WatchCop – Safer Software Execution through Hardware/Software Co-Design

Christian Ristig, René Fritzsche, Christian Siemers

Department of Computer Science, Clausthal University of Technology,
Julius-Albert-Straße 4, D-38678 Clausthal-Zellerfeld, Germany
christian.ristig@tu-clausthal.de
rene.fritzsche@tu-clausthal.de
christian.siemers@tu-clausthal.de

Abstract: This paper introduces a novel approach to support runtime execution monitoring with nearly no negative impact on runtime. The monitoring is configurable to application-specific requirements and supports real-time behaviour during development and runtime. An extension of the basic approach for monitoring contains watchdog capabilities for a fine-grained observation of runtime activities and a two-level support of recovery from program failures. It is shown how these capabilities may be used to ensure safe software execution.

1. INTRODUCTION

Safe software execution is a major requirement in many application areas. As nearly all applications in automation technology are software-based, and the software is executed through a von-Neumann-microprocessor, guaranteeing safe execution becomes a major issue also inside this application class.

To face this issue, most approaches address the development process. This resulted in models for the software process like the V-model and its extension V-model XT, in equivalent test models as part of the software development process, and in model-based development. The motivation for model-based approaches is mostly based on complexity handling and on handling safety issues, as software-generated source code might follow coding rules and said to be much more reliable than its hand-coded counterpart.

Nevertheless software-based systems remain complex. Beside their algorithmic complexity, where functions, methods and data structures interfere, time complexity is even harder to manage, as language support is still missing. Usually timing issues are mapped to the operating system, which is a second level of programmability often called “programming in the large”. The operating system will be able to manage tasks in a coarse-grained way, but the drawbacks are that single task behaviour is not influenced in a fine-grained manner, and that monitoring of task behaviour might be time consuming. As the operating system is just software being executed in most cases on the same processor, time consuming monitoring will negatively influence runtime behaviour.
In summary, safe software execution faces at least two dimensions: timing and algorithmic. For the algorithmic dimension, several approaches like lock-step execution of at least two processors or triple module redundancy (TMR) are well-known. They are more-or-less expensive in using computational resources and energy, but required fault coverage is obtained by them.

The now presented approach for the timing safety dimension uses a coprocessor for monitoring and timing issues instead of an operating system. The coprocessor is quite small and consists in its basic architecture of a set of registers, a monitoring memory of arbitrary size, limited algorithmic capacities and a control unit. Section 2.1 gives a more detailed description of the underlying hardware architecture.

The purpose of the basic architecture is to monitor the runtime behaviour of the program. The monitored data give an exact view of the runtime and may also be used for worst-case-execution-time analysis. This basic approach can be extended by some configurable comparing units to automatically monitor execution during runtime and to signal the processor critical situations in a fine-grained manner. This extension works like an advanced multi-watchdog and responsible for the name of this approach: WatchCop, a watchdog coprocessor. Section 2.2 shows more details of this extension.

The hardware/software interface is quite simple and uses only a small set of instructions to couple processor and coprocessor as discussed in section 3. More important as the number of coprocessor instructions is the usage of these instructions in the monitored program. Once the coprocessor is initialised, the instructions for monitoring are very rare in the binary code. Every label selected to monitor the program flow is translated into one cop2-instruction with specific parameter resulting in one execution cycle inside a RISC-based architecture. Section 4 discusses the use of the coprocessor instructions for monitoring software execution in detail.

It should be mentioned that this approach is not non-invasive therefore negative runtime impact can be minimised but won’t be zero. We followed the clearly structured coprocessor approach as a good balance between minimum impact on runtime and necessary changes in the microarchitecture.

Section 5 discusses some applications of the WatchCop for safe software execution, and finally section 6 gives a summary and an outlook for further work.
2. Hardware Architecture

2.1 Basic Architecture

Figure 1 shows the basic architecture of WatchCop. A similar approach to [Sa90] is used in the sense that all functionality is implemented within a coprocessor with a minimized impact on program execution time. While the approach in [Sa90] and other approaches [Ra05], [Fa08] and [Ir06] use checksums and trace functionality to ensure or monitor the correct program execution, our approach addresses mainly the timing behaviour as well as some other additional behaviour concerning program flow. Therefore, it may be used to control program flow as well as to support operating system capabilities.

The shown implementation in figure 1 (without the shaded extended part) contains a free running counter of arbitrary width (here 56 bit), a set of 4 coprocessor registers, a read and a write pointer to manage the ring buffer for storage of monitoring events, the monitoring memory of arbitrary size (here 1K depth with 64 bit width) and the control unit. While the arbitrary values were chosen for long-running counter implementation without overflow, maintaining the other 8 bit for label identification and for easy interfacing the internal 64 bit with a 32-bit microprocessor, these values may change frequently.

The basic approach works as a monitoring system. If the processor fetches a cop2-instruction with according arguments, this instruction is directed to the coprocessor and executed there. In detail, 8 bit of the argument are decoded as label number and stored together with the actual 56-bit counter value inside the RAM. This implements the monitoring functionality, while the read and write pointer manage the access using specific cop2 instructions.
2.2 Extended Architecture to implement Watchdog Functionality

The monitoring capacities of the basic architecture are extended by some watchdog functionality (see fig. 1 including the extended part). For this purpose, a set of configurable counters is integrated in the architecture. Each counter may be configured independently to a start value and to a label number. During program execution, each cop2-instruction containing this label (ref. section 4) as argument resets the counter to the start value.

On the other side, the counter is continuously decremented each clock cycle, and if an underflow occurs, a signal to the processor is set to active, because a defined runtime condition was not met. This is well-known watchdog functionality with the extension that several points in program flow might be used to monitor the runtime behaviour.

Different to known watchdog implementations, the severity level of this watchdog alert will be configurable and may change between two successive events. WatchCop uses at least two signalling lines to the processor, one for interrupt request, the other for reset. Therefore, a watchdog timer underflow can initialize an interrupt, e.g. if this is the first time, and may reset the processor, if this happens again. Applications of this feature are discussed in section 5.

3 Hardware/Software Interface

The interface between WatchCop and the application software contains three parts (see also figure 2): The coprocessor instructions for initialization and monitoring, the signalling part addressing interrupt service resources of the processor, and the data exchange for initialization and monitoring evaluation. This reflects to the four main parts of software interface between main program and monitoring: Initialization, monitoring during program execution, reaction on event signalling and monitoring evaluation.

During initialization, the main processor may set several register to arbitrary values. After this period, the coprocessor normally works autonomously, and only few instructions are inserted into normal program flow. This was designed to reduce the impact on program execution as much as possible while maintaining the coprocessor approach to preserve the main processor from redesign, as mentioned before.

The signalling part was designed to immediately inform the main processor about events like watchdog timer overflow (1st and 2nd level), therefore signalling is mapped on interrupt requests of arbitrary priority. In most cases, the 2nd overflow of one watchdog timer will generate a highest priority interrupt request, in many cases of non-maskable kind.
4 Instruction Set

The chosen instructions were picked from the standard instruction-set of MIPS-based microprocessors in order to be able to use unmodified development tools and compilers. Due to this fact language-checkers and optimizing strategies may still be applicable after inserting our monitor-instructions to the code. Three commands are necessary for using all WatchCop functionality:

- The mtc2-instruction (move to coprocessor 2 register) copies data from processor register to a specified coprocessor register. This may be used for initialization of some functions, e.g., timeout values for the watchdog timer or all kinds of configuration setup.

- The mfc2-instruction (move from coprocessor 2 register) copies data from coprocessor register to processor register. This will be frequently used, e.g., for reading status register or accessing monitored values in the event list.
- The cop2-instruction (coprocessor 2) is used as general purpose format for all other instructions. This format normally holds some bits for free use, in our implementation 25. Figure 3 shows the usage of the bits: Bits 23 through 16 contain a function number, bits 7 through 0 a label number.

The list of actually implemented functions for cop2-instruction is shown in table 1.

Table 1. Used subformats for some cop2-instruction

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Operation</td>
<td>None</td>
<td>Do nothing</td>
</tr>
<tr>
<td>Reset_Monitoring</td>
<td>None</td>
<td>Monitoring is stopped, all pointer are reset</td>
</tr>
<tr>
<td>Start_Monitoring</td>
<td>None</td>
<td>Monitoring is (re-)started, from this point on every store_event is recorded</td>
</tr>
<tr>
<td>Stop_Monitoring</td>
<td>None</td>
<td>Monitoring is stopped, but data are maintained</td>
</tr>
<tr>
<td>Set_Read_Pointer</td>
<td>None</td>
<td>Sets the read pointer to first/last value</td>
</tr>
<tr>
<td>Get_value_from_list</td>
<td>None</td>
<td>Gets the value from the list in RAM pointed to by the read pointer</td>
</tr>
<tr>
<td>Increment/decrement</td>
<td>None</td>
<td>The read pointer is incremented/decremented (in circular way)</td>
</tr>
<tr>
<td>read_pointer</td>
<td>Label (8 bit)</td>
<td>The actual clock counter is stored in RAM including the label</td>
</tr>
<tr>
<td>Store_Event</td>
<td>Label (8 bit)</td>
<td>The 56-bit value in CP2 R2/R3 is copied into the according watchdog timer</td>
</tr>
<tr>
<td>Set_Timeout_Value</td>
<td>Watchdog Timer (8 bit)</td>
<td>The 56-bit value in CP2 R2/R3 is copied into the according watchdog timer</td>
</tr>
</tbody>
</table>

5 Applications with Use of WatchCop

5.1 Applications Using the Basic Architecture

The first major application that is supported by WatchCop is the development, specifically the test of real-time applications. For this purpose, the critical paths or parts to be tested must be instrumented by cop2-instructions with monitoring. Each time the program runs through this point a label containing the 56-bit counter value is written into the private RAM of the WatchCop, and may afterwards be read and analyzed for real-time behaviour.

As 256 different labels may be inserted into assembler code, some interference like “if the program runs through label xx, timing constraint on label yy is not met” between paths through the program may also be observed. This leads to a post-run analysis with exact timing labels with very low impact on the runtime (by the additional instructions). Using this basic feature during runtime is also possible and a good monitor to prove real-time behaviour, even after problems have occurred.
5.2 Applications Using the Extended Architecture

The full power of WatchCop is provided by the extended features. This is due to the fact that WatchCop has now a direct feedback channel by using signalling lines for interrupt request or reset, and therefore watchdog, challenge/response-functionality and extended support for scheduling may be included into the application.

5.2.1 n-Level-Reaction Scheme for Watchdog Events

First, the watchdog part inside can be used for a classical watchdog functionality. Up to 8 timer in the actual implementation can be configured as watchdog timer and may be started. Specific cop2-instructions will reset the timer to its initial value, if the program executes this instruction. Therefore, WatchCop supports 8 different watchdog activities during runtime.

We implemented a configurable 3-level reaction scheme. The first underflow of any watchdog timer results in an interrupt request, a highly prioritized interrupt/trap or in a reset. The level of reaction is configurable, and therefore the ‘classical’ functionality of a watchdog might be configured. After requesting the interrupt, an additional time is loaded into the watchdog timer register. This time must be explicitly configured during initialization, otherwise it is set to ‘0’, and the next level of reaction is instantaneously invoked.

If the processor reacts during that additional reaction time, the system appears to work properly, and the watchdog is reloaded with the original value. If not, the second reaction level is started and results in a trap (non-maskable interrupt) or a reset (if additional time is configured to 0). The third level is always a reset, but in other implementations even more level might be integrated. In this case the processor has run out of control, and the application must be restarted completely.

This may be used for additional functionality in the following way. If the processor receives the 1st-level interrupt request, the reaction inside the interrupt service routine (ISR) might be the reset of the watchdog timer. Nevertheless it is still not sure that the processor does not stay in a deadlock inside the main-program, because the reaction is performed inside the ISR.

To avoid this misinterpretation and to introduce a challenge/response-function, it is proposed to include the reaction not into the ISR but into normal program operation. The interrupt coming from coprocessor results into a software event to be handled inside main program (see fig. 4), and algorithmic capacities may also be included. In this case, the interrupt is interpreted as challenge, and the software of the processor responses.
5.2.2 Scheduling Support by WatchCop

Scheduling is basically supported by measuring time differences inside WatchCop. If the microprocessor system does not use an operating system, the possibility of using a cooperative approach for threads with an application-own scheduling might be used. If all cooperative threads work perfectly well and do not block, everything is okay, and the system works with high efficiency. All overhead from an operating system is omitted, but multithreading is still supported.

If the application is not as perfect as described, WatchCop can be used to obtain a good compromise between cooperative and preemptive scheduling. The basic principle can be still cooperative, but all threads are individually controlled by the n-level watchdog mechanism. In this case the planned reaction upon a watchdog timer underflow could be the cancelling of the blocking thread (which is definitely a serious issue) and continuing work with next event or next thread.

Furthermore, the TaskCombining approach as described in [Si05] or the TaskPair approach in [Ge01], both are supported. In this case a twofold reaction scheme for real-time applications is proposed. If reaction time is tide, at least one of a set of tasks is not executed but reacts with an emergency value. This reaction system is known as precise time, imprecise logic.

As shown in [Si05], the presence for timer support is essential for efficient implementation. In this case, the WatchCop may support with the built-in timer, because they can support the processor with according interrupts. One watchdog timer is used for each task inside the task-combining-system and is set to a value close to the deadline of this task with a reaction value of “trap”. If this value is reached without reset before, then the task is not able to react precisely, and the imprecise value (which must be computed earlier as discussed in [Si05]) is used. As long as this task has not started to perform computation and all others are ready or close to readiness, no time is wasted.
6 Summary and Outlook

We presented a system consisting of a coprocessor specialized to monitor execution time as well as a rudimentary software interface to use this system. The purpose is to implement a reaction system for deadlocks and other software errors to keep the software-based system operating.

This WatchCop is implemented as softcore using a MIPS32-compatible processor with coprocessor interface. The implementation of the coprocessor uses roughly 20% of the processor, with most of the silicon area is used for storing time/label values in the RAM. These values are the base for detecting time failures or weaknesses and for correction. As simple example, an operating system providing preemptive scheduling using the WatchDog capabilities could be implemented very easy and fast.

The next step will be to integrate the coprocessor into high level languages and to establish high-level language support. The roadmap to do this is to use normal C and to enhance this with special comments that are interpreted by a pre-compiler, and we plan to automatically generate a multithreading system from such enhanced C-code.

References


