

Use of Helper Threads for OS Support in the Multithreaded Embedded TriCore 2 Processor

Florian Kluge, Jörg Mische, Sascha Uhrig, Theo Ungerer
HiPEAC European Network of Excellence
University of Augsburg, Augsburg, Germany
{kluge|uhrig|ungerer}@informatik.uni-augsburg.de

Rafael Zalman
HiPEAC European Network of Excellence
Infineon Technologies AG, Munich, Germany
rafael.zalman@infineon.com

Abstract—Infineon equipped their TriCore 2 microcontroller with multithreading capabilities. As memory protection techniques are getting more important, it also implements a range-based memory protection system. Based on the multithreading capability a helper thread can run in a thread slot in separation from the real-time application thread to support embedded operating systems like OSEK or AUTOSAR OS used in automotive systems. We show that our concept can save more than 70% of task switching time by pre-loading the memory protection registers for the application that is predicted to be scheduled next. Also, we propose modifications to the TriCore 2 architecture that would support our concept.

I. INTRODUCTION

A multithreaded processor [14] is characterized by the ability to execute instructions of different threads within the processor pipeline simultaneously. The contexts of two or more threads of control are stored in separate on-chip hardware thread slots each including its own register set, instruction pointer, and processor status registers. Multithreading within one processor can be used to hide memory latencies (e.g. from instruction fetching or data loading) of one thread while executing another thread. Typically, application threads are loaded into the hardware thread slots. Another application domain for this kind of threads are helper threads that run separated from an application and support the application or a running operating system. Such helper threads are proposed for tasks like branch prediction [2], prediction of accessed memory addresses [3], [9], [16], exception handling [8], [15] and accelerated execution of loops [10]. In the embedded Java microcontroller Komodo helper threads are also used for the garbage collection of hard real-time threads [13] and dynamic preloading of software upgrades of running hard real-time threads [12].

In section II we present characteristics of the TriCore 2 architecture relevant for our work. In section III we develop a helper thread concept to utilize the TriCore 2's second thread and propose some changes to the TriCore 2 architecture. Section IV concludes this paper.

II. INFINEON TRICORE 2 ARCHITECTURE

The Infineon TriCore architecture defines a 32-bit microcontroller, which is mostly used for automotive applications. It combines a RISC load/store architecture with a DSP-like Harvard memory architecture.

The TriCore 2 is binary compatible to its predecessor, but provides a second hardware thread slot, which can be used to bridge long instruction fetch latencies [11]. Currently, the TriCore 2 architecture is licenced as an IP Core.

A. TriCore 2 Multithreading

The Infineon TriCore 2 features two hardware threads, T0 and T1 [5]. Generally, a program is executed in T0. When the pipeline impends to stall due to long instruction fetch latencies, execution can be transferred to the second thread T1. This thread usually is executed from fast on-chip scratchpad RAM. It is also possible to have T0 and T1 running alternately by setting for each thread a number of clock cycles it should run. Although, there are some restrictions to T1. Interrupt service requests will always be served by T0. Furthermore, T1 is only allowed to run with interrupts enabled. Disabling interrupts automatically transfers execution to T0.

As can be seen from these restrictions, the two threads of the TriCore 2 are not fully equal. Applications running in T1 are restricted to threads that need not to disable interrupts. Examples are the "untrusted" applications in AUTOSAR [1]. Another use for T1 would be as helper thread that supports a running application or operating system, as proposed in the next section. The TriCore 2 scheduler ensures that the real-time behaviour of the application thread in T0 is not disturbed by the execution of a helper thread in T1.

B. Memory Protection

Current automotive control units usually run several applications. There need to be ways to ensure that an application cannot be harmed by other applications, i.e. to prevent other applications from manipulating their data or even code. This can happen due to programming errors, for example. Current automotive software specifications, like AUTOSAR or Protected OSEK [4] pick up this problem by claiming the existence of some kind of hardware-based Memory Protection System (MPS).

Usually one of two kinds of MPS are implemented in current microcontrollers. The *page-based* approach allocates memory in the form of equal-sized pages, e.g. 1kB. It is possible, to have as many pages as desired for an applications. The management of these pages is usually done by the Memory Management Unit (MMU) or a special Memory Protection Unit (MPU).

The other technique is the *range-based* approach. Here, the CPU or MMU has some special registers, where memory ranges are described by lower and upper bounds. There are usually separate sets for data and code memory, differing in the kind of access privileges (Read/Write/eXecute).

The Infineon TriCore family offers a range-based memory protection system (MPS) with two to four Memory Protection Register (MPR) sets each for data and code memory [7] (see Figure 1).

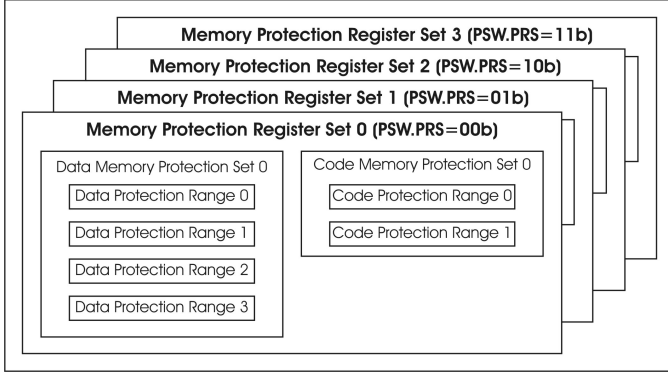


Fig. 1. Memory Protection Register Sets of the TriCore architecture [7]

The real number of MPR sets depends on the implementation of the processor. Each register set is made up of several range registers. Here again, the number of range registers is implementation dependent. Figure 1 shows the range register sets of the TriCore architecture, however, a TC1130 [6] TriCore processor implements only two of the possible four MPR sets. Most other TriCore implementations share this configuration with four data memory protection ranges and two code memory protection ranges (as far as they implement a MPS at all). For the TriCore 2 architecture the afore said also applies [5]. From the available MPR sets, at each time exactly one can be active, while the others are not considered. The active set is referenced in the `PSW.PRS` bits, as shown in Figure 1. If within one set there are overlapping ranges, the least restrictive access privileges are applied to the memory access.

III. A HELPER THREAD FOR THE TRICORE 2 MICROCONTROLLER

A. Design

Within an AUTOSAR OS implementation, each application is assigned its own Memory Protection Register Set. During scheduling, the MPR set must be changed along with the application's other context data. On a TC1130, the whole process of switching from one application to another (determining the next application and switching the context data) takes about 1400 clock cycles.

As the software on such a node usually is statically configured, it is simple to determine the next application at each point of time. The only exception to this rule is the occurrence of interrupts. Here the regular flow of execution may be

disturbed, but as it is induced from outside the processor, we cannot do anything about this case.

Now, ideally we have a helper thread running that predicts the next application and loads all context data in advance into the processor. Thus, at the point of scheduling the processor would only have to switch from one context set to another. This is not possible, because the processor contains only two sets of context data that both are in use already for the application thread and for the helper thread. So this technique would require a third set of context data.

However, we can speculatively determine the next task to be scheduled and pre-load its Memory Protection Registers. Here we assume a minimum of two MPR sets from which only one can be active at a time. So the other one could be used for the pre-loading of memory protection registers.

All calls to operating system functions will be done using the `syscall` trap that transfers the execution into a privileged mode. The operating system itself will then have full access to all memory areas. Thereby we assume the OS is correctly implemented. Thus, we would get by with the two available register sets.

Although, there is one drawback to our concept. As mentioned above, the currently selected MPR set of a running task is referenced in the `PSW` register of the CPU (`PSW.PRS`, see figure 1). This register is saved at each `call` instruction into the context save area and restored at the corresponding `return`. Now, if a task is assigned another MPR set than it had before its last preemption, all these values in the task's context save area need to be adjusted. The complexity of this operation depends linearly on the depth of the task's current call stack and would nullify our gained speed-up. Therefore, we could not yet evaluate our concept on a real TriCore 2 processor.

B. Proposed Architectural Changes

To overcome these problems and make good use of the second hardware thread, we propose the following changes to the TriCore 2 architecture:

- Implementation of all four memory protection register sets, and
- Split `PSW.PRS` into a local bit (`PRSL`) that is saved with each context, and a global bit (`PRSG`) that is not put into the context save area.

Thus, we would have two sets of memory protection register sets, where the globally active one is referenced by the `PSW.PRSG` bit, and therein the actually active one is referenced by the `PSW.PRSL` bit.

With the additional MPR sets, we would also be able to protect the operating system. Thus, risks through programming bugs would be reduced.

Figure 2 shows how we intend to use the four MPR sets. Both sets with `PSW.PRSL=0` will be used for the operating system, i.e. they contain the same values. Thus they could be mapped onto the same hardware registers. The other two sets are used for the application. Now, the helper thread can preload the register values in the currently inactive set, and

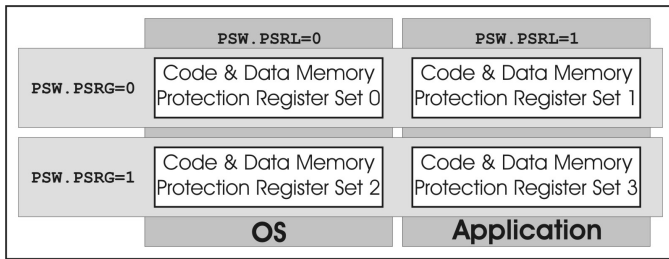


Fig. 2. Proposed organization of the Memory Protection Register Sets of the TriCore architecture

at an application switch, only the PSW.PSRG bit needs to be flipped, and the processor would automatically run with the correct MPR set.

C. Evaluation

We did some measurements using a TC1130 to find out what improvement would be possible. Here the complete process of scheduling takes about 1400 clock cycles. Thereof, determination of the task to be scheduled next amounts up to about 900 clock cycles. Nearly 300 cycles are needed for the swapping of OS management data. Loading of the memory protection registers amounts to 200 cycles. So if a helper thread predicts the next task correctly and already loads its memory protection registers, 1100 cycles (78%) of the total context switch time can be saved. For the remaining 300 cycles, we see no way for further improvements, as the data processed here directly depends on the program flow.

IV. CONCLUSION

We have presented a possible application for the TriCore 2's second hardware thread. As shown, the use of a helper thread for the pre-loading of memory protection registers would speed up application switching over 70%, if the next application is predicted correctly. Also, we have shown that only a small change in the processor architecture would be necessary to implement our proposed helper thread concept. As the TriCore 2 is traded as an IP Core, licensees would be able to easily adopt our proposed enhancements into their processors.

In the future, we intend to apply the proposed changes to a TriCore 2 simulator and evaluate the actual possible speed-up. The outcome of this evaluation is particularly dependent on the ratio of a correct task prediction.

V. ACKNOWLEDGEMENTS

This work was performed during a PhD internship of Florian Kluge at Infineon Technologies AG. The internship was supported by the EC Network of Excellence HiPEAC. Florian Kluge would like to thank Erik Norden for his mentoring during the internship.

REFERENCES

- [1] AUTOSAR AUTomotive Open System ARchitecture. <http://www.autosar.org>.
- [2] CHAPPELL, R. S., STARK, J., KIM, S. P., REINHARDT, S. K., AND PATT, Y. N. Simultaneous subordinate microthreading (ssmt). In *ISCA* (1999), pp. 186–195.
- [3] COLLINS, J. D., WANG, H., TULLSEN, D. M., HUGHES, C. J., LEE, Y.-F., LAVERY, D. M., AND SHEN, J. P. Speculative precomputation: long-range prefetching of delinquent loads. In *ISCA* (2001), pp. 14–25.
- [4] HIS STANDARD SOFTWARE WORKING GROUP. *OSEK OS Extensions for Protected Applications*, 1.0 ed., July 2003.
- [5] INFINEON TECHNOLOGIES AG. *Tricore™ 2 Architecture Manual*, 1.1 ed., June 2003.
- [6] INFINEON TECHNOLOGIES AG. *TC1130 User's Manual*, 1.3 ed., November 2004.
- [7] INFINEON TECHNOLOGIES AG. *Tricore™ 1 Core Architecture*, 1.3 ed., February 2005.
- [8] KECKLER, S. W., CHANG, A., LEE, W. S., CHATTERJEE, S., AND DALLY, W. J. Concurrent event handling through multithreading. *IEEE Trans. Computers* 48, 9 (1999), 903–916.
- [9] LUK, C.-K. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *ISCA* (2001), pp. 40–51.
- [10] MARCUELLO, P., GONZÁLEZ, A., AND TUBELLA, J. Speculative multithreaded processors. In *International Conference on Supercomputing* (1998), pp. 77–84.
- [11] NORDEN, E. Keynote: Multithreading for low-cost, low-power applications. In *ARCS* (2004), C. Müller-Schloer, T. Ungerer, and B. Bauer, Eds., vol. 2981 of *Lecture Notes in Computer Science*, Springer, pp. 4–5.
- [12] PFEFFER, M., AND UNGERER, T. Dynamic real-time reconfiguration on a multithreaded java-microcontroller. In *ISORC* (2004), IEEE Computer Society, pp. 86–92.
- [13] PFEFFER, M., UNGERER, T., FUHRMANN, S., KREUZINGER, J., AND BRINKSCHULTE, U. Real-time garbage collection for a multithreaded java microcontroller. *Real-Time Systems* 26, 1 (2004), 89–106.
- [14] UNGERER, T., ROBIC, B., AND SILC, J. A survey of processors with explicit multithreading. *ACM Comput. Surv.* 35, 1 (2003), 29–63.
- [15] ZILLES, C. B., EMER, J. S., AND SOHI, G. S. The use of multithreading for exception handling. In *MICRO* (1999), pp. 219–229.
- [16] ZILLES, C. B., AND SOHI, G. S. Execution-based prediction using speculative slices. In *ISCA* (2001), pp. 2–13.