A Concurrent Reactive Esterel Processor Based on Multi-Threading

Xin Li, Reinhard von Hanxleden
Real-Time and Embedded Systems Group
Dept. of Computer Science and Applied Mathematics
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40, D-24098 Kiel, Germany
{xli,rvh}@informatik.uni-kiel.de

ABSTRACT

Esterel is a concurrent synchronous language for developing reactive systems. As an alternative to the classical software and hardware synthesis paths, the *reactive processing* approach uses a specialized processor with an instruction set tailored to Esterel. A principal difficulty when compiling onto a reactive processor is the faithful, efficient implementation of concurrency. This paper presents a novel reactive processor architecture based on multi-threading, which allows the arbitrary nesting of preemption and concurrency, and is scalable to very high degrees of concurrency.

Categories and Subject Descriptors

C.1 [**Processor Architectures**]: Other Architecture Styles—*High-level language architectures*

General Terms

Languages, Performance

Keywords

Synchronous languages, Esterel, processor architecture, multi-threading

1. INTRODUCTION

The synchronous language Esterel has been developed for modeling reactive systems [6, 2], which typically are embedded systems that continuously react to their environment. To adequately express reactive behavior, Esterel offers control flow primitives that are much richer than that of traditional, sequential programming languages. An Esterel program typically consists of a collection of nested, concurrent threads, which may include preemption blocks and may themselves be included in preemption blocks, and whose execution is synchronized to a single, global clock.

SAC'06 April 23-27, 2006, Dijon, France

An Esterel program is classically either synthesized into hardware, using for example VHDL as an intermediate representation [3], or into software [6, 12]. The typical software synthesis approach is to first translate the (concurrent) Esterel program into a sequential language, such as C, and then to compile this further for a standard common-off-the-shelf (COTS) microprocessor. However, common processor architectures cannot handle concurrency and preemption directly; therefore, handling these control constructs correctly turns out to be not trivial and generally fairly expensive for classical software implementations.

Hence, an alternative approach proposed recently to improve the performance of a software implementation is to implement an Esterel program on a special-purpose *reactive processor* whose instruction set has been tailored to Esterel. The general appeal of reactive processors is that they offer the flexibility of software at a performance close to hardware implementations; another advantage is their predictability due to the direct mapping from Esterel specification to execution. This not not only makes timing predictions feasible [17], but also simplifies formal verification. Hence we envision a practical potential of reactive processors in embedded, reactive applications where short design turn-arounds or low volumes do not warrant a custom hardware design, but a classical software solution would be inappropriate as well due to requirements on predictability, price per unit, or also power consumption.

We distinguish patched reactive processors and custom reactive processors. The *patched reactive processor* approach combines a COTS processor core with an external hardware block, which implements additional Esterel-style instructions. This approach has been pioneered by the REFLIX and REPIC [19, 9, 20] designs. The EMPEROR architecture [11] is a multiprocessor variant, which also supports Esterel's concurrency operator "||"; however, it is not obvious how this design would support the arbitrary nesting of concurrency and preemption, and this solution is relatively hardware intensive.

The *custom reactive processor* approach involves a full-custom reactive core, whose instruction set and data path have been tailored exclusively for the processing of Esterel code. An example is the Kiel Esterel Processor family, with the KEP1 [16] and KEP2 [17] models. The major limitation of the KEP so far was that it did not support the || operator.

Hence, we still see limitations in the existing proposals, especially regarding their handling of concurrency. In particular, it appears that none of the architectures proposed so

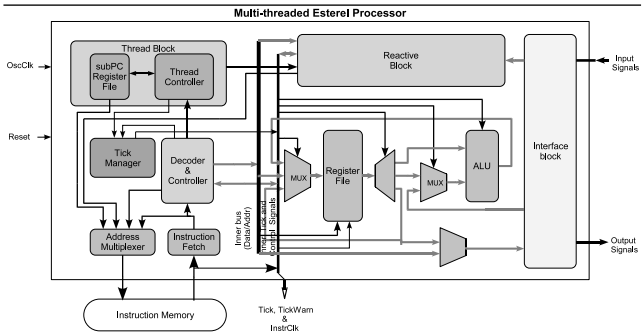

Figure 1: The architecture overview.

far allows the arbitrary nesting of preemption and concurrency operators, which is one of the key features of Esterel.

This paper presents a novel custom reactive processor architecture, the *Kiel Esterel Processor 3* (*KEP3*), that overcomes this limitation. A key concept realized in this architecture is that it offers concurrency *orthogonally* to the other reactive control flow behaviors, rather than providing concurrency *on top* of reactive behavior as is done in the multiprocessing approach. This is achieved by combining a single, sequential processing engine with separate control flow units for concurrency, preemption, signal testing, etc., which tightly interact with each other according to the Esterel semantics.

The rest of this paper is organized as follows. The next section gives an overview of the KEP3 architecture and its instruction set. Section 3 discusses KEP3's thread management in more detail, and Section 4 elaborates an example that illustrates the interaction of preemption and concurrency. Experimental results are given in Section 5. Finally, we conclude and outline future work in Section 6.

2. THE PROCESSOR ARCHITECTURE

The architecture of the KEP3, shown in Figure 1, is inspired by the three layers that constitute a reactive program [6], *i. e.*, the *interface* layer, the *reactive kernel*, and the *data handling* layer. The implementation of Esterel's reactive statements relies on the cooperation of the KEP3's Decoder & Controller, Reactive Block and Thread Block, which together form the Reactive Core. An interface block handles input reception and output production. The classical computations are performed by the Data Handling Block.

To illustrate the intricacies of the reactive control flow constructs and to illustrate the translation into the KEP3 assembler, we are considering the EXAMPLE Esterel module in Figure 2(a), introduced by Edwards [12] and also used by Closse *et al.* [10]. The most common Esterel statements, including all of the primitive reactive kernel statements, can be represented directly by single KEP3 assembler instructions. Other statements require statement expansion; the expansion rules applied here are shown in Figure 2(b). Note that the KEP3 also offers an instruction that directly corresponds to the sustain statement (see also the example presented later in Figure **??**); however, to properly synchronize threads, we must in this case expand the sustain into kernel statements. The resulting KEP3 assembler, shown in Figure 2(c), is still quite compact, with an instruction count that is comparable to the line count of the Esterel source code, despite the intermediate statement expansion.

2.1 Handling Preemption

In the KEP3 architecture, the Reactive Block contains a (configurable) number of Watcher modules that are responsible for implementing the various types of preemption offered by Esterel. Each Watcher module can be configured to a certain type of preemption, a certain trigger signal, and an address range that delineates the preemption block.

If during execution of the program the PC falls in the watched range and the trigger signal is present, the Watcher is responsible for triggering the corresponding changes in the control flow. The Reactive Block is responsible for coordinating the Watcher blocks in a way that reflects the Esterel semantics. Each Watcher in the Reactive Block is assigned an index number, which also defines its priority. A Watcher can be overridden by another Watcher with higher priority. Considering the preemption nest structure, it becomes clear that the higher priority preemption has a wider address range which covers the lower priority one. Therefore, the earlier preemption instruction in a preemption nest will be assigned to the higher priority Watcher. Since it is not necessary to continuously execute special instructions that check on the status of each watcher, the program does not slow down when entering a (nested) abortion block [16, 17]. The Watcher modules operate autonomously, thus also offering a certain type of concurrency, beyond the ||-operator.

2.2 Handling Concurrency

A hurdle when implementing concurrency is the need to interleave thread execution to allow communication among threads within the same logical tick. To handle this, the KEP3 employs a *multi-threaded architecture*. Each thread has an independent program counter (PC) and threads are scheduled according to their activation status and a dynamically changing priority. The priority of a thread is assigned when the thread is created (with the PAR instruction, as in "parallel," see below), and can be changed subsequently by executing a priority setting instruction (PRIO). Communication dependencies, which can be statically derived from the program, impose certain scheduling constraints, which determine how priorities must be assigned such that the interleaved thread execution obeys the semantics of the original program.

Figure 3 shows the architecture of the Thread Block, which is responsible for managing the threads. For each instruction cycle, it decides which thread to execute next, based on the current status of each thread. A context switch does not cost any extra clock cycles, and the lean design of the Thread Block still permits a comparatively high instruction frequency, see also the experimental results (Section 5).

A concurrent Esterel statement with n concurrent threads joined by the ||-operator is translated into KEP3 assembler as follows. First, threads are *forked* in addition to the previously existing thread(s). The fork is performed by a series of instructions that consist of n PAR instructions and one PARE instruction, which together initialize the Thread Block. Each PAR instruction creates one thread, by assigning a *start address*, which initializes the ThreadCurAddr that is associated with this thread in the Thread Block, and a non-negative priority. The *end address* is either given by

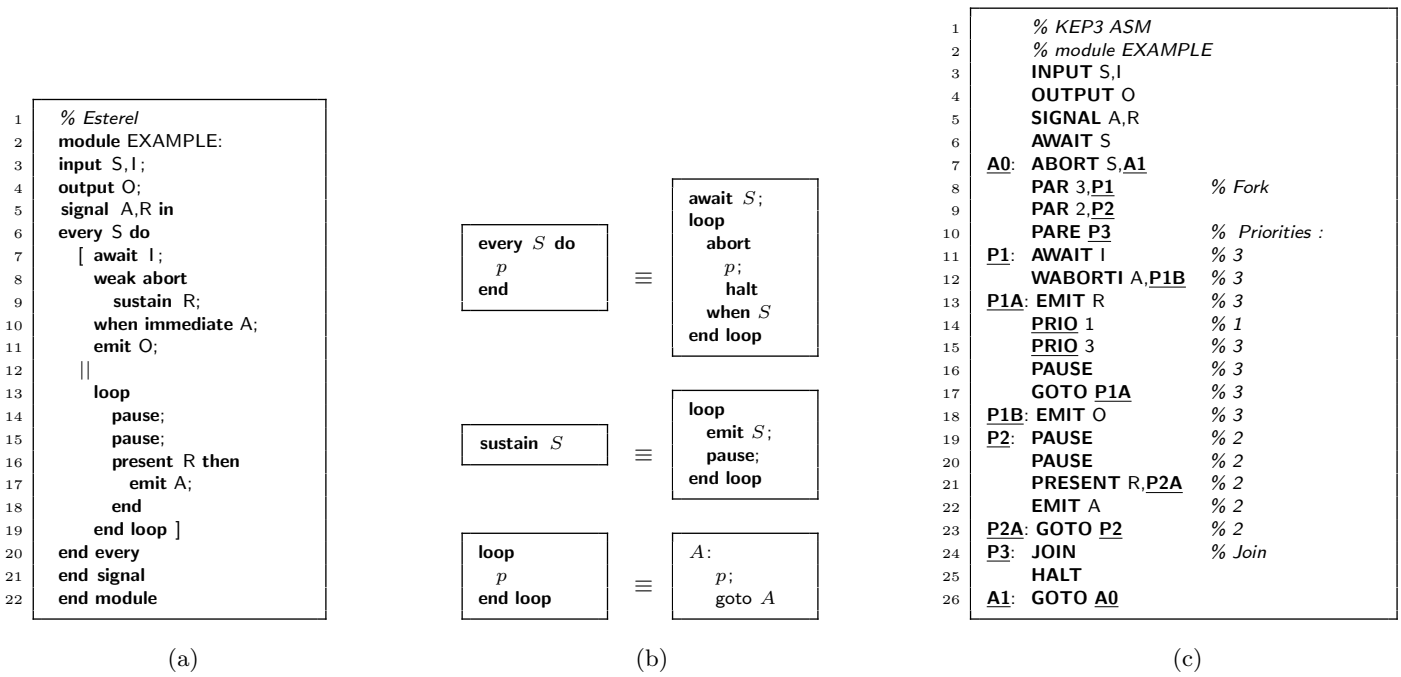

Figure 2: EXAMPLE: an Esterel module illustrating the parallel and preemption statements (a), the translation rules for every, sustain and loop (b), and the resulting KEP3 assembler program (c).

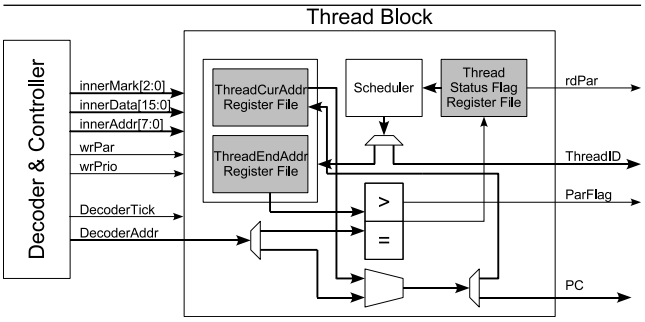

Figure 3: Architecture of the Thread Block.

the start address specified in a subsequent PAR instruction, or, if there is no more thread to be created, it is specified in a PARE instruction. The code block for the last thread is followed by a JOIN instruction, which waits for the termination of all forked threads and concludes the concurrent statement.

The Thread Block uses two status flags to keep track of each thread's status. The ThreadEnable flag indicates whether the thread is still running (*enabled*) or already terminated (*disabled*), and the ThreadActive flag indicates whether the thread should still be scheduled within the current logical tick (is *active*) or not (*inactive*). After a thread is created, those two flags are both set to '1', which means the thread is ready to be scheduled. However, the Scheduler will not become active until all of the thread configurations are finished. After the PARE instruction is executed, the activated threads can be invoked by the priority-based preemptive scheduling mechanism.

At the beginning of each instruction cycle, the Scheduler inspects all active threads and executes the thread with the highest priority; if there are several active threads with the same highest priority, the Scheduler executes the thread that has been created first (which precedes the other threads). Once a non-instantaneous instruction is executed, the ThreadActive flag will be set to '0', meaning that this thread will not be scheduled any more in the current tick. If all threads are inactive, the current tick is finished. At the start of the next tick, the ThreadActive flags of all enabled threads will again be set to '1'.

A thread termination could be caused by two reasons. One is that the thread finishes all statements in its body, in which case the expected fetch address will equal the ThreadEndAddr associated with that thread. The other is that the thread is aborted by an enclosing abortion, in which case the expected fetch address will be greater than the ThreadEndAddr. In the latter scenario, a ParFlag signal will be set to '1' to indicate to the Reactive Block that the thread is terminated by an abortion. The Thread Block and the Reactive Block tightly interact with each other through several control signals to ensure the proper handling of arbitrary preemption and concurrency control flow.

3. THREAD MANAGEMENT

The priority assigned during the creation of a thread and by a particular PRIO instruction is fixed. Due to the non-linear control flow, it is still possible that a given statement may be executed with varying priorities; in principle, the architecture would therefore allow a fully dynamic scheduling. However, we here assume that the given Esterel program can be executed with a statically determined schedule, which re-

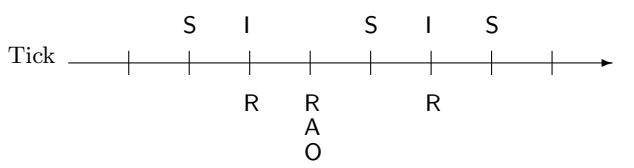

Figure 4: A possible execution trace of the EXAMPLE.

quires that there are no cyclic signal dependencies. This is a common restriction, imposed for example by the Esterel v7 [13] and the CEC [7] compilers. Note that there are also Esterel programs that are causally correct (that are *constructive* [5]), yet cannot be executed with a static schedule and hence cannot be directly translated into KEP3 assembler using the approach presented here. However, these programs can be transformed into equivalent, acyclic Esterel programs [18], which can then be translated into KEP3 assembler. Hence, the actual run-time schedule of a concurrent program running on KEP3 is *static* in the sense that if two statements that depend on each other, such as the emission of a certain signal and a test for the presence of that signal, are executed in the same logical tick, they are always executed in the same order relative to each other, and the priority of each statement is known in advance. However, the run-time schedule is *dynamic* in the sense that due to the non-linear control flow and the independent advancement of each program counter, it in general cannot be determined in advance which code fragments are executed at each tick. This means that the thread interleaving cannot be implemented with simple jump instructions; there has to be a run-time scheduling mechanism, as implemented in the Thread Block, that manages the interleaving according to the priority and the actual program counter of each active thread.

4. THE INTERACTION OF CONCURRENCY AND PREEMPTION

To study how the KEP3 combines concurrency and preemption, it is instructive to work through the example code in Figure 2(c), and using a possible execution trace in Figure 4, with input signals shown above the time line and local and output signals below the time line.

After starting the module, the initial thread (thread 0) is enabled and active. The control stays at the AWAIT S_6[1] and waits for signal S. For the example input trace, AWAIT S_6 is terminated by the presence of the signal S at the second tick. Next, the ABORT S, $A1_7$ configures Watcher0 to watch for signal S, followed by the PAR/PARE instructions that create two new threads. The Scheduler now has to handle all active threads, *i. e.*, threads 0, 1, and 2. Thread 1 has the highest priority and is scheduled first. In thread 1, the non-instantaneous statement AWAIT I_{11} causes the thread to become inactive, hence thread 2 is scheduled next. Similarly to thread 1, the $PAUSE_{19}$ delays thread 2 by one tick and causes it to become inactive. The last active thread, the thread 0 that forked the other threads, executes the $JOIN_{24}$ instruction to check the statuses of its incoming branched threads. Since two threads (threads 1 and 2) are still enabled, the JOIN does not terminate. Therefore, thread 0 becomes inactive as well, and the current tick is finished because all of threads are inactive.

When the third tick starts, all enabled threads are activated again. The Scheduler again starts with thread 1; now I is present, which terminates AWAIT I_{11}. Next, WABORTI A,$P1B_{12}$ makes Watcher1 immediately watch signal A, and execution continues through EMIT R_{13} and the priority setting instruction PRIO 1_{14}, which changes the priority of the currently executing thread to be lower than that of the thread 2. Therefore, the Scheduler blocks thread 1 and switches over to thread 2, where $PAUSE_{20}$ is executed and thus deactivates thread 2. Hence, thread 1 resumes with the PRIO 3_{15} instruction, which ensures that thread 1 is scheduled before thread 2 in the subsequent tick, before it becomes deactivated by $PAUSE_{16}$.

Similarly, in the fourth instant, after thread 1 has been blocked by the PRIO 1_{14} instruction, the thread 2 resumes from $PAUSE_{20}$ and executes PRESENT R,$P2A_{21}$ to test the presence of signal R. Since the signal R was emitted by thread 1, the PRESENT instruction will not cause a branch, and EMIT A_{22} is executed, before control moves back to $PAUSE_{19}$. Hence, the control is handed over to thread 1 again. Note that the program counter is in the watching range of Watcher1, which is triggered by A. The Priority-Controller maps Watcher2's outputs to the Reactive Block's output, and the Decoder & Controller checks the rdAbort, weakFlag, rdSuspend, rdAWAIT, and so on, simultaneously. As this is a weak abort, the abort body is still executed for the current instant; that is, the PRIO 3_{15} is executed, then the $PAUSE_{16}$ is fetched. Since it is a non-instantaneous statement, the Reactive Core will ignore it and instead leave the abort block. Therefore, the EMIT O_{18} is executed, and as thread 1 then reaches its end address, it is disabled and deactivated, and thread 0 is resumed. Since the thread 2 is still enabled, the JOIN still does not terminate.

At the next tick, the disabled thread 1 will not be scheduled, and control starts from the terminated $PAUSE_{19}$ instruction. As S is present, the Watcher0 is triggered. Since this is a strong abortion, the controller responds to it immediately. The Thread Block gets the ReturnAddr 26, *i. e.*, the next instruction address behind the body of abortion S, as the next fetch address. Note that the ReturnAddr is greater than the ThreadEndAddr of the current thread, hence, thread 2 is disabled and deactivated. The ParFlag signal is set as '1' to denote that this thread is terminated by an outer abortion. Now that all of the incoming branch threads are disabled, the JOIN instruction in the thread 0 terminates. Since the ParFlag is set, the execution of thread 0 responds to the active abortion. The control jumps to GOTO $A0_{26}$.

The execution scheme can also handle several threads within a preemption body. The seventh tick in Figure 4 illustrates how a triggered abortion overrides two enabled threads. Similar as in the fifth tick, the triggered Watcher0 causes the ReturnAddr to be 26, which is greater than the ThreadEndAddr of the thread 1. As a result, the thread 1 is disabled and deactivated, and thread 2 executes. At this point, the program counter is still in the watching range of the Watcher0. The Reactive Block's rdAbort is still '1' and the weakFlag is still '0' to denote a triggered strong abortion. Hence, thread 2 is also disabled and deactivated by the abortion.

[1]To aid readability, we here use the convention of subscripting instructions with the line number where they occur.

Table 1: The codes size and RAM usage comparison for EXAMPLE between the KEP3-A (see Table 3), MCS51, and Microblaze processors.

	KEP3-A	MCS51			Microblaze		
		V5	V7	CEC	V5	V7	CEC
Code size (words)	22	462	1051	839	464	1136	482
Code size (bytes)	88	724	1455	1119	1856	4544	1928
RAM Usage (bytes)	14	23	98	39	48	52	52

Table 2: Comparison of the codes sizes in words (one word equals four bytes), and comparison of RAM usage in bytes.

Module	Threads/ Preempt. Depth	Code Size (words)				RAM Usage (bytes)			
		Microblaze			KEP3	Microblaze			KEP3
		V5	V7	CEC		V5	V7	CEC	
BELT	2/3	617	1255	483	31	84	84	52	14
ABCD	4/1	1357	1547	1396	107	112	112	504	24
RUNNER	2/5	688	1323	608	37	88	84	60	14
ARBITER12	36/1	3162	1703	3909	317	256	172	88	156

5. EXPERIMENTAL RESULTS

The EXAMPLE module in Figure 2 is used to compare the concurrency and preemption handling abilities between the KEP3 and other implementations. We use the Esterel Compiler V5.92 (V5), the Esterel Compiler V7 (V7), and the CEC compiler 0.3 (CEC) to synthesize the module to C programs, which are then compiled onto the 32-bit Microblaze soft processor core, and the 8-bit micro-controller MCS51[2]. Table 1 compares the resource usages.

To evaluate the performance of the KEP3 further, we use some standard test cases [4, 1, 8]. The modules were first manually translated into the KEP3 assembler program and then compiled to the KEP3 executable code. This is then compared with software synthesis results of the V5, the V7 and the CEC compilers. We use the Microblaze as the reference point. Table 2 compares executable code size and RAM usage between the KEP3 implementation and the Microblaze software implementation. The optimized data path of the KEP3 results on average in an 88% reduction of codes size and 33% reduction of RAM usage when compared with the best result of the Microblaze implementation.

As mentioned in the introduction, the KEP3 has been designed to be highly configurable. Table 3 compares five different KEP3 variants which include different elements to target various applications. The KEP3-E is configured to allow a comparison with the EMPEROR2, which can handle 2 threads via two-processors [9, 11][3]. Note that every RePIC can handle an abortion nest of depth 4, but due to the architecture of EMPEROR, those abort handling elements cannot nest between the different processors directly. Hence the EMPEROR2 contains 8 abort handling elements, but can only deal with abortion nests of depth 4. As an approximation, we compare this with the KEP3-E that offers a level 6 preemption nesting depth. As a result, for the similar processor configuration as the EMPEROR2, the KEP3-E uses 56% less resources and achieves a 1.6 times instruction clock speedup—and the KEP3 typically takes significantly less instructions to implement the same behavior.

[2]The code for the MCS51 has been compiled by the Keil C51 compiler V6.12; here, a MCS51's instruction *word* represents a complete assembler line. For the Microblaze, code was compiled by gcc version 2.95.3-4. In each implementation, the default optimization is used.

[3]The KEP3 has been synthesized onto different Xilinx FPGAs, including the XC2S100-6TQ144, the XC2V1000-4FG456, and the XC3S1500-4FG676. The figures reported here are for the XC2S100-6TQ144, which should allow a fair comparison with the EMPEROR2, which is based on an ALTERA EP20K200EFC484-2, as the basic units of those two chips have similar structures, functions, and speeds [17].

Table 3: Performance comparison between the KEP3 and EMPEROR.

	KEP3-A	-B	-C	-D	-E	EMP.2
I/O signals	11/11	16/16	11/11	32/32	24/24	24/24
Valued I/O signals	2/2	2/2	3/3	3/3	2/2	2/2
Thread Cnt	4	16	16	32	2	2
Preemption Nest depth	2	4	6	8	6	4+4
Counter Value Range	255	65535	65535	65535	1	1
Variable Register Cnt	16	64	32	64	128	64+64
Datapath Width (bit)	16	16	16	16	8	8
Logic Cells	1670	2474	2692	4020	2086	4761
Max Osc Freq (MHz)	52.75	45.31	39.96	39.48	42.68	35.38
Instruction Freq (MHz)	17.58	15.10	13.32	13.16	14.23	8.84

Table 4: Extending the KEP3-E to different threads.

Thread Number	2	4	8	16	32	64	102	126
Logic Cells	2086	2170	2306	2466	2946	3758	4768	5564
Max Osc Freq (MHz)	42.68	42.68	42.68	42.51	42.68	42.68	40.26	40.26

Finally, to illustrate the scalability of the KEP3 to high degrees of concurrency, Table 4 shows the resources usage and maximum system frequency of KEP3-E when its thread number is increased from 2 to 126. Using resources comparable to the EMPEROR2, the KEP3-3E can handle 102 threads directly, and the instruction frequency is still 1.5 times higher.

6. CONCLUSIONS AND OUTLOOK

This paper presents the KEP3, a concurrent, configurable Esterel processor. It employs a multi-threaded reactive architecture which consists of a reactive core and an optimized data path for the direct execution of Esterel programs. The KEP3 supports Esterel's concurrency operator || in a very precise, direct and efficient way. It also supports full Esterel preemption statements, *i. e.*, the delayed and immediate abort, weak abort, and suspend. One of the strengths of Esterel is the clean orthogonalization of the different reactive control flow constructs, which allows to combine them in an arbitrary fashion; this is fully supported by the KEP3. As the KEP2 predecessor, the KEP3 also handles valued signals, signal counters, local signal declarations (respecting reincarnation), and the pre operator. Further instructions have been added or extended for concurrent execution, such as an exit instruction that properly deals with concurrent exceptions of different priorities irrespective of the order in which they are executed. A more detailed description of the processor and a discussion of its interleaved execution model and the generation of the appropriate priority settings, as well as further experimental results, can be found in a technical report [15].

Ignoring the limitations of the multiprocessing approach with respect to the ability to combine concurrency and preemption, one might argue that the multiprocessing approach has an efficiency advantage over a multi-threading approach, which still relies on sequential execution. However, one should note that threads in Esterel programs typically interact rather tightly, with signals communicating back and

forth within a logical tick, imposing strong synchronization requirements. Unlike classical parallel programming, where an originally sequential algorithm is divided into coarse-grained code fragments that can be executed in parallel to achieve a speedup over a single processor implementation, the concurrent programming in Esterel mainly serves to separate concerns, not to improve efficiency. Quoting Girault [14]: "[T]he source program is *parallel* and not sequential like in a classical programming language [...]. But this *parallelism of expression* is used by the programmer to conceive his/her application in terms of parallel modules cooperating to achieve the desired behavior. It is therefore not related to the *parallelism of execution*, which is due to the fact that the target architecture is distributed." In fact, we suspect that for the type of concurrency found in synchronous languages such as Esterel, a sequential, multi-threaded architecture may very well lead to higher efficiency than a multiprocessing approach, due to the tight link between independent threads that allows very efficient synchronization among the threads. However, substantiating this would require a further systematic comparison.

Regarding future improvements of the KEP3, we plan to add a reconfigurable logic block to allow the efficient detection of compound events. What we see as even more promising at this point is to explore the efficient compilation from Esterel onto the KEP3, in combination with a static analysis of the maximal reaction time in the presence of concurrency.

7. REFERENCES

[1] F. Balarin, P. Giusto, A. Jurecska, C. Passerone, E. M. Sentovich, B. Tabbara, M. Chiodo, H. Hsieh, L. Lavagno, A. Sangiovanni-Vincentelli, and K. Suzuki. *Hardware-Software Co-Design of Embedded Systems, The POLIS Approach*. Kluwer Academic Publishers, Apr. 1997.

[2] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The Synchronous Languages Twelve Years Later. In *Proceedings of the IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, Jan. 2003.

[3] G. Berry. Esterel on Hardware. *Philosophical Transactions of the Royal Society of London*, 339:87–104, 1992.

[4] G. Berry. *The Esterel v5 Language Primer*. Draft Book, 1999.

[5] 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.

[6] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[7] CEC: The Columbia Esterel Compiler. `http://www1.cs.columbia.edu/~sedwards/cec/`.

[8] Estbench Esterel Benchmark Suite. `http://www1.cs.columbia.edu/~sedwards/software/estbench-1.0.tar.gz`.

[9] C. Chow, J. S.Y.Tong, M. Dayaratne, P. S. Roop, and Z. Salcic. RePIC - A New Processor Architecture Supporting Direct Esterel Execution. School of Engineering Report No. 612, University of Auckland, 2004.

[10] E. Closse, M. Poize, J. Pulou, P. Venier, and D. Weil. SAXO-RT: Interpreting Esterel semantic on a sequential execution structure. In F. Maraninchi, A. Girault, and E. Rutten, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, July 2002.

[11] M. W. S. Dayaratne, P. S. Roop, and Z. Salcic. Direct Execution of Esterel Using Reactive Microprocessors. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP)*, Apr. 2005.

[12] S. A. Edwards. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(2), Feb. 2002.

[13] Esterel web. `http://www-sop.inria.fr/esterel.org/`.

[14] A. Girault. A survey of automatic distribution method for synchronous programs. In F. Maraninchi, M. Pouzet, and V. Roy, editors, *International Workshop on Synchronous Languages, Applications and Programs (SLAP'05)*, Electronic Notes in Theoretical Computer Science, Edinburgh, UK, Apr. 2005. Elsevier Science.

[15] X. Li and R. v. Hanxleden. A concurrent reactive esterel processor based on multi-threading. Technischer Bericht 0509, Christian-Albrechts-Universität Kiel, Institut für Informatik und Praktische Mathematik, Nov. 2005. `http://www.informatik.uni-kiel.de/reports/2005/0509.html`.

[16] X. Li and R. v. Hanxleden. The Kiel Esterel Processor - a semi-custom, configurable reactive processor. In S. A. Edwards, N. Halbwachs, R. v. Hanxleden, and T. Stauner, editors, *Synchronous Programming - SYNCHRON'04*, number 04491 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005. `http://drops.dagstuhl.de/opus/volltexte/2005/159`.

[17] X. Li, J. Lukoschus, M. Boldt, M. Harder, and R. v. Hanxleden. An Esterel Processor with Full Preemption Support and its Worst Case Reaction Time Analysis. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 225–236, New York, NY, USA, Sept. 2005. ACM Press.

[18] J. Lukoschus and R. v. Hanxleden. Removing cycles in Esterel programs. In F. Maraninchi, M. Pouzet, and V. Roy, editors, *International Workshop on Synchronous Languages, Applications and Programming (SLAP'05)*, Edinburgh, Apr. 2005.

[19] P. S. Roop, Z. Salcic, and M. W. S. Dayaratne. Towards Direct Execution of Esterel Programs on Reactive Processors. In *4th ACM International Conference on Embedded Software (EMSOFT 04)*, Pisa, Italy, Sept. 2004.

[20] Z. Salcic, P. S. Roop, M. Biglari-Abhari, and A. Bigdeli. REFLIX: A Processor Core with Native Support for Control Dominated Embedded Applications. *Elsevier Journal of Microprocessors and Microsystems*, 28:13–25, 2004.