LIGHT-WEIGHT, PREDICTABLE REACTIVE PROCESSING— THE KIEL ESTEREL PROCESSOR

Xin Li and Reinhard von Hanxleden
{xli, rvh}@informatik.uni-kiel.de
Dept. of Computer Science, Christian-Albrechts-Universität Kiel
http://www.informatik.uni-kiel.de/rtsys/

Abstract

The Kiel Esterel Processor architecture provides direct hardware support for reactive control flow, which keeps executables fast and compact and minimizes power consumption.

1. Introduction

Embedded real-time systems must react continuously to stimuli from their environment, which makes their control-flow patterns differ from those of traditional systems that transform a given input to an output at their own pace. Reactive processors provide direct hardware support for reactive control flow, which keeps executables fast and compact and results in lower power consumption than with traditional architectures [8]. The *Kiel Esterel Processor* (*KEP*) [7] is a reactive processor that has an instruction set derived from the Esterel language [1], which is a synchronous language designed for reactive programming. In particular, the *KEP* directly provides concurrency and preemption primitives, with deterministic behavior and predictable Worst Case Reaction Time (WCRT) [6].

This exhibit presents an FPGA-based implementation of the *KEP*, which has proven very competitive to classical processor designs. For a standard suite of Esterel benchmarks, the code size is typically an order of magnitude smaller than that of the MicroBlaze, a 32-bit COTS RISC processor core. The worst case reaction time is typically improved by 4×, and energy consumption is also typically reduced to a quarter [4].

2. The KEP Architecture

The main challenge when designing a reactive architecture is the handling of control. In the *KEP*, the Reactive Multi-Threading Core (RMC) is responsible for managing the control flow of all threads. Fig. 1 shows the architecture of the RMC. It contains dedicated hardware units to handle concurrency, preemption, exceptions, and delays.

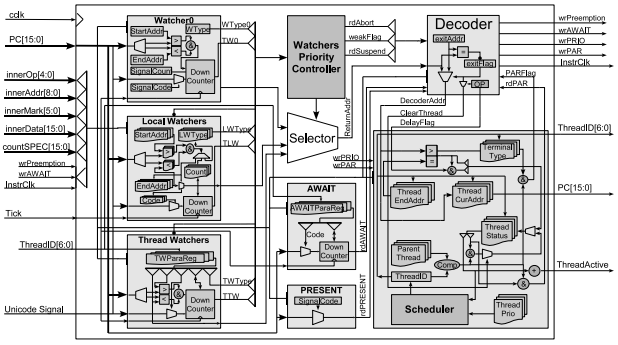

Figure 1: The architecture of the *KEP* Reactive Multi-threaded Core (RMC).

To implement concurrency, the *KEP* employs a multi-threaded architecture, where each thread has an independent program counter (PC) and threads are scheduled according to their statuses and dynamically changing priorities. The scheduler is very light-weight. In the *KEP*, scheduling and context switching do not cost extra instruction cycles, only changing a thread's priority costs an instruction. The priority-based execution scheme allows on the one hand to enforce an ordering among threads that obeys the constraints given by Esterel's semantics, but on the other hand avoids unnecessary context switches. If a thread lowers its priority during execution but still has the highest priority, it simply keeps executing.

The RMC provides a configurable number of Watcher units, which detect whether a signal triggering a preemption is present and whether the PC is in the corresponding preemption body [5]. When preemptions are nested and triggered simultaneously, the Watcher Priority Controller decides which must take precedence. The *KEP* Watcher are designed to permit arbitrary nesting of preemptions, and also the combination with the concurrency operator. However, in practice this often turns out to be more general than necessary, and hence wasteful of hardware resources. Therefore, the *KEP* also includes trimmed-down versions of the Watcher.

Max. threads	2	10	20	40	60	80	100	120
Slices	1295	1566	1871	2369	3235	4035	4569	5233
Gates (k)	295	299	311	328	346	373	389	406

Table 1: Extending a *KEP* to different threads.

The *KEP* is highly configurable, including the possible degree of concurrency. As Table 1 shows, the hardware usage (on a Xilinx 3S1500-4fg676 FPGA)increases only 4× when the concurrency increases 60× when measured in slices, and even just 1.4× when measured in equivalent gates. The clock rate does not vary significantly, it is around 60 MHz; one instruction takes three clock cycles. For comparison, the MicroBlaze which with the same memory size (BRAM) employs 309k gates.

3. The Compiler

We have implemented a compiler for the *KEP* based on the CEC infrastructure [2]. A central problem for compiling Esterel onto the *KEP* is the need to manage thread priorities during their creation and their further execution. In the *KEP* setting, this is not merely a question of efficiency, but a question of correct execution. We have devised a priority algorithm to compute a priority assignment that respects the Esterel semantics as well as the execution model of the *KEP*. The complexity of the algorithm is in practice linear in program size [4].

4. Validation

The exhibit also demonstrates the validation test bench (see Fig. 2) that has been used in the development of the *KEP*. The user interacts via a host work station with an FPGA Board, which contains the *KEP* as well as some testing infrastructure. First, an Esterel program is compiled into an *KEP* object file (.ko) which is uploaded to the FPGA board. Then, the host provides Input events to the *KEP* and reads out the generated Output events. This also yields the number of instructions per tick, from which we can deduce the worst case reaction time for the given trace. The input events can be either provided by the user interactively, or they can be supplied via a .esi file. The host can also compare the Output results to an execution trace (.eso). We use EsterelStudio V5.0 to compute trace files with state and transition coverage. Our regression suite currently contains some 500 benchmarks.

5. Summary and Outlook

The *KEP* demonstrates a custom processor design for the efficient execution of concurrent reactive programs. However, the underlying model of computation, with threads keeping their individual program counters and a priority based scheduling, could also be emulated by classical processors. Furthermore, it would be interesting to implement a virtual machine that has an instruction set similar to the *KEP*. An ongoing project is to implement the *KEP* itself in Esterel, which should not only a challenging benchmark, but could also be used to synthesize a virtual machine. We are also investigating to augment the *KEP* with external hardware to speed up the computation of signal expressions [3]. Finally, we are also interested in developing an advanced energy management methodology, which is based on the WCRT information, to further save power consumption.

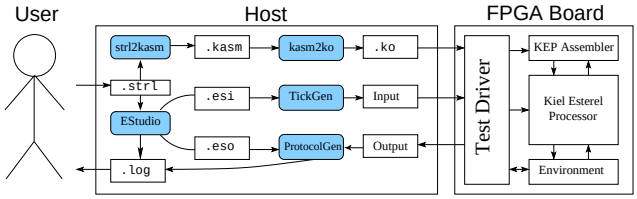

Figure 2: The structure of the KEP evaluation platform

References

[1] G. Berry. The Foundations of Esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte.

[2] S. A. Edwards. CEC: The Columbia Esterel Compiler. http://www1.cs.columbia.edu/~sedwards/cec/.

[3] S. Gädtke, X. Li, M. Boldt, and R. von Hanxleden. HW/SW Co-Design for a Reactive Processor. In *Proceedings of the Student Poster Session at the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, Ottawa, Canada, June 2006.

[4] X. Li, M. Boldt, and R. von Hanxleden. Mapping Esterel onto a multi-threaded embedded processor. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, San Jose, CA, October 21–25 2006.

[5] X. Li and R. von Hanxleden. A concurrent reactive Esterel processor based on multi-threading. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC'06), Special Track Embedded Systems: Applications, Solutions, and Techniques*, Dijon, France, April 23–27 2006.

[6] C. T. Marian Boldt and R. von Hanxleden. Worst case reaction time analysis of concurrent reactive programs. In *Proceedings of the Workshop on Model-driven High-level Programming of Embedded Systems (SLA++P07)*, Braga, Portugal, Mar. 2007.

[7] R. von Hanxleden. The Kiel Esterel Processor Homepage. http://www.informatik.uni-kiel.de/rtsys/kep/.

[8] R. von Hanxleden, X. Li, P. Roop, Z. Salcic, and L. H. Yoong. Reactive processing for reactive systems. *ERCIM News*, (66):28–29, Oct. 2006. http://www.ercim.org/publication/Ercim_News/EN67.pdf.