating systems have been proposed in [14, 1, 11, 15]. The Real-Time Unit, RTU by Lindh et. al. [1, 11], is a co-processor with support for real-time kernel services such as process scheduling and management (create, terminate, etc), inter-process communication (IPC, message send/receive), synchronisation (semaphores), and I/O interrupt handling. The RTU, which supports scheduling of both single- and multiprocessor systems, runs in parallel with the target system's processor(s). Processors interface with the RTU by memory-mapping to its processor-independant register interface. Via this interface, *service-calls* are placed by writing to dedicated service-call registers.

Figure 3 shows the basic building blocks of the RTU. The core part is the scheduler which schedules processes on-line (pre-emptive priority scheme) and dispatches process execution. Connected in between the scheduler and the programming/bus interface, a set of functional modules implements the various services in the RTU, such as management of the scheduler, IPC, semaphores, clock and timer management. Process context-switching is notified to CPUs using interrupts causing handlers in software to perform the actual context-switching.

2.2 A SARA CompactPCI System

Desribed in [9] is a SARA implementation on a CompactPCI (CPCI [16]) computer system. A CPCI system

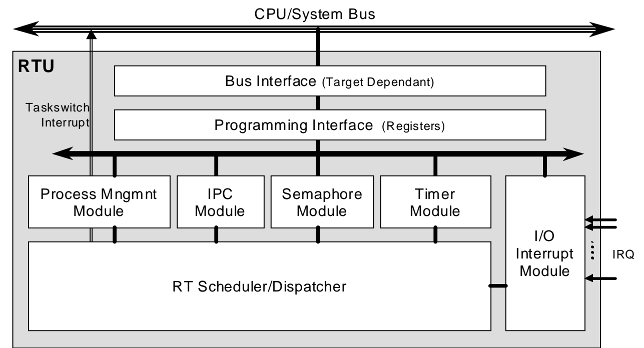

Figure 3. Basic building blocks of the RTU

has 8 slots where CPU-boards can be inserted. The first slot, slot 0, is dedicated as the *system slot* which requires that the CPU-board on that slot handles arbitration and clock distribution on the CPCI backplane. Figure 4 shows the current SARA implementation with 3 PowerPC-750 CPU-boards. The RTU, which resides on a PMC-board (PCI Mezzanine Card [16]), is attached to the system board from where it can communicate with all CPUs in the system (see also Figure 10).

All CPU-boards have local memory and a local PCI-bus. Processes that are allowed to migrate between CPUs require global memory to hold their Process Control Blocks (PCB). This global memory can be defined out of local memories on all CPU-boards. Currently, global memory is allocated at the system board only.

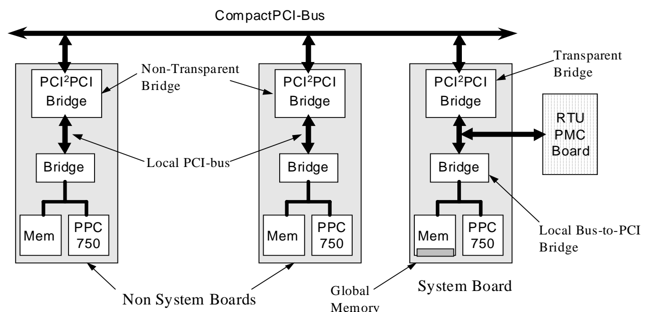

Figure 4. A SARA system based on CompactPCI-board computers [9]

3 A Monitoring System for Hardware-Accelerated Real-Time Operating Systems

3.1 Overview

The proposed monitoring system aims at providing means for on-chip observability at the system-level in single- and multiprocessor real-time systems. The monitor, which we call MAMon (short for Multiprocess Application Monitor), is based on the following assumptions about the monitored target system:

- The target's RTOS is supported by a hardware Real-Time Kernel (RTK), like the RTU or a similar component as described in Section 2.1.
- The RTK holds information about the state of every process in the system, inter-process communication activities, timers, interrupts, etc.
- The RTK must allow external access to internal (vital) signals and data. Since the RTU was available to us as a *soft* IP-component (HDL source), access to all signals and data is straightforward in VHDL.

The architecture of MAMon, shown in Figure 5, consists of two major parts: the *Integrated Probe Unit* (IPU, Section 3.2) which is the hardware part of MAMon, and a *host* computer system. Like an IP-block, the IPU is integrated with the hardware RTK at the VHDL level. In a SoC the IPU may also be connected to processor busses, I/O components, and other hardware logic in order to extract information at various levels of abstraction. In the synthesized hardware (e.g. ASIC or FPGA implementation), the IPU monitors the RTK in run-time, and collects events regarding the system-level behaviour of the real-time application. The collected events are timestamped each and then transferred over a high-speed parallel communication port to the host computer where they are stored in a database. In an integrated framework (Section 3.5) the database serves as an event repository which can be used by monitoring application tools to provide event-based debugging, performance analysis, assessment of design constraints, etc.

In certain cases there is a need to generate events from software, for instance, to mark code checkpoints (flags), or to report register and memory contents required for lower-level analysis. Such events are produced by inserting software instructions (software probes) that writes to a dedicated register connected in between the IPU and the system/processor bus.

3.2 The Integrated Probe Unit

Figure 6 shows a block-diagram of the IPU's internal organisation.

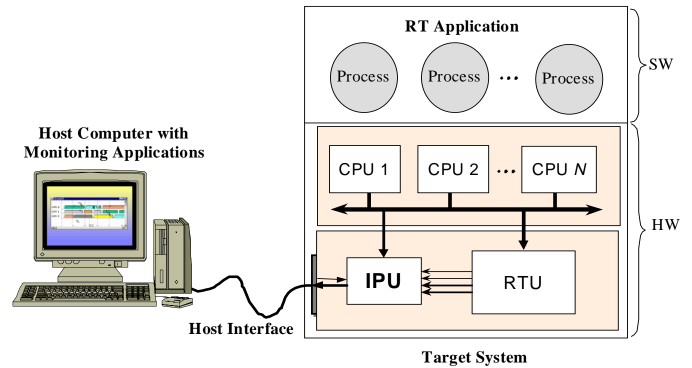

Figure 5. Overview of MAMon

3.2.1 Event Detector

The *Event Detector*, seen in top of Figure 6, is responsible for the detection of events and for collecting event samples. Detection of events is performed by comparing input event signals with pre-defined event *condition expressions*. The input signals are hard-wired (in HDL) from selected points in the RTU. Over-sampling of the input signals is not required because the pre-defined events will never occur simultaneously. When an event is detected, a sample is collected and stored immediately along with a timestamp in the local FIFO buffer.

An event-sample comprises the event-type, the timestamp, and an event-defined parameter field, see Figure 7. The parameter field is used to store additional information about an event. For instance, for a *task-switch event* to be sufficiently informative, the parameter field contains the new task's id-number and the CPU it was scheduled to run on. For a *send message event*, the parameter field may contain the id-number of the receiving task and the pointer to the message, and so on.

The timestamp comes from the 48-bit Timer module which denotes the absolute system time given in nanoseconds. The Timer is updated at the resolution of the system clock frequency.

To support detection of software probes, the IPU provides a simple interface that can be used by external decode logic. A single strobe line is all that is required to indicate a software write-access, and to signal the IPU to latch incoming data.

3.2.2 FIFO

The FIFO buffer is needed during transient over-loads of events while the host computer is busy reading event data over the parallel port. FIFO buffer dimensioning is described in Section 3.4. The FIFO is built onto on-chip dual-ported RAM with parameterizable (generic) size and port

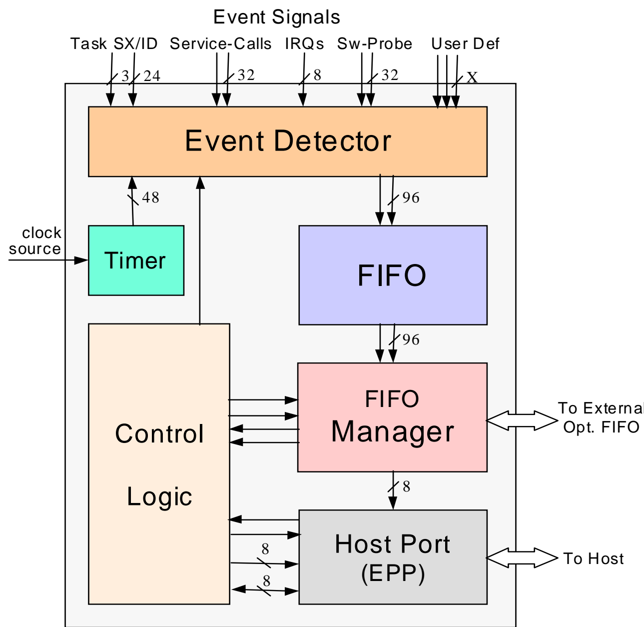

Figure 6. The Integrated Probe Unit

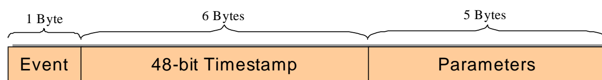

Figure 7. The event sample format

width. Moreover, it has signals that indicate when the buffer becomes full, or half-full.

3.2.3 FIFO Manager

The FIFO Manager mainly provides a byte-wide interface for the Host Port to read event data from the FIFO. In circumstances when the required FIFO size is not feasible on-chip, e.g. in FPGA implementations, the FIFO Manager can also be used to extend the FIFO using external RAM. In this case, the FIFO Manager will also take care of flushing the contents of the on-chip FIFO out to the external RAM. The option to use external RAM can be set via the Host Port.

3.2.4 Host Port

The Host Port is responsible for taking care of host-inititated acquisition of event data. It also provides the host with a programming interface to read the status of the IPU and to control its behaviour (the Control Logic in Figure 6).

Since FIFO buffering is limited it is important that event samples are transferred to the host with a guaranteed high communication bandwidth. Therefore, the Host Port implements the bi-directional *Enhanced Parallel Port* protocol (EPP 1.9 [4]). In theory the EPP supports transfer rates up to 2MB/s (approx. 160k events/s).

To indicate availability of events in the FIFO the Host Port can be programmed to generate an interrupt to the host computer. When this feature is enabled, it can be set into one of three modes:

- Interrupt whenever new events arrive
- Interrupt when the FIFO buffer is half-full
- Interrupt when the FIFO buffer is full

The first two modes are useful when continous monitoring is desired. The third mode is more useful if the IPU is set to sample from a given command until the FIFO becomes full, and then stop. Providing the ability to choose the interrupt mode gives a customised solution that best suits the capabilities of the host computer performance, the tools, or the user. When the interrupt function is disabled, events can still be acquired in *polled* mode.

3.3 Events

Currently the Event Detector supports detection of four types of events; *Taskswitches, Service-Calls, Interrupts, and Software Probes*. The conditions for these events are hardcoded in the Event Detector. Therefore, the size of the Event Detector logic is linearly proportional to the number of supported events. Given below is a description of each event-type; its condition(s) and related data to be collected.

3.3.1 Taskswitch events

For a taskswitch to be detected, the IPU is connected directly to the scheduler module in the RTU. Whenever a taskswitch is to occur, the scheduler asserts an interrupt signal and indicates the next task's id along with the CPU it is to run on. Upon detection of this event the following packet is produced.

TSW_EVT	*TIMESTAMP*	*CPU_NR*	–	*TASK_ID*
1B	6B	1B	2B	2B

3.3.2 Service-Call events

A service-call is detected whenever software writes to a *Service-Call Register* in the RTU, i.e. to indicate a service-request. For each CPU in the system there exists one Service-Call Register in the RTU's register-interface. These registers are connected to the IPU as well. An event of this type produces the following packet.

SVC_EVT	*TIMESTAMP*	*CPU_NR*	*REG_VALUE*
1B	6B	1B	4B

3.3.3 Interrupt events

The RTU supports handling of external interrupts by associating tasks with the interrupts. When an interrupt is asserted the RTU's interrupt module tells the scheduler to start the associated task. To detect this event, the interrupt lines are connected to the IPU along with the associated tasks' id. An interrupt event produce the follwing packet.

IRQ_EVT	*TIMESTAMP*	*IRQ_NR*	–	*TASK_ID*
1B	6B	1B	2B	2B

3.3.4 Software-Probe events

A software probe is similar to a service-call request in that software writes to registers in the RTU. However, these register are dedicated to MAMon and are connected only to the IPU. Values written to these registers can be used for profiling, measurements and debugging purposes. A software probe event produce the following packet.

SWP_EVT	*TIMESTAMP*	*REG_NR*	*REG_VALUE*
1B	6B	1B	4B

3.4 Performance and FIFO Dimensioning

3.4.1 Input rate

The rate at which the EDU detects and stores events in the FIFO buffer depends on the system freqency; the higher frequency, the higher the input rate to the buffer. The EDU requires 2 clock cycles to store one event in the buffer. Since the currently supported events (see previous section) cannot occur consecutively within 2 clock cycles, no events will be missed. This implies that the worst condition corresponds to an event occuring every 2 clock cycles. With a clock freqency of 10 MHz, the input rate is 1 occurence per 200 ns. It is also assumed that the input rate follows a Poisson statistical distribution, as described in [12].

3.4.2 Output rate

The output rate for emptying the event FIFO buffer is largerly determined by the performance of the host interface communication link, the EPP port in this case. In theory, EPP supports transfer rates up to 2 MB/s [4]. When using a PC running Linux as the MAMon host computer, and a standard bi-directional parallel port interface, we could reach a maximum transfer rate of 1.3 MB/s. We therefore estimate the time to transfer one event-packet to 10 μs, assuming a transfer involves 13 byte reads; 12 for event data plus 1 for reading the IPU's status register (to check for event availability). The rate at which events are stored in the database is not considered since it is much faster than the communication bandwidth.

3.4.3 FIFO buffer dimension

To eliminate buffer overflow the FIFO buffer must be large enough to handle the worst case input flow while the MAMon host system is busy flushing the buffer. By applying a queueing analysis (adopted from [12]) we can estimate the required buffer size. This analysis assumes two facts: 1) events can arrive concurrently while the buffer is flushed, and 2) that the MAMon host system starts flushing the buffer at latest when the buffer is *half* full. The first assumption is fulfilled as the FIFO buffer is dual-ported. The second assumption requires that the host either polls continously for new events, or uses the half-full-buffer interrupt mode.

Let k be half the FIFO buffer size, R the mean input rate, and T the transfer time per event-packet. Assuming that the input rate follows the Poisson distribution, then, $\mathcal{P}(k)$ is defined as the probability that the buffer has k arrivals in time T (i.e. that half the buffer fills up within T). The probability function is,

$$\mathcal{P}(k) = \frac{(RT)^k}{k!} e^{-RT}$$

In determining the total buffer size ($2k$) it is assumed that the probability of filling up half the buffer is at a minimum, for instance 0.5%, given that it takes kT time to flush the first buffer half. That is,

$$\mathcal{P}(k) + \mathcal{P}(k+1) + \mathcal{P}(k+2) + \ldots < 0.005$$

Using $T = 10\mu s$ and $R = 1/200ns$, gives 70 as the best value for k. Hence, the FIFO buffer must handle no less than 140 events.

3.5 The Monitoring Application Framework

To provide the user with a platform for event-based performance analysis and debugging, we have developed an integrated framework for monitoring applications. Our goal is not to develop a complete monitoring envrionment, but to show the capabilities with our hardware monitoring approach.

The framework is developed mainly in Java and uses an SQL database to store the event histories. Figure 8 shows this framework's architecture. At the bottom lies the IPU interface module which is mainly used to transfer event samples from the IPU into the SQL database, and to control the behaviour of the IPU. The IPU interface module runs as a separate process, but is controlled from the Java framework via the Java Native method Interface (JNI). JNI is required becasue this module is written in C/C++ as it is strongly dependant on the underlying architecture for communicating with the EPP interface. The SQL database is

run by the MySQL DBMS (www.mysql.com). We choose MySQL for its speed and capabilities to handle our amounts of events, and because it is free for educational purposes. The database and IPU interface constitutes the base of our framework.

The Java application forms the actual framework which provides an integrated interface to control the monitoring process, to collect events into the database, and to query the event database in various ways. Using this interface we can now easily implement application specific monitoring tools that are plugged into the framework.

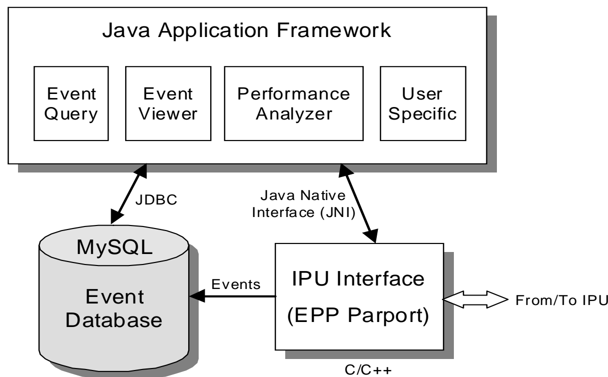

Figure 8. MAMon's Application Framework

An example monitoring tool is the *Event Viewer* that displays portions of the event history. Such a tool can be useful for finding and analysing erroneous execution patterns. The Event Viewer tool, shown in Figure 9, collects events from the database, and displays them along a time-line. Apart from standard functions such as zooming and scrolling, there is also support for time-markers that are used for timing measurements, and search-markers that can be used to locate event conditions and patterns.

Another example tool is the *Event Query* tool (also shown in Figure 9) which provides a user-friendly interface to query the database for event conditions and execution patterns. The output from the query may be output textually to screen or to a file, or graphically by linking its results with the Event Viewer tool.

The event database is also suitable for other post-analysis, such as extraction of performance indexes for use in diagrams and histograms showing task's execution time, processor utilisation, IPC frequencies, interrupt response times, etc.

4 Physical Hardware Implementation

In this section we present some implementation details on a prototype of MAMon for a SARA CompactPCI system (described in Section 2.2).

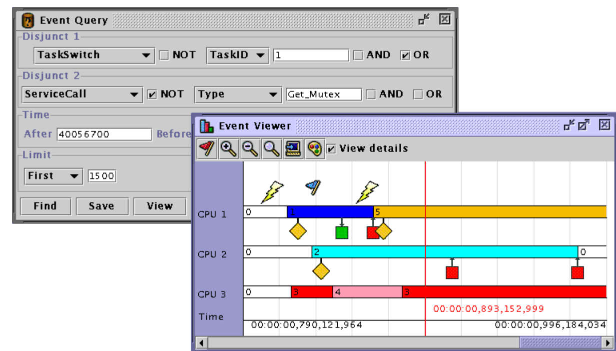

Figure 9. Screenshot of Event Query & Event Viewer tools

4.1 The Hardware Prototype

The IPU is implemented together with the Real-Time Unit[2] on a Xilinx Virtex-1000 FPGA [5]. All modules are designed in VHDL which is either textually entered or automatically generated from state and block diagrams drawn in Renoir (graphical hardware design tool, by Mentor Graphics). The FPGA is mounted on a PMC-board and connects to the SARA-system via a PCI bus-interface chip (PLX-bridge), see Figure 10. The host system of MAMon connects to the parallel-port connector (left in Figure 10) with a IEEE-1284C cable [4]. Because RAM-cells are limited inside the FPGA, a 128kB SRAM module (on backside of the board) is used to extend the internal event FIFO buffer.

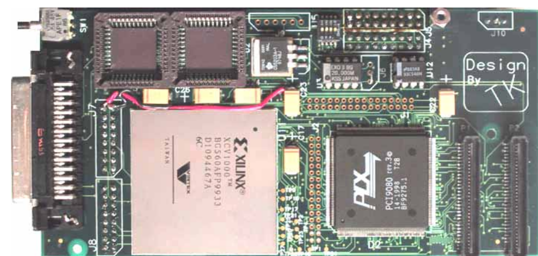

Figure 10. PMC-board with a Xilinx Virtex-1000 FPGA and PLX-bridge [9]

4.2 Physical Footprint

Table 1 shows some area figures from a synthesis to a Xilinx Virtex-1000 target. Although these figures are target-specific, they could serve as a reference for estimating the

[2]The RTU in this prototype was synthesised to support 128 tasks with 64 priorities.

equivalent area requirements for other silicon technologies. Xilinx's FPGA technology can be described as matrices of Configurable Logic Blocks (CLBs) where each CLB contains two Function Generators (FGs) and two D-Flip-Flops. According to Xilinx, a Virtex-1000 FPGA has a capacity of "1 million gates" [5].

Resource	*IPU*	*RTU*	*Avail*	*Utilisation*
CLB Slices	181	3276	12288	28.13%
FGs.	361	6553	24576	28.13%
Flip-Flops	254	2580	24576	11.53%

Table 1. Area figures for a Xilinx V1000 FPGA

As shown in the table, the IPU makes up only 5% of the total number of CLBs for the whole design. What is not shown in the figures, however, is the area costs for the event FIFO buffer. This is becasue FIFO memory was mapped onto RAM cells built-in the FPGA (called Block-Select RAM). However, calculating the area costs for memory is straightforward in many technologies. Currently the event FIFO buffer can store 16 events where each is 96 bits wide (12 bytes), i.e. 1536 bits are required.

The host interface port, currently implemented as an EPP parallel port, requires 15 I/O pins; 8 for data, and 7 for control. On a chip with limited pinouts it could be preferrable to multiplex these pins with other I/O, or choose an interface with less ports, e.g. a synchronous serial port. Moreover, an additional 29 I/O pins are used to interface with the external SRAM used to extend the event FIFO buffer. As this memory is optional, this overhead can be removed if the internal event FIFO can be fitted on-chip.

5 Prototype Evaluation

The prototype system was validated in a number of small tests on both single- and multiprocessor targets. With no intrusion on neither the execution or the timing behaviour of the target system the prototype was able to monitor task-switches, service-calls, and external interrupts. Monitoring of software probes (hybrid monitoring) was also accomplished but with a minimal intrusion equal to the delay of a 32-bit PCI-transfer per probe (@33MHz = 30ns). To illustrate a proof of concept we present hereunder an example were we analyse a deadlock situation using the monitor.

Example: Deadlock Detection

The program in this example illustrate a typical situation where two tasks need to synchronise before proceeding to a next step, in this case opening a pair of fuel valves. The deadlock occurs due to an in-planted synchronisation error between the two tasks T1 and T2 which execute on processors CPU1 and CPU2 respectively. Figure 11 shows the pseudo-code for the tasks. The tasks synchronise with mutual sending and receiving of messages over the VCB (described in Section 2). Task T1 uses the blocking *sendwait()* call to send a message and wait for the other party to send as well. For a proper synchronisation, task T2 should also call sendwait(), but due to a programming error the *receive()* call was used instead. This results in a deadlock since T1 cannot resume, and T2 will get blocked the second time it calls receive().

```
Global VCB Slot_T1;

Task T1() {
   Slot_T1 = Connect_to_VCB();
   LOOP {
      Compute_X;
      ...
      Slot_T1.sendwait(slot_T2);
      Open_valve1();
      ...
      Close_valve1();
   }
}
```

```
Global VCB Slot_T2;

Task T2() {
   Slot_T2 = Connect_to_VCB();
   LOOP {
      Compute_Y;
      ...
      Slot_T2.receive(); // Bug!
      Open_valve2();
      ...
      Close_valve2();
   }
}
```

Figure 11. Deadlock example in pseudo-code

To locate the erroneous bug, we first monitor the target system and collect the system-level events into the host database. The Event Query tool is then used to perform a filtered search in the event database. Using predicate disjuncts and conjuncts in the query we can easily find the first and last occurences of the tasks of interest. Figure 12 shows a text-dump from the query tool. Rows 1-3 shows that T1 starts and attempts to connect to the VCB. Rows 4-6 shows the similar sequence for T2. The sendwait() call in T1 is mapped to the VCB primitives *VCB_Put* and *VCB_Get* seen on rows 7-8. After that T1 gets blocked, the IDLE task starts on that processor (row 9). At row 10, T2 receives the message from T1 without blocking, and later when it attempts to receive again at row 11 it gets blocked too. The same sequence of events can also be depicted by the Event Viewer tool, see Figure 13. Horizontal bars indicate executing tasks, and the icons beneath indicate service-calls.

6 Conclusions

This paper has described a monitoring system and its implementation for non-intrusive monitoring of real-time systems. The monitoring system, called MAMon, integrates a probe component with a hardware Real-Time Kernel in order to non-intrusively detect and collect process-level events at the target system. Via a parallel communication link, the

##.	Event	Timestamp	CPU	Subtype/Parameters	
1.	**TaskSwitch**	00:00:00,815,083,600	CPU1	**T1**	
2.	ServiceCall	00:00:00,821,241,000	CPU1	VCB_Alloc	0x140101
3.	ServiceCall	00:00:00,821,255,200	CPU1	VCB_Open	0x020101
4.	**TaskSwitch**	00:00:00,828,930,200	CPU2	**T2**	
5.	ServiceCall	00:00:00,834,384,300	CPU2	VCB_Alloc	0x140202
6.	ServiceCall	00:00:00,834,395,000	CPU2	VCB_Open	0x020202
7.	ServiceCall	00:00:00,846,497,300	CPU1	VCB_Put	0x107102
8.	ServiceCall	00:00:00,864,018,800	CPU1	VCB_Get	0x10F501
9.	**TaskSwitch**	00:00:00,864,044,800	CPU1	**IDLE**	
10.	ServiceCall	00:00:00,908,777,500	CPU2	VCB_Get	0x108502
11.	ServiceCall	00:00:00,979,800,300	CPU2	VCB_Get	0x108402
12.	**TaskSwitch**	00:00:00,979,826,300	CPU2	**IDLE**	

Figure 12. Text-dump in the Event Query Tool

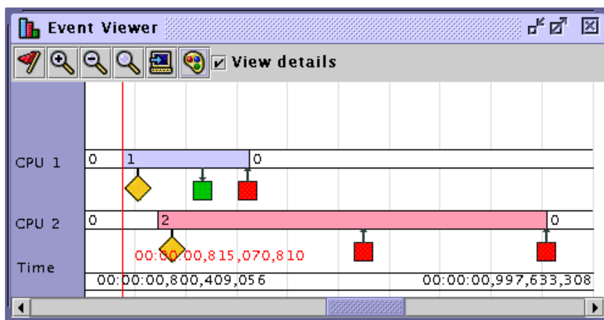

Figure 13. Deadlock seen in Event Viewer Tool

collected events are transferred to a host computer system where they are stored in a database. Built onto the database, a set of monitoring applications provides post analysis features such as event-debugging, profiling, and performance evaulations based on the collected events. While our approach is non-intrusive, it also overcomes the difficulties in extracting execution information residing on-chip of processors and in Systems-on-Chip (SoC). Monitoring occurs at full system speeds, both in single- and multiprocessor targets. Although the monitor's probe component is hardware dependant in that it is tightly coupled to a hardware real-time kernel, the solution is independant of target processor architectures and no software overhead is required.

The paper also describes the option to use MAMon as a hybrid monitoring system for monitoring at lower abstraction levels, e.g. functional and data levels. In this case instrumentation of the software is required and will introduce execution delays (although minimised).

To our knowledge, the proposed idea is novel and introduces a new alternative to monitoring, particularly useful in systems with hardware-accelerated real-time operating systems, and in SoCs. Future work include further validation of the MAMon concept for monitoring of real-case applications, and for systems built on SoC hardware.

References

[1] J. Adomat, J. Furunas, L. Lindh, and J. Starner. Real-time kernel in hardware rtu: A step towards deterministic and high performance real-time systems. In *8th Euromicro Workshop on Real-Time Systems*, LAquila, Italy, June 1996. EUROMICRO.

[2] S. E. Chodrow, F. Jahanian, and M. Donner. Run-time monitoring of real-time systems. In *Proceedings of the Real-Time Systems Symposium*, pages 74–83. IEEE, December 1991.

[3] M. Collin, R. Haukilahti, M. Nikitovic, and J. Adomat. Socrates - a multiprocessor soc in 40 days. In *Conference on Design, Automation and Test in Europe 2001, Designers Forum*, Munich, Germany, March 2001.

[4] Enhanced parallel port v. 1.9. IEEE 1284.

[5] Xilinx inc. http://www.xilinx.com/prs_rls/ibmpartner.htm. 2100 Logic Drive, San Jose, CA 95124-3400.

[6] J. Gait. A probe effect in concurrent programs. *Software - Practise and Experience*, 16(3):225–233, March 1986.

[7] D. Haban and D. Wybranietz. A hybrid monitor for behavior and performance analysis of distributed systems. *IEEE Trans. on Software Engineering*, 16(2):197–211, February 1990.

[8] F. Jahanian, R. Rajkumar, and S. C. Raju. Runtime monitoring of timing constraints in distributed real-time systems. *Real-Time Systems*, 7(3):247–273, November 1994.

[9] T. Klevin and L. Lindh. Scalable architectures for real-time applications and use of bus-monitoring. In *Proc. 6th international conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, China, December 1999.

[10] L. Lindh, T. Klevin, and J. Furunas. Scalable architectures for real-time applications - SARA. In *CAD & CG'99*, December 1999.

[11] L. Lindh, J. Starner, J. Furunas, J. Adomat, and M. E. Shobaki. Hardware accelerator for single and multiprocessor real-time operating systems. In *Seventh Swedish Workshop on Computer Systems Architecture*, Goteborg, Sweden, June 1998.

[12] A.-C. Liu and R. Parthasarathi. Hardware monitoring of a multiprocessor system. *IEEE Micro*, pages 44–51, October 1989.

[13] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–621, December 1989.

[14] L. D. Moleksy, K. Ramamritham, C. Shen, J. A. Stankovic, and G. Zlokapa. Implementing a predictable real-time multiprocessor kernel - the spring kernel. In *IEEE Workshop on Real-Time Operating Systems and Software*, May 1990.

[15] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, and M. Imai. Hardware implementation of a real-time operating system. In *Proceedings of TRON'95*, pages 34–42, 1995.

[16] PCI Industrial Computers Manufacturers Group. *CompactPCI Specification rev. 2.1*.

[17] H. Tokuda, M. Koreta, and C. Mercer. A real-time monitor for a distributed real-time operating system. *ACM Sigplan Notices*, 24(1):68–77, January 1989.

[18] J. J. Tsai, K.-Y. Fang, and H.-Y. Chen. A noninvasive architecture to monitor real-time distributed systems. *Computer*, 23(3):11–23, March 1990.