INSTITUT FÜR INFORMATIK

UND PRAKTISCHE MATHEMATIK

A Concurrent Reactive Esterel Processor Based on Multi-Threading

Xin Li, Reinhard von Hanxleden

Bericht Nr. 0509
November 2005

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

A Concurrent Reactive Esterel Processor Based on Multi-Threading

Xin Li, Reinhard von Hanxleden

Bericht Nr. 0509
November 2005

e-mail:
xli@informatik.uni-kiel.de,
rvh@informatik.uni-kiel.de

An abbreviated version of this report will appear in the *Proceedings of the 21st ACM Symposium on Applied Computing (SAC'05)*, Special Track Embedded Systems: Applications, Solutions, and Techniques, Dijon, France, April 23–27, 2006

Abstract

Esterel is a concurrent synchronous language for developing reactive systems. Classically, Esterel programs are either compiled into software, such as a C program that is then executed on a standard microprocessor, or into hardware. An alternative approach, which tries to combine the efficiency of a software solution with an efficiency close to a hardware solution, is the *reactive processing* approach, where the Esterel program is executed on a specialized processor.

A principal difficulty when compiling onto a reactive processor is the faithful, efficient implementation of concurrency. One approach for implementing concurrency is multiprocessing, where sequential reactive processors are replicated and tightly synchronized to achieve concurrency. This allows concurrency at the top level; however, it is not obvious how this approach supports the arbitrary nesting of preemption and concurrency that is permitted in Esterel. This paper presents a novel reactive processor architecture, the Multi-threaded Esterel Processor (MEP), which overcomes this limitation and also scales well to large degrees of concurrency. Rather than replicating sequential processors, the MEP combines a single main processing element with independent control units that implement the concurrency and preemption operators and that allow the arbitrary combination of these operators in the executed program.

Contents

List of Figures

List of Tables

1 Introduction

The synchronous language Esterel has been developed for modeling reactive systems [6, 2], which typically are embedded systems that continously react to their environment. As a system-level language, it gives an abstract, well-defined and executable description of the application, and can be synthesized into sequential languages for further compilation. 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 include preemption blocks and may themselves be included in preemption blocks, and whose execution is synchronized to a single, global clock.

A fundamental concept of Esterel is the *signal*. Signals are used to communicate internally and with the environment. The execution of an Esterel program is divided into logical *instants*, or *ticks*, which also determine the sampling of input signals and the generation of output signals. The synchrony of Esterel implies that the outputs generated from given inputs occur at the same logical instant; that is, the generation of outputs is (conceptually) simultaneous with the inputs, and computations are (conceptually) instantaneous. Signals are *present* or *absent* throughout an instant, indicating the occurrence of certain events, and they may also carry a value.

The concurrency operator ("||") groups concurrent threads together into a compound concurrent statement. This statement terminates when all participating threads have terminated. When several threads are active concurrently, they may communicate back and forth instantaneously, that is, within the same logical tick; this makes the compilation of a concurrent Esterel program onto a sequential processor a non-trivial task.

Esterel also supports various types of preemption, including suspension and weak and strong abortion [4]. An abortion statement kills its *abort body* upon a specific *trigger signal*. In *strong abortion*, expressed by abort, the body does not receive control at the instant when the trigger occurs. For *weak abortion*, performed by weak abort, the body receives control for a last time at the abortion instant. A *suspension*, performed by suspend, freezes the state of a body for the instant when the trigger event occurs.

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.

An alternative approach proposed recently to improve the performance of a software implementation is to implement an Esterel program on a *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. In the *patched reactive processor* approach, a COTS processor core is combined with an external hardware block, which implements additional Esterel-style instructions [9, 21]; this approach has also been extended into a multiprocessor version [11]. 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 [16, 17]. The architectures proposed so far already demonstrated the efficiency and principal feasibility of the reactive processing approach; however, we still see limitations in

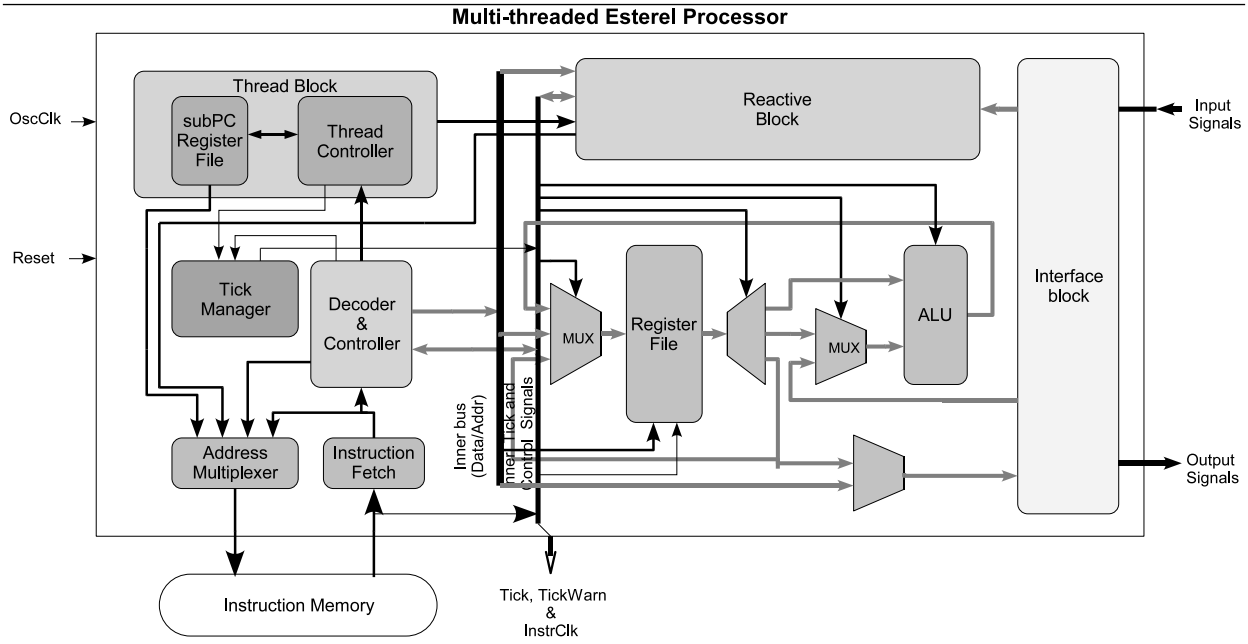

Figure 1: The architecture overview

the existing proposals, especially regarding their handling of concurrency. In particular, it appears that none of the architectures proposed so 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, which we call the *Multi-threaded Esterel Processor* (*MEP*), 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 freely 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 MEP architecture and its instruction set. Section 3 discusses MEP's thread management in more detail, and Section 4 elaborates an example that illustrates the interaction of preemption and concurrency. Section 5 discusses related work, experimental results are given in Section 6. Finally, we conclude and outline future work in Section 7.

2 The Processor Architecture

The architecture of the MEP, 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. In the MEP, a Reactive Core decides what computations and what outputs must be generated when it reacts to inputs. The implementation of Esterel's reactive statements relies on the cooperation of the MEP'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. Before describing the architecture in further detail, we give an overview of the MEP instruction set in the following.

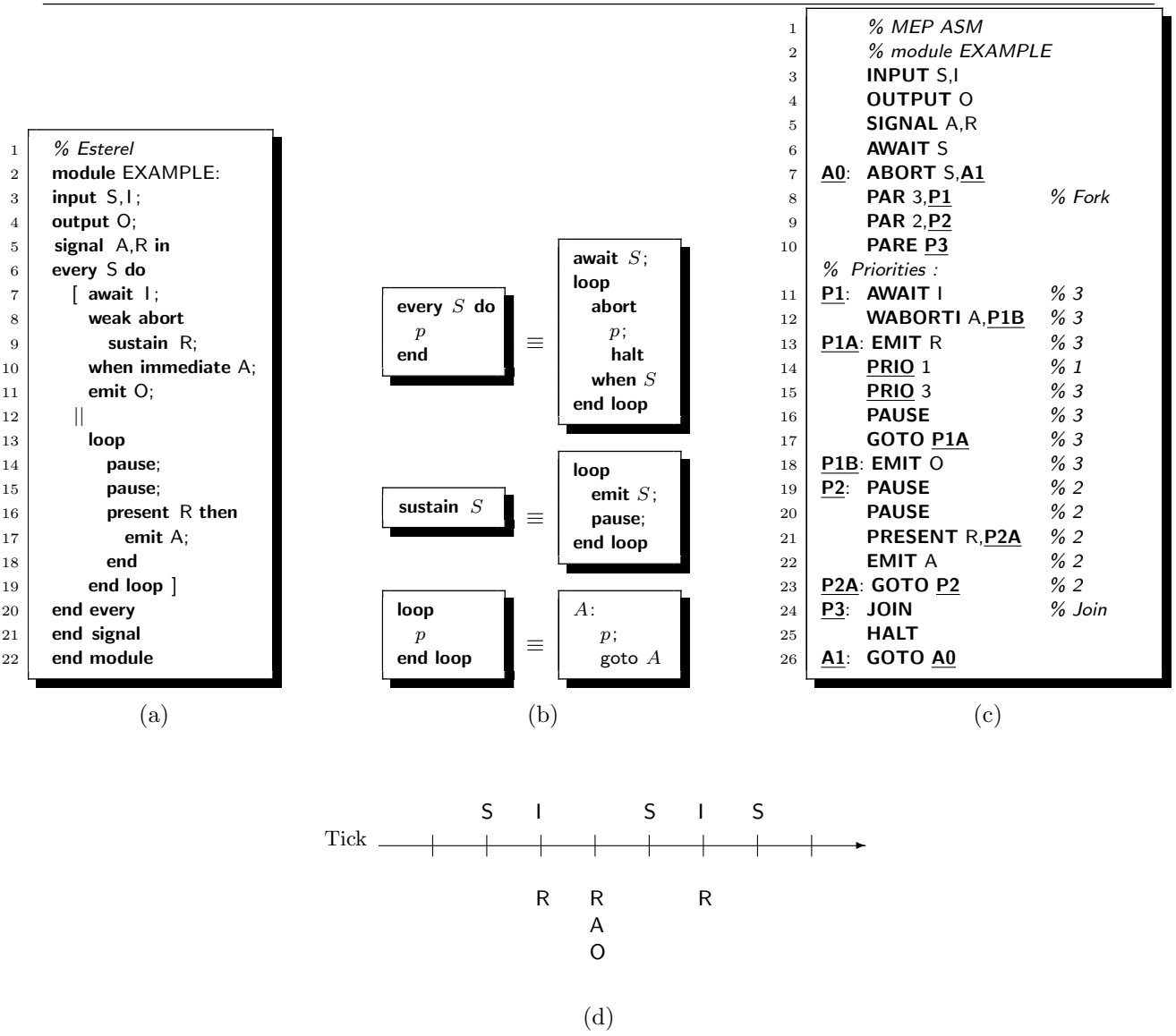

Figure 2: EXAMPLE: an Esterel module illustrating the parallel and preemption statements (a), the translation rules for the every, sustain and loop statements that are used when compiling the program (b), the resulting MEP assembler program (c), and a possible execution trace (d).

2.1 The MEP Instruction Set

The MEP uses a 32-bit wide instruction word and a 16-bit data bus. This gives a range of up to 65535 for the signal counters, which indicate how often a signal has to occur before for example an abort is triggered. The most common Esterel statements, including all of the primitive reactive kernel statements, can be represented directly by single MEP assembler instructions.

The MEP executes each MEP assembler instruction within one *instruction cycle*, indicated by the InstrClk output pin, and each instruction cycle takes three *clock cycles* to execute, which are clocked by the OscClk input pin. The computation of each logical tick requires a sequence of instructions; the tick boundaries are indicated by the Tick output pin. In general, the number of instructions that must be executed to compute the reaction of the current tick varies, which would result in a varying tick frequency; however, the MEP can be configured via a special instruction to a fixed tick length. Obviously, this tick length must be long enough to accommodate the longest instruction sequence that is possible within one tick; if a tick cannot be computed within the specified tick length, this is signaled to the environment via the TickWarn pin.

To illustrate the intricacies of the reactive control flow constructs and to illustrate the translation into the MEP assembler, we are considering the EXAMPLE Esterel module in Figure 2(a), introduced by Edwards [12] and also used by Closse *et al.* [10]. In this module, two concurrent threads are enclosed in an every block, which restarts both threads whenever the input signal S is present (except for the initial tick, when S is ignored, as the every is not "immediate"). The first thread initially waits for the input signal I, and then sustains the local signal R up to the first tick when the local signal A occurs; it then emits the output signal O. The second thread initially idles for two ticks, and then at every other tick emits A whenever R is present. A possible execution trace is shown in Figure 2(d), with input signals shown above the time line and local and output signals below the time line. All signals are absent at the initial tick; at the second tick, the presence of S triggers the start of both threads; at the third tick, I causes the first thread to start sustaining R; and at the fourth tick, the second thread tests R, which is still emitted by the first thread, and emits A, which in turn causes the first thread to perform a weak abort of the sustain R statement. The weak abort means that R is still emitted during this instant—otherwise, we would have a *causality error* in this example, as the emission of R would cause its absence. At this point, at the fourth tick, the first thread has terminated, but the second thread is still in its infinite loop, thus the overall concurrent statement has not terminated yet. However, at the fifth tick, the presence of S causes the (strong) abortion of the concurrent statement, followed immediately—within the same tick—by a restart of both threads. Hence, at the sixth tick, the presence of I causes the first thread to resume sustaining R—only to be again aborted by another occurrence of S at the seventh tick. As this example demonstrates, there may be intricate dependencies among concurrent threads, and preemption and concurrency can be freely nested within each other.

A characteristic of Esterel is that there is a very small number of *kernel statements* and a comparatively large number of *derived statements*. The derived statements can always be replaced by equivalent constructions that involve only kernel statements; the derived statements are merely syntactic sugar, *i. e.*, convenient shorthands for the programmer. Hence, it would theoretically suffice to just implement the kernel statements to implement a reactive processor for Esterel. However, there are derived statements, such as await, which are fairly frequent, and always expanding them into kernel statements would increase the resulting code and slow down execution significantly. This makes the choice of the instruction set for an Esterel processor not trivial; the MEP consciously aims to make a good trade off between a lean instruction set and compact, efficient code.

Considering again the EXAMPLE module from Figure 2(a), there are some Esterel statements that the MEP handles directly, such as the signal declarations, await, (weak) abort, emit, or pause. Other statements, in this example every, sustain, and loop, require statement expansion; the expansion rules applied here are shown in Figure 2(b). Note that the expansion rule for loop involves a goto, which is not part of Esterel, but is part of the MEP instruction set. Furthermore, note that the MEP also offers an instruction that directly corresponds to the sustain statement (see also the example presented later in Figure 4); however, to properly synchronize threads, we must in this case expand the sustain into kernel statements, as illustrated later in Section 3.2. The resulting MEP assembler is shown in Figure 2(c). Some of the instructions, namely those regarding preemption and concurrency, will be explained in the sequel. One may however note already the overall compactness of the code, which has an instruction count that is comparable to the line count of the Esterel source code, despite the intermediate statement expansion.

For implementing the reactive control flow in the MEP, we focus on two issues: how to handle Esterel preemption statements, which must test their trigger signals at each logical tick, and how to interleave the execution of concurrent threads in such a manner that the semantics of the original program is preserved. Each issue is treated in the following.

2.2 Handling Preemption

In the MEP 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, such as weak abortion, a certain trigger signal, and an address range that delineates the preemption block. For example, considering the MEP assembler code in Figure 2(c), the ABORT S, $A1_7$ instruction[1], which implements the abort that resulted from expanding the every statement in the Esterel source, configures a Watcher to perform a strong abortion for the block ranging from the subsequent instruction (line 8) to the instruction preceding label A1 (line 25) whenever the trigger signal S occurs.

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. Note that Esterel allows the arbitrary nesting of preemption blocks of different types, for example a strong abortion may be nested within another weak abortion, which may be nested in a suspension. 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. This scheme is called *Inside/Outside Preemption Range Watching* [16]. Note that here, unlike in the patched processor approach, it is not necessary to continuously execute special instructions that check on the status of each watcher, meaning that the program does not slow down when entering a (nested) abortion block. The Watcher modules operate autonomously, thus also offering a certain type of concurrency, beyond the dedicated concurrency operator.

2.3 Handling Concurrency

A hurdle when implementing concurrency is the need to interleave thread execution to allow communication among threads within the same logical tick. As already illustrated in the example in Figure 2, a thread may be executed partially, then control may jump to another thread, and later return to the first thread, all within the same tick. To handle this, the MEP employs a *multi-threaded architecture*, where 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). As discussed later on, 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 6).

A concurrent Esterel statement with n concurrent threads joined by the ||-operator is translated into MEP assembler as follows. First, threads are *forked*, meaning n new threads will be created, 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 main thread that starts the program is implicitly assigned priority 0, which is the lowest possible priority.) For example, at line 8 in the MEP code shown in Figure 2(c), the PAR 3, P1 creates a thread with priority 3 that starts at label P1. The *end address* of each thread is the address of the instruction immediately after the last instruction belonging to this thread; this

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

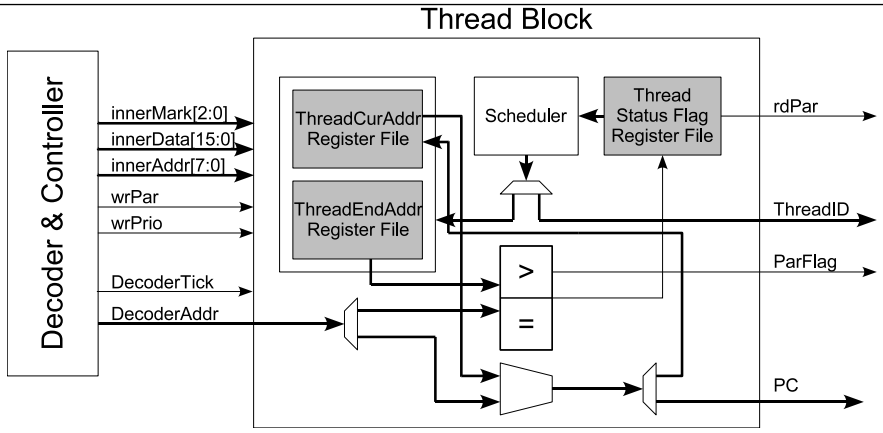

Figure 3: Architecture of the Thread Block

address is stored by the Thread Block as the ThreadEndAddr of this thread. The end address is either given by 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. In the EXAMPLE code, the fork instructions create two threads, the first with address range from line 11 up to (but excluding) line 19, and the second ranging from line 19 up to (but excluding) line 24. Furthermore, the Thread Block still separately keeps track of the incoming thread that executed the fork instructions; for this thread, ThreadCurAddr is set to the ThreadEndAddr of the thread created last. The fork instructions are followed by MEP code for each of the created threads, in the specified address ranges. 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.

In addition to the ThreadCurAddr and ThreadEndAddr, 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. Each thread, including the main thread that starts the program, has its private await counter register, which in case an await-type statement is executed keeps track of how often the specified trigger signal has yet to occur to trigger the await statement. The Reactive Block provides an AWAIT Element for each thread.

At the beginning of each instruction cycle, the Scheduler inspects all active threads. If there are multiple active threads, the Scheduler 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, such as an AWAIT or a PAUSE, 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.

We now have presented the MEP architecture with its instruction set and the elements that are responsible for implementing these instructions, focusing on concurrency and preemption. However, there still remains the issue of how threads must be coordinated such that their execution adheres to the Esterel semantics, which we will discuss further in the following section.

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 requires 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 MEP assembler using the approach presented here. However, these programs can be transformed into equivalent, acyclic Esterel programs, using *e. g.* the technique presented by Lukoschus and von Hanxleden [18], which can then be translated into MEP assembler. Hence, the actual run-time schedule of a concurrent program running on MEP 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 that manages the interleaving according to the priority and the actual program counter of each active thread.

3.1 Thread Ordering in the EXAMPLE

To illustrate how the MEP implements concurrent operations, consider again the Esterel module EXAMPLE in Figure 2, which contains nested preemption and concurrent threads. The generation of the MEP assembler for the EXAMPLE module is in general straightforward; as mentioned before, most common Esterel statements can almost literally be translated into corresponding MEP instructions, and there are direct equivalence rules for the remaining statements. The only non-trivial aspect of code generation is the assignment of priorities when executing threads concurrently. To understand how these priorities are assigned, let us consider the thread scheduling constraints that must be obeyed to run the EXAMPLE module faithful to Esterel's semantics. As we already noted when first considering this example in Section 2.1, the two threads enclosed in the every block can communicate back and forth via the R and A signals, within a logical tick. In this case, there is a *dependency* between the statements that (may) emit these signals (the *dependency sources*) and the statements where these signals are tested (the *dependency sinks*).

First, let us consider the dependency involving R. It is clear which instruction is the dependency source: the EMIT R_{13} instruction. It is also clear which is the dependency sink: the PRESENT R_{21} instruction. This lets us formulate the first dependency present in the EXAMPLE module: whenever the EMIT R and the PRESENT R instructions are executed within the same logical tick, the execution of the EMIT must precede the execution of the PRESENT. In the terminology used in the Event Graph (EG) formalism developed by Closse *et al.* [10], this corresponds to an *order link*. Adopting their notation, we express this scheduling constraint as

$$\text{EMIT } R_{13} \xrightarrow{<<} \text{PRESENT } R_{21}. \tag{1}$$

Now let us consider the dependency involving A. The dependency source is the EMIT A_{22} instruction, in the second thread. However, it is less obvious which is the dependency sink, which we defined as the "statements where these signals are tested." The first thread reacts to A when it has entered the weak abort block, in which case A triggers the abort. Hence, whenever we execute a statement in that block, which in the MEP assembler are the instructions between the $WABORTI_{12}$ and the label $P1B_{18}$, we should also watch for the presence of A. However, closer inspection yields that as this is a weak abort, it suffices to check at the end of each logical tick whether the block is aborted, that is, whenever we execute a non-instantaneous instruction. In this case, the only non-instantaneous instruction in the abort block is the $PAUSE_{16}$. This yields the second scheduling constraint:

$$\text{EMIT } A_{22} \xrightarrow{<<} \text{PAUSE}_{16}. \tag{2}$$

What remains now is to assign priorities in such a way that the scheduling constraints (1) and (2) are met. In EXAMPLE, the first thread is started with priority 3, and the second thread is started with priority 2. If none of the threads would execute any PRIO instruction, the first thread would always be executed before the second thread. This would suffice to ensure meeting Dependency (1); however, Dependency (2) would be violated. To remedy this, the first thread must yield to the second thread before it executes the $PAUSE_{16}$ instruction, to give the second thread a chance to test for the presence of R and to emit an A—and it must regain control again before that PAUSE instruction, to perform the abortion in case A is present. This is achieved by the two PRIO instructions in lines 14 and 15; it is also easy to see that Dependency (1) is still met.

3.2 Assigning Priorities

To obtain a more general understanding of how the priority mechanism influences the order of execution, recall that at the start of each instant, all enabled threads are activated, and are subsequently scheduled according to their priorities. Furthermore, each thread is assigned a priority upon its creation. Once a thread is created, its priority remains the same, unless it changes its own priority with a PRIO instruction, in which case it keeps that new priority until it executes yet another PRIO instruction, and so on. Neither the scheduler nor other threads can change its priority. Note also that a PRIO instruction is considered instantaneous; the only non-instantaneous instructions, which delimit the logical ticks, are the PAUSE instruction and derived instructions, such as AWAIT and SUSTAIN. This mechanism has a couple of implications:

- At the start of a tick, a thread is resumed with the priority corresponding to the last PRIO instruction it executed during the preceding ticks, or with the priority with which it has been created if it has not executed any PRIO instructions. In particular, if we must set the priority of a thread to ensure that at the beginning of a tick the thread is resumed with a certain priority, it does not suffice to execute a PRIO instruction at the beginning of that tick; instead, we must already have executed that PRIO instruction in the preceding tick.

- A thread is executing only if no other active thread has a higher priority. Once a thread is executing, it continues to execute until it executes a non-instantaneous statement, or until its priority is lower than that of another active thread. While a thread is executing, it is not possible for other inactive threads to become active; furthermore, while a thread is executing, it is not possible for other threads to change their priority. Hence, the only way for a thread's priority to become lower than that of other active threads is to execute a PRIO instruction that lowers its priority below that of other active threads.

So far, we have generated the MEP assembler code manually, including the priority assignment; a compiler for this is currently under development. Regarding a general, systematic scheme for assigning thread priorities, we should first convince ourselves that it is always possible to come up with such a set of priorities that the semantics of the original program is preserved. To that end, recall the restriction we imposed on the Esterel program regarding the dependency structure: we forbid cyclic dependencies, there must always exist a static ordering (schedule) of statements, which

can be obtained for example from the Event Graph [10] for the program or from the Concurrent Control Flow Graph [12].

The next question then is whether our priority control mechanism always allows to enforce such a schedule. To answer this question, we first consider the start of a logical tick: as observed above, in this case all threads are activated, and if we want to enforce an order in which each thread starts to execute, we can do so by placing a PRIO instruction immediately before the non-instantaneous instruction that denotes the start of the tick. In some cases, this may require expanding the non-instantaneous statement until the PAUSE statement becomes exposed, as has been the case in the EXAMPLE module, where we had to dismantle the original SUSTAIN R statement.

Next, we consider the situation where the schedule demands a context switch after a tick has been started. As observed above, a thread can yield to another thread by lowering its priority below the priority of that other thread. As the program cannot have cyclic dependencies (scheduling constraints) and schedules are static and finite, we can thus assign priorities to each code segment, where each statement that must precede another instruction must have a higher priority than that other instruction. Then each thread may continuously lower its priority while executing a tick, always yielding to the next thread on the schedule; if this next thread is not active, it follows the next thread, and so on, until all threads have finished their tick. Only towards the end of a tick a thread might have to raise its priority again—not to influence the scheduling of the current tick, but to get the proper thread ordering at the beginning of the next tick. Regarding the potential increase in code size due to additional PRIO instruction, a conservative estimate—that disregards the ordering from the priorities assigned during thread creation—is that for each dependency in the program, there may be up to two PRIO instructions needed to enforce it.

4 The Interaction of Concurrency and Preemption—The EXAMPLE Module

To study how the MEP combines concurrency and preemption, it is instructive to work through the example code in Figure 2(c), again using the execution trace in Figure 2(d).

After starting the module, the initial thread (thread 0) is enabled and active. The control stays at the AWAIT S_6 instruction 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, as already discussed in Section 2.3. 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 second 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 third 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 PriorityController 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 process mentioned above can also handle several threads within a preemption body. The seventh tick in Figure 2(d) 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, the thread 2 is also disabled and deactivated by the abortion. Therefore, unlike the processor behavior in the fourth tick, no signal is emitted.

5 Related Work

As mentioned in the introduction, most work on the synthesis of Esterel has focused on the synthesis into hardware [3] or into software [6, 12]. In addition, Esterel has also been used in hw/sw co-design [1]. Only fairly recently one has started to consider the execution of Esterel program on special-purpose reactive processors.

To our knowledge, Dayaratne, Roop, Salcic *et al.* [19, 20, 11] were the first to propose this approach, and they have developed a series of patched reactive processors. Their EMPEROR architecture, based on the earlier RePIC processor, was also the first and to our knowledge the so far only architecture that aims to support Esterel's concurrency operator directly. Here, every Esterel thread is assigned to an independent RePIC processor, a separate thread control unit handles the synchronization and communication between processors. The fork and join instructions are used to assign threads. To illustrate, consider the module SLAP in Figure 4(a), which contains three threads, and the corresponding EMPEROR code for each thread, in Figure 4(b1/b2/b3) (taken from [11]). The code is generated by the EMPEROR Esterel Compiler, which uses an Unrolled Concurrent Control Flow Graph (UCCFG) that also unveils thread dependencies. This information is used to generate sync instructions, to ensure that signals are not tested before they are emitted. This works in this example, as signals are always emitted unconditionally; however, it is not clear how this would work in case of arbitrary control flow, as the sync instruction appears to only test for the guaranteed presence of a signal, and not for its guaranteed absence. It appears that this could be remedied, for example using a priority base scheme as proposed here. What we see as the more serious limitation is that it is not clear how this design would support the arbitrary nesting of concurrency and preemption. Furthermore, this solution is relatively hardware intensive. For a comparison of overall compactness, Figure 4(c) lists the corresponding interleaved MEP assembler program.

Another custom reactive processor implementation is the Kiel Esterel Processor (KEP) pro-

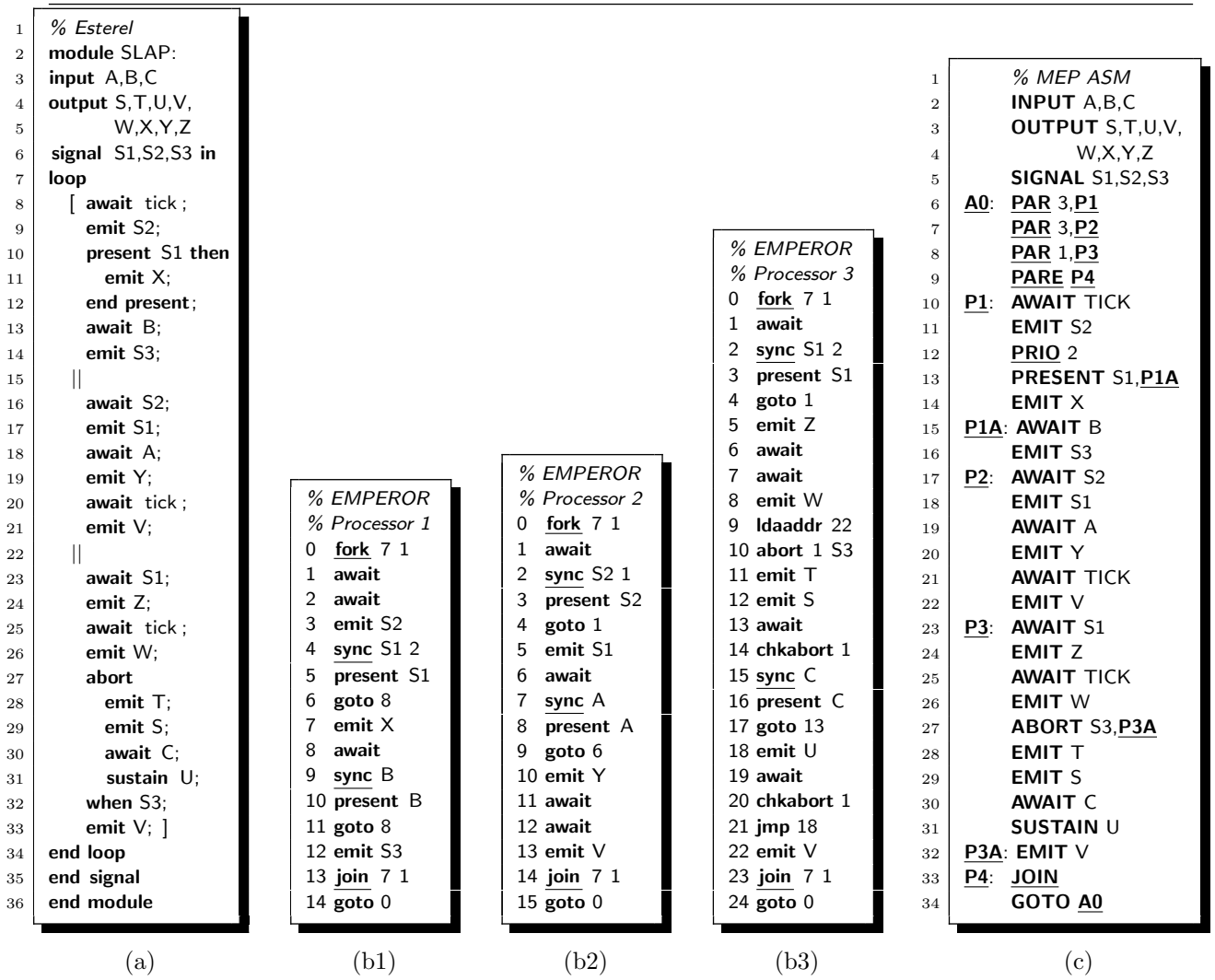

Figure 4: An Esterel module involving the parallel statement (a), and the corresponding assembler code for EMPEROR (b1/b2/b3) and for the MEP (c).

posed by Li, von Hanxleden *et al.* [16, 17, 15]. To our knowledge, their design was the first to properly handle weak and strong abortion, and is the only one to handle suspension. Their preemption handling is also fairly efficient, in that it does not require additional instructions to continuously check for the presence of abortion triggers. Furthermore, the authors also presented an efficient method to compute its worst case reaction time. However, the major limitation of the KEP is that it does not support the || operator; it appears that this could be remedied using the interleaving approach presented in this paper.

6 Experimental Results

The EXAMPLE module in Figure 2 is used to briefly compare the concurrency and preemption handling abilities between the MEP 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. Note that due

[2]The code for the MCS51 has been compiled by the Keil C51 compiler V6.12. The level 8 (default) optimization is used. The lengths of MCS51's instructions vary; here, a *word* represents a complete assembler line. For the Microblaze, code was compiled by gcc version 2.95.3-4. The level 2 (default) optimization is used.

	MEP-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 1: Comparison of codes size and RAM usage for EXAMPLE between the MEP-A (see Table 3), MCS51, and Microblaze processors.

Module	Thread Cnt/ Preemption nesting	Code Size (in words)				RAM Usage (in bytes)			
		Microblaze			MEP	Microblaze			MEP
		V5	V7	CEC		V5	V7	CEC	
SPEED	1/2	276	1081	253	11	52	48	36	6
CONTROLLER	1/1	560	1226	487	27	72	88	68	4
DEBOUNCE	1/1	392	1198	299	28	72	68	48	12
ALARM_COMPARE	1/1	315	1109	265	14	56	60	40	8
SPEEDOMETER	1/2	328	1145	293	20	64	60	44	8
DASHBOARD_TIMER	1/1	617	1388	541	65	100	96	68	20
FRC	1/1	375	1163	313	18	76	80	48	8
BAT_DIAG	1/0	487	1274	378	66	80	76	56	16
VER_ACC_DIAG	1/0	433	1229	303	41	72	68	52	12
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
LONG_SPEED_STRAT	1/0	573	1306	319	56	80	72	52	12

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

to our thread handling mechanism, threads will incur RAM resources for recording the thread address (ThreadCurAddr), the await counter, and the thread end address. Therefore, the formula for computing RAM usage in byte is: $RAM_{usage} = Reg_{num} * 2 + 4 + Thread_{num} * 5$. Here, Reg_{num} equals the occurred variable registers number, and $Thread_{num}$ equals the maximum degree of concurrency.

To evaluate the performance of the MEP further, we use some standard test cases [4, 1, 8]. Those modules are typical Esterel applications, which not only contain the reactive statements, but also include arithmetic and logical data handling. The modules were first manually translated into the MEP assembler program and then compiled to the MEP 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 MEP implementation and the Microblaze software implementation. The optimized data path of the MEP results on average in an 90% reduction of codes size and 67% reduction of RAM usage when compared with the best result of the Microblaze implementation. In practice, the majority of Esterel statements can be translated into MEP assembler instructions word by word.

As mentioned in the introduction, the MEP has been designed to be highly configurable. Table 3 compares five different MEP variants which include different elements to target various applications. The MEP-E offers similar functions to the EMPEROR2 (an EMPEROR version which can handle 2 threads). 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 MEP-E that offers a level 6 preemption nesting depth. When considering clock frequencies, note also that the RePIC uses four clock cycles to execute an instruction cycle, whereas the MEP uses only three clock cycles; *i. e.*, when the processors run at the same clock frequency, the MEP's instruction cycle

	MEP-A	-B	-C	-D	-E	EMPEROR2
Input/Output	11/11	16/16	11/11	32/32	24/24	24/24
Valued Input/Output	2/2	2/2	3/3	3/3	2/2	2/2
Thread Number	4	16	16	32	2	2
Preemption Nest	2	4	6	8	6	4+4
Counter Value Range	255	65535	65535	65535	1	1
Variable Register Num	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 3: Performance comparison between the MEP and EMPEROR.

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

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

period is just 75% of that of the RePIC's. Regarding the logic cell count, implementations of the MEP and the EMPEROR2 are based on different FPGA chips. However, the basic units of those two chips have similar structures, functions, and speeds [17]. Therefore, we can assume that logic cell counts are comparable. As a result, for the similar processor configuration as the EMPEROR2, the MEP-E uses 56% less resources and achieves a 1.6 times instruction clock speedup—and the MEP typically takes significantly less instructions to implement the same behavior.

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

7 Conclusions and Outlook

This paper presents the MEP, 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 MEP 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 MEP. Valued signals and signal counters, local signal declarations, and the pre operator are also supported directly.

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 of these approaches.

Regarding future improvements of the MEP, we plan to add a reconfigurable logic block to allow the efficient detection of compound events, such as waiting for the simultaneous occurrence of two signals; so far, these have to be handled by expanding them into separate individual signal tests. However, what we see as even more promising at this point is to explore the efficient compilation from Esterel onto the MEP, in combination with a static analysis of the maximal reaction time in the presence of concurrency.

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. KEP2 (Kiel Esterel Processor 2): The Esterel Processor. Technischer Bericht 0506, Christian-Albrechts-Universität Kiel, Institut für Informatik und Praktische Mathematik, Apr. 2005. `http://www.informatik.uni-kiel.de/reports/2005/0506.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, M. Biglari-Abhari, and A. Bigdeli. A New Reactive Processor with Architecture Support for Control Dominated Embedded Systems. In *IEEE International Conference on VLSI Design*, pages 189–194. IEEE CS Press, Jan. 2003.

[20] 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.

[21] 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.