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 is 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 quite expensive in using computational resources and energy, but required fault coverage is obtained by them.

1.1 The Organisation of This Paper

The approach in this paper addresses runtime issues. A coprocessor is used to monitor and/or control the runtime behaviour of a set of tasks. This coprocessor is configured by the main processor at initialisation time and later fed by runtime information also provided by the main processor, but the build-in control algorithms work independently from main processor after configuration. As the coprocessor is kept small and simple, the additional overhead in silicon area and energy consumption remains small.

Therefore the coprocessor is used for monitoring and timing issues instead of an operating system. The coprocessor 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 is responsible for the name of this approach: WatchCop, a *watch*dog *cop*rocessor. Section 2.2 shows more details of this extension.

Section 3 discusses the hardware/software interface between processor and coprocessor. This interface is kept quite simple and uses only a small set of instructions to couple both units. 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 3.2 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 4 discusses some applications of the WatchCop for safe software execution, including some classical approaches like challenge/response, and finally section 5 gives a summary and an outlook for future work.

1.2 Related Work

The observation of program execution is an issue since several decades. The authors in [Ma88] give a very good survey of approaches in the 1980s to take care of program execution. These early works are focussed on memory access errors – now a part of a memory protection system – and on illegal opcodes – inside modern architectures a task of the exception system. Most of the surveyed approaches in [Ma88] have found their way into the microprocessor and closely coupled units.

The approach in [Na05] on the other side addresses the actual requirements for reliability and real-time execution. The authors use a formal method to extract models from source code written in Ada for verification. The information obtained by verification are then used to arm a specifically designed chip called SafetyChip. Real-time monitoring is performed by observing the communication between application and run-time kernel, and any deviance from the predefined behaviour is signalled to the system.

The authors in [Sa90] use an approach to implement control functionality within a coprocessor for any program part, not only specific parts. The main constraint of this approach is to minimise impact on program execution time. To ensure this, the approach in [Sa90] as well as other approaches ([Ra05], [Fa08] and [Ir06]) use checksums and trace functionality to ensure or monitor the correct program execution in the sense of arithmetically correct program flows but not in timely manner.

In contrast to the briefly discussed approaches, WatchCop is used to monitor and control program execution concerning its timing behaviour first to ensure real-time functionality and secondly to ensure correct program execution at all. The main constraints are to minimise the impact on runtime – specifically by reducing the amount of additional instructions in the program flow – and the impact on processor microarchitecture – specifically by avoiding any impact on the architecture inside the execution pipeline.

Consequently the WatchCop approach shows most similarity to the work published in [Na05]. In contrast to that, WatchCop may observe any program flow, even if a run-time kernel is not available, and a formal execution model is not required to enhance the design with monitoring capacity. Nevertheless such a model is very useful, and concerning WatchCop, we are following the way to enhance a language model and a compiler to generate the monitoring information semi-automatically.

Compared to all previously discussed approaches, WatchCop shows the following differences:

1. WatchCop monitors and controls the timing behaviour, not just the control flow behaviour of the program and couples runtime and real-time.

2. While maintaining a minimum of negative impact on runtime behaviour as well as the original processor architecture, WatchCop may monitor all kinds of program flows with no restriction.

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.

The architecture shows two major parts, containing the basic part and the extended part. The basic part carries all necessary parts for monitoring applications including an interface to obtain the monitored values. This addresses operating system issues in the sense that actual runtime data are available for use in scheduling decisions.

The extended part on the other side contains direct control mechanisms and all necessary interfaces. This was designed to directly control runtime behaviour and to recover from exceptional operating conditions like deadlocks. This may be used to control program flow as well as to support operating system capabilities.

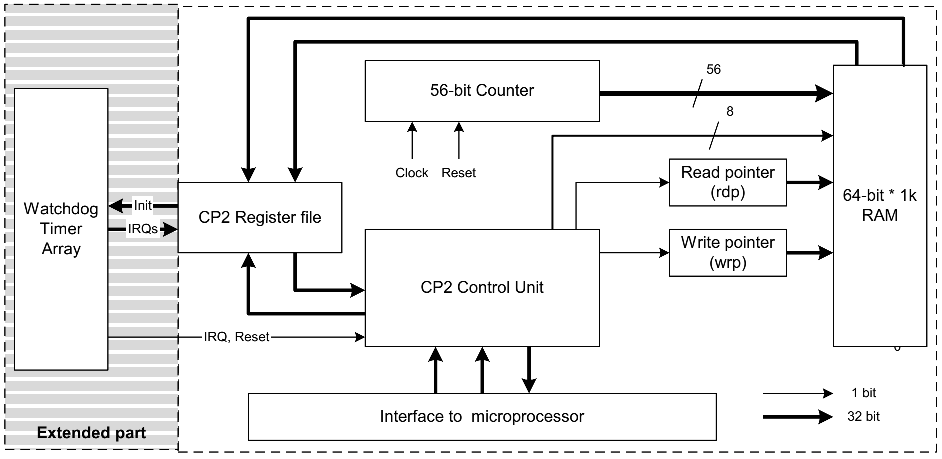

Figure 1. Basic/extended (left) microarchitecture of WatchCop

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 pointers 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 3) as argument resets the corresponding 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 4.

3 Hardware/Software Interface

The interface between WatchCop and the application software contains three parts (see also figure 2): The coprocessor instructions for configuration during initialisation phase and for 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 initialisation, the main processor may set several register to arbitrary values for configuring the coprocessor. After this period, the coprocessor normally works autonomously without executing an instruction flow provided by the main processor. Only few instructions are inserted into normal program flow to synchronise the coprocessor with instruction flow of the main processor. This interface 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.

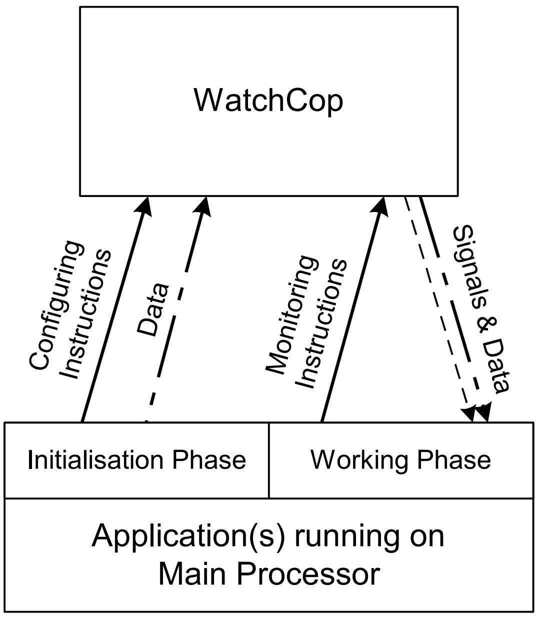

Figure 2. Interfaces between WatchCop and main processor

3.1 Hardware Interface between WatchCop and Main Processor

The hardware interface to/from coprocessor and main processor consists of a bus system for instructions of arbitrary size, e.g. 32 bit width, optionally of a data bus system of arbitrary size, and of few signalling lines. While the instruction bus system is mandatory, the same lines could be used for transmitting additional data, if they are capable of bidirectional data transfer, when data transfer from WatchCop to main processor is desired. The latter could be required for transmitting monitored values from WatchCop to main processor for further evaluation.

The signalling part was designed to immediately inform the main processor about events like watchdog timer underflow. During first project evaluation it appeared to be desirable to implement a signalling system with more than one level, and a two-level system was chosen. The 1st and 2nd level signalling is mapped on interrupt requests of arbitrary priority. In most cases, the 2nd level underflow of one watchdog timer will generate a highest priority interrupt request, in many cases of non-maskable type.

3.2 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:

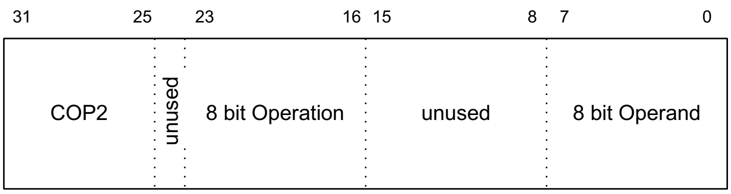

Figure 3. Binary format of COP2-instructions in WatchCop

- 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.

Operation	Operand	Descrpition
No_Operation	None	Do nothing
Reset_Monitoring	None	Monitoring is stopped, all pointer are reset
Start_Monitoring	None	Monitoring is (re-)started, from this point on every Store_Event is recorded
Stop_Monitoring	None	Monitoring is stopped, but data are maintained
Set_Read_Pointer	None	Sets the read pointer to first/last value
Get_value_from_list	None	Gets the value from the list in RAM pointed to by the read pointer.
Increment/decrement read_pointer	None	The read pointer is incremented/decremented (in circular way)
Store_Event	Label (8 bit)	The actual clock counter is stored in RAM including the label
Set_Timeout_Value	Watchdog Timer (8 bit)	The 56-bit value in CP2 R2/R3 is copied into the according watchdog timer

Table 1. Used subformats for some cop2-instruction

The list of actually implemented functions for cop2-instruction is shown in table 1. Actually this may be extended for additional functions if required, because the instruction set space is not densely used in the current implementation..

4 Applications with Use of WatchCop

4.1 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 will 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, or as information base for operating system decisions.

This approach was successfully tested during testing real-time applications using several threads to monitor specifically inter-thread communications. These communications, either implemented in blocking (waiting) or non-blocking fashion, show runtime-specific and even data-specific behaviour. Therefore the observation of realistic behaviour might be essential to consider modifications to the design, and it is hard work to realise these realistic constraints only by simulation.

4.2 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.

4.2.1 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 initialised by the coprocessor results into a software event which in turn must 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.

```
void interrupt vISR()
{
   ...
   switch( cp2IRQStatus )
   {
      case WATCHDOG_LEVEL_1:
         setEvent( EVENT_WATCHDOG_L1 );
         break;
   ...
   }
}
```

```
void main()
{
   int tempEvent;
   ...
   while( 1 )
   {
      getEvent( &tempEvent )
      switch( tempEvent )
      {
         case EVENT_WATCHDOG_L1:
            resetWatchdogTimer();
            break;
   ...
   }
}
```

Figure 4. Code fragment to integrate challenge/response-functionality

4.2.2 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.

This scheduling was successfully implemented, and we call it forced-cooperative due to the fact that preemptive scheduling only occurs when exceptional operating conditions are observed and threads block. This is comfortably supported by WatchCop, even if the forced scheduling times vary from thread to 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.

5 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. The pre-compiler extracts the timing constraint from source code and generates a user constraint file, which is then interpreted for code generation.

This approach is first published in [Fr08] and [Fr10], and WatchCop is the ideal hardware architecture to execute the necessary monitoring and measurement for inter-thread scheduling. The formal models in [Na05], which are essential for defining the correct functionality of the proposed Safety Chip, are here replaced by language constructs.

While the integration of high-level language support and WatchCop into a development tool and framework is the actual next step, the plan is to automatically generate a multithreading system from such enhanced C-code including WatchDog capabilities to support multithreading and monitor the complete system.

References

[Fa08] N. Farazmand, M. Fazeli, S.G. Miremadi, "FEDC: Control Flow Error Detection and Correction for embedded systems without program interruption". *ares, pp.33-38, 2008 Third International Conference on Availability, Reliability and Security, 2008.*

[Fr08] R. Fritzsche, G.Kemnitz, C. Siemers, "TEC: Time-Enhanced C – Erweiterung einer imperativen Programmiersprache um Zeitvorgaben". *Tagungsband Embedded Software Engineering, S. 427-430, Sindelfingen, Germany, Dezember 2008 (in German language).*

[Fr10] R. Fritzsche, C. Siemers, "Scheduling of Tme-Enhanced C (TEC)". *Accepted for publication in Proceedings of World Automation Conference 2010 (WAC 2010), Kobe, Japan, September 2010.*

[Ge01] M. Gergeleit, "A Monitoring-based Approach to Object-Oriented Real-Time Computing," *Otto-von-Guericke-Universität Magdeburg, Universitätsbibliothek, 2001, http://diglib.uni-magdeburg.de/Dissertationen/2001/margergeleit.pdf*

[Ir06] K. Irrgang, J. Braunes, R.G. Spallek, S. Weisse, T. Gröger, "A new Concept for Efficient Use of Complex On-Chip Debug Solutions in SOC based Systems". *Embedded World 2006 Conference, pp. 215-223 (2006). Franzis Verlag, Poing, 2006, ISBN 3-7723-0143-6.*

[Ma88] A. Mahmood, E.J. McCluskey, "Concurrent Error DetectionUsing Watchdog Processors – A Survey". *IEEE Transactions on Computers 37(2), pp. 160-174 (1988).*

[Na05] G. Naeser, L. Asplund, J. Furunäs, “Safety Chip – A Time Monitoring and Policing Device”. *SIGAda 05, pp. 63–68 (2005).*

[Ra04] A. Rajabzadeh, M. Mohandespour, G. Miremadi, "Error Detection Enhancement in COTS Superscalar Processors with Event Monitoring Features". *prdc, pp.49-54, 10th Pacific Rim International Symposium on Dependable Computing (PRDC'04), 2004*

[Ra05] A. Rajabzadeh, S.G. Miremadi, "A Hardware Approach to Concurrent Error Detection Capability Enhancement in COTS Processors," *prdc, pp.83-90, 11th Pacific Rim International Symposium on Dependable Computing (PRDC'05), 2005*

[Sa90] N.R. Saxena, E.J. McCluskey, “Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums”. *IEEE Transactions on Computers 39(4), pp. 554-559 (1990).*

[Si05] C. Siemers, R. Falsett, R. Seyer, K. Ecker, “Reliable Event-Triggered Systems for Mechatronic Applications,” ”. *The Journal of Systems and Software 77, Elsevier, 2005, pp. 17–26.*