STARPro — A new multithreaded direct execution platform for Esterel

Simon Yuan1  Sidharta Andalam2  Li Hsien Yoong3  Partha S. Roop4  Zoran Salcic5

Electrical and Computer Engineering Department
University of Auckland
Auckland, New Zealand

Abstract
We propose a fully pipelined, multithreaded, reactive processor called STARPro for direct execution of Esterel. STARPro provides native support for Esterel threads and their scheduling. In addition, it also natively supports Esterel’s preemption constructs, instructions for signal manipulation, and a notion of logical ticks for synchronous execution. As a side effect of the proposed architecture, we have developed a new intermediate format called UCCFGsd (unrolled concurrent control-flow graph with surface and depth) that closely resembles the Esterel source. A compiler based on UCCFGsd has been developed for code generation. We have synthesized STARPro and have carried out a range of benchmarking experiments. Experimental results reveal substantial improvement in performance and code size compared to software compilers. We also excel in comparison to recent reactive architectures, by achieving an average speed-up of 37% in worst-case reaction times, and a speed-up of 38% in average-case reaction times. This has been achieved by utilizing fewer hardware resources, while incurring an average code size increase of 46%.

Keywords: Compilation, concurrency, Esterel, reactive processors, synchronous

1 Introduction

The programming language Esterel [4] belongs to the family of synchronous languages [2]. Due to its formal semantics, all correct Esterel programs are guaranteed to be reactive and deterministic [3]. These properties greatly simplify the formal verification of programs, while at the same time, provide predictable runtime behaviour. Hence, there has been a great deal of interest in using Esterel for the design and validation of a special class of embedded systems, called reactive systems [10].

Esterel provides constructs to describe concurrently executing statements. Each concurrent component executes in lock-step, evolving in discrete instants of time,
1 module demoloop :
2 input A, R ;
3 output B, C, D, E ;
4 abort
5 loop
6     await A ;
7     emit B ;
8     present C then
9     emit D
10    end
11    end loop
12 ||
13 loop
14     present B then
15     emit C
16     end ;
17     pause ;
18     present B else
19     emit E
20    end
21 when R
22 end module

Fig. 1. The demoloop example: (a) Esterel source (b) Unrolled Concurrent Control-Flow Graph (UCCFGsd)

known as a tick. Such synchronous execution is achieved by taking a snapshot of input signals at the start of each tick, performing some computation, and emitting all outputs before the start of the next. Concurrent statements may communicate back and forth with each other within a tick, making such communication conceptually instantaneous. Such synchronous execution guarantees that each reaction in Esterel is atomic in every possible sense. This makes race conditions, common in concurrent programming, impossible in Esterel.

While such powerful features make it intuitive to write specifications in Esterel, its compilation and efficient execution has been non-trivial. We illustrate some aspects of this complexity using the example shown in Fig. 1(a).

The demoloop example demonstrates three key features of the language: concurrency, synchronous preemption, and instantaneous broadcast communication. It consists of a single module, with its input and output interface signals declared on lines 2 and 3 respectively.

The program consists of two parallel threads, beginning from lines 5–11, and lines 13–21 respectively. Both threads are enclosed by their respective non-terminating loops. The ‘∥’ operator is used to denote their synchronous concurrency. The first thread begins by awaiting for input A (line 6). The await statement always pauses for the first instant, and will continue to do so in subsequent instants until its delay predicate (A in this example) becomes true. Once A becomes present, B
will be emitted.

The emission of B is broadcast to the second thread, which instantaneously reacts to it by emitting C (lines 14–16). The emission of C, in turn, provokes the instantaneous emission of D back in the first thread (lines 8–10). Meanwhile, in the second thread, the pause statement on line 17 marks the end of its tick, and implies synchronization with the first thread before the start of the next instant. The possibility of instantaneous dialog, as well as this implicit synchronization at each instant between concurrent components, are two factors that make the efficient software implementation of Esterel challenging.

In the subsequent instant, the first thread will again wait for another occurrence of A. If A is not present this time, the latter thread will respond by emitting E (lines 18–20). The ability to react instantaneously to both signal presence and absence is a crucial issue, making the scheduling of programs in Esterel non-trivial.

Meanwhile, should the signal R become present at any time after the first instant, both the parallel threads will be preempted at precisely the same instant by the abort construct enclosing them on lines 4–22. This will instantaneously terminate the program. This distinct behaviour for the start and resumption instants are referred to as the surface and depth behaviours respectively [16].

Several approaches exist for dealing with these complexities in the compilation and execution of Esterel programs. These include hardware compilation [3], software compilation for general-purpose microprocessors [6,16,9], and architecture-specific compilation for reactive processors optimized for Esterel [7,11]. While the translation of Esterel to digital circuits in hardware is relatively straightforward, the generation of efficient software code has been challenging. Software compilers typically map Esterel programs into another language, such as C, so that they can be executed on standard microprocessors. Consequently, concurrent statements in Esterel need to be interleaved and appropriately scheduled in order to produce an equivalent sequential program. This requires artificial synchronization mechanisms to be added to preserve Esterel’s semantics. Such mechanisms introduce extra execution overhead and increase the required memory footprint.

The architecture-specific approach relies on custom microprocessors that have been augmented with an instruction set, which enables the efficient mapping of Esterel statements to assembly code. This approach yields very compact software code, as well as efficient execution, and will be the focus of this paper. We present a novel multithreaded processor, named STARPro (Simultaneous multiThreaded Auckland Reactive Processor), and an Esterel compiler for it, that achieves significant speed-up and code size compaction over traditional methods for software implementations of Esterel.

Multithreading in our processor design differs to conventional multithreaded processor in the sense that threads are interleaved rather than executing in parallel. The hardware provides the facility to store and context switching between threads.

The rest of this paper is organised as follows. Section 2 reviews previous work related to architecture-specific execution of Esterel. Section 3 then presents STARPro’s processor architecture, which is followed by a description of its instruction set architecture (ISA) in Section 4. Section 5 will cover aspects on the code generation from the intermediate format and the execution semantics. In Section 6,
we show the experimental results obtained for some benchmarks. We finally end with some concluding remarks in Section 7.

2 Related work

The EMPEROR multiprocessor architecture [7] was the first attempt at the direct execution of Esterel using a set of reactive processor cores. These cores communicate and synchronize with each other using a thread control block to achieve synchronous execution. It executed Esterel programs by resolving signal dependencies during run-time using a dual-rail encoding of signals [19]. This approach, while achieving good execution times, required excessively high hardware resources.

In contrast to the approach taken in EMPEROR, new contributions were also made to the idea of reactive processing through the KEP series of processors [13,12,14,11]. The KEP series of processors are custom designed architectures that have evolved with incremental support for executing Esterel. The most recent processor, KEP3a [11], is capable of preserving the semantics of the full language. It also provides a multithreaded execution platform to support the concurrency in Esterel. This approach has yielded impressive code size compaction and execution times, thus affirming again the benefits of reactive processors for executing Esterel.

However, there are many improvements that could be made over KEP’s approach to reactive processor design and Esterel execution. At present, KEP3a employs a non-pipelined architecture, which supports Esterel’s semantics almost entirely in hardware. This approach results in a complex hardware design, with a consequently lower operating clock frequency.

In contrast, this paper presents a novel multithreaded processor, named STARPro, that provides an alternative approach to direct execution compared to KEP3a. STARPro uses variable tick lengths and a pipelined architecture to obtain much better average performance compared to KEP3a. This has been achieved using far fewer logic gates for processor implementation, while maintaining code sizes that are comparable to KEP3a for a given Esterel program.

Plummer et al [15] have explored another approach of executing Esterel using a virtual machine (VM). A VM provides Esterel supporting instructions for direct execution, similar to the way STARPro works. The key difference is a virtual machine is implemented as software, where STARPro is a hardware platform. Both approaches are superior in code size when compared with traditional Esterel compilers, however the VM approach is significantly slower than traditional Esterel compilers [15] and the VM cannot handle host procedure calls written in for example C.

3 The STARPro Processor Architecture

STARPro’s design was based on an existing processor, called REMIC [17]. REMIC is a three-stage pipelined reactive processor that was inspired by Esterel, though it was not designed to provide support for executing Esterel. REMIC has a Reactive Functional Unit (RFU), attached to the control unit and data path of the processor core, that provides instruction set support for efficient handling of asynchronous I/O in reactive applications. The RFU, however, is not well-suited for Esterel programs,
which require I/O to be handled synchronously. Hence, we have developed the Esterel Support Unit (ESU) to replace the RFU within REMIC, as illustrated in Fig. 2(a). The ESU still interfaces with the control unit and the datapath as before, but enables synchronous handling of signals, as well as multithreading to support concurrency in Esterel.

The ESU itself (see Fig. 2(b)) consists of the Abort Handling Block (AHB) for dealing with preemptions, and the Thread Control Block (TCB) for multithreading support. Unlike most other simultaneous multithreading processors, STARPro does not use separate register files for each thread, but it does, however, provide separate program counters and auxiliary registers for abort handling for each thread. In the following, we will first explain the TCB and AHB, before discussing how the two interact.

3.1 The Thread Control Block (TCB)

The purpose of the TCB is twofold: it is used to store thread context, and to perform thread scheduling. As depicted in Fig. 2(c), the TCB itself is composed of a scheduler, a thread table, and a TCB control unit.
The thread table stores the current program counter and the abort context\(^6\) associated with the current thread. Both the program counter and the abort context are sufficient to fully describe a thread’s context in STARPro. The number of threads that can be stored in the thread table is parameterizable in our design, and is limited only to the hardware resources available.

The thread table is indexed by the Thread ID register. The entry indexed by that register determines the thread which is currently being executed. When the \texttt{LD\_TCB} signal is asserted, write access is enabled to the table for a thread context to be saved. Switching between threads then become a simple matter of changing the value stored in the thread ID register. A new thread ID value is loaded through the \texttt{Rx} bus connected to the datapath. During the processor’s reset, the thread ID register will be initialized to zero. Consequently, the ID of the root thread of all programs will be assigned a default value of zero by the STARPro compiler.

The other remaining important component of the TCB is the scheduler. The scheduler stores the priority and a notion of a \textit{local tick} for each thread. We say that the \textit{local tick} for a thread has elapsed whenever a \texttt{pause} statement in it is reached. This differs from the global \textit{tick} for an entire Esterel program, which only elapses when all running threads have completed their local ticks. In STARPro, the \texttt{pause} statement is mapped to the \texttt{PAUSE} instruction, which is used within the processor to indicate the completion of the \textit{local tick} for a given thread.

The scheduler will always select the thread with the highest priority for execution. In doing so, it ignores all the threads that have either completed their local ticks, or are otherwise inactive. A thread is considered to be inactive if its priority number is set to the lowest possible priority. When the local tick of all the currently active threads elapse, the global tick completes, and a compiler-generated management thread is selected to sample new inputs and to clear all output signals for the next global tick.

The distinction between local and global ticks is actually the key idea that facilitates the use of variable tick durations in STARPro. This idea was first introduced in [7], and has been adapted for our current design. By relying on the completion of individual local ticks to determine the final duration of a global tick, the global tick duration is dynamically changed and equal to the actual computational time required.

### 3.2 The Abort Handling Block (AHB)

The AHB is used to monitor aborting signals, and to trigger the appropriate preemptions if necessary. In Esterel, the priority of the abort construct depends on the level of its nesting. An outer abort construct will always have higher priority than those nested below it. The AHB supports this feature by providing hardware-based priority resolution for the abort constructs. The depth of nested aborts is fully parameterizable in our design. Fig. 2(d) depicts an AHB that has been configured with four levels of aborts for each thread.

The AHB relies on the abort context provided by the TCB to trigger abortions. An abort context consists of the following elements:

\(^6\) The abort context will be described in Section 3.2.
• **Rx**: This is the bus that connects to a 16-bit register selected from the register file in datapath. The register has to be loaded with the status of I/O signals in a bunch of 16s at a time from memory. It is updated at every tick, and is used by the AHB to evaluate the status of the aborting signals.

• **ASR** (Abort Signal Register): This stores the ID of the signal which needs to be monitored during execution of an abort body.

• **AAR** (Abort Address Register): This stores the continuation address, to which the thread must jump, should preemption happens.

• **ATF** (Abort Type Flags): STARPro supports the different types of abortions in Esterel. Abortions can either be *strong* or *weak*, and may be either *immediate* or *non-immediate*. These are orthogonal to each other, resulting in four distinct behaviours for abortions in Esterel.

• **ALC** (Abort Level Count): Each thread can consist of an arbitrary number of nested aborts. This register is incremented as the depth of nested aborts increases.

The TCB stores the ASR, ATF, and ALC for each thread, and provides these abort context of the current running thread to the AHB. The AHB does not contain any memory element and it is purely control. When the AHB detects the presence of the preemtions signal, it provides an index (CA_SEL in Fig. 2(b)) that selects the continuation address (AAR, stored in the TCB), as well as an updated ALC, back to the TCB. The TCB directly provides the continuation address to the datapath, and hence the AAR is the only abort context not passed to the AHB. The activation and deactivation of abort levels are also controlled by the TCB control unit.

The most significant difference between the AHB and the preemption *watchers* in KEP is how the preemtion is monitored. STARPro relies on explicit checks at appropriate times using an instruction, where the *watchers* in KEP relies on a physical tick signal in hardware. The correctness of abort semantics of the AHB relies on the compiler at compile time, where the *watchers* rely on the runtime hardware behaviour. The difference in the two approaches results in simpler preemption hardware design for STARPro.

The AHB relies on the control unit to indicate to it when to check for aborting conditions. This is necessary to preserve Esterel's synchronous preemption, and to correctly implement both strong and weak abortions. This indication from the control unit is provided using STARPro's **CHKABORT** instruction. When the **CHKABORT** signal arrives, the AHB control unit will check for abortions in the following manner:

• For strong abortions, the AHB starts by evaluating the status of aborting signals, beginning from the outermost to the innermost abort level.

• For weak abortions, the AHB starts by evaluating the status of aborting signals, beginning from the innermost to the outermost abort level.

We describe the reason for this difference. An abort construct in Esterel may contain an abort handler. If an abort handler exists, the handler will be executed when an abortion takes place. A weak abort offers the current executing abort body one last chance to complete the current tick before preemting it.

Let us now consider the scenario where a weak abort is nested within another weak abort, and both of them have an associated abort handler. In the instant
where the aborting signals for both constructs are present, the program will first execute the inner abort handler up to, but not including, the pause statement (if any). Execution will then branch to the outer abort handler. This chaining of weak abort handlers is the reason behind the different order of checking between the two types of abort constructs. By checking a weak abort beginning at the innermost level, the preemption can be propagated from the inner to the outer levels of aborts.

4 The STARPro Instruction Set Architecture

STARPro uses a 32-bit instruction format. Apart from the common instructions found on a typical RISC processor, we introduce additional Esterel-oriented instructions to support multithreading, signal testing, and preemption. The syntax and description of these instructions are summarized in Table 1.

The number of I/O signal ports is parameterizable. I/O signals are memory-mapped, which enables signal manipulation to be also done using instructions that read from and write to memory. This design allows any arbitrary arithmetic or logic operation to be performed on signals. STARPro also does not have any dedicated instruction for strong immediate aborts. Instead, this is derived using the ABORT instruction, together with the PRESENT instruction to test for the aborting condition in the starting instant.

We illustrate the reactive instructions using the example in Fig. 1(a). The equivalent STARPro assembly code for that example is shown in Fig. 3. We start, first, by explaining the reactive instructions used in this program, and defer the discussion on the translation process to Section 5.

Starting with ABORT on line 20, the first abort level is configured here to watch for signal 14 (signal R). Then, the program forks two concurrent threads. This is accomplished using the SPAWN instruction on lines 22 and 24, which initializes thread 1 and 2 to start at label T1 and T2 respectively. Line 26 creates the special global tick handler thread. The PCHANGE instruction on lines 29 and 30 set the initial priority of thread 1 and 2. Finally, the CSWITCH instruction on line 31 completes the thread-forking process by setting the current (in this case, the root) thread inactive.

### Table 1: Esterel-oriented instructions

<table>
<thead>
<tr>
<th>Instruction Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPAWN Reg StartAddr</td>
<td>Creates a new thread</td>
</tr>
<tr>
<td>CSWITCH Priority</td>
<td>Context switches to a thread and updates the current thread priority</td>
</tr>
<tr>
<td>PAUSE Reg</td>
<td>Marks the end of a tick and context switch to a thread</td>
</tr>
<tr>
<td>PCHANGE Reg Priority</td>
<td>Changes the priority of a thread</td>
</tr>
<tr>
<td>PRESENT Sig Reg ElseAddr</td>
<td>Checks the presence of a signal</td>
</tr>
<tr>
<td>ABSENT Sig Reg ElseAddr</td>
<td>Checks the absence of a signal</td>
</tr>
<tr>
<td>ABORT Sig Addr</td>
<td>Initializes the AHB for strong abortion</td>
</tr>
<tr>
<td>WABORT Sig Addr</td>
<td>Initializes the AHB for weak abortion</td>
</tr>
<tr>
<td>WIABORT Sig Addr</td>
<td>Initializes the AHB for weak immediate abortion</td>
</tr>
<tr>
<td>CHKABORT Reg Type</td>
<td>Checks for preemption of type Type (strong/weak) only</td>
</tr>
<tr>
<td>ENDABORT</td>
<td>Deactivates the current abort level</td>
</tr>
</tbody>
</table>

At this point, either thread can be scheduled as both have the same priority in the starting instant. The scheduler selects whichever it
finds first and does a context switch to the selected thread. The PAUSE instructions found on several lines across thread 1 and 2 essentially does the same thing as CSWITCH, except that the PAUSE, in addition, also marks the end of a local tick for the currently executing thread.

In order to achieve a simpler hardware design, the abort constructs are kept local to the threads that they have been declared in. When a thread is forked, the aborts within it are duplicated in the child threads, as was done in [7]. Due to this, thread 1 and thread 2 begin with an ABORT instruction on lines 36 and 55 respectively. These two lines do the same initialization as was done on line 20. Inside the abort body, the CHKABORT instruction is appropriately inserted at local tick boundaries, such as on lines 38 and 61. As the mnemonic suggests, it checks for the abort at the point of execution of this instruction. It requires a register to be selected and the abortion type (strong or weak) to be given. The abortion type operand of a CHKABORT instruction allows the AHB to check only the type of aborts initialized with the same type and ignores the other type. When the end of an abort body is reached, the ENDBORT instruction (see line 33) is used to deactivate the current abort level, and it will not be checked again until it is reactivated. The ENDBORT marks the end of an abort body, and the instruction following it is simply a branch to the address of the next instruction after the abort construct.

The PRESENT instruction, found in many places such as line 39, is functionally equivalent to Esterel’s present statement. It tests for the presence of a signal. If it is present, the following instruction executes, otherwise the else-address is taken. The ABSENT instruction is similar to PRESENT, except that it checks for a signal’s
absence instead. It is provided for code compaction, and to avoid unnecessary
branching so as to minimize the flushing of the processor’s pipeline.

5 Code generation and execution semantics

In order to generate assembly code from the Esterel source, the STARPro compiler
uses an intermediate format, called the \textit{unrolled concurrent control-flow graph with
surface and depth} (UCCFG$_{sd}$), to represent a given Esterel program. We first
present the UCCFG$_{sd}$, and then, describe how assembly code is generated from it.

5.1 Unrolled Concurrent Control-Flow Graph

The UCCFG$_{sd}$ is a variant of the UCCFG intermediate format, which was first
introduced in [7]. However, the UCCFG is not capable of fully preserving Es-
terel’s semantics, especially for statements that have distinct start and resumption
behaviours (also known as \textit{surface} and \textit{depth} behaviours), like that of the \textit{await}
statement described in the example of Fig. 1(a). Some statements, like \textit{emit}, are log-
ically instantaneous, while others, like the \textit{await} statement, consumes time (\textit{ticks}).
Such non-instantaneous statements have distinct surface and depth behaviours.

To overcome this, we have modified the original UCCFG format, and extended
it to explicitly capture both the surface and depth behaviour of every statement
in Esterel. This approach adapts the technique used in [16], where the start and
resumption behaviours are differentiated using distinct \textit{surface code} and \textit{depth code}.

In [16], each pass of the control-flow graph (CFG) represents an execution of
just one \textit{tick}. Thus, to compute the reaction for multiple \textit{ticks}, the CFG would
have to be executed within a loop. The selection of the appropriate surface and
depth code in each pass of the graph is accomplished using state variables. In
contrast, STARPro can directly preserve state information during execution through
its \textit{PAUSE} instruction, which essentially mimics Esterel’s \textit{pause} statement by keeping
the program counter for each thread unchanged until the start of the next \textit{tick}.

In UCCFG$_{sd}$, tick boundaries are marked by \textit{pause} nodes, denoted as an arrow
with a black bar on the right, as depicted in Fig. 1(b). Using these \textit{pause} nodes,
the loop required to execute the CFG of [16] can be completely unrolled. Hence,
instead of using a switch statement to select between the surface and depth code as
done in [16], code for STARPro can be conveniently represented in this form:

\[
surface(\text{code}); depth(\text{code})
\]

Using this approach, Esterel statements can be mapped to UCCFG$_{sd}$ nodes
rather intuitively. The mapping of the \textit{abort} statement, however, would merit
further elaboration. This is actually done in two stages: first, by marking the start
and end of the body, and subsequently, by placing the \textit{check abort} node at the
desired points. Depending on the type of the abort, placement of the \textit{check abort}
 nodes varies with respect to the \textit{tick} boundary. To handle the four types of aborts,
we use the following general rules:

\begin{itemize}
  \item A \textit{strong abort} always checks for preemption at the start of a \textit{tick}. Therefore, a
  \textit{check abort} node is placed immediately after each \textit{pause} node.
\end{itemize}
- A weak abort always checks for preemption at the end of a tick. Therefore, a check abort node is placed immediately before each pause node.
- The immediate version of a strong abort checks for preemption before entering the abort body. A present node is simply added before the abort node to test for the aborting condition.
- The non-immediate version of a weak abort also has the check abort nodes inserted before the pause node of the first instant. The reason for this is described below.

The handling of a non-immediate weak abort is subtle when its abort body contains a loop. The first pass through the loop is different from all subsequent passes, as the surface part of the loop body gets folded back into the depth after the first pass. In this case, the abortion condition need not be checked during the first pass of the loop, but would need to be done in subsequent passes. In order to handle this, the AHB has been designed to ignore the first CHKABORT instruction encountered for weak non-immediate aborts using an additional status bit.

The demoloop example contains a strong abortion. In Fig. 1(b), this is indicated through the start abort node. Within the abort body, a check abort node is placed after each pause node in the two forked threads. An end abort node is placed at the end of the abort body. The start and end abort pair, thus, defines the scope of the abort in the graph. The two sibling threads in Fig. 1(b) presented here do not end with end abort nodes. These two threads will never reach the end of the abort body due to the loops. For this same reason, the two threads will only join should the abort take place via the check abort nodes. The last check abort node below the join node will only have an effect if the abort in the root thread has an abort handler, which is not the case in this example.

Following the preliminary construction of the UCCFG_{sd}, the nodes are clustered into distinct sets to facilitate their static scheduling. This is similar to that done in the CEC compiler [9]. However, unlike CEC, our scheduling is done in hardware using a priority instruction, similar to [11].

Each pause node marks the end of a cluster. Additionally, nodes may also be separately clustered due to data dependency arcs, as can be seen from the demoloop example of Fig. 1(b). A context switch node is inserted at such points. The clusters are then assigned priorities based on the depth of the dependency chain. In the case of cluster C3 in thread 1, the depth of the dependency chain is two (C2 to C6, C6 to C3), and hence, it is assigned a priority of two.

The starting clusters of the two forked threads in the demoloop example have the same priority, and thus it makes no difference in terms of program behaviour which cluster to be scheduled for the first instant. If C1 is to be executed first, the pause node in C1 marks the end of the local tick for thread 1, and it will no longer be scheduled until all active threads have completed their local tick. Subsequently the context switch node in C5 will only cause a context switch to itself in this case. The context switch node in C5 is not redundant because the signal producer (C2) can potentially execute in the same instant as the consumer (C6). Thus breaking the first instant of the consumer thread into two clusters (C5 and C6) ensures the potential producer will always be executed prior to the consumer by assigning the consumer cluster with a lower priority.
5.2 Handling schizophrenic programs

Statements in an Esterel program may potentially be executed multiple times within a single *tick*. Such programs are referred to as *schizophrenic* [3,18]. This phenomenon may result in a single local signal declaration in Esterel being executed multiple times within a *tick*. Esterel compilers typically handle this by creating multiple copies of the same signal (known as *incarnations* [3]) for each new signal declaration that may potentially occur within the *tick*. This not only complicates the compilation process, but also significantly leads to an increase in memory footprint due to code duplication.

STARPro’s ISA is able to handle schizophrenic programs correctly without requiring multiple incarnations of a signal to be created. Local signals are simply implemented as variables in STARPro. Whenever the local signal is declared (re-declared when looping back), the corresponding variable will be (re-)initialized. This effectively introduces a fresh copy of the signal by replacing the previous incarnation. This does not pose any problem even for local signals that are shared between multiple threads, as Esterel’s semantics always ensure that parallel statements are synchronously terminated before the local signal enclosing them can be re-declared. This prevents any thread from entering a new scope of the local signal, while other threads are still in the previous scope.

5.3 Code generation

The nodes in the UCCFG.sd map very closely to STARPro instructions. Code generation from the UCCFG.sd is greatly simplified as there is almost a direct mapping between nodes and assembly instructions. For example, the *context switch* and *pause* nodes directly translate to the CSWITCH and PAUSE instructions respectively.

The less straightforward ones in Fig. 1(b) are *fork* and *join* nodes. Forking involves the following actions: spawning each child thread, setting the priority and join status (stored as a variable) of each thread, and finally, context switching to one of the child threads and marking the parent thread as inactive. Lines 21 to 31 in Fig. 3 are the translated output for the *fork* node. Joining requires checking the join status, and making sure that all the child threads in the same fork are ready to join before reviving the parent thread. In the demoop example, thread 2 would finish before thread 1. When thread 2 reaches the *join* node, it clears the corresponding bit in the JOIN variable, and checks the join status to see if all other sibling threads are ready to join. It then deactivates itself by executing the CSWITCH instruction with a priority of 255. These are shown on lines 47 to 53 and 68 to 74 in Fig. 3. When all threads are ready to join (the JOIN variable evaluates to zero), the last executing thread of the fork revives the parent thread by changing its priority to a priority lower than the currently executing cluster. When the CSWITCH instruction is next executed, the scheduler will select the parent thread.

6 Experimental results

STARPro was synthesized on both CycloneII and Spartan3 FPGA. Its hardware resource usage on Spartan3 is presented in Table 2 for comparison with KEP3a
Since the number of threads supported by STARPro is parameterizable, we synthesised the design for 2 to 512 threads to examine the relationship between the resource usage and the number of threads. Table 2 also shows this relation for KEP3a, when configured for 2 to 120 threads. The table clearly shows that STARPro consumes far less resources than KEP3a for a given number of threads, using an order of magnitude less for the starting configuration of two threads. The number of I/O ports (memory mapped) configured for STARPro will not significantly change the hardware resource usage as accessing I/O ports is a purely generic memory operation.

<table>
<thead>
<tr>
<th>STARPro</th>
<th>Max. Threads</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices</td>
<td>@167MHz</td>
<td>320</td>
<td>391</td>
<td>488</td>
<td>678</td>
<td>851</td>
<td>1478</td>
<td>2710</td>
<td>5251</td>
<td>10137</td>
</tr>
<tr>
<td>Gates (k)</td>
<td></td>
<td>24</td>
<td>24</td>
<td>26</td>
<td>29</td>
<td>52</td>
<td>89</td>
<td>193</td>
<td>322</td>
<td>682</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>KEP3a</th>
<th>Max. Threads</th>
<th>2</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>60</th>
<th>80</th>
<th>100</th>
<th>120</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices</td>
<td>@60MHz</td>
<td>1295</td>
<td>1566</td>
<td>1871</td>
<td>2369</td>
<td>3235</td>
<td>4035</td>
<td>4569</td>
<td>5233</td>
</tr>
<tr>
<td>Gates (k)</td>
<td></td>
<td>295</td>
<td>299</td>
<td>311</td>
<td>328</td>
<td>346</td>
<td>373</td>
<td>389</td>
<td>406</td>
</tr>
</tbody>
</table>

Table 2
Quantitative comparison of hardware resource usage

Fig. 4. Comparison of hardware resource usage between KEP3a and STARPro

Fig. 4 portrays the difference in usage of resources between the two processors. Both processors exhibit a linear relation between the number of logic gates and the level of multithreading support provided. KEP3a, however, incurs a very high initial gate usage at 295k, while STARPro only consumes an initial gate count of 22.9k. This gap narrows as the number of threads increases. Extrapolating this graph reveals that the resource usage of STARPro and KEP3a will meet when the number of threads reach 9000. This means that for an application with more than 9000 threads, the resource usage of STARPro will begin to exceed that of KEP3a. Since most realistic embedded applications are anticipated to use well below 9000 threads, STARPro can be considered to be more efficient in terms of resource usage.

The benchmark programs presented here have been selected from EstBench [8]. All the selected programs are also present in [11] for comparison. The benchmarks were evaluated in three aspects. First, we compare the worst-case and average reaction times for KEP3a and STARPro. The optimized results for KEP3a were taken from [11]. Then, we compare the generated code size. Finally, we show the effects of a pipelined architecture in terms of the speedup obtained.

To evaluate STARPro’s compiler, we compared it against four other Esterel compilers, namely CEC v0.4 [9], EEC2 [19], and the V5 [4] and V7 Esterel compilers [5]. These compilers produced C code from the Esterel source, which we compiled
for the NIOS-II [1] 32-bit RISC processor. NIOS is a softcore processor, provided
by Altera as part of its development tools for its CycloneII FPGA. All C programs
were compiled using the nios2-elf-gcc compiler with level-2 optimization (-O2).

We start by comparing the execution times of the two reactive architectures,
KEP3a and STARPro. Execution traces were generated using Esterel Studio’s
Coverage Analysis tool, which were also used for the benchmarks in [11]. The worst-
case and average-case reaction times for KEP3a and STARPro are shown in Table 3.
Although KEP3a has almost a one-to-one mapping between Esterel statements and
assembly instructions, STARPro is still able to achieve, on average, a 37% speed-up
in worst-case reaction time (WCRT), and a 38% speed-up in average-case reaction
time (ACRT). The exception where KEP3a significantly excels in performance is the
runner example. The example involves counting of signal occurrences. In KEP3a,
such counting is done in hardware, whereas STARPro relies on software to do this.

The code sizes for the software compilers were obtained from the size of the
object files generated by the nios2-elf-gcc compiler. The approach taken by
KEP3a and STARPro consistently resulted in much more compact code compared
to the conventional software approach, as depicted in Table 4. The runner example
again shows that code sizes are more compact with KEP3a’s hardware-oriented
approach. STARPro has on average 40% larger code size than KEP3a.

Table 5 shows the performance gain from a pipelined architecture. The clock
cycles shown in the table represent the total number of clock cycles required to
complete each program with a given execution trace. The same applies to the
instruction count. Multiplying the instruction count by three, we obtain the total
number of clock cycles required for a non-pipelined processor. The effect of pipelining
results in an average speedup of 1.83.

In summary, execution of Esterel using reactive processors yields much better
code size and execution times compared to conventional software approaches. The
STARPro architecture proposed here, also provides much better execution times with less hardware resources compared to the latest KEP processor, while suffering only a minimal code size penalty. In general, the STARPro architecture is simpler than KEP3a’s, as its instructions are also much simpler. Unlike KEP3a, STARPro does not have a one-to-one mapping of Esterel statements to its ISA. Instead, it relies on a combination of hardware and software. Consider, for example, count delays in Esterel. KEP has direct support for this in its ISA, while STARPro does this in software. Also, for abortions in Esterel, the AHB requires CHKABORT instructions to be inserted at appropriate points to emulate Esterel behaviour. This approach leads to a slight code size penalty compared to KEP3a. However, STARPro’s simpler hardware not only operates at a higher frequency, but also executes Esterel programs faster, in both the worst and the average cases.

7 Conclusions

We have presented a direct execution platform for Esterel with multithreading support. Esterel programs compiled for STARPro are significantly faster than Esterel software compilers, while achieving smaller code size at the same time. In comparison to an existing Esterel-optimized processor, KEP3a, STARPro achieves superior execution times, while suffering minimal code size penalties. This has been accomplished with a simpler hardware design, which at the same time, consumes significantly less hardware resources. The ability of the pipelined STARPro processor to operate at 167MHz in contrast to the non-pipelined operating frequency of KEP3a of only 60MHz further adds to the elegance of the STARPro approach.

Both the STARPro hardware and compiler have received minimal optimization at this stage. The hardware has a lot of room for further reduction in resource usage. The hardware scheduler, in particular, can be improved to scale gate usage more optimally with increase in the number of supported threads.

On the compiler side, host procedures in Esterel (usually implemented in C) cannot yet be compiled directly to STARPro. It is envisioned that the STARPro compiler would eventually be able to support C data computation, as well as the reactive part of Esterel.

8 Acknowledgement

The authors would like to acknowledge the assistance of Rohan Aggrawal in STARPro’s implementation, and Chia-Hao Chou for his contribution to the design.
References


