Synchronous Deterministic Parallel Programming for Multi-Cores with ForeC

Eugene Yip, Alain Girault, Partha S Roop, Morteza Biglari-Abhari

Synchronous Deterministic Parallel Programming for Multi-Cores with ForeC

EUGENE YIP, Software Technologies Research Group, University of Bamberg, Germany
ALAIN GIRAULT, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, France
PARTHA S. ROOP, Department of ECE, The University of Auckland, New Zealand
MORTEZA BIGLARI-ABHARI, Department of ECE, The University of Auckland, New Zealand

Embedded real-time systems are tightly integrated with their physical environment. Their correctness depends both on the outputs and timeliness of their computations. The increasing use of multi-core processors in such systems is pushing embedded programmers to be parallel programming experts. However, parallel programming is challenging because of the skills, experiences, and knowledge needed to avoid common parallel programming traps and pitfalls. This paper proposes the ForeC synchronous multi-threaded programming language for the deterministic, parallel, and reactive programming of embedded multi-cores. The synchronous semantics of ForeC is designed to greatly simplify the understanding and debugging of parallel programs. ForeC ensures that ForeC programs can be compiled efficiently for parallel execution and be amenable to static timing analysis. ForeC's main innovation is its shared variable semantics that provides thread isolation and deterministic thread communication. All ForeC programs are correct by construction and deadlock-free because no nondeterministic constructs are needed. We have benchmarked our ForeC compiler with several medium-sized programs (e.g., a 2.274 line ForeC program with up to 26 threads and distributed on up to 10 cores, which was based on a 2.155 line non-multi-threaded C program). These benchmark programs show that ForeC can achieve better parallel performance than Esterel, a widely used imperative synchronous language for concurrent safety-critical systems, and is competitive in performance to OpenMP, a popular desktop solution for parallel programming (which implements classical multi-threading, hence is intrinsically nondeterministic). We also demonstrate that the worst-case execution time of ForeC programs can be estimated to a high degree of precision.

CCS Concepts: • **Computing methodologies → Parallel programming languages**; • **Software and its engineering → Semantics**; *Source code generation.*

Additional Key Words and Phrases: programming language, semantics, parallelism, synchronous, determinism, reactive programming, multi-core, worst-case execution time, code generation

ACM Reference Format:
Eugene Yip, Alain Girault, Partha S. Roop, and Morteza Biglari-Abhari. 2023. Synchronous Deterministic Parallel Programming for Multi-Cores with ForeC. *ACM Trans. Program. Lang. Syst.* 45, 2, Article 11 (June 2023), 78 pages. https://doi.org/10.1145/3591594

Authors' addresses: Eugene Yip, Software Technologies Research Group, University of Bamberg, 96047 Bamberg, Germany; Alain Girault, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France; Partha S. Roop, Department of ECE, The University of Auckland, Auckland 1142, New Zealand; Morteza Biglari-Abhari, Department of ECE, The University of Auckland, Auckland 1142, New Zealand.

0164-0925/2023/6-ART11 $15.00
https://doi.org/10.1145/3591594

1 INTRODUCTION

We present the programming language **ForeC**[1], aimed at programming *parallel safety-critical embedded systems* deployed on *multi-core processors*. The most important challenges raised by the design and implementation of such a system are:

(A) It must be *dependable* and *functionally safe* [22, 37, 93]. Certification is often required w.r.t. safety standards such as DO-178B [102], IEC 61508 [60], and ISO 26262 [62] depending on the application domain. These safety standards require programs to be deterministic, understandable, and maintainable.

(B) It must be *real-time* and *reactive* because it interacts *continuously* with its *environment* [53], at a speed imposed by this environment. The system's environment can be a physical process that it monitors and controls (e.g., a nuclear power plant), the physical environment it moves into (e.g., the 3D space for an unmanned aerial vehicle), or both. Interactions with the environment are performed through input and output signals, and the system must react to new inputs as soon as possible. The key point is that the correctness of an embedded system depends both on the output of its computations and on the timeliness of completing these computations [74, 127].

(C) It must be *concurrent* because the physical process that it interacts with is almost always a collection of several independant parts (e.g., a flight control system monitors and controls the plane's rudders, ailerons, engines, and so on). Concurrency is sometimes called "expression parallelism" and it concerns the source program.

(D) When the platform it is deployed on is parallel (e.g., a multi-core processor), the system must also be *parallel*. Parallelism is sometimes called "execution parallelism" and it concerns the generated code.

(E) It must involve *complex data processing*, which requires complex data structures (arrays, matrices, structs, pointers, ...) and control instructions (conditionals, switches, loops, ...). In other words, it must allow both data-centric and control-centric applications to be programmed.

These challenges are difficult to address *jointly* in a programming language. We present in the following sub-sections how ForeC achieves this.

1.1 Safety-Critical Data Processing

This part addresses challenges (A) and (E). ForeC extends C with specific language features to address the challenges (B) and (C), as will be explained in the next sub-sections. The advantages of choosing C are that it provides the essential data and control structures to program both low-level and high-level systems, that many compilers exist for basically all possible instruction sets, and that many libraries exist for basically all peripherals. For these reasons, C is a popular programming language used in the safety-critical embedded domain. This is how ForeC addresses challenge (E).

Still, C's semantics [61] include unspecified and undefined behaviors [70]. Thus, to address challenge (A), strict coding guidelines [56, 63, 85] are used by safety-critical programmers to help write well defined programs that are deterministic, understandable, maintainable, and easier to debug [43, 54]. The coding guidelines can be grouped into three main concerns:

code clarity to avoid as much as possible the occurrence of ambiguous code statements;

defensive programming to eliminate nondeterminism due to unspecified and undefined behaviors;

runtime reliability to prevent the occurrence of runtime errors even when the program has been written correctly.

[1]ForeC is pronounced "foresee", because the goal of the language is to help programmers foresee the execution and timing behavior of their programs.

1.2 Concurrent and Parallel Programming

This part addresses challenges (C) and (D). On the one hand, *parallel programs* have been developed with the goal of executing several threads simultaneous over multiple compute resources. C is a popular language for programming embedded systems with support for multi-threading and parallelism provided by third-party libraries, compilers, and runtime support [34]. Notable examples include Pthreads [115], OpenMP [91], and MPI [84]. Unfortunately, these multi-threading solutions are inherently *nondeterministic* [73] because they permit nondeterministic behaviors, e.g., *race conditions* over shared variables. More generally, the lack of formal semantics for the programming model can lead to ambiguous behaviors.

Besides the non-determinism issue, parallel programming is challenging because of *performance* issues. Indeed, several studies [75] have shown that, without careful tuning, parallel programs executed on multi-cores can perform worse than their sequential counterparts!

On the other hand, *concurrent programs* use the idea of simultaneous thread execution to simplify the modeling of concurrent behaviors at design time. When the system being programmed is intrinsically concurrent, as it is the case of many embedded real-time systems, using a concurrent programming language is essential. The concurrent threads are then typically compiled away to produce sequential code. Otherwise, the programmer has to manually interleave all the concurrent behaviors, which is error prone and time consuming.

ForeC addresses challenges (C) and (D) by being a multi-threaded extension of C, equipped with a compiler that generates parallel code for execution on multi-core processors.

1.3 Synchronous Programming Languages

This part addresses challenges (A), (B), and (C). ForeC is a *synchronous* programming language. In the same spirit that procedural imperative programming languages (e.g., C, Pascal, Ada, ...) ease the design and implementation of sequential programs by providing advanced sequential control structures (procedures, functions, loops, conditionals), synchronous programming languages ease the design and implementation of concurrent reactive programs by providing *ideal* temporal control structures (temporal loops, preemption, concurrency, synchronization barrier).

Synchronous languages [10] are based on a rigorous mathematical model; this facilitates program comprehension, system verification by formal methods [10], and generation of correct-by-construction implementations [40, 87]. Figure 1 depicts a synchronous program, defined as a set of concurrent communicating threads, within its physical environment. A synchronous program reacts to its environment by sampling its inputs, allowing its threads to execute computations, and emitting corresponding outputs back to the environment. Inputs are sampled to avoid the need to use interrupts which are sources of unpredictable delays that degrade a system's timing-predictability. Conceptually, each *reaction* is triggered by the ticking of a logical global clock.

Threads contain *synchronization barriers*: when a thread executes and reaches its next synchronization barrier, it pauses and we say that the thread has completed its *local tick*. When all threads in the program have completed their respective local ticks, we say that the program has completed its *global tick*. This constitutes a reaction.

Central to synchronous languages is the *synchrony hypothesis* [10], which states that the execution of each reaction is considered to be atomic and instantaneous. If a program's global clock is fast enough (faster than the occurence of events from the environment), it gives the illusion to the user that the program reacts *continuously* to its environment. Of course this requires the program's reaction time to be shorter than its clock period. In classical synchronous languages, e.g., Esterel [15], concurrent threads communicate instantaneously with each other (dashed arrows in Figure 1) thanks to the synchrony hypothesis. This is the key element that makes temporal and concurrent

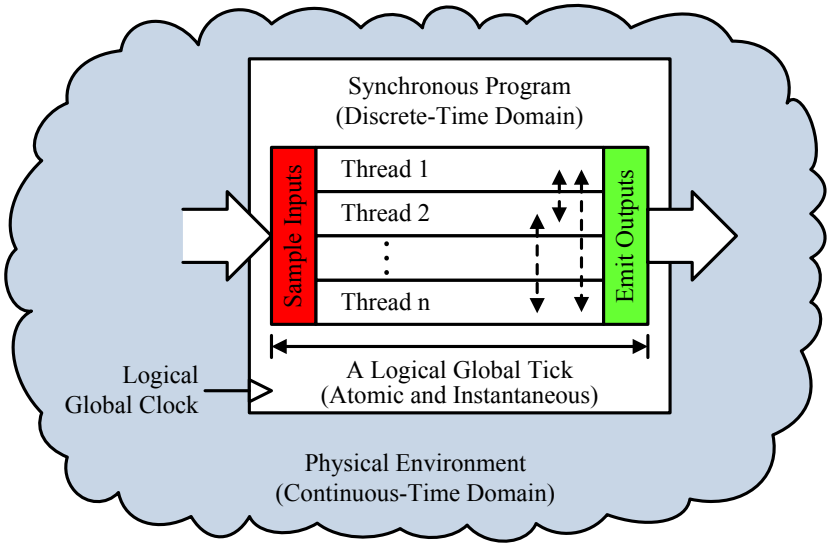


Fig. 1. Synchronous model of computation.

```
1  input int A, B, R;
2  output int O;
3
4  void main(void) {
5    while (1) {   // Loop each R
6      O=0;
7      abort {
8        par(tA(),tB());   // Await A and await B
9        O=1;   // Emit O
10       while (1) { pause; O=0; }
11     } when (R==1);
12   }
13 }
14 void tA(void) {
15   do { pause; } while (A!=1);
16 }
17 void tB(void) {
18   do { pause; } while (B!=1);
19 }
```

Fig. 2. ABRO synchronous program in ForeC.

behaviors easier to to reason with. Once the embedded system has been implemented, the synchrony hypothesis has to be validated by ensuring that the worst-case execution time [126] of any global tick must not exceed the minimal inter-arrival time of the inputs.

We use the ForeC language to illustrate key features of the synchronous paradigm in Figure 2. The ForeC program implements the well-known ABRO example [14][2], which has the following specification: "*Emit an output* `O` *as soon as two inputs* `A` *and* `B` *have occurred. Reset this behavior*

[2]ABRO is the "hello world" counterpart of synchronous programs.

each time input R *occurs.*" Note that the "emit ... as soon as ..." and the "reset ... each time ... occurs" parts of this specification are tricky to program correctly. Synchronous programming languages precisely offer dedicated control structures and a deterministic semantics to do it.

Lines 1 and 2 define the reactive interface of the program, which consists of the input and output variables, respectively with the `input` and `output` keywords. In this example, `0` and `1` encode respectively the absence or presence of these variables. To wait for inputs `A` and `B` to occur, the parallel threads `tA` and `tB` are forked using a `par` statement (line 8). Thread `tA` *pauses* at each tick until input `A` becomes present (line 14). This is achieved with a `pause` statement, which demarcates the end of a thread's local tick and acts as a *synchronization barrier* across all threads to complete a global tick. Once threads `tA` and `tB` have both terminated due to the respective occurrence of inputs `A` and `B`, the `par` statement also terminates. Execution continues on line 9 and output `O` is made present. The program then waits at each tick and sets output `O` back to absent (line 10). The above behavior is enclosed by an `abort` statement (lines 7–11) that terminates its body when input `R` is present. Together with an outer loop (line 5), the reset behavior is achieved on every occurrence of `R`.

In classical synchronous languages, threads communicate instantaneously by emitting *signals* and testing for either their *presence* or *absence*. Synchronous programs are considerably difficult to parallelize [44, 66, 134] due to the need to resolve instantaneous thread communication and the associated causality issues. At runtime, all potential signal emitters must be executed before all testers of a signal. A causality issue arises if this is not possible. E.g., this occurs in the statement "`present X else emit X end`" which has no behavior in Esterel because the signal `X` cannot be absent and present during the same global tick. Interestingly, the Synchronous Constructive (SC) model of computation [109] offers a more permissive but still consistent notion of causality, called *Sequentially Constructive Causality* [121, 122], and can for instance give a behavior to the statement "`present X else emit X end`" thanks to the *init-update-read* principle. Thus, causality is commonly resolved by *compiling away* the concurrency into a sequential intermediate representation [40]. From the resulting sequential program, low-level parallelism may be extracted and distributed code can be produced [8, 44, 66, 134].

Classical synchronous languages only support basic data computations and delegate complex data computations to a host language, for instance C, but this makes challenge (E) impossible to address. Consequently, C-based synchronous languages have been developed to provide data handling at the language level with support for synchronous concurrency, preemption, and thread communication; we can cite Reactive C [18], Esterel C Language (ECL) [72], and PRET-C [3]. Such languages appeal to C programmers because the barrier to learn a synchronous language is reduced. However, they are not parallel, meaning that their compiler only generates sequential code.

To summarize this part: Challenge (A): ForeC is equipped with a formal semantics that is deterministic, which lends itself to formal verification; Challenge (B): The synchrony hypothesis can be validated thanks to static timing analysis; Challenge (C): ForeC is also a synchronous concurrent programming language.

1.4 Contributions

We propose the ForeC parallel programming language for simplifying the deterministic parallel programming of embedded multi-core systems. All the synchronous languages designed for the single-core era must be reinvented to address the programming of multi-cores. To this end, ForeC is a C-based synchronous language designed specifically for the programming of multi-cores.

ForeC brings together the formal deterministic semantics of synchronous languages and the benefits of C's control and data structures. For instance, one can fork threads statically with the `par` statement. Complex temporal control statements can be used (e.g., preemption with the `abort`

statement) and temporal loops can be programmed by relying on the C loop constructs. A key innovation is ForeC's shared variable semantics that provides thread isolation and deterministic thread communication. This is made possible with the `pause` statement (synchronization barrier): schematically, when each thread has reached its `pause`, a global tick occurs and shared variables are resynchronized with a so-called `combine` function à la Esterel. Besides, each thread works with its own local copy of each shared variable (thread isolation). Several combine policies are provided and users can program their own combine functions, allowing a great variety of behaviors for shared variables.

ForeC belongs to the family of time-predictable programming languages, so for instance threads are statically allocated by the compiler to the available cores (as described in an architecture specification file provided by the user). This allows static timing analysis thanks to the ForeCast tool. Through benchmarking, we demonstrate that ForeC can achieve better parallel performance than Esterel and be competitive with OpenMP. Finally, although the formal semantics of ForeC presented here is for threads that synchronize at the same clock rate, we have developed an extension [45] that supports multi-rate threads.

1.5 Paper Organization

The ForeC language is introduced in Section 2 and its formal semantics is presented in Section 3. The compiling approach is described in Section 4 and the benchmarking results in Section 5. Section 6 reviews the literature on parallel and synchronous programming languages. Section 7 concludes this paper. Finally, several appendices provide additional material, including the proofs for reactivity and determinism (Appendix A), illustrations of how the semantics works (Appendix B), and illustrations of the combine policies and the behavior of shared variables (Appendix C).

2 THE FOREC LANGUAGE

Execution platforms have evolved from single-cores to multi-cores. Hence, all the synchronous languages designed earlier (e.g., Esterel [15], Lustre [50], Signal [49], Esterel C Language [72], Reactive Shared Variables [19], and PRET-C [3]) must be remodeled to address the challenges raised by multi-cores. Over 30 years of R&D has demonstrated that synchronous programming languages are very well suited to the design of safety-critical real-time systems [21, 111]. Moreover, the ideal modeling of time brought by the synchrony hypothesis makes them good candidates for PRET programming. This motivates our proposed ForeC language that is dedicated to the programming of multi-cores. ForeC inherits the benefits of synchrony, such as determinism and reactivity, along with the benefits and expressive power of the C language, such as control and data structures. This is unlike conventional synchronous languages, which treat C as an external host language. A key goal of ForeC is to provide deterministic shared variable semantics that is agnostic to scheduling. This goal is essential for the reasoning and debugging of parallel programs. This section presents ForeC with a UAV running example. Parallel programming patterns, such as point-to-point and broadcast communication, fork-join, map-reduce, software pipelining, and early-termination, can be readily applied in ForeC programs, but this topic is outside the scope of this paper.

2.1 Overview and Syntax

ForeC is a synchronous language that extends a *safety-critical subset* of C [16, 64] (see Section 1.1) with a minimal set of synchronous constructs. Figure 3 gives the extended syntax of ForeC and Table 1 summarizes the informal semantics. We briefly describe the statements, type specifiers, and type qualifiers allowed in the C subset:

Statement	*st*	::= *c_st* | `pause` | `par`(*st*, *st*) | `weak`? `abort` *st* `when immediate`? (*cond*) | *st*; *st*
Variable Declaration	*var_decl*	::= *c_storage*? *tq*? *c_type var_id init*? *comb*
Type Qualifier	*tq*	::= *c_tq* | `input` | `output` | `shared`
Combine Clause	*comb*	::= `combine` *policy* `with` *func_id*
Combine Policy	*policy*	::= `all` | `mod` | `new`

Fig. 3. Syntactic extensions to C.

Table 1. ForeC constructs and their semantics

`input`: Type qualifier to declare an input, the value of which is updated by the environment at the start of every global tick.
`output`: Type qualifier to declare an output, the value of which is emitted to the environment at the end of every global tick.
`shared`: Type qualifier to declare a shared variable, which can be accessed by multiple threads.
`combine` *policy* `with` *func_id*: Combine clause of a shared variable declaration to specify the policy and function for reconciling parallel accesses.
`pause`: Pauses the executing thread until the next global tick.
`par`(*st*, *st*): Forks two statements *st* as parallel threads. The `par` terminates when both threads terminate (join back).
`weak`? `abort` *st* `when immediate`? (*cond*): Preempts its body *st* when the condition *cond* evaluates to *true*. The optional `weak` and `immediate` keywords modify its temporal behavior.

C statements (*c_st*): Expressions in a statement can only be constants, variables, functions, pointers, and arrays that are composed with the logical, bitwise, relational, and arithmetic operators of C. Although the use of pointers and arrays is allowed, they can make static dataflow analysis difficult [24] because of pointer aliasing. Thus, we assume that pointers are never reassigned to point to other variables. Direct management of the memory is not allowed, i.e., neither `malloc` nor `free`. All C control statements, except `goto`, can be used. These are the selection statements (`if–else` and `switch`), loop statements (`while`, `do–while`, and `for`), and jump statements (`break`, `continue`, and `return`).

C storage class specifiers (*c_storage*): The C `typedef`, `extern`, `static`, `auto`, and `register` specifiers can be used.

C type specifiers (*c_type*): All the C primitives can be used, e.g., `char`, `int`, and `double`. Custom data types can be defined using `struct`, `union`, and `enum`.

C type qualifiers (*c_tq*): All the C `const`, `volatile`, and `restrict` qualifiers can be used.

The additional ForeC statements include a barrier (`pause`), fork/join (`par`), and preemption (`abort`) statement. Like C, extra properties can be specified for a variable using type qualifiers: the ForeC type qualifiers are for an environment interface (`input` and `output`), or for sharing

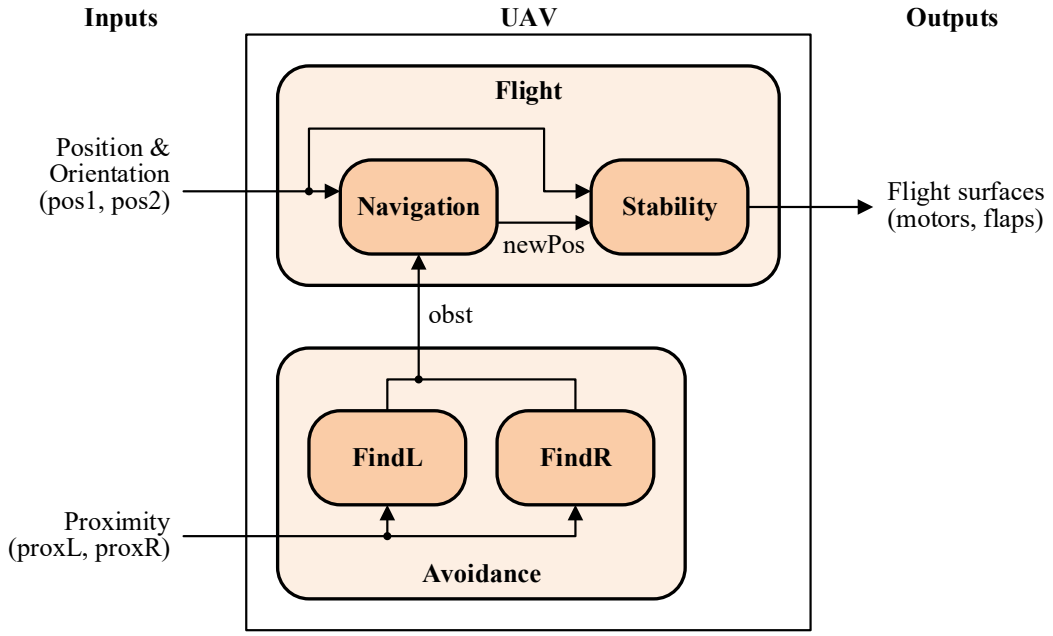


Fig. 4. Tasks of the UAV.

among threads (`shared`). When declaring a `shared` variable (see *var_decl* in Figure 3), the `combine` clause is needed to specify the policy and function for reconciling parallel accesses by threads (see Section 2.6). In addition to the strict C-coding guidelines described in Section 1.1, ForeC forbids the use of recursive function calls and recursive thread forking to ensure static WCRT analyzability.

As a running example to illustrate the ForeC language, we describe the design of an unmanned aerial vehicle (UAV) inspired by the Paparazzi project [88]. A UAV is a remotely controlled aerial vehicle commonly used in surveillance operations. Figure 4 presents the functionality of the UAV as a block diagram of tasks. The UAV consists of two parallel tasks called `Flight` and `Avoidance`. The `Flight` task consists of two parallel tasks called `Navigation` and `Stability`. The `Navigation` task localizes the UAV with on-board sensors, updates the flight path, and sends the desired position to the `Stability` task. The `Stability` task controls the flight surfaces to ensure stable flight to the desired position. The `Avoidance` task consists of two parallel tasks called `FindL` and `FindR`. These tasks use on-board sensors to detect obstacles around the UAV and sends collision avoidance data to the `Navigation` task.

Figure 5 is a ForeC implementation of the UAV system represented abstractly in Figure 4. Figure 6 is a possible execution trace of Figure 5 to help illustrate the execution of ForeC programs. Section 1.3 described the execution behavior of synchronous programs. To recap, the threads of a synchronous program execute in lock-step to the ticking of a *global clock*. In each global tick, the threads sample the environment, perform their computations, and emit their results to the environment. When a thread completes its computation, we say that it has completed its *local tick*. When all the threads complete their local ticks, we say that the program has completed its *global tick*. In Figure 6, the first three global ticks are demarcated along the left-hand side.

```
1  #include <uav.h>
2  input int pos1,pos2,proxL,proxR;
3  output int motors=0,flaps=0;
4
5  void main(void) {
6    shared int obst=0 combine new with min;
7    par(Flight(&obst),Avoidance(&obst));
8  }
9
10 void Flight(shared int *obst) {
11   shared int newPos=0 combine new with plus;
12   par(Navigation(&newPos,obst),Stability(&newPos));
13 }
14
15 void Navigation(shared int *newPos,shared int *obst) {
16   while (1) {
17     *newPos=plan(pos1,obst);
18     pause;
19   }
20 }
21
22 void Stability(shared int *newPos) {
23   while (1) {
24     motors=thrust(pos2,newPos);
25     flaps=angle(pos2,newPos);
26     pause;
27   }
28 }
29
30 void Avoidance(shared int *obst) {
31   while (1) {
32     par(FindL(obst),FindR(obst));
33     pause;
34   }
35 }
36
37 void FindL(shared int *obst) { *obst=find(proxL); }
38 void FindR(shared int *obst) { *obst=find(proxR); }
39
40 int min(int th1,int th2) {
41   if (th1<th2) {
42     return th1;
43   } else {
44     return th2;
45   }
46 }
47
48 int plus(int th1,int th2) {
49   return (th1+th2);
50 }
```

Fig. 5. A ForeC program for the UAV running example.

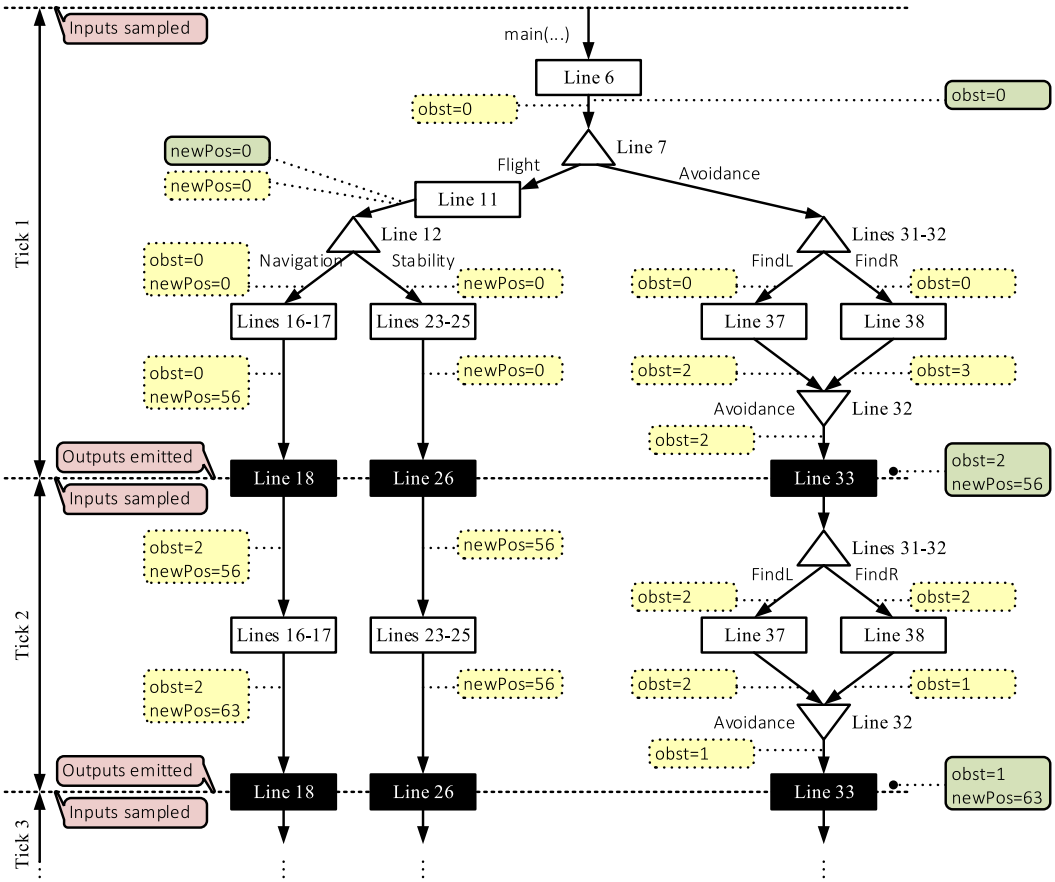


Fig. 6. Possible execution trace for Figure 5.

In Figure 5, the UAV program starts with the inclusion of a C header file (line 1) for the functions used in the program and the global variable declarations (lines 2–3) to interface with the environment. Line 2 declares inputs to capture sensor readings. Inputs are read-only and their values are updated by the environment at the start of every global tick. Line 3 declares outputs for the actuation commands for the flight motors and surfaces. Outputs emit their values to the environment at the end of every global tick. Inputs and outputs can only be declared in the program's global scope. The left-hand side of Figure 6 shows the sampling of inputs and emission of outputs at the start and end of each global tick, respectively.

Like traditional C programs, the function `main` (line 5) is the program's main entry point and serves as the initial thread of execution. Line 6 declares a variable that can be shared amongst threads (see Section 2.4). In Figure 6, the states of shared variables are given inside solid round boxes at specific points along the execution trace. Line 6 declares a shared variable `obst` to store information about the closest obstacle.

At line 7, the `par` statement forks the `Flight` (line 10) and `Avoidance` (line 30) functions into two parallel *child* threads. We refer to the threads by their function names, e.g., the `Flight` and `Avoidance` threads. The forking of threads is represented in Figure 6 as triangles. At line 12, the `Flight` thread forks two more parallel child threads, `Navigation` (line 15) and `Stability` (line 22),

creating a hierarchy of threads. The par is a blocking statement that terminates only when both its child threads have terminated and joined together. The joining of threads is represented in Figure 6 as inverted triangles.

After the Navigation, Stability, FindL, and FindR threads have forked, they start executing their respective bodies. For example, the Navigation thread enters the while-loop (line 16) and computes a new desired position for the UAV and stores it in the shared variable newPos (declared at Line 11). Next, the pause statement *pauses* the thread's execution (line 18), acting as a synchronization barrier. In Figure 6, the pause statements are shown as black rectangles and the program completes a global tick when all the threads pause, indicated by a dotted horizontal line across the pause statements.

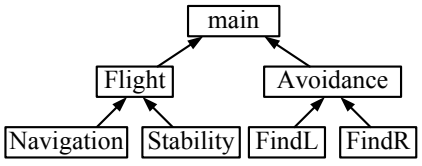
Every time a thread starts its local tick, it creates *local copies* of all the shared variables that its body accesses (reads or writes). The local copies are initialized at the start of the global tick with the values that have been resynchronized at the end of the previous global tick. We use combine functions to compute these resynchronized values (details below). The shared variables declared in the program remain distinct from the threads' local copies. When a thread needs to access a shared variable, it accesses its local copies instead. Thus, the changes made by a thread cannot be observed by other threads, yielding mutual exclusion and thread isolation. Moreover, only sequential reasoning is needed within a thread's local tick. In Figure 6, the states of a thread's copies are shown inside dotted round boxes throughout the execution trace. For example, when the Navigation thread starts its first local tick, it has a copy of obst and newPos (values equal to 0). When its local tick ends, its copy of newPos has been set to 56.

To enable thread communication, the copies of each shared variable are automatically *combined* into a single value when the threads join and when the global tick ends. This is achieved by a programmer-specified *combine function*. In Figure 5, the combine function for obst (line 6) is min (line 40), specified by the combine clause. The combine clause also specifies that only the copies with new values are combined (new since the last global tick). In global tick one of Figure 6, the FindL and FindR threads set new values (2 and 3) to their copies of obst. When these threads join, these new values are combined to 2 and assigned to their parent (Avoidance) thread's copy of variable obst. Meanwhile, the Navigation thread only reads its copy of obst. Thus, when global tick one ends, the value of the shared variable obst is set to 2 by the min function. Had there been more copies with new values, then these copies would have been combined and assigned to obst before the next global tick started. We say that the shared variables are *resynchronized* at the end of each global tick. In Figure 6, the resynchronized values are shown on the right inside solid round boxes, e.g., obst = 2 and newPos = 56. The shared variables start each global tick with their resynchronized values. For the first global tick only, the resynchronized value of a shared variable is its initialization value.

Appendix C describes more examples of combine functions and how more than two copies are combined. The following sections elaborate on the details of local and global ticks, fork/join parallelism, shared variables, and preemption.

2.2 Local and Global Ticks

We say that a thread completes its *local tick* when it pauses or terminates. When a (parent) thread forks some child threads, the parent completes its local tick when all of its children complete theirs. For example, in Figure 5, the Avoidance thread starts its first local tick by forking the child threads FindL and FindR (line 32). Assuming that the find function does not pause (i.e., its body does not contain a pause statement), both child threads complete their local tick by terminating. After the child threads join, the Avoidance thread reaches a pause (line 33) and completes its first local tick. A program completes its *global tick* when all its threads have completed their respective local ticks.



(a) Thread genealogy.

`Flight`: Child of `main`.
`Flight`: Parent of `Navigation` and `Stability`.
`Flight`: Sibbling of `Avoidance`.
`Flight`: Independent of `Avoidance`, `FindL`, and `FindR`.

(b) Genealogy relations for thread `Flight`.

Fig. 7. Thread genealogy for Figure 5.

At the next global tick, the paused threads start their next local tick from their respective `pause`s. For brevity, we shorten "global tick" into "tick" and use "local tick" as before.

2.3 Fork/Join Parallelism

The `par` statement enables the forking of parallel threads. We use the well known terminology related to parallel programming. The *parent* thread is the thread that executes the `par` statement to fork its *child* threads. The *parent* thread is also the *ancestor* of its child threads and of their nested child threads. Child threads forked by the same `par` statement are *siblings*. Since `par` is a blocking statement, threads can never execute in parallel with their ancestors. We define threads that do not satisfy any ancestor relationship as being *independent*: these threads can execute in parallel.

The thread genealogy of a program summarizes all the parent-child thread relationships that can potentially arise at runtime, which is used to compute a total order for thread scheduling (see Section 4.2). Such a thread genealogy can be determined statically by inspecting the program's control-flow. When a function is forked more than once by different parent threads, the thread instances in the genealogy need to be distinguished by unique names. Figure 7a shows the thread genealogy of the UAV program. Each node is a thread and arrows are drawn from the children to their parent thread. Figure 7b exemplifies the thread genealogy.

2.4 Shared Variables

All variables in ForeC follow the scoping rules of C. By default, all variables are *private* and can only be accessed (read or write) by one thread throughout its scope. To allow a variable to be accessed by multiple threads, it must be declared as a *shared* variable by using the `shared` type qualifier. Thanks to a control flow analysis, we compute for each variable the list of threads that access it. To avoid naming ambiguities, each thread is assigned a unique label. Conditional statements (`if-else`, loops, and `abort`s) are assumed to be executed unconditionally by control flow analysis. Loops are analyzed only for one iteration to determine their variable accesses (recall that ForeC forbids the recursive use of function calls and thread forking). Thus, any misuse of private variables is easy to detect at compile time.

Appendix C.1 describes how shared variables are passed by value and by reference. The ForeC semantics ensures that shared variables can be safely accessed by the parallel threads without the need for mutual exclusion constructs. The goal is to provide a deterministic shared variable semantics that is agnostic to scheduling, which is essential for designing and debugging parallel programs. Within each tick, the accesses to a shared variable from two threads may occur in sequence or in parallel:

Definition 1. *Accesses from two threads are in* sequence *if both threads are not independent or if the accesses occur in different ticks.*

Definition 2. *Accesses from two threads are in* parallel *if both threads are independent and the accesses occur in the same tick.*

Improperly managed parallel accesses to a shared variable can cause race conditions, leading to nondeterministic behavior. For example, two parallel writes to a shared variable can partially overwrite each other's value, leading to functional nondeterminism. A parallel read and write to a shared variable can result in the read returning the variable's value before, during, or after the write has completed. Table 9 in Section 6 details the solutions that exist for enforcing mutual exclusion on shared variables, usually by serialising the parallel accesses. Parallel accesses can be interleaved in many ways (influenced by the programmer, compiler, and runtime system), and relying on a particular interleaving for correct program behavior is brittle and error prone.

We propose a shared memory model that permits shared variables to be accessed deterministically in parallel, without needing the programmer to explicitly use mutual exclusion. The goals are:

Isolation: To provide isolation between threads to enable the local reasoning of each thread. That is, the execution of a thread's local tick can be understood by only knowing the values of the variables at the start of the thread's local tick.

Determinism [81]: To ensure deterministic execution regardless of scheduling decisions. This guarantees that deterministic outputs are always generated at the end of each tick.

Parallelism: To minimize the need to serialize parallel accesses to shared variables. This maximizes the amount of parallel execution that can occur at runtime, which is important for improving the program's performance.

The key mechanism is that all threads access their own *local copies* of the shared variables, and these copies are *resynchronized* every time threads join and when the tick ends.

2.5 Copying of Shared Variables

Every time a thread starts its local tick, it creates *local copies* of all the shared variables that its body accesses (reads or writes). When a thread is forked, its initial copy of a shared variable is created from its parent's copy if it exists, otherwise, from the shared variable's resynchronized value. A parent thread that is blocked on a `par` statement does not create any copies of the shared variables until the `par` statement terminates. For example, in tick two of Figure 6, the threads `main`, `Flight`, and `Avoidance` make no local copies. The child threads `Navigation`, `Stability`, `FindL`, and `FindR` must create their local copies from the shared variables' resynchronized values, e.g., `obst` = 2 and `newPos` = 56. A shared variable declared inside a thread can be shared among its child threads by *passing a reference* (using a pointer) to the child threads (e.g., `obst` on line 7 of Figure 5). When a shared variable is passed by reference to an ordinary function (e.g., `obst` on line 17), the function uses the calling thread's copy of the shared variable.

2.6 Resynchronization of Shared Variables

For any shared variable, its copies created by parallel threads are *resynchronized* every time the program completes its tick (before outputs are emitted) and when child threads join. Resynchronizing at specific program points ensures that the semantics of shared variables is agnostic to scheduling. We use combine functions to compute the value of resynchronized shared variables. *Combine functions must be deterministic, associative, and commutative.* The signature of any combine function, C, is $C : Val \times Val \rightarrow Val$. The two input parameters are the two copies to be combined. When a `par` statement terminates, the copies from the terminating child threads are combined and assigned to their parent thread's copies of shared variables. A combine function being deterministic, associative, and commutative, it suffices to have a binary function from $Val \times Val$ to combine the copies of a shared variable from an arbitrary number of threads and still obtain a deterministic

Preemption Condition	
Cond ::= *U_bool_op Cond*	*// Unary Boolean condition.*
| *Cond B_bool_op Cond*	*// Binary Boolean condition.*
| *Cond* ? *Cond* : *Cond*	*// Ternary condition.*
| (*Cond*)	*// Grouping.*
| *Exp B_comp_op Exp*	*// Binary comparison.*
| *val* | *var* | ...	*// Expressions can also be immediate values, variables, array accesses, struct accesses, or pointers.*
Unary Boolean Operator	*U_bool_op* ::= ! *// Negation.*
Binary Boolean Operator	*B_bool_op* ::= || | && *// Logical operators.*
Binary Comparison Operator	*B_comp_op* ::= > | >= | < | <= | == | !=

Fig. 8. Syntax of preemption conditions.

```
1  typedef enum {OK,ERROR,WARN,TERM} State;
2  input State comms;  // Additional input.
3  ...
4  void main(void) {
5    shared int obst=0 combine new with min;
6    abort {
7      par(Flight(&obst),Avoidance(&obst));
8    } when (comms==TERM);
9    safeDescent();
10 }
```

Fig. 9. Figure 5 extended with preemption.

value. Appendix C describes more examples of combine functions and how more than two copies are combined.

It can be useful to ignore some of the copies when resynchronizing a shared variable. This is achieved by specifying a *combine policy* that determines what copies will be ignored. The combine policies are `new`, `mod`, and `all`, specified when declaring the variable in the `combine` clause, e.g., "`combine new with`". The `new` policy ignores copies that have the same value as their shared variable, i.e., have not changed during the tick. The `mod` policy ignores copies that were not assigned a value during the tick, i.e., have not appeared on the lefthand side of an assignment. This differs from the `new` policy because an assignment semantically equivalent to "`x=x`" will be taken into account by the `mod` policy, but not by the `new` policy. The default policy is `all` where all copies are taken into account. Note that the combine function is not invoked when only one copy remains. Instead, that copy becomes the resynchronized value. Appendix C provides extensive illustrations comparing the behavior of the combine policies.

2.7 Hierarchical Preemption

Inspired by Esterel [15], the `abort` *st* `when` (*cond*) statement provides preemption [13], which comprises the termination of *st* when *cond* evaluates to *true*. Preemption provides a convenient means to model *hierarchical state machines* [15, 52] succinctly. The condition *cond* must be side-effect free, produced from the syntax shown in Figure 8.

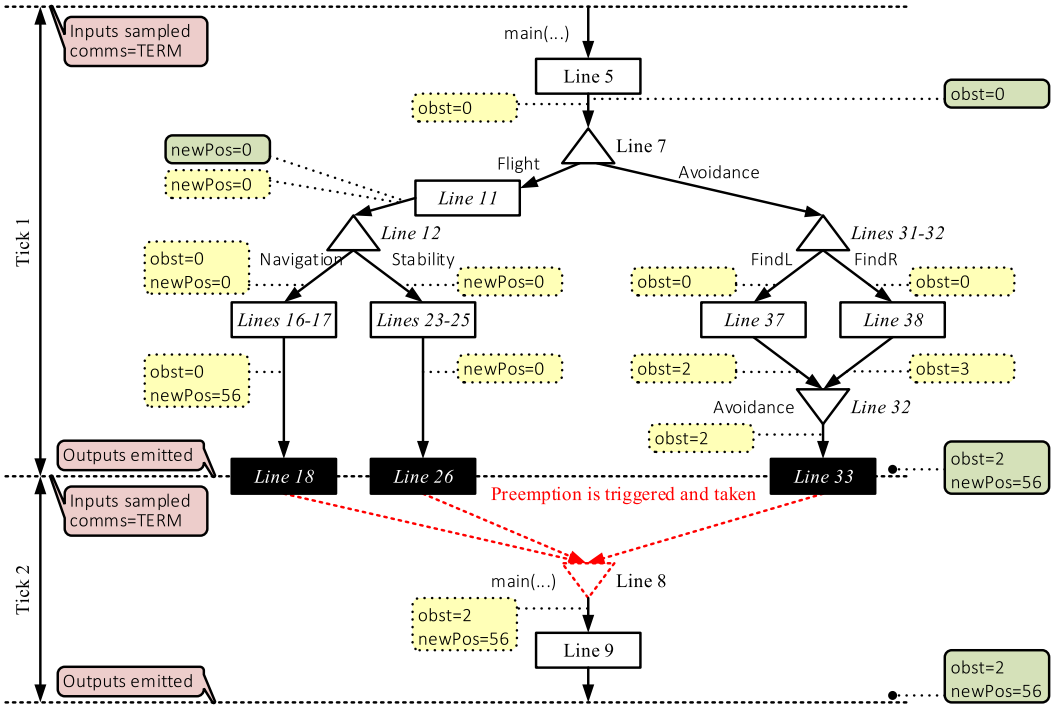


Fig. 10. Possible execution trace for Figure 9.

We illustrate this in Figure 9, where the `main` function of the UAV has been extended to respond to external commands through the input `comms` (line 2). The value of `comms` can be `OK`, `ERROR`, `WARN`, or `TERM` (line 1). The `abort` statement on line 6 preempts the execution of all the UAV tasks when `TERM` is received. A possible execution trace of the program of Figure 9 is given in Figure 10. The *italicized* line numbers in Figure 10 refer to the line numbers in Figure 5, while the non-italicized line numbers refer to the line numbers in Figure 9.

The preemption of the `abort` must be *triggered* before the `abort` body can be terminated. Preemption is never taken when the `abort` body executes for the first time (e.g., tick one in Figure 10). At the start of each subsequent tick, *cond* is evaluated before the `abort` body can execute. This allows shared variables in the condition to be evaluated with their resynchronized value. If *cond* evaluates to *true* (any non-zero value following the C convention), then the preemption is triggered and the `abort` statement is terminated. At the start of tick two in Figure 10, preemption is triggered because the input `comms` is equal to `TERM`. The `abort` statement also terminates if its body terminates normally.

Preemptions in ForeC differ from those in Esterel because Esterel uses signals for thread communication instead of shared variables. Signals in Esterel are either present or absent in each tick and this information is propagated instantaneously among the threads. Thus, preemptions in Esterel are triggered instantaneously, whereas preemptions in ForeC are triggered with a delay of one tick because the condition *cond* is evaluated using values computed in the previous tick.

Like in Esterel [13], the optional `weak` and `immediate` keywords change the temporal behavior of preemptions: `weak` delays the termination of the `abort` body until the body cannot execute any further, e.g., reaches a `pause` statement; and `immediate` allows preemption to be triggered

immediately when execution reaches the `abort` for the first time. This results in four variants of `abort`, detailed below. These variants are not syntactic sugar; they cannot be expressed in terms of one another [13]. To illustrate these four different behaviors, Figure 11a presents an `abort` with the optional keywords commented out:

Non-immediate and strong `abort` This is the default preemption behavior, summarized in Figure 11b. In tick one, the `main` thread sets its copy of `s` to 1 and prints "1". Next, the threads `t0` and `t1` set their copies of `s` to 2 and 5, respectively. When the tick ends, using the combine policy `all`, the resynchronized value of `s` is 7. In tick two, the `abort`'s preemption is triggered and the `abort` body is terminated, resulting in "7" being printed.

Non-immediate and weak `abort` This behaviour is summarized in Figure 11c. The execution of tick one proceeds identically to the non-immediate and strong `abort` variant. In tick two, the `abort`'s preemption is triggered. However, the termination of the `abort` body is delayed until threads `t0` and `t1` complete their local ticks. This allows `t0` and `t1` to set their copies of `s` to 3 and 6, respectively. Thus, "9" is printed.

Immediate and strong `abort` This behaviour is summarized in Figure 11d. In tick one, the `main` thread sets its copy of `s` to 1 and prints "1". Next, the `abort`'s preemption condition is evaluated immediately. Intuitively, because "1" was printed for the value of `s`, the condition `s>0` should evaluate to *true*. The counter-intuitive result of *false* would occur if the resynchronized value of `s` was used. Thus, when execution reaches an immediate `abort`, the condition *cond* is evaluated immediately with the thread's copies of the shared variables. In subsequent ticks, the resynchronized values of the shared variables are used. In tick one of Figure 11d, because the preemption has been triggered, the `abort` body is terminated without executing.

Immediate and weak `abort` This behaviour is summarized in Figure 11e. In tick one, the `main` thread sets its copy of `s` to 1 and prints "1". Next, the `abort`'s preemption is triggered immediately. However, the termination of the `abort` body is delayed until threads `t0` and `t1` complete their local ticks. This allows `t0` and `t1` to set their copies of `s` to 2 and 5, respectively. Hence, "7" is printed.

The `abort` statements can be nested to create a hierarchy of preemptions. In such a case, the outer abort has precedence over the inner aborts. Figure 12 is an example of an immediate and weak `abort` (line 3) with a nested immediate and strong `abort` (line 5). In tick one, preemption is triggered for the outer weak `abort`. The variable `x` is set to 2 and the inner strong `abort` preempts immediately without executing its body. Next, `x` is set to 5 and the outer weak `abort` takes its preemption when it reaches the `pause` on line 6. Finally, "5" is printed.

2.8 Bounded Loops

The synchrony hypothesis requires each tick to execute in bounded time, which in turn means that all statements need to have bounded execution times. Loop constructs (`for` and `while`) are problematic because they can have unbounded iterations, leading to unbounded execution times. Thus, if a loop construct is used, then the programmer must guarantee that it always terminates or executes a `pause` in each iteration. Guaranteeing that a loop always executes a `pause` may not be decidable when `pause` statements are enclosed by `if`-statements. For this reason, the compiler makes conservative assumptions to prove whether a loop always executes a `pause` in each iteration. For example, a loop is assumed to always execute a `pause` in each iteration if its body has at least one statement that always executes a `pause`. An `if`-statement is assumed to always execute a `pause` if both its branches always execute a `pause`. An `abort` statement is assumed to never execute a `pause`. A `par` statement is assumed to always execute a `pause` if at least one of its child threads always executes a `pause`. Based on these principles, the compiler performs a structural induction

```
1  shared int s=0 combine all with plus;
2  int plus(int th1,int th2) { return (th1+th2); }
3  void main(void) {
4    s=1; printf("%d",s);
5    /* weak */ abort {
6      par({s=2;pause;s=3;pause;s=4;},
7          {s=5;pause;s=6;pause;s=7;});
8    } when /* immediate */ (s>0);
9    printf("%d",s);
10 }
```

(a) Example code.

Tick 1: "1" is printed. s = plus(2,5) = 7.

Tick 2: Preemption is triggered and the abort body is terminated. "7" is printed.

(b) Non-immediate and strong abort.

Tick 1: "1" is printed. s = plus(2,5) = 7.

Tick 2: Preemption is triggered.

s = plus(3,6) = 9. The abort body is terminated. "9" is printed.

(c) Non-immediate and weak abort.

Tick 1: "1" is printed. Preemption is triggered and the abort body is terminated. "1" is printed again.

(d) Immediate and strong abort.

Tick 1: "1" is printed. Preemption is triggered. s = plus(2,5) = 7. The abort body is terminated. "7" is printed.

(e) Immediate and weak abort.

Fig. 11. Abort variants.

```
1  void main(void) {
2    int x=1;
3    weak abort {
4      x=2;
5      abort { x=3;pause;x=4; } when immediate (x==2);
6      x=5;pause;
7      x=6;
8    } when immediate (x==1);
9    printf("%d",x);
10 }
```

Fig. 12. Nesting of preemptions.

on the program's control-flow to conservatively prove whether every loop in the program will always execute a pause in each iteration.

Regarding loops that do not contain a pause, inspired by PRET-C [3], we have extended the syntax of loops to also allow the programmer to write bounded loops, shown in the first column of Table 2. The "#n" after the loop header specifies that only up to n iterations can be executed. The second column of Table 2 gives the structural translation of bounded loops.

Table 2. Structural translations of bounded loops

Bounded Loop	**Translation**
`for (init; cond; update) #n {st}`	`int cnt=0;` `for (init; cond && (cnt<n); (update, cnt++)) {st}`
`while (cond) #n {st}`	`for ( ; cond;) #n {st}`
`do {st} while (cond) #n`	`int first=1;` `for ( ; cond && (first==0); first=0) #n {st}`

2.9 Multi-Rate Extension

All threads of a ForeC program run in lockstep, hence all the inputs are always handled at the same *rate*, and all the outputs are always produced at the same rate. This unique rate is the sequence of global ticks of the ForeC program, and we call it the (main) rate of the ForeC program. However, when inputs are produced by the environment are different rates, this monolithic behavior is inefficient. For this reason, we have proposed in [45] a *multi-rate* conservative extension of ForeC. In short, with this extension, each input and output can be sampled and emitted at its own rate, relative to the program's main rate. By conservative, we mean that it is backward compatible with the (single-rate) ForeC language presented in this paper.

To keep this paper's focus on the formal semantics of ForeC and its compilation, we only provide here a summary of the multi-rate extension. Concretely, *Multi-rate ForeC* extends ForeC with logical rates, which are multipliers or dividers of a parent rate. This results in a tree of rates, the root of which is the main rate of the ForeC program. Moreover, the user defines a concrete period for the main program rate, in microseconds. This *concrete rate* is specified in an architecture description file that is taken into account by the ForeC compiler. It follows that concrete rates can be derived for all logical rates. All threads in `par` statements can be annotated with their desired logical rate. When no rate is provided, a child thread has the same rate as its parent thread. Implicitly, the inputs read (resp. outputs written) by a thread are sampled from (resp. emitted to) the environment at the rate of this thread, therefore not necessarily at the main rate of the program.

Threads synchronize for a global tick whenever their local ticks end at the same *absolute (concrete) time*, at which point their shared variables are resynchronized, their outputs are emitted, and their inputs are sampled. The global tick is *total* if all executing threads participate, otherwise it is *partial*. Multi-rate support for preemption is not addressed in [45] and it will be the topic of future work.

2.10 Assessment of ForeC

In summary, the main strengths of ForeC are:

- By bridging the differences between synchronous-reactive programming and general-purpose parallel programming, ForeC enables the deterministic parallel programming of multi-cores (and avoids complex program parallelization techniques required by classical synchronous programming languages).
- Thread isolation is guaranteed by stipulating that threads work on local copies of the shared variables, which maximizes the opportunities for parallel execution.
- The shared variable semantics removes the burden of ensuring mutual exclusion from the programmer and ensures deadlock freedom.
- Resynchronizing shared variables after the threads have finished their respective local ticks ensures that the program behavior is agnostic to scheduling decisions.

- Determinism is guaranteed by the semantics of ForeC, instead of relying on ad-hoc determinism imposed by an implementation.
- All these features allow for local reasoning on each thread and simplifies the understanding and debugging of ForeC programs.

And the main limitations of ForeC are:

- Communication via shared variables between threads is delayed by one tick due to the resynchronization of shared variables. If the programmer wants to sequentially assign two values to a shared variable `x`, such that these two values are visible to another parallel thread, then the tick must be "broken" into two smaller ticks by adding an intermediary `pause` statement between the two assignments to `x`. This guarantees a deterministic parallel execution of the two threads. This design choice also avoids a (costly) causality analysis.
- The programmer needs to make explicit the resolution of parallel writes to shared variables. This is done within our disciplined framework of combine functions and policies. Hence, for a given desired behavior, the programmer must define an appropriate combine function.
- Copying and resynchronisation of shared variables at each global tick introduces some timing overhead. This is even more the case for shared variables of type array or large data structures.
- The threads that can be forked by a par statement are statically defined, e.g., one cannot fork n threads where n is determined at runtime (although this can be emulated with `if-else` or `switch-case` statements).

3 SEMANTICS OF FOREC

This section presents the semantics of ForeC as rewrite rules in the style of Structural Operational Semantics (SOS) [97]. The semantics is inspired by that of other imperative synchronous programming languages (Esterel [100] and PRET-C [3] in particular). The semantics is defined on a set of primitive ForeC constructs (the kernel of Table 3) from which the full ForeC constructs are derived. These kernel constructs are not used for compiling. They only consider a subset of the C language: the assignment operator (`=`), the statement terminator (`;`) for sequencing, and the `if` and `while` statements. Table 4 shows how the other ForeC constructs (Table 1) are translated into the kernel constructs (Table 3). This is exemplified by the translation of the ForeC constructs in Figure 13b into the kernel constructs in Figure 13c. The translations for `input`, `output`, and `pause` are straightforward. A `shared` variable is translated into a unique global variable and a `copy` kernel statement that is placed at the start of every thread body in the scope of the shared variable. The `copy` kernel statement initiates the copying of the shared variables when the threads are forked and when the threads start their local ticks. The `par` statement is translated by prefixing each thread body f with a unique identifier t to allow the body of one thread to be distinguished from another. The `par` kernel statement handles the resynchronization of the shared variables. In Esterel, `traps` [100] are used to translate `aborts` and other complex preemption statements. By contrast, a simpler `abort` translation is possible in ForeC because `abort` is the only type of preemption statement. Each `abort` is assigned a unique identifier a and translated into the `status` and `abort` kernel statements. The `status` kernel statement is needed to define the immediate behavior of an `abort`; it takes the unique identifier a and an expression, which is 0 (zero) for a non-immediate `abort` but is *cond* (the preemption condition) for an immediate `abort`. The `abort` kernel statement takes the unique identifier a and the `abort` body f. The following section describes the assumptions on ForeC kernel programs to simplify the presentation of the formal semantics. The notations, semantic functions, and rewrite rules are then presented. Important proofs concerning program reactivity and determinism [81, 113] are provided in Appendix A.

Table 3. ForeC kernel constructs. f is an arbitrary composition of kernel constructs, *var* is a variable, *exp* is an expression, t is a thread identifier, and a is an abort identifier. A question mark means that the preceding symbol is optional

Kernel Construct	Short Description
`nop`	Empty statement
f; f	Sequence operator
var = *exp*	Assignment operator
`while` (*exp*) f	Loop
`if` (*exp*) f `else` f	Conditional
`copy`	Creates a local copy for each shared variable of its enclosing thread
`pause`	Barrier synchronization
`par`(t:f, t:f)	Fork/join parallelism
`status`(a, *exp*)	Initial preemption status
`weak?` `abort`(a, f)	Abort

Table 4. Structural translations of the ForeC constructs (Table 1) to kernel constructs (Table 3)

ForeC Construct	ForeC Kernel Constructs
`input` and `output`	Translated into unique global variables.
`shared`	Translated into unique global variables and `copy` kernel statements are placed at the start of every thread body.
`pause`	`pause`
`par`(f_1, f_2)	`par`(t_1:f_1, t_2:f_2) Note that thread bodies f_1 and f_2 can be the same.
`weak?` `abort` f `when` (*cond*)	`status`(a, 0); `weak?` `abort`(a, f)
`weak?` `abort` f `when immediate` (*cond*)	`status`(a, *cond*); `weak?` `abort`(a, f)

3.1 Assumptions

We make the following assumptions about ForeC programs. (1) All programs follow safety-critical coding practices, as discussed in Section 2.1. Dynamic memory allocation (e.g., `malloc`) and unstructured jumps (e.g., `goto`) cannot be used, and loops must be bounded (i.e., with a "`#n`" annotation) or contain a `pause`. Moreover, C expressions may only be constants, variables, pointers, and arrays composed with the logical, bitwise, relational, and arithmetic operators of C. Arguments of functions and the right-hand side of assignment statements cannot contain any assignment operators. The sequencing operator "`,`" of C cannot be used. These assumptions limit us to a deterministic subset of the C language. (2) Functions and threads cannot be called or forked recursively, respectively. This assumption prevents the unbounded execution of functions and threads, leading to unbounded memory use and execution time.

To simplify the presentation of the semantics, we assume that the following transformations have been performed on ForeC programs. (1) Inlining of functions at their call sites, so that the semantics

```
1 input int i; output int o=0;
2 int plus(...) {...}
3 void main(void) {
4   shared int s=1 combine all with plus;
5   par({s++; pause;}, {s=1;});
6   abort {f(&s);} when (s>3);
7 }
8 void f(shared int *x) {*x=2;}
```

(a) Original program.

```
1 input int i; output int o=0;
2 int plus(...) {...}
3 shared int s combine all with plus;
4 void main(void) {
5   s=1;
6   par({s++; pause;}, {s=1;});
7   abort {s=2;} when (s>3);
8 }
```

(b) Transformed program.

```
1 int i; int o=0;
2 int plus(...) {...}
3 int s;
4 void main(void) {
5   copy; s=1;
6   par(t1:{copy; s++; pause;}, t2:{copy; s=1;});
7   status(a1,0); abort(a1, {s=2;});
8 }
```

(c) Translated kernel program.

Fig. 13. Example of transforming and translating a ForeC program into the kernel constructs.

can ignore function calls. (2) Renaming variables uniquely and hoisting their declarations up to the program's global scope, so that the semantics can ignore (static) memory allocation and focus on the semantics of private variables (accessible to only one thread) and shared variables. (3) Replacing pointers with the variables they reference, so that the semantics can ignore pointer analysis [24, 51]. Consider the program of Figure 13a that is transformed into the equivalent program of Figure 13b. The shared variable declaration for `s` (line 4 in Figure 13a) is hoisted to the global scope (line 3 in Figure 13b). The function `f` (line 8 in Figure 13a) is inlined into the `abort` body (line 7 in Figure 13b) and the pointer inside `f` is replaced by the variable `s` that it references.

3.2 Notation

The rewrite rules have the following form in the style of structural operational semantics (SOS) [97]:

$$\langle S\rangle\ t : f \xrightarrow[I]{k} \langle S'\rangle\ t : f'$$

This notation describes a program fragment f belonging to thread t, in the program state $\langle S\rangle$ and with inputs I, which reacts and modifies the program state to $\langle S\rangle'$, generates the completion code k, and becomes the new program fragment f'. All the (globally declared) `inputs` are stored in I. Let T be the set of all threads in the program. The state $\langle S\rangle$ is a pair $\langle E, A\rangle$ where:

- E is an environment that maps the program's global scope to the program's global variables and maps the threads' scopes to their local copies of shared variables. Specifically, E is a partial function that maps the global scope $\mathcal{G}$ and the set of threads T to a store ($Store$) of variables. Let $Id = T \cup \{\mathcal{G}\}$ be the set of all scopes; we then have $E : Id \hookrightarrow Store$. On the one hand, $E[\mathcal{G}]$ stores all the output, shared, and private variables in the program, which are all globally declared thanks to the program transformations of Section 3.1. On the other hand, for each t in T, $E[t]$ stores thread t's copies of the shared variables. The function E is partial because a store is only created for a thread if it accesses any shared variable.

GetParent(main) = main	GetParent(t1) = main	GetParent(t2) = main
GetShared($\mathcal{G}$) = {s}	GetShared(t1) = {s}	GetShared(t2) = {s}
GetCombine(s) = plus	GetPolicy(s) = all	GetExp(a1) = s>3

Fig. 14. Retrieving statically known information about Figure 13c.

The store itself (*Store*) is a partial function that maps variable ($var \in Var$) to a pair of value ($v \in Val$) and status ($sts \in Sts$): $Store : Var \hookrightarrow (Val, Sts)$. Statuses are used to define the behavior of the combine policies and can be `pre` (previous resynchronized value), `mod` (modified value), `cmb` (combined value), or `pvt` (for a private variable). In $E[\mathcal{G}]$, the status of a private variable is always `pvt` and the status of a shared variable is always `pre`. In $E[t]$, a thread's copy of a shared variable always starts each local tick with the status `pre`.
For example, $E = \{\mathcal{G} \to \{s \to (1, \texttt{pre})\}, t1 \to \{s \to (3, \texttt{mod})\}\}$ for a program that has a shared variable `s` with value 1 in the global scope and modified value 3 in the scope of thread $t1$. We use the notation $E[t1][s]$ to look up the value and status $(3, \texttt{mod})$ of `s` in $t1$'s store. We use the notations $E[t1][s].v$ and $E[t1][s].sts$ to look up its value and status, respectively. We use the notation $S.E$ to retrieve E from the program state S.

- A is a partial function that maps `abort` identifiers ($a \in \mathcal{A}$) to values ($v \in Val$) representing their preemption status, $A : \mathcal{A} \to Val$. An `abort` with a non-zero value means that its preemption condition is *true* and that it has been triggered, otherwise its condition is *false* and the `abort` has not been triggered.
For example, $A = \{a1 \to 1, a2 \to 0\}$ for a program that has `aborts` $a1$ and $a2$ with the statuses 1 and 0, respectively. We use the notation $A[a1]$ to look up the status of `abort` $a1$. We use the notation $S.A$ to retrieve A from the program state S.

The transition of a program fragment from f to f' is encoded by the completion code k, where:

$$k = \begin{cases} 0 & \text{If the transition terminates.} \\ 1 & \text{If the transition pauses.} \\ \bot & \text{Otherwise (the transition continues).} \end{cases}$$

3.3 Semantic Functions

The following sections describe the semantic functions that are used by the rewrite rules to ensure semantic conciseness.

3.3.1 Statically Known Information. The following semantic functions return statically known information about the program:

- GetParent(t): Returns the parent of thread t from the program's thread genealogy, e.g., Figure 7a. If t = `main`, then "`main`" is returned.
- GetShared($\mathcal{G}$): Returns the set of all shared variables declared in the program.
- GetShared(t): Returns the set of all shared variables that the body of thread t accesses (reads or writes).
- GetCombine(var): Returns the combine function of shared variable var.
- GetPolicy(var): Returns the combine policy of shared variable var.
- GetExp(a): Returns the preemption condition exp of `abort` a.

Figure 14 exemplifies the use of these functions on the program in Figure 13c.

3.3.2 Eval. The semantic function Eval(E, I, id, exp) follows the evaluation rules of C to evaluate the expression exp and return its value. The expression exp has a classical tree structure: it can be an atom (a constant, a variable, a string, ...), a unary arithmetic or Boolean operation (-, *, &, !, ~), a binary arithmetic or Boolean operation (+, -, *, /, %, ||, &&, ^, |, &, <<, >>, ==, !=, <, >, <=, >=), a function call with its arguments passed by value or by reference, an array, and so on. For the sake of simplicity, we do not give the details here [16, 89] and will only write the string of the expression when calling the Eval function. Finally, the Eval function returns the value of exp. Unlike in C, where expressions can have side-effects (which would be captured by the function Eval returning a *pair* (E', v) instead of only v), we have restricted ForeC expressions and functions to be side-effect free. During the evaluation, a variable's value is retrieved with the semantic function GetVal(E, I, id, var) described by Algorithm 1. The inputs to this algorithm are: the program's environment E, the inputs I, the identifier id of the store to try and retrieve the value from, and the variable var of interest. The output is a value v. If var is an input, then line 2 returns its value. Otherwise, if var is in id's store, then line 4 returns its value. Otherwise, line 6 returns the global value of var.

ALGORITHM 1: GetVal(E, I, id, var): Gets the value of a given variable.

Input: Program's environment E, inputs I, identifier id of the store to search, and variable var of interest.
Output: Value of var.

```
1 if var ∈ I then                                          // If var is an input.
2 |   return I[var]                                       // Return the input value of var.
3 else if var ∈ E[id] then                                 // Otherwise, if a local copy of var exists.
4 |   return E[id][var].v                                  // Return the value of var from id's store.
5 else
6 |   return E[G][var].v                                  // Otherwise, return the global value of var.
7 end
```

3.3.3 Copy. The semantic function Copy(E, t) creates in thread t the local copies of each shared variable $var \in$ GetShared(t) that it does not have. That is, if thread t already has a copy of the shared variable var, then Copy skips the copying of var. This conditional behavior is needed because the semantic function Copy may be invoked for a thread t that already has a subset of its required local copies. For example, when local copies are created for a parent thread that is resuming from the termination of a `par`, the combined values from its child threads must not be overwritten. The Copy function is described by Algorithm 2. The inputs to this algorithm are: the program's environment E and a thread t. The output is an updated environment E. Line 1 considers each shared variables that is accessed in the thread's body. For each shared variable, line 2 checks if a copy already exists. If it does not exist, then lines 4–5 copy the parent thread's copy if available, otherwise from the shared variable (line 7). Line 11 returns the updated environment E.

3.3.4 Combine. The semantic function Combine(E, t_1, t_2, t_0) combines all the copies of shared variables from two threads and is described by Algorithm 3. The inputs to this algorithm are: the program's environment E, two threads t_1 and t_2 to combine, and thread t_0 to store the combined values. The output is an updated environment E. Line 1 considers each shared variable var. Line 2 retrieves the shared variable's `pre` value (`preVal`). For the combine policy `all`, the copies of `var` from both threads are combined if they exist. Thus, line 3 gets the set of threads T that have a copy of `var`. If the combine policy is `new`, then line 5 keeps only the copies with values that differ from `var`'s `pre` value ($E[t][var].v \neq$ `preVal`) or copies that have been combined ($E[t][var].sts =$ `cmb`). If the combine

ALGORITHM 2: COPY(E, t): Copies all the shared variables needed by a thread.

Input: Program's environment E, and thread t.
Input: Updated environment E.

```
1  forall var ∈ GetShared(t) do                  // For all shared variables needed by thread t.
2      if var ∉ E[t] then                                  // If thread t does not have a copy.
3          if var ∈ E[GetParent(t)] then                        // If its parent has a copy.
4              v := E[GetParent(t)][var].v                  // Value of its parent's copy.
5              E[t][var ← (v, pre)]                           // Copy its parent's copy.
6          else                                // Otherwise, its parent does not have a copy.
7              E[t][var ← E[G][var]]      // Copy the shared variable from the global scope.
8          end
9      end
10 end
11 return E
```

ALGORITHM 3: COMBINE(E, t_1, t_2, t_0): Combines the copies of shared variables from two threads.

Input: Program's environment E, threads t_1 and t_2 to combine, and thread t_0 to store the combined values.
Output: Updated environment E.

```
1  forall var ∈ GetShared(G) do                                  // For all shared variables.
2      preVal := E[G][var].v                                          // Get the pre of var.
3      T := {t | t ∈ {t1, t2}, var ∈ E[t]}      // Policy all: Subset of {t1, t2} with copies of var.
4      if GetPolicy(var) = new then
           // Keep only the copies that differ from preVal or have been combined.
5          T := {t | t ∈ T, E[t][var].v ≠ preVal ∨ E[t][var].sts = cmb}
6      else if GetPolicy(var) = mod then
           // Keep only the modified or combined copies.
7          T := {t | t ∈ T, E[t][var].sts ∈ {mod, cmb}}
8      end
9      if |T| = 2 then                                  // If there are two copies to combine.
10         cf := GetCombine(var)                          // Get the combine function of var.
11         v := cf(E[t1][var].v, E[t2][var].v)                          // Combine the copies.
12         E[t0][var ← (v, cmb)]                      // Assign the combined value to t0.
13     else if |T| = 1 then                          // Otherwise, there is only one copy.
14         E[t0][var ← (E[t ∈ T][var].v, cmb)]                // Assign the only copy to t0.
15     end
16 end
17 E′ = {(id, store) | (id, store) ∈ E ∧ id ≠ t1 ∧ id ≠ t2}
18 return E′
```

policy is mod, then line 7 keeps only the modified or combined copies ($E[t][var].sts \in \{\text{mod}, \text{cmb}\}$). If two copies are found, then line 10 retrieves var's combine function (cf) and line 11 computes the combined value. Line 12 assigns the combined value to thread t_0 with the status cmb because it is now a combined value. If only one copy is found, then line 14 assigns the value of that copy to thread t_0. Moreover, it is assigned the status cmb to ensure that it continues to be combined under the new and mod combine policies. Line 18 returns the updated environment E restricted to t_0 (i.e., without thread t_1 and t_2's stores).

3.4 The Structural Operational Semantics

The operational semantics of the kernel constructs presented in Table 3 are now defined. Appendix B illustrates how ForeC programs execute under this semantics.

3.4.1 The nop *Statement.* The nop statement does nothing and terminates instantly:

$$\langle E, A\rangle\ t : \texttt{nop} \xrightarrow[I]{0} \langle E, A\rangle\ t : \tag{nop}$$

3.4.2 The copy *Statement.* The copy statement copies the shared variables needed by thread t and terminates instantly. The combining of the copies is handled by the par statement:

$$\langle E, A\rangle\ t : \texttt{copy} \xrightarrow[I]{0} \langle \textsc{Copy}(E, t), A\rangle\ t : \tag{copy}$$

3.4.3 The pause *Statement.* The pause statement rewrites into the copy statement and pauses. The copy statement ensures that thread t starts its next local tick by copying the shared variables it needs (the pre values are copied):

$$\langle E, A\rangle\ t : \texttt{pause} \xrightarrow[I]{1} \langle E, A\rangle\ t : \texttt{copy} \tag{pause}$$

3.4.4 The status *Statement.* Recall that the abort statement is translated into a status kernel statement that evaluates the preemption status, followed by an invocation of the abort kernel statement that accesses the result of the evaluated preemption status. The status statement sets abort a's preemption status to the value of the expression exp, and then it terminates instantly:

$$\langle E, A\rangle\ t : \texttt{status}(a, exp) \xrightarrow[I]{0} \langle E, A[a \leftarrow \textsc{Eval}(E, I, t, exp)]\rangle\ t : \tag{status}$$

3.4.5 The abort *Statement.* The abort of a executes its body f if its preemption has not been triggered. The body may have paused ($k = 1$) or may have executed some instantaneous statements ($k = \bot$):

$$\frac{\langle E, A\rangle\ t : f \xrightarrow[I]{k \in \{1,\bot\}} \langle E', A'\rangle\ t : f'}{\langle E, A\rangle\ t : \texttt{weak? abort}(a, f) \xrightarrow[I]{k} \langle E', A'\rangle\ t : \texttt{weak? abort}(a, f')}\,(A[a] = 0) \tag{abort-1}$$

The abort terminates normally if its body terminates and its preemption has not been triggered:

$$\frac{\langle E, A\rangle\ t : f \xrightarrow[I]{0} \langle E', A'\rangle\ t :}{\langle E, A\rangle\ t : \texttt{weak? abort}(a, f) \xrightarrow[I]{0} \langle E', A'\rangle\ t :}\,(A[a] = 0) \tag{abort-2}$$

The weak abort terminates normally if its body terminates, even if its preemption has been triggered:

$$\frac{\langle E, A\rangle\ t : f \xrightarrow[I]{0} \langle E', A'\rangle\ t :}{\langle E, A\rangle\ t : \texttt{weak abort}(a, f) \xrightarrow[I]{0} \langle E', A'\rangle\ t :}\,(A[a] \neq 0) \tag{abort-3}$$

The weak abort allows its body to execute instantaneous statements ($k = \bot$), even if its preemption has been triggered:

$$\frac{\langle E, A\rangle\ t : f \xrightarrow[I]{\bot} \langle E', A'\rangle\ t : f'}{\langle E, A\rangle\ t : \texttt{weak abort}(a, f) \xrightarrow[I]{\bot} \langle E', A'\rangle\ t : \texttt{weak abort}(a, f')}\,(A[a] \neq 0) \tag{abort-4}$$

The weak abort terminates if its body pauses and its preemption has been triggered, and then it rewrites into the copy statement because it may be the start of thread t's local tick[3]

$$\frac{\langle E, A\rangle\ t : f \xrightarrow[I]{1} \langle E', A'\rangle\ t : f'}{\langle E, A\rangle\ t : \texttt{weak abort}(a, f) \xrightarrow[I]{\perp} \langle E', A'\rangle\ t : \texttt{copy}}(A[a] \neq 0) \tag{abort-5}$$

The strong abort terminates without executing its body if its preemption has been triggered, and then it rewrites into the copy statement because it may be the start of thread t's local tick[4]:

$$\frac{A[a] \neq 0}{\langle E, A\rangle\ t : \texttt{abort}(a, f) \xrightarrow[I]{\perp} \langle E, A\rangle\ t : \texttt{copy}} \tag{abort-6}$$

3.4.6 The Assignment Operator (=). The assignment operator evaluates the expression exp into a value $v = \textsc{Eval}(E, I, t, exp)$. If var is a shared variable[5] (rule assign-shared), then the value v and status mod is assigned to the thread's copy in $E[t]$. Otherwise, if var is a private variable (rule assign-private), then the value v and status pvt is assigned to the global variable in $E[\mathcal{G}]$:

$$\frac{var \in \textsc{GetShared}(t)}{\langle E, A\rangle\ t : var{=}exp \xrightarrow[I]{0} \langle E[t][var \leftarrow (v, \texttt{mod})], A\rangle\ t :} \tag{assign-shared}$$

$$\frac{var \notin \textsc{GetShared}(t)}{\langle E, A\rangle\ t : var{=}exp \xrightarrow[I]{0} \langle E[\mathcal{G}][var \leftarrow (v, \texttt{pvt})], A\rangle\ t :} \tag{assign-private}$$

3.4.7 The if–else *Statement.* A conditional construct is rewritten into one of its branches, depending on the value of its condition exp:

$$\frac{\textsc{Eval}(E, I, t, exp) \neq 0}{\langle E, A\rangle\ t : \texttt{if}\ (exp)\ f_1\ \texttt{else}\ f_2 \xrightarrow[I]{\perp} \langle E, A\rangle\ t : f_1} \tag{if-then}$$

$$\frac{\textsc{Eval}(E, I, t, exp) = 0}{\langle E, A\rangle\ t : \texttt{if}\ (exp)\ f_1\ \texttt{else}\ f_2 \xrightarrow[I]{\perp} \langle E, A\rangle\ t : f_2} \tag{if-else}$$

3.4.8 The while *Statement.* The body of a loop statement is either unrolled once or it terminates, depending on the value of its condition exp:

$$\frac{\textsc{Eval}(E, I, t, exp) \neq 0}{\langle E, A\rangle\ t : \texttt{while}\ (exp)\ f \xrightarrow[I]{\perp} \langle E, A\rangle\ t : f;\ \texttt{while}\ (exp)\ f} \tag{loop-then}$$

$$\frac{\textsc{Eval}(E, I, t, exp) = 0}{\langle E, A\rangle\ t : \texttt{while}\ (exp)\ f \xrightarrow[I]{0} \langle E, A\rangle\ t :} \tag{loop-else}$$

[3]The abort may have had a par statement that paused. In this case, when the abort kernel statement preempts, thread t will start its local tick.

[4]In addition to footnote 3, the strong preemption prevents the execution of a copy statement inside the abort body.

[5]Recall from Section 3.2 that E maps the global and thread scopes to their own store of variables, $E : Id \hookrightarrow Store$. Variables are mapped to a value and status, $Store : Var \hookrightarrow (Val, Sts)$ where $Sts = \{\texttt{pre}, \texttt{mod}, \texttt{cmb}, \texttt{pvt}\}$. A private variable has the status pvt, a shared variable has the status pre, and a thread's copy of a shared variable starts each local tick with the status pre. The notation $E[t][var]$ returns the value and status (v, sts) of thread t's copy of var.

3.4.9 The Sequence Operator (;) For a sequence of program fragments, the first fragment f_1 must terminate before the second fragment f_2 can be rewritten. In other words, the (seq-left) rule applies up to the micro-step during which f_1 emits the completion code 0. At this point, the (seq-right) rule applies. The (seq-left) rule emits the completion code of the first fragment:

$$\frac{\langle E, A\rangle\ t : f_1 \xrightarrow[I]{k \in \{1,\bot\}} \langle E', A'\rangle\ t : f_1'}{\langle E, A\rangle\ t : f_1; f_2 \xrightarrow[I]{k} \langle E', A'\rangle\ t : f_1'; f_2} \quad \text{(seq-left)}$$

$$\frac{\langle E, A\rangle\ t : f_1 \xrightarrow[I]{0} \langle E', A'\rangle\ t :}{\langle E, A\rangle\ t : f_1; f_2 \xrightarrow[I]{\bot} \langle E', A'\rangle\ t : f_2} \quad \text{(seq-right)}$$

3.4.10 The `par` *Statement.* The `par` statement allows both of its child threads, t_1 and t_2, to execute instantaneous statements in parallel. The parent thread is t_0:

$$\frac{\langle E, A\rangle\ t_1 : f_1 \xrightarrow[I]{\bot} \langle E', A'\rangle\ t_1 : f_1' \qquad \langle E, A\rangle\ t_2 : f_2 \xrightarrow[I]{\bot} \langle E'', A''\rangle\ t_2 : f_2'}{\langle E, A\rangle\ t_0 : \mathtt{par}(t_1 : f_1,\ t_2 : f_2) \xrightarrow[I]{\bot} \langle E^A, A^A\rangle\ t_0 : \mathtt{par}(t_1 : f_1',\ t_2 : f_2')} \quad \text{(par-1)}$$

E^A and A^A are the *aggregated* environment and preemption statuses, respectively, and are required for the following reason. Threads t_1 and t_2 always modify the starting environment E in a mutually exclusive manner. Indeed, the (assign-shared) rule only allows a thread to access its own copies of shared variables and the (assign-private) rule only allows a thread to access its own private variables. This means that thread t_1's new program environment E' contains the old variables of thread t_2 and t_2's nested child threads, and vice versa for E''. Thus, variables that changed in E' or E'' are aggregated to form E^A by taking the union of the changes in E' (i.e., $E' \setminus (E' \cap E)$) and in E'' (i.e., $E'' \setminus (E'' \cap E)$) with the remaining unchanged variables (i.e., $E' \cap E''$). Note that intersecting two environments, e.g., $E' \cap E''$, produces a new environment containing the variables that have the same values and statuses in E' and E''. Thus, $E^A = (E' \setminus (E' \cap E)) \cup (E'' \setminus (E'' \cap E)) \cup (E' \cap E'')$. Similarly, the preemption statuses that changed in A' and A'' are aggregated to form $A^A = (A' \setminus (A' \cap A)) \cup (A'' \setminus (A'' \cap A)) \cup (A' \cap A'')$. In Esterel, such aggregation is not required because signals are broadcasted instantaneously among all threads.

If a child thread can complete its local tick, by pausing or terminating, then it will wait for its sibling to complete its local tick. The waiting is captured by stopping the child thread from taking its transition:

$$\frac{\langle E, A\rangle\ t_1 : f_1 \xrightarrow[I]{k \in \{0,1\}} \langle E', A'\rangle\ t_1 : f_1' \qquad \langle E, A\rangle\ t_2 : f_2 \xrightarrow[I]{\bot} \langle E'', A''\rangle\ t_2 : f_2'}{\langle E, A\rangle\ t_0 : \mathtt{par}(t_1 : f_1,\ t_2 : f_2) \xrightarrow[I]{\bot} \langle E'', A''\rangle\ t_0 : \mathtt{par}(t_1 : f_1,\ t_2 : f_2')} \quad \text{(par-2)}$$

$$\frac{\langle E, A\rangle\ t_1 : f_1 \xrightarrow[I]{\bot} \langle E', A'\rangle\ t_1 : f_1' \qquad \langle E, A\rangle\ t_2 : f_2 \xrightarrow[I]{k \in \{0,1\}} \langle E'', A''\rangle\ t_2 : f_2'}{\langle E, A\rangle\ t_0 : \mathtt{par}(t_1 : f_1,\ t_2 : f_2) \xrightarrow[I]{\bot} \langle E', A'\rangle\ t_0 : \mathtt{par}(t_1 : f_1',\ t_2 : f_2)} \quad \text{(par-3)}$$

The `par` pauses if both of its child threads pause. The changes made to E and A are aggregated into E^A and A^A, respectively, as defined for the (par-1) rule. The copies of shared variables from the child threads are combined and assigned to their parent thread, thanks to the semantic function

Combine:

$$\frac{\langle E, A\rangle\ t_1 : f_1 \xrightarrow[I]{1} \langle E', A'\rangle\ t_1 : f_1' \qquad \langle E, A\rangle\ t_2 : f_2 \xrightarrow[I]{1} \langle E'', A''\rangle\ t_2 : f_2'}{\langle E, A\rangle\ t_0 : \mathsf{par}(t_1 : f_1,\ t_2 : f_2) \xrightarrow[I]{1} \langle \textsc{Combine}(E^A, t_1, t_2, t_0), A^A\rangle\ t_0 : \mathsf{par}(t_1 : f_1',\ t_2 : f_2')} \tag{par-4}$$

Otherwise, the par terminates if both of its child threads terminate. The completion code is $\bot$ because the parent thread t_0 resumes its execution. The par rewrites into the copy statement because it may be the start of the parent thread's local tick[6]

$$\frac{\langle E, A\rangle\ t_1 : f_1 \xrightarrow[I]{0} \langle E', A'\rangle\ t_1 : \qquad \langle E, A\rangle\ t_2 : f_2 \xrightarrow[I]{0} \langle E'', A''\rangle\ t_2 :}{\langle E, A\rangle\ t_0 : \mathsf{par}(t_1 : f_1,\ t_2 : f_2) \xrightarrow[I]{\bot} \langle \textsc{Combine}(E^A, t_1, t_2, t_0), A^A\rangle\ t_0 : \mathsf{copy}} \tag{par-5}$$

If only one child thread terminates while the other pauses, then the terminated child thread rewrites into the nop statement and the par pauses:

$$\frac{\langle E, A\rangle\ t_1 : f_1 \xrightarrow[I]{0} \langle E', A'\rangle\ t_1 : \qquad \langle E, A\rangle\ t_2 : f_2 \xrightarrow[I]{1} \langle E'', A''\rangle\ t_2 : f_2'}{\langle E, A\rangle\ t_0 : \mathsf{par}(t_1 : f_1,\ t_2 : f_2) \xrightarrow[I]{1} \langle \textsc{Combine}(E^A, t_1, t_2, t_0), A^A\rangle\ t_0 : \mathsf{par}(t_1 : \mathsf{nop},\ t_2 : f_2')} \tag{par-6}$$

$$\frac{\langle E, A\rangle\ t_1 : f_1 \xrightarrow[I]{1} \langle E', A'\rangle\ t_1 : f_1' \qquad \langle E, A\rangle\ t_2 : f_2 \xrightarrow[I]{0} \langle E'', A''\rangle\ t_2 :}{\langle E, A\rangle\ t_0 : \mathsf{par}(t_1 : f_1,\ t_2 : f_2) \xrightarrow[I]{1} \langle \textsc{Combine}(E^A, t_1, t_2, t_0), A^A\rangle\ t_0 : \mathsf{par}(t_1 : f_1',\ t_2 : \mathsf{nop})} \tag{par-7}$$

3.4.11 Tick Completion. A tick completes if the main thread pauses or terminates. If the main thread is executing a par statement, then a tick completes when all its child threads and nested child threads have paused or terminated. The shared variables are resynchronized (from E' to E''), the preemption statuses are reevaluated (from A' to A''), the outputs are emitted, and the inputs are resampled:

$$\frac{\langle E, A\rangle\ main : f \xrightarrow[I]{k \in \{0,1\}} \langle E', A'\rangle\ main : f'}{\langle E, A\rangle\ main : f \xrightarrow[I]{k} \langle E'', A''\rangle\ main : f'} \tag{tick}$$

The rules for the par statement ensures that, when the tick completes, the store of main in E' has the combined values from all its child threads. The shared variables[7] are resynchronized by assigning the combined values from $E'[main]$ to their corresponding shared variables in the global store $E'[\mathcal{G}]$. The main's store is then removed from E'. Thus, for all var in $E'[main]$, we have $E'' = E'[\mathcal{G}][var \leftarrow (E'[main][var].v, \mathsf{pre})] \setminus \{main\}$. All the preemption statuses are updated by evaluating their preemption conditions with the resynchronized shared variables in $E''[\mathcal{G}]$. Thus, for all a in A', we have $A'' = A'[a \leftarrow \textsc{Eval}(E'', I, \mathcal{G}, \textsc{GetExp}(a))]$.

[6]The par statement may have paused. In this case, when the par terminates, the parent thread t_0 will start its local tick.
[7]Recall from Section 3.2 that E maps the global and thread scopes to their own store of variables, $E : Id \hookrightarrow Store$. Variables are mapped to a value and status, $Store : Var \hookrightarrow (Val, Sts)$. In $E[\mathcal{G}]$, shared variables have the status pre. The notation $E[t][var]$ returns the value and status (v, sts) of thread t's copy of var.

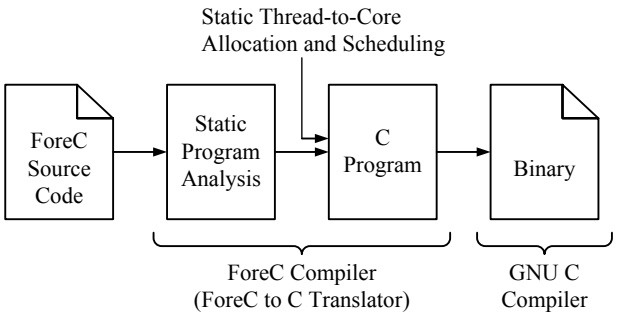


Fig. 15. Overview of compiling ForeC programs.

3.5 Reactivity and Determinism

The semantics of the ForeC kernel constructs (Section 3.4) can be used to formally prove two essential properties of safety-critical programs, *reactivity* and *determinism* [81, 113]. A program is reactive if it always responds to changes in the environment, i.e., it produces outputs and does not deadlock (see Definition 3 and Theorem 4). A program is deterministic if, for a given set of inputs from the environment, there is at most one set of outputs produced by the program. In terms of semantic derivation rules, a program is deterministic if there is at most one derivation tree in response to the environment (see Definition 16 and Theorem 17).

For synchronous programs, the definitions of reactivity and determinism are normally based on a program's tick, which is a sequence of transitions. However, the state of a ForeC program also depends on the initial valuations of its variables. Thus, we define a stronger notion of reactivity and determinism based on program transitions and prove them using structural induction. The proofs themselves are quite standard for synchronous programs and can be found in Appendix A.

4 COMPILING FOREC FOR PARALLEL EXECUTION

This section first describes a predictable parallel architecture (section 4.1). It then presents how the ForeC compiler generates code for direct (bare metal) execution on this architecture (Section 4.2). The chosen compilation strategy generates code that is amenable to static timing analysis and achieves good execution performance, as will be illustrated by the benchmarking results in Section 5. In Section 4.9, we will extend the compiler to generate code for execution on an operating system. Figure 15 is an overview of the compilation process. The first step checks the syntax of the ForeC source code. This includes checking whether all threads have been defined and whether all variables accessed by multiple threads have been declared with the `shared` qualifier. The second step translates the ForeC statements into equivalent C code. Bootup and thread scheduling routines are generated for each core. ForeC threads are statically allocated and statically scheduled on each core. The final step is to compile the generated C program with the GNU C compiler; GNU's computed `goto` extension is used to implement fast context-switching. This section describes the generation of C code. For brevity, we omit inputs and outputs because we follow existing approaches [100] for creating the reactive interface.

4.1 A Time-Predictable Embedded Multi-Core Architecture

To estimate a program's worst-case execution time (WCET) reasonably tightly, a detailed timing model of the underlying processor architecture is needed. However, general-purpose processors sacrifice worst-case performance to focus on improving the average-case via speculative features [95]

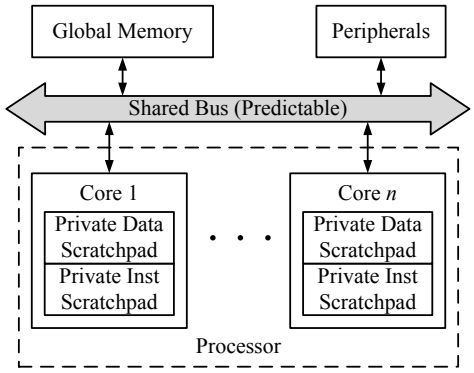


Fig. 16. Example of a predictable multi-core embedded architecture.

such as out-of-order execution, branch prediction, data forwarding, superscalar execution, and caches. However, such optimizations are difficult to model precisely and may cause *timing anomalies* [80] where a local WCET does not lead to the program's WCET, leading to a degradation in time-predictability that is undesirable for real-time embedded systems. These issues are exacerbated by the use of multi-cores because the cores can interfere with each other's timing behaviour.

The PREcision Timed (PRET) machine [38, 39] and the PRedictability Of Multi-Processor Timing (PROMPT) [32, 68] design philosophies advocate for the design of *predictable hardware architectures*, while not impairing performance. In particular, the architecture shall provide *timing isolation* between cores, i.e., the actions of a core must not influence the timing behavior of another. The architecture shall be *timing compositional*, i.e., the timing behavior must be repeatable and be free of timing anomalies. Examples of unpredictable hardware features with possible predictable alternatives include:

- replacing caches with fast software managed memories, called scratchpads [125],
- replacing out-of-order execution with better code generation from the compiler [35, 36, 96, 106, 117], and
- deactivating high-performance bus features, e.g., burst transfers or pipelining, and using fair time-sharing arbitration policies, e.g., round-robin or time division multiple access (TDMA) [108].

PRET or PROMPT-based processors are simpler to understand, model, and analyze. Multi-PRET [55] implements a multi-core PRET architecture by connecting n multi-threaded PRET cores, called FlexPRET [136], over a TDMA bus. MERASA [117] is a predictable multi-core processor that supports hard and non-real-time threads. Hard real-time threads access scratchpads for better predictability, while non-real-time threads access caches for better performance. An analyzable memory controller arbitrates the shared bus accesses from the cores.

In this paper, we have used a representative PRET-based multi-core processor [39] for benchmarking, illustrated in Figure 16. It consists of n identical Xilinx MicroBlaze [128] cores, each configured with a three-stage, in-order, timing anomaly-free pipeline connected to private data and instruction scratchpads. The scratchpads are statically allocated and loaded at compile time. A shared bus with TDMA arbitration connects the cores to shared resources, such as global memory and peripherals. For benchmarking purposes, we have developed a multi-core MicroBlaze simulator that significantly extends an existing one [124] by supporting cycle-accurate simulation, an arbitrary number of cores, and a shared bus with TDMA arbitration.

4.2 Static Thread Scheduling

This section deals exclusively with ForeC threads. We illustrate the static thread scheduling with the example of Figure 17a. The programmer statically allocates the threads to the cores and passes the allocations into the compiler. The scheduling is *static* and *non-preemptive*. Thus, threads execute without interruption until they reach a *context-switching point*: a `par` or `pause` statement, or the end of their body. The semantics of shared variables (see Section 2.4) ensures that threads execute their local ticks in isolation, e.g., independently of their siblings or their parent's siblings. The compiler defines a total order for all the threads. The total order is based on the depth-first traversal of the thread hierarchy. Figure 17b depicts the thread hierarchy of the ForeC program from Figure 17a, where numbers indicate the total order. A lower number means higher execution priority. Figure 17c shows a possible thread allocation chosen by the programmer for two cores, in their thread scheduling order. When a thread reaches a `par`, its child threads are forked for execution on their allocated cores. The core that executes the parent thread is called the *master* core, while the cores that execute the child threads are *slave* cores.

Based on the the thread allocation and scheduling order shown in Figure 17c, Figure 17d is a possible execution trace. The trace for both cores ("Core 1" and "Core 2") progresses downwards from the top of Figure 17d. Thread executions are shown as white segments in the trace and each one has the thread's name and the executed lines of code from Figure 17a. The compiler generates *synchronization routines* to manage the thread executions on the master and slave cores. These routines are shown as shaded segments in the trace and each has the routine's name, which is prefixed with "`m`" or "`s`" to identify whether the routine is for a master or slave core, respectively. The names are suffixed with an integer to identify the unique id assigned to each `par` (with a depth-first traversal of the thread hierarchy starting from the root). For example, the `mFork1`, `sFork1`, `mJoin1`, and `sJoin1` routines in Figure 17d all manage the threads forked by the `par` with id = 1 (line 6 of Figure 17a). Table 5 summarizes the behavior of the routines. The `mFork` and `sFork` routines manage the forking of child threads (Section 4.4). The `mJoin` and `sJoin` routines manage the joining of child threads (Section 4.4). The `mSync` and `sSync` routines manage the global tick synchronization of all the cores (Section 4.8). In Figure 17d, the synchronization between the routines are shown as arrows, which are also marked with information that the fork and join routines send between themselves. When execution reaches a `par`, the corresponding `mFork` routine sends the `par`'s unique id to its `sFork` routines, otherwise the integer -1 (`OTHER`) is sent. On a slave core, when all child threads of a `par` have terminated, the corresponding `sJoin` routine sends the integer 0 (`TERM`) to its `mJoin` routine, otherwise the integer -1 (`OTHER`) is sent.

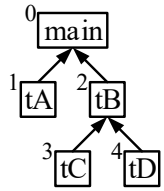
The threads and synchronization routines are statically scheduled on each core with *doubly linked lists*. Each node (defined in Figure 18) of a linked list represents a thread or a synchronization routine and stores its continuation point (`pc`) and the links to its adjacent nodes (`prev` and `next`). A node's `pc` is initially set to the start of the thread or routine's body. Each core starts its scheduling by jumping to the `pc` of its first node. When a context-switching point is reached during the execution of a thread or routine, a jump is made to the `pc` of the next node. A core will only execute the threads and routines in its linked list. Thus, inserting or removing a thread or routine from the list controls whether it is included or excluded, respectively, from execution. The remainder of this section describes how a ForeC program is compiled into a C program and how the linked lists are created and used to implement the ForeC semantics.

4.3 Structure of the Generated Program

Figure 19 shows a simplified version of the C program generated for the ForeC program in Figure 17a. All line numbers refer to Figure 19. The generated C program contains:

```
1  shared int x=0 combine all with plus;
2  void main(void) {
3    abort {
4      x=1;
5      pause;
6      par(tA(),tB());     // id = 1
7    } when (x > 1);
8  }
9
10 void tA(void) {
11   x=x+1;
12   pause;
13   x=x+1;
14 }
15 void tB(void) {
16   par(tC(),tD());       // id = 2
17 }
18 void tC(void) { int a=1; ... }
19 void tD(void) { ... }
20
21 int plus(int th1,int th2) {
22   return (th1+th2);
23 }
```

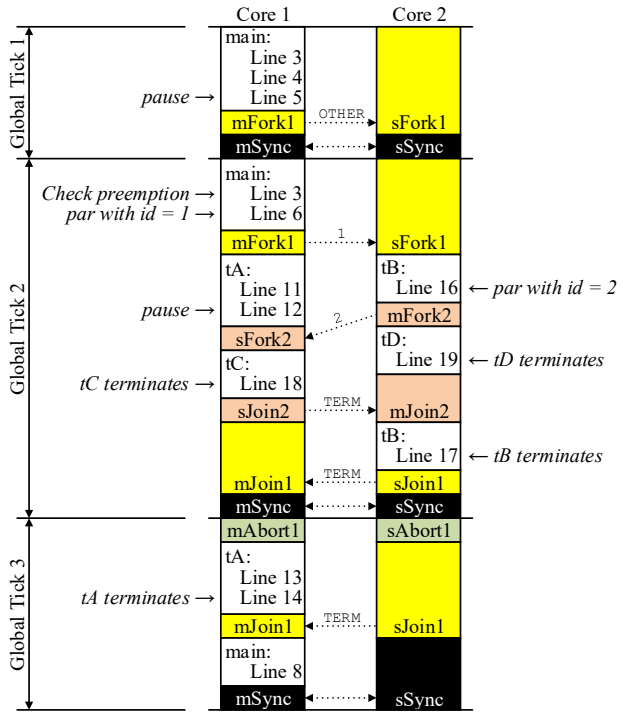
(a) Example ForeC program.



(b) Total order.

Core 1	Core 2
main	tB
tA	tD
tC	

(c) Thread allocation.



(d) Possible execution trace of the compiled program.

Fig. 17. Example ForeC program to be compiled.

Table 5. Summary of the synchronization routines

`mFork`: Uses a non-blocking send to notify the slave cores whether or not the parent thread has forked. `sFork`: Blocks until it receives whether or not the parent thread has forked.
`mJoin`: Blocks until it receives whether the child threads on other cores have terminated. Then, it notifies the slave cores whether the parent thread has resumed. `sJoin`: Uses a non-blocking send to notify the master core whether or not its child threads have terminated. Then, it blocks until it receives whether the parent thread has resumed.
`mSync`: Synchronizes with all the cores, performs the housekeeping tasks, and then synchronizes with all the cores again to start the next global tick. `sSync`: Synchronizes with all the cores and waits for the next synchronization to start the next global tick.
`mAbort` and `sAbort`: Evaluates the preemption condition of an `abort`.

```
1  // Node definition
2  typedef struct _Node {
3    void *pc;
4    struct _Node *prev, *next;
5  } Node;
6  // Insert node n2 after n1
7  #define insert(n1,n2)          \
8    n2.prev = &n1;               \
9    n2.next = n1.next;           \
10   n1.next->prev = &n2;         \
11   n1.next = &n2
12 // Remove node n2 from the list
13 #define remove(n2)             \
14   n2.prev->next = n2.next;     \
15   n2.next->prev = n2.prev
```

Fig. 18. Definition of a linked list node and its operations in `node.h`.

- Global declarations and functions from the ForeC program (lines 4–6).
- Global declarations for storing the execution states of the threads and implementing the shared variables (lines 9–13).
- `main` function (line 16) with the bootup routine (lines 34–41), the synchronization routines (lines 44–138), and the threads (lines 141–186).

When the cores enter the `main` function, they execute the bootup routine to initialize their linked lists. First, a node is created for each thread and each synchronization routine (lines 18–32). Second, the nodes are linked together to create the initial linked list for each core (lines 34–41). These initial lists are illustrated in the second row of Table 6. To avoid the need to create stacks for each thread to maintain their local variables, the local variables are given unique names and hoisted up to the global scope (e.g., `tC`'s local variable `a` on line 5). However, functions executed on the same core will share the same stack space. To avoid stack corruption, all the functions must execute atomically, i.e., without interruption. Although the scheduling routines dominate the generated code in Figure 19, their code remains constant whatever the size of the user-defined

Table 6. Core 1 and 2's initial lists and subsequent lists when threads fork

Execution Point	Linked Lists
When the program starts	**Core 1:** main ⇄ mFork1 ⇄ mSync **Core 2:** sFork1 ⇄ sSync
When `main` forks (id = 1)	**Core 1:** mAbort1 ⇄ tA ⇄ sFork2 ⇄ mJoin1 ⇄ mSync **Core 2:** sAbort1 ⇄ tB ⇄ mFork2 ⇄ sJoin1 ⇄ sSync
When `tB` forks (id = 2)	**Core 1:** mAbort1 ⇄ tA ⇄ tC ⇄ sJoin2 ⇄ mJoin1 ⇄ mSync **Core 2:** sAbort1 ⇄ tD ⇄ mJoin2 ⇄ sJoin1 ⇄ sSync

threads (which could be arbitrarily large). Benchmarking in Section 5 reveals that ForeC programs can achieve much higher multi-core speed ups than Esterel and be competitive to OpenMP on the same benchmark programs, while having high time-predictability with at most 3.2% overestimation in worst-case reaction times via static timing analysis.

4.4 The par Statement

The execution of a ForeC program starts with its `main` thread. The slave cores must wait for their allocated threads to be forked. *The global tick in which threads fork and join can only be determined at runtime.* Hence, before a core executes a thread, it must check that no other higher priority thread allocated to it will be forked. Otherwise, the higher priority thread must be executed first. This is achieved by executing an `mFork` routine after a parent thread completes its local tick. It uses a *non-blocking send* to notify the slave cores whether or not the parent thread has forked. Thus, the `sFork` routine of each slave core *blocks* until it receives whether or not the parent thread has forked. To ensure correct scheduling order, the `sFork` routine has the same execution priority as the parent thread. When a fork does occur, the `mFork` and `sFork` routines instruct their cores to suspend the parent thread and to schedule the child thread. In the first global tick of Figure 17d, `mFork1` notifies `sFork1` that thread `main` has not forked (`OTHER` is sent). In the second global tick, `mFork1` notifies `sFork1` that thread `main` has forked (`1` is sent).

Before a core executes a parent thread that was suspended by a fork, it must check that all of its child threads have terminated. This is achieved by executing an `mJoin` routine after the child threads on the master core have completed their respective local ticks. It *blocks* until it receives whether or not the child threads on the slave cores have terminated. When all child threads have terminated, the `mJoin` routine instructs the master core to resume the parent thread. Thus, each slave core executes an `sJoin` routine after its child threads complete their respective local ticks. It uses a *non-blocking send* to notify the master core whether or not the child threads on the slave core have terminated. In the second and third global ticks of Figure 17d, `sJoin1` notifies `mJoin1` that thread `tB` has terminated (`TERM` is sent). The `mJoin` routine is also responsible for combining the shared variables, but we defer this discussion to Section 4.6.

We now describe the C code that is generated for each `par` statement and how the synchronization routines are incorporated into the linked lists. The last two rows of Table 6 visualizes core 1 and 2's linked lists when threads `main` and `tB` fork (Figure 19, lines 153 and 177 respectively). Each `par` statement is assigned a unique positive integer `id` by the compiler. In Figure 19, lines 153–156 is an example of the C code that is generated for a `par` statement. Line 154 sets the parent thread's execution state to `id` and sets the parent thread's `pc` to be immediately after the `par` statement.

```
#include "node.h"   // Figure 18

// Programmer-defined
int x=0;    // Shared variable
int a_tC;   // tC's local variable
int plus(int th1,int th2) {return th1+th2;}

// Compiler-defined
enum State {OTHER=-1,TERM=0};
int mainState=OTHER, tAState=OTHER,
    tBState=OTHER, tCState=OTHER,
    tDState=OTHER;
int x_main, x_tA, x_tB, x_tC, x_tD;

// Entry point
void main(void) {
  // Nodes for the linked lists
  Node main={.pc=&&main},
       tA={.pc=&&tA}, tB={.pc=&&tB},
       tC={.pc=&&tC}, tD={.pc=&&tD};
  Node mFork1={.pc=&&mFork1},
       sFork1={.pc=&&sFork1},
       mJoin1={.pc=&&mJoin1},
       sJoin1={.pc=&&sJoin1};
  Node mFork2={.pc=&&mFork2},
       sFork2={.pc=&&sFork2},
       mJoin2={.pc=&&mJoin2},
       sJoin2={.pc=&&sJoin2};
  Node mAbort1={.pc=&&mAbort1},
       sAbort1={.pc=&&sAbort1};
  Node mSync={.pc=&&mSync},
       sSync={.pc=&&sSync};
  // Create initial linked lists
  if (core == 1) {
    main.prev=main.next=&main;
    insert(main,mFork1); insert(mFork1,mSync);
    goto *main.pc;
  } else if (core == 2) {
    sFork1.prev=sFork1.next=&sFork1;
    insert(sFork1,sSync); goto *sFork1.pc;
  } else { while(1); }

// Forking
mFork1: {
  send(mainState);
  if (mainState == 1) {
    Insert the nodes mAbort1,tA,sFork2, and mJoin1
    after mFork1.
    remove(main); remove(mFork1); goto *tA.pc;
  } else { goto *mFork1.next->pc; }
}
sFork1: {
  receive(mainState);
  if (mainState == 1) {
    Insert the nodes sAbort1,tB,mFork2, and sJoin1
    after sFork1.
    remove(sFork1); goto *sFork2.pc;
  } else { goto *sFork1.next->pc; }
}

mFork2: {
  send(tBState);
  if (tBState == 2) {
    Insert the nodes tD and mJoin2 after mFork2.
    remove(tB); remove(mFork2); goto *tD.pc;
  } else { goto *mFork2.next->pc; }
}
sFork2: {
  receive(tBState);
  if (tBState == 2) {
    Insert the nodes tC and sJoin2 after sFork2.
    remove(sFork2); goto *tC.pc;
  } else { goto *sFork2.next->pc; }
}

// Joining
mJoin2: {
  receive(tCState);
  x_tB=plus(x_tC,x_tD,x);   // Combine
  if (tCState == TERM
      && tDState == TERM) {
    tBState=OTHER; send(tBState);
    insert(mJoin2,tB); remove(mJoin2);
    goto *tB.pc;
  } else {
    send(tBState); goto *mJoin2.next->pc;
  }
}
sJoin2: {
  send(tCState); receive(tBState);
  if (tBState == OTHER) { remove(sJoin2); }
  goto *sJoin2.next->pc;
}
mJoin1: {
  receive(tBState);
  x_main=plus(x_tA,x_tB,x);   // Combine
  if (tAState == TERM
      && tBState == TERM) {
    mainState=OTHER; send(mainState);
    insert(mJoin1,main); remove(mAbort1);
    remove(mJoin1); goto *main.pc;
  } else {
    send(mainState); goto *mJoin1.next->pc;
  }
}
sJoin1: {
  send(tBState);
  receive(mainState);
  if (mainState == OTHER) { remove(sJoin1); }
  goto *sJoin1.next->pc;
}

// Preempting
mAbort1: {
  if (x > 1) {
    Remove the linked nodes between mAbort1
    and mJoin1 inclusive.
    main.pc = &&abort1; goto *main.pc;
  } else { goto *mAbort1.next->pc; }
}
```

Fig. 19. Example of the C program generated for the ForeC source code of Figure 17a with the thread allocation of Figure 17c.

```
121 sAbort1: {
122     if (x > 1) {
123         Remove the linked nodes between sAbort1
124         and sJoin1 inclusive.
125         goto *sAbort1.next->pc;
126     } else { goto *sAbort1.next->pc; }
127 }
128
129 // Synchronizing
130 mSync: {
131     barrier();
132     x=x_main; emitOutputs(); sampleInputs();
133     barrier();
134     goto *mSync.next->pc;
135 }
136 sSync: {
137     barrier(); barrier(); goto *sSync.next->pc;
138 }
139
140 // Threads
141 main: {
142     copy(x_main,x);
143     /* abort */ {
144         x_main=1;
145
146         // pause;
147         main.pc=&&pause1;
148         goto *main.next->pc;
149         pause1:;
150         if (x > 1) { goto abort1; }
151         copy(x_main,x);
152
153         // par(tA(),tB()); with id=1
154         mainState=1; main.pc=&&join1;
155         goto *main.next->pc;
156         join1:;
157         copy(x_main,x);
158     } // when (x_main > 1);
159     abort1: exit(0);
160 }
161 tA: {
162     copy(x_tA,x_main);
163
164     x_tA=x_tA+1;
165
166     // pause;
167     tA.pc=&&pause2; goto *tA.next->pc;
168     pause2: copy(x_tA,x);
169
170     x_tA=x_tA+1;
171
172     // Termination
173     tAState=TERM; remove(tA);
174     goto *tA.next->pc;
175 }
176 tB: {
177     // par(tC(),tD()); with id=2
178     tBState=2; tB.pc=&&join2; goto mFork2;
179     join2:;
180     // Termination
181     tBState=TERM;
182     remove(tB);
183     goto *tB.next->pc;
184 }
185 tC: { a_tC=1; ... }
186 tD: { ... }
187
188 }  // End of "void main(void)"
```

Fig. 19. (Continued.) Example of the C program generated for Figure 17a.

Line 155 is a context-switch to the parent thread's `mFork` routine. Lines 44–51 is an example of the C code that is generated for an `mFork` routine. Line 45 sends the parent thread's execute state to the slave cores. If the parent thread has forked, then lines 47–49 insert the allocated child threads and then an `mJoin` routine into the linked list. The parent thread and `mFork` routine are removed from the linked list. The end of line 49 is a context-switch to the first node that was inserted. Otherwise, if the parent thread has not forked, then line 50 is a context-switch to the next node in sequence.

If a child thread can fork its own threads, then further `mFork` and `sFork` routines need to be inserted into the linked lists to ensure that such nested threads are forked. Recall from the example in Figure 17a that thread `main` has a `par` statement that forks the child thread `tB`, which itself forks threads `tC` and `tD`. Thus, when thread `main` executes its `par` statement, thread `tB` is inserted into the linked list along with its fork routine (named `mFork2`) on line 55 in Figure 19. Similarly, a fork routine (named sFork2) is inserted into the other core's linked list on line 47. Both routines are defined on lines 61–74.

Recall that the slave cores have an `sFork` routine in their initial linked list. Lines 52–59 show an example of the C code that is generated for an `sFork` routine. Line 53 blocks until it receives whether the parent thread has forked. If the parent thread has forked, then line 55 inserts the allocated child threads and then an `sJoin` routine into the linked list. The `sFork` routine is removed from the linked list. The end of line 57 is a context-switch to the first node that was inserted. Otherwise, if the parent thread has not forked, then line 58 is a context-switch to the next node in sequence.

Lines 180–183 in Figure 19 is an example of the C code that is generated for the end of a child thread to handle thread termination. Line 181 sets the thread's execution state to `TERM`. Line 182 removes the thread from the linked list. Line 183 is a context-switch to the next node in sequence. Lines 94–105 is an example of the C code that is generated for an `mJoin` routine. Line 95 blocks until it receives the execution state of each child thread. If all the child threads have terminated, then line 99 sets the execution state of the parent thread to `OTHER` and sends this state to its slave cores. Lines 100–101 insert the parent thread back into the linked list and removes the nodes associated with the `par` statement. This is followed by a context-switch to the parent thread. Otherwise, if some child threads have not terminated, then line 103 is is a context-switch to the next node in sequence. Lines 106–111 is an example of the C code that is generated for an `sJoin` routine. Line 107 sends the execution state of each child thread to the master core. Line 108 blocks until it receives whether the parent thread has been resumed. If the parent thread has been resumed, then line 109 removes the `sJoin` routine from the linked list. Finally, line 110 is a context-switch to the next node in sequence.

4.5 The pause Statement

The `pause` statement is a context-switching point and lines 146–149 in Figure 19 is an example of the C code that is generated. Line 147 sets the current thread's `pc` to be immediately after the `pause` statement. Line 148 is a context-switch to the next node in sequence. In the next global tick, execution will resume from the statement immediately after the `pause` statement.

4.6 Shared Variables

Shared variables are hoisted up to the program's global scope to allow all cores to access them (e.g., line 4 in Figure 19). The copies of shared variables are implemented as unique global variables (e.g., line 13) to allow them to be combined on different cores. In each thread, all shared variable accesses are replaced by accesses to their copies (e.g., lines 144 and 164). The shared variables are copied at the start of each local tick, i.e., at the start of each thread body, and after each `pause` and `par` statement. For example, the shared variable `x` on line 4 is copied by thread `main` on lines 142, 151, and 157. As defined by the (par-4), (par-5), (par-6), and (par-7) semantic rules given in Section 3.4, the `par` statement is responsible for combining the copies of shared variables. More precisely, when the child threads of a `par` statement complete their respective local ticks, their copies of shared variables are combined. The combined result is assigned to their parent thread. This combine process is implemented by the `mJoin` routine (e.g., line 96) because it waits for the child threads to complete their respective local ticks. The final values of the shared variables are computed by the `mJoin` routine of thread `main`.

4.7 The abort Statement

We begin by describing the C code that is generated for an `abort` that does not have the optional `immediate` or `weak` keywords. Conditional jumps, using the preemption condition, are inserted after each `pause` statement in the `abort` body. For example, lines 143–158 in Figure 19 is the code generated for the `abort`; a conditional jump is inserted on line 150 after the `pause` statement. The preemption condition `x>1` is used in the conditional jump. If the preemption condition evaluates to *true*, a jump is made to the statement immediately after the `abort` (e.g., line 159). If a `par` statement is inside the `abort` body, then the preemption condition must be evaluated before the threads can execute. For example, in the third global tick of Figure 17d, the cores use the `mAbort` and `sAbort` routines to evaluate the preemption condition on line 7 of Figure 17a. It is safe to evaluate the preemption conditions in parallel because they are side-effect free (Section 2.7). Thus, when a fork occurs, an `Abort` routine is inserted before the child threads in the linked lists. For a master core,

```
/* abort */ {
  if (x > 1) { goto abort1; }
  x_main=1;

  // pause;
  main.pc=&&pause1;
  goto *main.next->pc;
  pause1:;
  if (x > 1) { goto abort1; }
  copy(x_main,x);

  // par(tA,tB) with id=1
  mainState=1;
  goto *main.next->pc;
  join1:;
  copy(x_main,x);
} // when (x_main > 1);
```

(a) Immediate and strong abort.

```
int triggered = 0;
/* abort */ {
  x_main=1;

  // pause;
  main.pc=&&pause1;
  goto *main.next->pc;
  pause1:;
  triggered = (x > 1);
  copy(x_main,x);

  // par(tA,tB) with id=1
  mainState=1;
  goto *main.next->pc;
  join1:;
  copy(x_main,x);
} // when (x_main > 1);
```

(b) Non-immediate and weak abort.

```
int triggered = 0;
/* abort */ {
  triggered = (x > 1);
  x_main=1;

  if (triggered) { goto abort1; }
  // pause;
  main.pc=&&pause1;
  goto *main.next->pc;
  pause1:;
  triggered = (x > 1);
  copy(x_main,x);

  // par(tA,tB) with id=1
  mainState=1;
  goto *main.next->pc;
  join1:;
  copy(x_main,x);
} // when (x_main > 1);
```

(c) Immediate and weak abort.

Fig. 20. C code for the immediate and weak variants of the abort on lines 143–158 of Figure 19.

lines 114–120 in Figure 19 is an example of the C code that is generated for an mAbort routine. Line 115 evaluates the preemption condition. If it evaluates to *true*, then line 116 removes the nodes associated with the par statement. Line 118 sets the parent thread's pc to be immediately after the abort statement and context-switches to the parent thread. Otherwise, if the preemption condition evaluates to *false*, then line 119 is a context-switch to the next node in sequence. For a slave core, lines 121–127 is an example of the C code that is generated for an sAbort routine and is similar to that of an mAbort. Line 122 evaluates the preemption condition. If it evaluates to *true*, then line 123 removes the nodes associated with the par statement. Line 125 is a context-switch to the next node in sequence. Otherwise, if the preemption condition evaluates to *false*, then line 126 is a context-switch to the next node in sequence.

The optional immediate keyword allows the preemption condition to be evaluated before the abort body is executed for the first time. Thus, an additional conditional jump, using the preemption condition, is inserted at the start of the abort body. Figure 20a is an example of the C code that would be generated if the abort on lines 143–158 in Figure 19 was an immediate abort. The optional weak keyword delays the jumping to the end of the abort body when the preemption condition evaluates to *true*. Thus, the conditional jump is divided into two parts: (1) the evaluation of the preemption condition and (2) the resulting jump. The evaluation is inserted directly after each pause statement and the jump is inserted directly before each pause statement. If a par statement is inside a weak abort, then the mAbort and sAbort routines are inserted after the child threads in the linked lists. Figure 20b is an example of the C code that would be generated if the abort on lines 143–158 in Figure 19 was a weak abort. Finally, Figure 20c is an example of the C code generated if it was an immediate and weak abort.

4.8 Global Tick Synchronization

The global tick is implemented by inserting a Sync routine that implements *barrier synchronization* at the end of each linked list. This synchronization is shown at the end of each global tick in

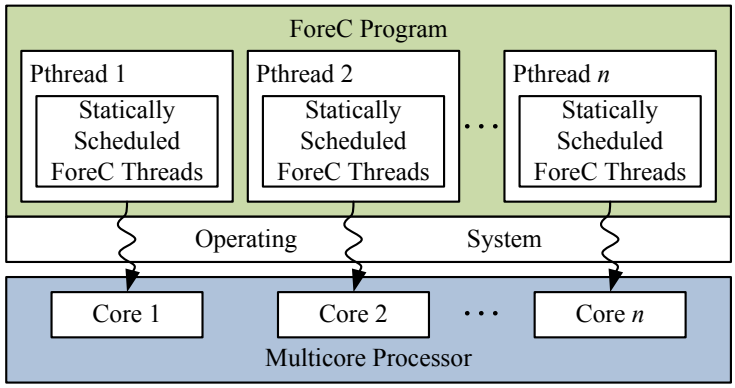


Fig. 21. Using Pthreads to adapt the generated code for multi-cores.

```
1  // Compiler–defined
2  #include <pthread.h>
3  pthread_t cores[2];
4  ...
5  // Entry point
6  void main(int argc,char ** argv) {
7     pthread_create(&cores[0], ... ,forecMain, ...); pthread_create(&cores[1], ... ,forecMain, ...);
8     pthread_join(cores[0],NULL); pthread_join(cores[1],NULL);
9  }
10 // Original main function from Figure 19
11 void *forecMain(void *args) { ... }
```

Fig. 22. Example Pthreads program.

Figure 17d. Lines 130–135 is the `mSync` routine C code generated for the master core. Line 131 is a barrier synchronization for the end of the tick. Line 132 performs the following housekeeping tasks: finalizing the values of the shared variables, emitting outputs, and sampling inputs. Line 133 is a barrier synchronization to signal the start of the next global tick. Line 134 is a context-switch to the first node in the linked list. For the remaining slave cores, lines 136–138 is the `sSync` routine C code. Line 137 are barrier synchronizations for the end of the tick and the start of the next tick. This is followed by a context-switch to the first node in the linked list.

4.9 Generating Programs for Execution on Operating Systems

By default, the ForeC compiler generates bare metal embedded C code. This section describes how the ForeC compiler is extended to generate executable code for operating systems. To utilize multiple cores in a system, a program must create multiple threads that the operating system can schedule. To this aim, we modify the ForeC compiler to generate a Pthread [115] for each core in the system. Each Pthread is responsible for executing the ForeC threads statically allocated to the same core, as shown in Figure 21. In effect, a fixed pool of Pthreads executes the ForeC threads and the cost of creating each Pthread is only incurred once. Although the Pthreads will be dynamically scheduled by the operating system, the original ForeC threads will still follow their static schedule. Finally, the generated Pthreads program is compiled with any GNU C compiler.

For the ForeC program of Figure 17a, Figure 22 is a simplified extract of the generated Pthreads program. In addition to the global declarations shown in Figure 19, there are now Pthreads-related declarations (lines 2–3) and a new `main` function for creating the Pthreads (line 6). The original `main` function from Figure 19 (line 16) is renamed as `forecMain` (line 11). When the operating system executes the `main` function, the Pthreads start executing the `forecMain` function and, hence, the statically allocated ForeC threads.

4.10 Discussion

This section has presented the compilation of ForeC programs for direct execution on parallel hardware architectures. The compilation is syntax-driven and templates are used to generate code for each ForeC construct. Light-weight synchronization routines are generated to manage the forking and joining of threads across the cores. The use of linked lists to manage the scheduling of threads and routines is inspired by that of the Columbia Esterel Compiler [40].

The code generation is structural, meaning that a nesting of ForeC constructs is compiled into a nesting of each construct's generated code. The advantages of our static scheduling approach include: (1) a light-weight scheduling of ForeC threads, and (2) an easier analysis (e.g., WCET computation) because all scheduling decisions are known beforehand. However, the disadvantages include: (1) the inability to dynamically load balance the ForeC threads to utilize the idle cores, and (2) the need to recompile the program to target a different number of cores.

Memory fences in C (e.g., `atomic_thread_fence` [61]) are not used to implement the semantics of shared variables because (1) the reading of inputs and the writing of outputs for global tick synchronisation already requires barrier synchronization among the cores, making memory fences redundant for finalizing shared variables, and (2) memory fences on shared variables are unable to isolate the accesses of one thread from the accesses of another thread, which is needed during each local tick.

Finally, the ForeC compiler does not check the associativity and commutativity of combine functions (the general problem is undecidable [28]). Therefore it falls on the programmer's responsibility to guarantee these two properties. The SOS semantics of Section 3 does not guarantee that the copies of a given shared variable are always resynchronized in the same order. If that was the case, then the requirement for associativity and commutativity could be relaxed.

5 FOREC BENCHMARKING

This section quantitatively evaluates ForeC's parallel execution performance on a mixture of data and control dominated benchmark programs. The comparison is with Esterel, a widely used synchronous language for real-time safety-critical systems, and with OpenMP, a popular framework for parallel programming. The static timing analysis of ForeC programs using the reachability technique is described in a previous paper [130], which showed that the worst-case reaction time (WCRT) [17] of ForeC programs could be estimated to a high degree of precision. Such time-predictability is very useful for the implementation of real-time, safety-critical, embedded systems. We highlight the key timing analysis results in this section. Our compilation approach offers good parallel execution that is amenable to static timing analysis. To our knowledge, no other synchronous language achieves parallel execution and timing predictable as good as ForeC.

5.1 Benchmark Programs

The following benchmark programs are used in the evaluation:

- `FlyByWire` is based on the real-time UAV benchmark called PapaBench [88]. `FlyByWire` is a control dominated program with several tasks managing the UAV's motors, navigation, timer, and operation mode.
- `FmRadio` [98] is based on the GNU Radio Package [47], which transforms a fixed stream of radio signals into audio. The history of the radio signals is used to determine how the remaining stream should be transformed. `FmRadio` is data orientated.
- `Life` simulates Conway's Game of Life [42] for a fixed number of iterations and a given grid of cells. In each iteration, the outcome of each cell can be computed independently. `Life` presents a good mixture of data and control dominated computations.
- `Lzss` uses the Lempel-Ziv-Storer-Szymanski (LZSS) [112] algorithm to compress a fixed amount of text. Multiple sliding windows are used to search different parts of the text for repeated segments that can compressed. `Lzss` presents a good mixture of data and control dominated computations.
- `Mandelbrot` computes the Mandelbrot set for a square region of the complex number plane. The Mandelbrot set for each point in the region can be computed independently, making `Mandelbrot` data dominated.
- `MatrixMultiply` computes the matrix multiplication of two equally sized square matrices. Each element in the result can be computed independently, making `MatrixMultiply` data dominated.
- `Pi` computes the value of pi to 20 decimal places using the infinite series $\pi = 4 \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$.

5.2 Performance Evaluation

We create C (non-multi-threaded), ForeC, Esterel, and OpenMP versions of each benchmark program and handcrafted each for best performance. We use *Speedup* as the performance metric to compare ForeC, Esterel, and OpenMP with respect to the execution time of the C version:

$$Speedup(P) = \frac{\text{Execution time of the sequential C version}}{\text{Execution time of } P}$$

where P is either the ForeC, Esterel, or OpenMP version of the benchmark program being tested. The higher the speedup the better.

5.2.1 Comparison with Esterel. The static and dynamic thread scheduling approaches of Yuan et al. [132, 134] for the parallel execution of Esterel programs has proved to perform well on an Intel multi-core and on a (simulated) Xilinx MicroBlaze multi-core. However, compiler support is only available for the dynamic approach on MicroBlaze. Thus, we evaluate Esterel on a MicroBlaze multi-core with dynamic scheduling. Yuan et al.'s dynamic approach relies a special hardware FIFO queue to allocate the threads to the cores. Threads are added to the queue when they are forked by other threads. Each core retrieves a thread from the queue and executes it until it terminates or reaches a context-switching point for resolving signal statuses. A core makes a context-switch by adding the executing thread back to the queue and retrieving a different thread from the queue. Threads are removed from the queue when they terminate. All the cores can access the FIFO queue in parallel and each access takes two clock cycles to complete. For benchmarking, the MicroBlaze multi-core simulator described in Section 4.1 is extended with the necessary hardware queue. The configuration of the simulator is detailed in the caption of Table 7. The static scheduling approach presented in Section 4 is used to parallelize the ForeC programs.

Table 7 shows the implementation details of the ForeC and Esterel versions of the benchmark programs. In Esterel, it is usual to define data computations in a more capable host language such as C. Hence, for the "Lines of Code" column of Table 7, the first number is the lines of Esterel code and

Table 7. ForeC versus Esterel benchmarks: Average speedup on four cores, normalized to sequential runtime. Xilinx MicroBlaze details: 4 single-threaded cores, three-stage pipeline, no speculative features (no branch prediction, caches, or out-of-order execution), 16 KB private data and instruction scratchpads on each core (1 cycle access time), 64 KB global memory (5 cycle access time), TDMA shared bus (5 cycle time slots per core), Benchmarks compiled with GCC-4.1.2 -O0.

	Lines of Code		No. of Threads		Average Speedup	
Benchmark	**ForeC**	**Esterel**	**ForeC**	**Esterel**	**ForeC**	**Esterel**
`Life`	212	139+111	4 (4)	7 (4)	**3.23**	2.28
`Lzss`	485	42+421	4 (4)	4 (4)	**3.20**	2.42
`Mandelbrot`	381	220+337	8 (8)	18 (9)	**3.68**	1.20
`MatrixMultiply`	162	51+53	16 (8)	16 (8)	**3.87**	**3.87**

the second number is the lines of host C-code (excluding header files). The "No. of Threads" column specifies the total number of threads forked by the programs and, in brackets, the total number of threads that can execute together in parallel. The benchmark programs are compiled for bare-metal execution and do not need operating system support. Yuan et al.'s compilation approach [132] uses an intermediate format called GRaph Code (GRC) [100], which transforms the program into an acyclic execution graph to help schedule the resolution of signals. Executing the GRC from the root to its leaves corresponds to one tick of the program. To decide which GRC states need to be executed during each tick, a set of internal variables are updated as the GRC is executed. The GRC can obfuscate the parallelism and, for most programs, the ForeC compiler (see Section 4) can generate more efficient code requiring less context-switching. The same input vector is given to the ForeC and Esterel versions of a benchmark program to ensure the same final output. When a program terminates, the simulator returns the execution time in clock cycles.

Table 7 shows the speedups achieved by ForeC and Esterel on four cores. ForeC shows superior performance compared to Esterel, apart from `MatrixMultiply`, even though Esterel uses dynamic scheduling with hardware acceleration. The need to resolve instantaneous signal communication in Esterel can lead to significant runtime overheads. All possible signal emitters must be executed before any signal consumers can execute and this invariant is achieved using a signal locking protocol [132] that is costly. In comparison, shared variables in ForeC only need to be resolved at the end of each tick. The significance of the overhead is evident in the `Mandelbrot` results, where the Esterel version has 24 unique signals and only achieves a speedup of 1.2×. In fact, when `Mandelbrot` is executed on a single core, Esterel's execution time is already 58% longer than the C version. ForeC's execution time is only 0.2% longer than the C version. Because of minimal data dependencies in `MatrixMultiply`, combine functions are not needed in the ForeC version and signals are not needed in the Esterel version. Thus, the scheduling overheads for the ForeC and Esterel versions are minimal, resulting in near identical speedups.

Ju et al. [66] provide a multi-core static scheduling approach for Esterel. However, we cannot compare with that work because speedup results for multi-core execution were not reported.

5.2.2 Comparison with OpenMP. The purpose is to evaluate ForeC's competitiveness compared to OpenMP, which has been tailored for general-purpose and high-performance computing. Table 8 shows the implementation details of the ForeC and OpenMP versions of the benchmark programs. Intel VTune Profiler [59] was used to profile and guide the parallelization of the OpenMP versions of the benchmarks. It provided insight into the regions of code that are most time consuming and,

Table 8. ForeC versus OpenMP benchmarks: Average speedup on four cores, normalized to sequential runtime.
(L)aptop details: Intel Core i7-1060NG7 at 1.2 GHz, 4 cores, Hyper-Threading disabled, Turbo Boost disabled, Speed Shift disabled, Speed Step disabled, 8 MB L3 cache, 16 GB RAM, Windows 10, Clang 10.0.0, OpenMP 4.5, Benchmarks compiled with optimizations disabled.
(S)erver details: Intel Xeon Gold 6240 CPU at 2.60GHz, 18 cores, Hyper-Threading disabled, Turbo Boost disabled, Speed Shift disabled, Speed Step disabled, 24 MB L3 cache, 64 GB RAM, Ubuntu 20.04, Clang 10.0.0, OpenMP 4.5, Benchmarks compiled with optimizations disabled.

	Lines of Code		No. of Threads		Average Speedup	
Benchmark	**ForeC**	**OpenMP**	**ForeC**		**ForeC**	**OpenMP**
`FlyByWire`	241	227	8 (7)	L	2.16	**3.29**
				S	1.48	**3.11**
`FmRadio`	481	382	12 (6)	L	1.88	**2.16**
				S	**1.90**	1.84
`Life`	325	268	10 (8)	L	2.53	**3.69**
				S	2.16	**3.68**
`Lzss`	593	552	4 (4)	L	3.11	**3.53**
				S	3.43	**3.55**
`Mandelbrot`	111	89	4 (4)	L	3.85	**3.93**
				S	3.88	**3.99**
`MatrixMultiply`	156	121	7 (4)	L	**3.22**	3.05
				S	**3.73**	3.43
`Pi`	80	55	4 (4)	L	**4.18**	4.04
				S	3.93	**3.95**

therefore, top candidates for parallelization. Testing was carried out on a laptop and on a server and their configurations are detailed in the caption of Table 8.

The OpenMP versions use the OpenMP dynamic and static thread scheduling pragmas. Static scheduling was used in benchmarks (e.g., `FlyByWire` and `Pi`) when we could determine at compile time the so called *chunk size* (the workload and number of loop iterations that each thread needs to perform). Dynamic scheduling was used in benchmarks (e.g., `MatrixMultiply` and `Mandelbrot`) when the chunk size of each thread could not be made equal or could not be determined at compile time. For dynamic scheduling, the chunk size of each thread is determined by the OpenMP runtime. Using dynamic scheduling does introduce slight overheads, especially thread locking, but these overheads should be amortized across the overall run of the benchmarks. This OpenMP scheduling approach is in contrast to the ForeC approach, where all scheduling is static and determined by the ForeC compiler.

Table 8 shows the speedups achieved by ForeC and OpenMP when the benchmark programs are executed over four cores on a laptop (L) and server (S). The speedups are averaged over 3 of the shortest executions times observed for each program to take into account the potential effects of long term use, e.g., filling and flushing of the cache, and background processes of the platforms. The speedups achieved by ForeC and OpenMP vary widely across different benchmarks, between the same benchmarks, and across the laptop and server platforms.

On the server platform, the speedups achieved by ForeC are quite close to those of OpenMP for `Lzss`, `Mandelbrot`, and `Pi`, while ForeC is better than OpenMP for `FmRadio` and `MatrixMultiply`. However, the speedups achieved by ForeC for `FlyByWire` and `Life` are noticeably worse. The poor ForeC result for `FlyByWire` is due to a computationally heavy function being called by multiple threads, creating unbalanced workloads on the cores. The ForeC version would achieve better speedup if the function was duplicated for each of its invocations and parallelized individually for a balanced workload. The OpenMP version benefits by having its (statically computed) chunk sizes scheduled at run-time for better load balancing. Workload imbalance is also an issue for the ForeC version of `Life`, since not all threads compute the same number of cells in each iteration and threads are forked frequently for fine-grained parallelism.

The poor parallel performance of OpenMP for `MatrixMultiply`, which is a data-dominant program, is likely due to the OpenMP runtime accessing its own data and causing matrix elements to be evicted from fast L1 and L2 caches. For example, when executed on a single core, OpenMP's execution time is already 15% longer than the sequential C version, while ForeC is only 1% longer. Intel VTune reports that the OpenMP version allocates a total of 5.6MB, of which 4.3MB is for the matrices. This is significant because the server platform only has 32KB of L1 and 1MB of L2 caches per core. In contrast, the ForeC version only allocates a total of 4.4MB. When the execution time of OpenMP on four cores is compared to itself on a single core, a speedup of 3.9× is obtained. For the other benchmarks, the overhead for OpenMP is much less significant.

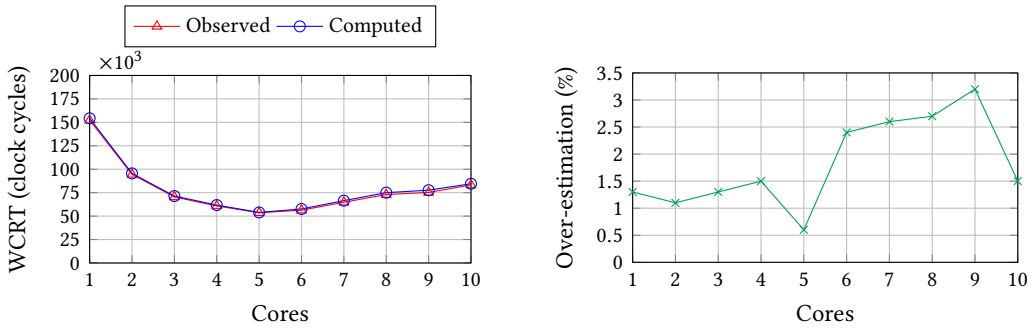
On the laptop platform, the speedups achieved follow a similar trend to those of the server platform. However, the performance gap between ForeC and OpenMP is narrower for `FlyByWire`, `Mandelbrot`, and `Life`, and wider for `FmRadio` and `Lzss`. These differences are likely due to the processor architectures and underlying operating systems. A surprising result can be seen for `Pi` where ForeC and OpenMP achieve super-linear speedup.

As the purpose of the benchmarking is to demonstrate the competitiveness of ForeC programs in a more general setting, without optimizing benchmarks to specific systems, the results are encouraging for the use of ForeC to develop high performing parallel programs. Moreover, recall that determinism is enforced by ForeC's formal semantics, while it is not the case for OpenMP (and other runtime environments).

5.3 Time-Predictability

We developed a C++ static timing analysis tool [130], called ForeCast, that statically analyzes the WCRT of ForeC programs on the embedded PRET multi-core architecture described in Section 4.1. We highlight the key findings of our previous paper [130] on the static WCRT analysis of ForeC programs. Benchmarking was performed on our MicroBlaze multi-core simulator with the configuration detailed in the caption of Figure 23. ForeCast itself was executed on an Intel Core 2 Duo 2.20 GHz desktop computer with 3 GB RAM running Linux 2.6.38. We highlight the results of the benchmark program called `802.11a` [98], which is production code from Nokia that tests various signal processing algorithms needed to decode 802.11a data transmissions. `802.11a` has complex data and control dominated computations. `802.11a` is implemented in 2,274 lines of ForeC code and forks up to 26 threads, of which 10 can execute in parallel. `802.11a` was distributed on up to 10 cores and ForeCast was used to compute the WCRT of each possible distribution.

In hard real-time systems, computing the WCRT is extremely difficult [126] and uses many over-approximations, which result in a computed WCRT that is often quite far from the *observed* WCRT. It is therefore useful to assess the *tightness* of the computed WCRT. To do this, `802.11a` was executed on the MicroBlaze simulator for one million global ticks or until the program terminated. Test vectors were generated to elicit the worst-case program state by studying the program's



(a) WCRT results for `802.11a` in clock cycles.

(b) WCRT over-estimations for `802.11a`.

Fig. 23. Time-predictability results for `802.11a`. Xilinx MicroBlaze details: single-threaded cores, three-stage pipeline, no speculative features (no branch prediction, caches, or out-of-order execution), 8 KB private data and instruction scratchpads on each core (1 cycle access time), 32 KB global memory (5 cycle access time), TDMA shared bus (5 cycle time slots per core and, thus, a 5×(number of cores) cycle long bus schedule), Benchmarks compiled with MB-GCC-4.1.2 -O0 and decompiled with MB-OBJDUMP-4.1.2.

control-flow. The simulator returned the execution time of each global tick and the longest was taken as the *observed* WCRT.

The observed and computed WCRTs of `802.11a` (in clock cycles) are plotted as a line graph in Figure 23a. It shows that ForeCast is very precise, even as the number of cores increases. The *over-estimation* of the computed WCRT is calculated as follows:

$$\text{WCRT Over-estimation} = \frac{\text{Computed WCRT} - \text{Observed WCRT}}{\text{Observed WCRT}} \times 100\%$$

Figure 23b depicts the WCRT over-estimations for `802.11a` as a line graph. We can see that ForeCast computes WCRTs that are at most 3.2% longer than the observed WCRTs. This is an excellent result because classical real-time systems show an over-estimation of, e.g., 2.3–13.3% [101] and 10–105% [77] for concurrent programs on up to 4 cores.

Figure 23a shows the impact of multi-core execution. The WCRT decreases when the number of cores is increased from 1 to 5 cores. However, the workload of the threads become can no longer be balanced 6 or more cores. In fact, the WCRT for 5 cores corresponds to the WCET of a reaction of a particular thread that has been allocated to its own core. Thus, no further WCRT improvements can be achieved with more than 5 cores and the WCRT worsens due to increasing scheduling overheads and global memory access costs.

We performed additional experiments to compare the observed worst-case execution times (WCETs) of ForeC and Esterel versions of `Life`, `Lzss`, `Mandelbrot`, and `MatrixMultiply` on embedded multi-cores. Since these programs are not reactive (they terminate), the WCRT is irrelevant so we observe the WCET instead, i.e., the total time that the programs take to execute from start to finish.[8] We limited the `Life` program to 10,000 iterations of its main loop. The same input vector was given to the ForeC and Esterel versions of a benchmark program to ensure the same final output. The Esterel programs were compiled using Yuan et al.'s approach [132]. For each benchmark program in Figure 24, the WCETs for ForeC and Esterel are plotted. Apart from `MatrixMultiply`, the observed WCETs for ForeC were much shorter than those for Esterel. Unfortunately, the static timing analysis of multi-core Esterel programs has not been developed, preventing an objective

[8]The entire execution of a benchmark program occurs over multiple ticks.

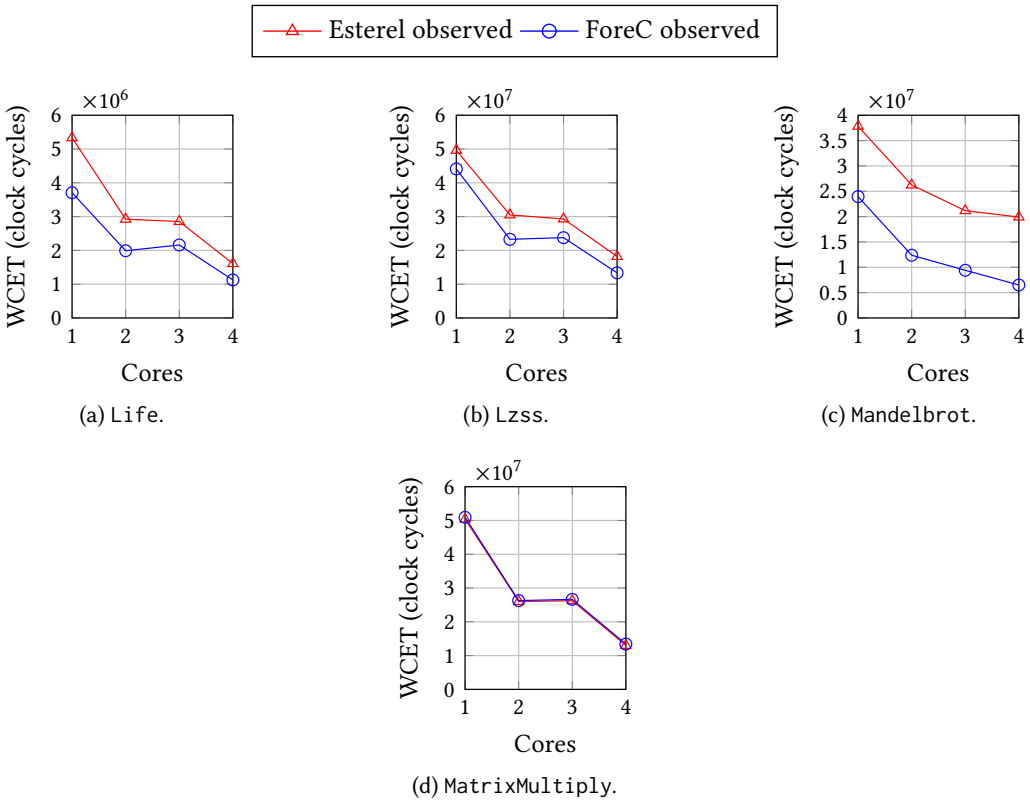


Fig. 24. Observed WCETs for ForeC and Esterel. Platform details in caption of Figure 7.

comparison of time-predictability. To compute WCETs or WCRTs for Esterel that are as tight as ForeCast, the dynamic resolution of signal statuses would need to be analyzed extremely carefully to rule out infeasible execution paths.

6 RELATED WORK

Designing embedded systems that are time predictable remains an open challenge [6]. Moreover, the increased adoption of embedded multi-cores requires more programmers to become parallel programming experts. Table 9 highlights different approaches to enforcing *mutual exclusion* on shared variables, usually by interleaving parallel accesses into a strict sequence. Some of the examples we report in Table 9 may rely on multiple approaches, but we have only placed them under their main approach. As argued by Lee [73], the adoption of parallelism in sequential languages, like C [61], discards important properties, such as determinism, predictability, and understandability. Thus, inconsistent values may be observed for shared variables and significant effort has to be devoted to tame nondeterminism in parallel programs [79].

6.1 Concurrency and Parallelism Issues

Deterministic runtime support. Runtime environments that enforce deterministic thread scheduling and memory accesses have been developed for Pthreads (Kendo [90] and CoreDet [11]), OpenMP (DOMP [5]), and MPI (DetMP [135]). In Kendo and CoreDet, all thread interactions are

Table 9. Existing solutions for avoiding race conditions in parallel programs

Programming Constructs: These are written in the host language to provide programmers with mechanisms to achieve mutual exclusion. Examples include locks, monitors, memory fences, transactional memory, message passing, and parallel data structures. Using such constructs correctly can be tedious and error prone for large programs and may lead to other errors [73, 79, 82], e.g., deadlocks, starvation, or priority inversion.
Language Semantics: A memory model defines how threads communicate over shared memory without race conditions, e.g., by defining admissible write-read orderings that are sequentially constructive [122]. The resulting memory model may restrict itself to certain applications. Examples include SharC [4], SHIM [119], ΣC [48], unfederated Lingua Franca [78], and Concurrent Revisions [23].
Static Analysis: A compiler or static analyzer can identify and alert the programmer about race conditions in their program (e.g., Parallel Lint [67]) and may try to automatically resolve them by serializing the parallel accesses (e.g., Sequentially Constructive Concurrency [122]). However, programmer guidance is needed when race conditions cannot be automatically resolved.
Runtime Support: Programs are executed by a runtime that dynamically enforces deterministic execution and memory accesses. Examples include dOS [12], Kendo [90], CoreDet [11], DOMP [5], DetMP [135], and federated Lingua Franca [78]. However, understanding the program's behavior at compile time remains difficult because the determinism is only enforced at runtime, which also makes the static computation of tight WCRTs difficult.
Hardware Support: Parallel accesses can be detected and resolved by hardware, preventing race conditions from happening. Examples include Ultracomputer's combine hardware [107] and some shared bus arbitration policies (e.g., round-robin, TDMA, and priority). However, the timestamps of the accesses affect how they are interleaved, i.e., race conditions are prevented but nondeterminism is not.

mapped deterministically onto a logical timeline, which progresses independently of physical time. Program execution is divided into alternating parallel and serial phases, similar to the Bulk Synchronous Parallel (BSP) [118] programming model. In the parallel phase, threads execute until they all reach a synchronization point, e.g., a lock acquisition, memory access, or predefined number of executed instructions. In the serial phase, threads take turns to resolve their memory accesses or lock acquisitions. Threads in CoreDet also maintain their own version of the shared memory state, which are resynchronized in every serial phase. This concept has been formalised by concurrent revisions [23]. However, understanding a program's behavior at compile time remains difficult because determinism is only enforced at runtime. Thus, if the program is modified, e.g., to fix a bug, then a vastly different runtime behavior is possible. Moreover, program execution is not portable across the runtimes because each may enforce its own notion of determinism.

Parallel C languages. An alternative is to directly extend and modify the C language with deterministic parallelism, such as SharC [103], CAT [41], SHIM [114], ΣC [48], and ForkLight [69]. These solutions allow the asynchronous forking and synchronized joining of threads, but lack a convenient mechanism for preempting groups of threads. Unlike ForeC, their time-predictability has not been demonstrated and this complicates their use in real-time safety-critical embedded systems.

Distribution of synchronous programs. The classic synchronous languages [10] are Esterel [15] (imperative), Lustre [50] (dataflow), Signal [49] (multi-clock), and Lucid Synchrone [31] (functional), and are well suited to control-dominated and safety-critical systems [10, 26]. A key advantage is that the burden of ensuring race-free communication is with the language semantics, and not left to the programmer. Concurrency in synchronous languages is a logical concept to help programmers manage concurrent inputs, rather than to specify parallel execution. Thus, concurrency is typically compiled away to resolve causality and to generate sequential code [40, 100], although some do generate concurrent tasks [27, 86, 87, 92] for execution on single-cores.

Techniques exist to distribute the code of a synchronous program over multiple processors, but it is motivated by the desire to execute computations closer to sensors or actuators, which may be separated geographically. The distribution [44] of synchronous programs is notoriously difficult due to the signal communication model and the need to maintain monotonic signal values [8, 134]. First, *causality analysis* [100] is needed to ensure that the presence or absence of all signals can be determined exactly in each global tick (a corollary is that programs can react to the *absence* of signals). Second, the compiler must generate code for resolving signal statuses at runtime. A common approach is to compile away the concurrency and to generate a sequential program [40, 100]. Third, the sequential program is partitioned into subprograms and distributed to execute on their allocated processors. When distributing a synchronous program, some desynchronization [9, 20, 46] is needed among the concurrent threads. That is, the concurrent threads execute at their own pace, but sufficient inter-thread communication is used to preserve the original synchronous semantics. Synchronized Distributed Executive (SynDEx) [99] is a tool that automatically distributes synchronous programs and considers the cost of communication between the processors. The use of *futures* has been proposed as a method for desynchronizing long computations in Lustre [30]. A *future* is a proxy for a result that becomes known at a later time and may be computed in parallel with other computations.

Code distribution can also be achieved by extracting parallelism from an intermediate representation of the sequentialized code [7, 8, 25, 44, 66, 94, 131, 134]. The techniques differ in the heuristics used to partition and distribute the sequentialized code to achieve sufficient parallelism, and some apply to general-purpose multi-cores [8, 66, 134]. Due to control and signal dependencies, the opportunities for extracting parallelism from a sequential program is limited. Yuan et al. [132, 134] offer static and dynamic thread scheduling approaches for Esterel on multi-cores. For the static approach, threads are statically load-balanced across the cores and signal statuses are resolved at runtime. The load balancing is based on the threads' estimated worst-case execution times, rather than on their actual times which could be much shorter. The dynamic approach instead relies on a custom hardware queue to store the threads that cores can execute.

The distribution of synchronous programs over multi-threaded and multi-core reactive processors has been studied extensively [33, 76, 105, 131, 133]. Reactive processors handle the scheduling of threads in hardware, thereby simplifying code generation. However, causality analysis and runtime signal resolution are still required. Signal resolution may impede parallel performance, for instance because a thread must wait for a signal's status to be resolved before it can be read.

In contrast, ForeC is much easier to compile because causality analysis is not needed for delayed communication over shared variables (Section 2.4) and the threads specified by the programmer can be distributed directly over the available cores. ForeC threads have more opportunities to execute in parallel because execution dependencies only exist at tick boundaries.

C-based synchronous languages. To increase their uptake with embedded programmers, C-based synchronous programming languages have been developed, starting with Reactive C [18]. The key characteristic of Reactive C is that communication between concurrent threads is *delayed* by one tick,

which simplifies the causality analysis, but forbids programs that react to the absence of an input to be written. Following this idea, other extensions of C have been proposed, such as Reactive Shared Variables [19], Esterel C Language (ECL) [72], and PRET-C [3]. A more recent proposal has been the *sequential constructive semantics*, adopted by Synchronous C (SCL) [120, 122], SCCharts [121], and SCEst [109], which allows concurrent writes to shared variables if a sequentially consistent interpretation can be guaranteed for their final values. This is known as the *init-update-read* protocol. This has later been generalized in two directions: Firstly by *scheduling policies* [1] where each shared variable type comes with a *policy interface* that formally defines the admissibility and precedence of its access methods (e.g., presence, emit, ...); And secondly by *scheduling directives* [110] where causality issues can be avoided (or fixed) by adding total order relationships between blocks of instructions, which can override the initial sequentially constructive schedule. In both cases, unlike ForeC, their inherent sequential execution semantics hinders their suitability for multi-core execution.

ForeC enables the deterministic parallel programming of multi-cores, and it merges deterministic concurrency offered by synchronous languages with deterministic shared states offered by runtime solutions such, as concurrent revisions. ForeC helps to bridge the differences between synchronous-reactive programming and general-purpose parallel programming, making deterministic parallelism accessible to traditional embedded C programmers. Its shared variable semantics guarantees deadlock freedom and alleviates the burden of ensuring mutual exclusion from the programmer. Thread isolation is guaranteed by stipulating that threads work on local copies of shared variables. The resynchronization of shared variables after threads reach their local tick makes program behavior agnostic to scheduling decisions. All these features simplify the understanding and debugging of ForeC programs.

WCET analysis. After a synchronous program has been implemented, it is necessary to validate the synchrony hypothesis. That is, the worst-case execution time [126, 127] (WCET) of any global tick must not exceed the minimal inter-arrival time of the inputs. This is known as worst-case reaction time (WCRT) analysis [17, 83] and various techniques have been developed for single-cores [2, 17, 29, 65, 71, 83, 104, 123] and multi-cores [66, 129, 130].

Combine functions. ForeC's combine functions are inspired by Esterel [100] but similar solutions can be found in other parallel programming frameworks, e.g., OpenMP's `reduction` operators [91], MPI's `MPI_Reduce` and `MPI_Gather` functions [84], Intel Thread Building Blocks' `tbb::parallel_reduce` function and `tbb::combinable` data type [58], Intel Cilk Plus' `reducer` data types [57], and Unified Parallel C's collective functions [116]. These solutions could be made available in ForeC as combine functions with particular combine policies. Appendix C provides more extensive examples of ForeC's combine functions and describes how they work together with the combine policies.

6.2 Detailed Comparison with Esterel, SCEst, PRET-C, and Concurrent Revisions

This section compares ForeC with Esterel [15], Sequentially Constructive Esterel (SCEst) [109], PRET-C [3], and concurrent revisions [23], summarized in Table 10. Concurrent revisions is a programming model that supports the forking and joining of asynchronous threads. When a thread is forked, a snapshot of the shared variables is created and any changes performed by the thread are only applied to this snapshot. Like ForeC, this ensures thread isolation during parallel execution. When two threads join, their snapshots are merged together using a deterministic *merge function*.

ForeC and PRET-C are intended for applications with control and data parallelism. Control parallelism is not a strength of concurrent revisions because its semantics does not consider (reactive) inputs or outputs. In concurrent revisions, threads are forked asynchronously, allowing a

Table 10. Comparing ForeC with Esterel, PRET-C, and concurrent revisions

Property	**Esterel** [15]	**SCEst** [109]	**PRET-C** [3]	**ForeC**	**Concurrent revisions** [23]
Causal Programs	Not always		Yes, by construction		
Use for Parallelism	Control		Control and data		Data
Model of Computation	Synchronous				Asynchronous
Reactive Interface	Yes				No
Preemption	Yes				No
Dynamic Parallelism	No				Yes
Thread Communication Method	Pure or valued signals	Pure or valued signals, or shared variables	Shared Variables		
Thread Communication Speed	Instantaneous	Instantaneous (sequentially constructive)	Instantaneous (sequential)	Delayed to the end of every tick	Delayed to thread join
Resynchronization of Shared Variables or Valued Signals	Combine functions (`mod` values)		Not required	Combine functions with policies	Merge functions (`all` values)
Commutative and Associative Parallelism	Yes		No (Sequential)	Yes if combine functions are commutative and associative	Yes if merge functions are commutative and associative

parent thread to execute alongside its children. Hence, the parent thread can fork as many parallel threads as needed at runtime. The `rjoin` construct can be used to force the parent thread to wait for its children to join. In contrast, threads are forked synchronously in ForeC, Esterel, SCEst, and PRET-C, meaning that the parent thread can only fork a fixed number of child threads and must wait for them to join.

Threads in PRET-C are executed in a strict sequential order, which is unsuitable for multi-core execution. However, this strict order ensures that only one thread is executing at any time and that shared variables are accessed in a thread safe manner. Consequently, thread communication is instantaneous (i.e., within the same tick) in the sequential order, but delayed by one tick in the reverse order. Threads in concurrent revisions modify their own snapshot of the shared variables, which are only merged when threads join. The *merge function* always considers both copies, i.e., equivalent to ForeC's combine policy `all`. Thus, thread communication only occurs when child threads join. In contrast, ForeC threads may execute over several ticks and thread communication is only delayed to the end of every tick.

Esterel and SCEst threads communicate instantaneously by emitting and receiving pure signals (either present or absent) or valued signals (pure signals with associated values) during each tick. All potential emissions of a signal must be performed by the concurrent threads before the signal can be read. For each valued signal, all its emitted values are combined using a programmer-specified commutative and associative combine function, i.e., equivalent to ForeC's combine policy `mod`. Signal statuses are reset to absent at the start of every tick. Note that only Esterel and SCEst are able to react to the absence of a signal. This allows behavior to be executed when a signal

is guaranteed to never be emitted during a tick, such as "`present X else emit Y`". However, non-causal statements can be written, such as "`present X else emit X`", which the Esterel compiler rejects because it has no behavior. SCEst can accept programs rejected by Esterel when causality issues due to non-concurrent signal accesses can be resolved by a sequential interpretation (the "init-update-read" policy).

Esterel and SCEst's parallel construct for forking threads is commutative and associative, thanks to the requirement that all combine functions must be commutative and associative. PRET-C's parallel construct is neither commutative nor associative because of its sequential semantics. In concurrent revisions and ForeC, the commutativity and associativity of their parallel construct is guaranteed if their respective combine and merge functions are also commutative and associative.

Preemptions in ForeC and PRET-C are inspired by Esterel, but behave slightly differently. Preemptions in Esterel and SCEst are triggered instantaneously, whereas preemptions in ForeC and PRET-C are triggered with a maximum delay of one tick. PRET-C, however, only supports immediate preemptions. Concurrent revisions does not support preemptions. Esterel programs may be non-causal [10] because of instantaneous feedback cycles, but ForeC, PRET-C, and concurrent revisions programs are always causal by construction thanks to delayed communication.

7 CONCLUSIONS AND FUTURE DIRECTIONS

A common approach to developing cyber-physical systems is to program an embedded ARM multi-core with C and Pthreads and to use an RTOS to manage the execution. Although high performance can be achieved with this approach, time-predictability is sacrificed. This paper proposed the ForeC language for the deterministic, parallel, and reactive programming of parallel architectures. Section 2 provided an in-depth description of ForeC. Unlike existing C-based synchronous languages, ForeC is designed specifically for parallel programming. The semantics of ForeC is designed to give programmers the ability to express explicit parallelism while ensuring that ForeC programs can be compiled efficiently for parallel execution and be amenable to static timing analysis. ForeC's main innovation revolves around its shared variable semantics that provides thread isolation and deterministic communication. The behavior of a shared variable can be tailored to the application at hand by specifying a suitable *combine function* and *policy*. All ForeC programs are correct by construction (no race conditions, no deadlocks) because mutual exclusion constructs are not needed. The formal semantics greatly simplifies the understanding and debugging of parallel programs. Section 4 presented a compilation approach that used non-preemptive static thread scheduling. The key strategy was to preserve the ForeC threads and to use light-weight context-switching and simple scheduling routines to preserve the language semantics.

Planning communication between threads is as difficult in ForeC as in any parallel programming language with shared variables (e.g., OpenMP), but, compared to such languages, there is no risk of race conditions thanks to ForeC's combine functions, thread isolation, and the deterministic semantics. We claim that this makes program writing and debugging easier. Assigning threads to cores and scheduling threads are not critical points because the ForeC semantics is preserved whatever the core assignment chosen by the users in their architecture specification. From this specification, the ForeC compiler statically allocates the threads, so no manual effort is required from the users. Allocating optimally the threads to the cores is very complex, and in fact NP-complete as in other similar parallelization problems. To achieve a good sub-optimal allocation, the ForeCast timing analyzer generates a timed sequence diagram for the program's WCRT: it combines the WCET of each thread with the global thread scheduling and inter-core synchronizations. This snapshot allows the programmer to see how balanced the workloads were across the cores and to devise, possibly, a better core allocation. In addition, the ForeC compiler detects loops that are potentially instantaneous (this is an over-approximation because the problem is undecidable),

variables that are accessed by multiple threads but have not been annotated as shared variables, and shared variables whose combine functions would never be invoked because at most only one copy would need to be combined. All the above-mentioned points demonstrate that, thanks to its semantics, compiler, and associated ForeCast tool, ForeC alleviates as much as possible the difficult task of writing safe and efficient parallel code for reactive systems.

For future work, the ForeC compiler could be improved to generate more efficient code that remains amenable to static timing analysis. In particular, different static scheduling strategies could be explored for different parallel programming patterns. Currently, scheduling priorities are assigned to ForeC threads by traversing the thread hierarchy in a depth-first manner. However, assigning scheduling priorities in a breadth-first manner could produce more efficient schedules in some cases. The allocation of ForeC threads could be refined automatically by feeding the WCRT results of the ForeCast analyzer into the ForeC compiler.

ACKNOWLEDGMENTS

This work was supported in part by the RIPPES INRIA International Lab. We wish to acknowledge Pascal Fradet and Jean-Bernard Stefani for useful discussions on ForeC, and the reviewers for their very helpful suggestions.

REFERENCES

[1] Joaquín Aguado, Michael Mendler, Marc Pouzet, Partha S. Roop, and Reinhard von Hanxleden. 2018. Deterministic Concurrency: A Clock-Synchronised Shared Memory Approach. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 86–113.

[2] Sidharta Andalam, Partha S. Roop, and Alain Girault. 2011. Pruning Infeasible Paths for Tight WCRT Analysis of Synchronous Programs. In *Design, Automation Test in Europe Conference Exhibition (DATE)*. 1 – 6.

[3] Sidharta Andalam, Partha S. Roop, Alain Girault, and Claus Traulsen. 2014. Predictable Framework for Safety-Critical Embedded Systems. *IEEE Trans. Comput.* 63, 7 (July 2014), 1600 – 1612.

[4] Zachary Anderson, David Gay, Rob Ennals, and Eric Brewer. 2008. SharC: Checking Data Sharing Strategies for Multithreaded C. *SIGPLAN Not.* 43, 6 (June 2008), 149 – 158.

[5] Amittai Aviram and Bryan Ford. 2011. Deterministic OpenMP for Race-Free Parallelism. In *Proceedings of the 3rd USENIX conference on Hot topic in parallelism* (Berkeley, CA) *(HotPar)*. USENIX Association.

[6] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard Von Hanxleden, Reinhard Wilhelm, and Wang Yi. 2014. Building Timing Predictable Embedded Systems. *ACM Transactions on Embedded Computing Systems* 13, 4, Article 82 (March 2014), 37 pages.

[7] Daniel Baudisch, Jens Brandt, and Klaus Schneider. 2010. Dependency-Driven Distribution of Synchronous Programs. In *Distributed, Parallel and Biologically Inspired Systems*, Mike Hinchey, Bernd Kleinjohann, Lisa Kleinjohann, Peter Lindsay, Franz Rammig, Jon Timmis, and Marilyn Wolf (Eds.). IFIP Advances in Information and Communication Technology, Vol. 329. Springer Boston, 169 – 180.

[8] Daniel Baudisch, Jens Brandt, and Klaus Schneider. 2010. Multithreaded Code from Synchronous Programs: Extracting Independent Threads for OpenMP. In *Design, Automation and Test in Europe (DATE)*. EDA Consortium, 949 – 952.

[9] Albert Benveniste, Benoît Caillaud, and Paul Le Guernic. 2000. Compositionality in Dataflow Synchronous Languages: Specification and Distributed Code Generation. *Inf. Comput.* 163, 1 (Nov. 2000), 125 – 171.

[10] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. 2003. The Synchronous Languages 12 Years Later. *Proc. IEEE* 91, 1 (Jan. 2003), 64 – 83.

[11] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 2010. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *Proceedings of the 15th ASPLOS on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, Pennsylvania, USA) *(ASPLOS)*. ACM, 53 – 64.

[12] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. 2010. Deterministic Process Groups in dOS. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI)*. 1 – 16.

[13] Gérard Berry. 1993. Preemption in Concurrent Systems. In *Conference on the Foundations of Software Technology and Theoretical Computer Science, FST&TCS'93 (LNCS, Vol. 761)*, R.K. Shyamasundar (Ed.). Springer-Verlag, Bombay, India, 72–93.

[14] Gérard Berry. 2000. *The Esterel v5 Language Primer.* Technical Report. Centre de Mathématiques Appliquées, Ecole des Mines and INRIA.

[15] Gérard Berry and Georges Gonthier. 1992. The Esterel Synchronous Programming Language: Design, Semantics and Implementation. *Science of Computer Programming* 19, 2 (1992), 87 – 152.

[16] Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning* 43, 3 (2009), 263 – 288.

[17] Marian Boldt, Claus Traulsen, and Reinhard von Hanxleden. 2008. Worst Case Reaction Time Analysis of Concurrent Reactive Programs. *Electronic Notes in Theoretical Computer Science* 203, 4 (June 2008), 65 – 79.

[18] Frédéric Boussinot. 1991. Reactive C: An Extension of C to Program Reactive Systems. 21, 4 (April 1991), 401 – 428.

[19] Frédéric Boussinot. 1993. *Reactive Shared Variables Based Systems.* Technical Report 1849. INRIA.

[20] Jens Brandt, Mike Gemunde, and Klaus Schneider. 2009. Desynchronizing Synchronous Programs by Modes. In *9th International Conference on Application of Concurrency to System Design (ACSD).* 32 – 41.

[21] Dominique Brière, Denis Ribot, Daniel Pilaud, and Jean-Louis Camus. 1994. Method and Specification Tools for Airbus On-board Systems. In *Avionics Conference and Exhibition.* ERA Technology, London, UK.

[22] Giuseppe Buja and Roberto Menis. 2012. Dependability and Functional Safety: Applications in Industrial Electronics Systems. *IEEE Industrial Electronics Magazine* 6, 3 (2012), 4 – 12.

[23] Sebastian Burckhardt and Daan Leijen. 2011. Semantics of Concurrent Revisions. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the joint European conferences on theory and practice of software* (Saarbrucken, Germany) *(ESOP/ETAPS).* Springer-Verlag, 116 – 135.

[24] Marcio Buss, Daniel Brand, Vugranam Sreedhar, and Stephen A. Edwards. 2010. A Novel Analysis Space for Pointer Analysis and Its Application for Bug Finding. *Sci. Comput. Program.* 75, 11 (Nov. 2010), 921 – 942.

[25] Paul Caspi, Alain Girault, and Daniel Pilaud. 1999. Automatic Distribution of Reactive Systems for Asynchronous Networks of Processors. *IEEE Transactions on Software Engineering* 25, 3 (May 1999), 416 – 427.

[26] Paul Caspi and Oded Maler. 2005. From Control Loops to Real-Time Programs. In *Handbook of Networked and Embedded Control Systems*, Dimitrios Hristu-Varsakelis and William S. Levine (Eds.). Birkhauser Boston, 395 – 418.

[27] Paul Caspi, Norman Scaife, Christos Sofronis, and Stavros Tripakis. 2008. Semantics-Preserving Multitask Implementation of Synchronous Programs. *ACM Trans. Embed. Comput. Syst.* 7, 2, Article 15 (Jan. 2008), 40 pages.

[28] Arthur Charlesworth. 2002. The undecidability of associativity and commutativity analysis. *ACM Trans. Program. Lang. Syst.* 24, 5 (2002), 554–565.

[29] Etienne Closse, Michel Poize, Jacques Pulou, Joseph Sifakis, Patrick Venter, and Daniel Weiland Sergio Yovine. 2001. TAXYS: A Tool for the Development and Verification of Real-Time Embedded Systems. In *Computer Aided Verification*, Gerard Berry, Hubert Comon, and Alain Finkel (Eds.). Lecture Notes in Computer Science, Vol. 2102. Springer Berlin / Heidelberg, 391 – 395.

[30] Albert Cohen, Léonard Gérard, and Marc Pouzet. 2012. Programming Parallelism with Futures in Lustre. In *Proceedings of the 10th ACM International Conference on Embedded Software* (Tampere, Finland) *(EMSOFT).* ACM, 197 – 206.

[31] Jean-Louis Colaço and Marc Pouzet. 2003. Clocks as First Class Abstract Types. In *Embedded Software*, Rajeev Alur and Insup Lee (Eds.). Lecture Notes in Computer Science, Vol. 2855. Springer Berlin Heidelberg, 134 – 155.

[32] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza Burguiere, Jan Reineke, Benoit Triquet, and Reinhard Wilhelm. 2010. Predictability Considerations in the Design of Multi-Core Embedded Systems. *Embedded Real Time Software and Systems (ERTS)* (2010).

[33] M. W. Sajeewa Dayaratne, Partha S. Roop, and Zoran Salcic. 2006. Direct Execution of Esterel Using Reactive Microprocessors. In *International Workshop on Synchronous Languages, Applications, and Programming (SLAP).* Vienna.

[34] Javier Diaz, Camelia Munoz-Caro, and Alfonso Nino. 2012. A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. *Parallel and Distributed Systems, IEEE Transactions on* 23, 8 (Aug. 2012), 1369 – 1386.

[35] Huping Ding, Yun Liang, and Tulika Mitra. 2013. Shared Cache Aware Task Mapping for WCRT Minimization. In *18th Asia and South Pacific Design Automation Conference* (Yokohama, Japan).

[36] Yiqiang Ding and Wei Zhang. 2011. Multicore-Aware Code Positioning to Improve Worst-Case Performance. *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (2011), 225 – 232.

[37] William R. Dunn. 2003. Designing Safety-Critical Computer Systems. *Computer* 36, 11 (2003), 40 – 46.

[38] Stephen A. Edwards, Sungjun Kim, Edward A. Lee, Isaac Liu, Hiren D. Patel, and Martin Schoeberl. 2009. A Disruptive Computer Design Idea: Architectures with Repeatable Timing. In *Proceedings of IEEE International Conference on Computer Design (ICCD)* (Lake Tahoe, CA). IEEE.

[39] Stephen A. Edwards and Edward A. Lee. 2007. The Case for the Precision Timed (PRET) Machine. In *Proceedings of the 44th Annual Design Automation Conference (DAC)* (San Diego, California). ACM, 264 – 265.

[40] Stephen A. Edwards and Jia Zeng. 2007. Code Generation in the Columbia Esterel Compiler. *EURASIP Journal on Embedded Systems* 2007 (2007).
[41] Cormac Flanagan and Shaz Qadeer. 2003. Types for Atomicity. In *Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (New Orleans, Louisiana, USA) *(TLDI)*. ACM, 1 – 12.
[42] Martin Gardner. 1970. Mathematical Games: The Fantastic Combinations of John Conway's new Solitaire Game "Life". *Scientific American* (Oct. 1970), 120 – 123.
[43] Gernot Gebhard, Christoph Cullmann, and Reinhold Heckmann. 2011. Software Structure and WCET Predictability. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems (OpenAccess Series in Informatics (OASIcs), Vol. 18)*, Philipp Lucas, Lothar Thiele, Benoit Triquet, Theo Ungerer, and Reinhard Wilhelm (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 1 – 10.
[44] Alain Girault. 2005. A Survey of Automatic Distribution Method for Synchronous Programs. In *International Workshop on Synchronous Languages, Applications and Programs, SLAP'05 (ENTCS)*, F. Maraninchi, M. Pouzet, and V. Roy (Eds.). Elsevier Science.
[45] Alain Girault, Nicolas Hili, Eric Jenn, and Eugene Yip. 2019. A Multi-Rate Precision Timed Programming Language for Multi-Cores. In *Forum for Specification and Design Languages (FDL)*. IEEE, 8 pages.
[46] Alain Girault, Xavier Nicollin, and Marc Pouzet. 2006. Automatic Rate Desynchronization of Embedded Reactive Programs. *ACM Trans. Embed. Comput. Syst.* 5, 3 (Aug. 2006), 687 – 717.
[47] GNU. 2013. GNU Radio. [Online] http://gnuradio.org.
[48] Thierry Goubier, Renaud Sirdey, Stephane Louise, and Vincent David. 2011. ΣC: A Programming Model and Language for Embedded Manycores. In *Algorithms and Architectures for Parallel Processing*, Yang Xiang, Alfredo Cuzzocrea, Michael Hobbs, and Wanlei Zhou (Eds.). Lecture Notes in Computer Science, Vol. 7016. Springer Berlin Heidelberg, 385 – 394.
[49] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. 1991. Programming Real-Time Applications with SIGNAL. *Proc. IEEE* 79, 9 (Sept. 1991), 1321 – 1336.
[50] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The Synchronous Data Flow Programming Language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305 – 1320.
[51] Ben Hardekopf and Calvin Lin. 2011. Flow-Sensitive Pointer Analysis for Millions of Lines of Code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 289 – 298.
[52] David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8, 3 (1987), 231 – 274.
[53] D. Harel and A. Pnueli. 1985. On the Development of Reactive Systems. In *Logic and Models of Concurrent Systems, NATO*. Springer-Verlag.
[54] Les Hatton. 2004. Safer Language Subsets: An Overview and a Case History, MISRA C. *Information and Software Technology* 46, 7 (2004), 465 – 472.
[55] Nicolas Hili, Alain Girault, and Eric Jenn. 2019. Worst-Case Reaction Time Optimization on Deterministic Multi-Core Architectures with Synchronous Languages. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE.
[56] Gerard J. Holzmann. 2006. The Power of 10: Rules for Developing Safety-Critical Code. *IEEE Computer* 39, 6 (2006), 95 – 97.
[57] Intel. 2012. Intel Cilk Plus. [Online] http://software.intel.com/en-us/intel-cilk-plus.
[58] Intel. 2012. Intel Thread Building Blocks. [Online] http://threadingbuildingblocks.org.
[59] Intel. 2022. Intel® VTune™ Profiler. [Online] https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.
[60] International Electrotechnical Commission. 2010. IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems. Standard.
[61] ISO/IEC JTC1/SC22/WG14. 2011. ISO/IEC 9899:2011.
[62] ISO/TC22/SC3. 2011. ISO 26262-1:2011. Standard.
[63] Jet Propulsion Laboratory. 2009. JPL Institutional Coding Standard for the C Programming Language. Standard 1.0. [Online] http://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_C.pdf.
[64] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC)*. USENIX Association, 275 – 288.
[65] Lei Ju, BachKhoa Huynh, Abhik Roychoudhury, and Samarjit Chakraborty. 2012. Performance Debugging of Esterel Specifications. *Real-Time Systems* 48, 5 (2012), 570 – 600.
[66] Lei Ju, Bach Khoa Huynh, Abhik Roychoudhury, and Samarjit Chakraborty. 2010. Timing Analysis of Esterel Programs on General-Purpose Multiprocessors. In *Proceedings of the 47th Design Automation Conference (DAC)*

(Anaheim, California). ACM, 48 – 51.
[67] Andrey Karpov. 2011. Parallel Lint. [Online] http://software.intel.com/en-us/articles/parallel-lint.
[68] Daniel Kastner, Marc Schlickling, Markus Pister, Christoph Cullmann, Gernot Gebhard, Reinhold Heckmann, and Christian Ferdinand. 2012. Meeting Real-Time Requirements with Multi-Core Processors. In *Computer Safety, Reliability, and Security*, Frank Ortmeier and Peter Daniel (Eds.). Lecture Notes in Computer Science, Vol. 7613. Springer Berlin Heidelberg, 117 – 131.
[69] Christoph W. Kessler and Helmut Seidl. 1999. ForkLight: A Control-Synchronous Parallel Programming Language. In *High-Performance Computing and Networking*, Peter Sloot, Marian Bubak, Alfons Hoekstra, and Bob Hertzberger (Eds.). Lecture Notes in Computer Science, Vol. 1593. Springer Berlin Heidelberg, 525 – 534.
[70] Andrew Koenig. 1988. *C Traps and Pitfalls*. Addison-Wesley. I–XI, 1–147 pages.
[71] Matthew Kuo, Roopak Sinha, and Partha S. Roop. 2011. Efficient WCRT Analysis of Synchronous Programs using Reachability. In *48th ACM/EDAC/IEEE Design Automation Conference (DAC)* (San Diego, USA). 480 – 485.
[72] Luciano Lavagno and Ellen Sentovich. 1999. ECL: A Specification Environment for System-Level Design. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference (DAC)* (New Orleans, USA).
[73] Edward A. Lee. 2006. The Problem with Threads. *Computer* 39 (2006), 33 – 42.
[74] Edward A. Lee. 2009. Computing Needs Time. *Commun. ACM* 52, 5 (May 2009), 70 – 79.
[75] Markus Levy and Thomas M. Conte. 2009. Embedded Multicore Processors and Systems. *IEEE Micro* 29, 3 (2009), 7 – 9.
[76] Xin Li and Reinhard von Hanxleden. 2012. Multithreaded Reactive Programming - the Kiel Esterel Processor. *IEEE Trans. Comput.* 61, 3 (March 2012), 337 – 349.
[77] Yun Liang, Huping Ding, Tulika Mitra, Abhik Roychoudhury, Yan Li, and Vivy Suhendra. 2012. Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores. *Real-Time Systems* 48, 6 (2012), 638 – 680.
[78] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. 2021. Toward a Lingua Franca for Deterministic Concurrent Systems. *ACM Trans. Embed. Comput. Syst.* 20, 4 (2021), 36:1–36:27.
[79] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. ACM, 329 – 339.
[80] Thomas Lundqvist and Per Stenstrom. 1999. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*. 12 – 21.
[81] Florence Maraninchi. 1992. Operational and Compositional Semantics of Synchronous Automaton Compositions. In *CONCUR*, W.R. Cleaveland (Ed.). Lecture Notes in Computer Science, Vol. 630. Springer Berlin Heidelberg, 550 – 564.
[82] Charles E. McDowell and David P. Helmbold. 1989. Debugging Concurrent Programs. *ACM Comput. Surv.* 21, 4 (Dec. 1989), 593 – 622.
[83] Michael Mendler, Reinhard von Hanxleden, and Claus Traulsen. 2009. WCRT Algebra and Interfaces for Esterel-Style Synchronous Processing. In *Proceedings of the Design, Automation and Test in Europe (DATE)*. Nice, France.
[84] Message Passing Interface Forum. 2012. MPI: A Message-Passing Interface Standard. Standard 3.0.
[85] Motor Industry Software Reliability Association. 2013. MISRA-C: 2012: Guidelines for the Use of the C Language in Critical Systems. , 226 pages. Standard.
[86] Marco Di Natale, Liangpeng Guo, Haibo Zeng, and Alberto Sangiovanni-Vincentelli. 2010. Synthesis of Multitask Implementations of Simulink Models With Minimum Delays. *IEEE Transactions on Industrial Informatics* 6, 4 (2010), 637 – 651.
[87] Marco Di Natale and Haibo Zeng. 2012. Task Implementation of Synchronous Finite State Machines. In *Design, Automation Test in Europe Conference Exhibition (DATE)*. 206 – 211.
[88] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. 2006. PapaBench: A Free Real-Time Benchmark. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*.
[89] Michael Norrish. 1999. Deterministic Expressions in C. In *Proceedings of the 8th European Symposium on Programming Languages and Systems (ESOP)*. Springer-Verlag, 147 – 161.
[90] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: Efficient Deterministic Multithreading in Software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS)*. ACM, 97 – 108.
[91] OpenMP Architecture Review Board. 2013. OpenMP Application Program Interface. Standard 4.0.
[92] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. 2011. Multi-task Implementation of Multi-periodic Synchronous Programs. *Discrete Event Dynamic Systems* 21, 3 (2011), 307 – 338.
[93] Marco Paolieri and Riccardo Mariani. 2011. Towards Functional-Safe Timing-Dependable Real-Time Architectures. In *17th IEEE International On-Line Testing Symposium (IOLTS)* (Athens, Greece). 31 – 36.
[94] Virginia Papailiopoulou, Dumitru Potop-Butucaru, Yves Sorel, Robert De Simone, Loic Besnard, and Jean-Pierre Talpin. 2011. *From Concurrent Multi-Clock Programs to Concurrent Multi-Threaded Implementations*. Rapport de

recherche RR-7577. INRIA.
[95] David A. Patterson and John L. Hennessy. 1990. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[96] Sascha Plazar, Paul Lokuciejewski, and Peter Marwedel. 2008. A Retargetable Framework for Multi-Objective WCET-aware High-level Compiler Optimizations. In *Proceedings of The 29th IEEE Real-Time Systems Symposium (RTSS) WiP.* Barcelona, Spain, 49 – 52.
[97] Gordon D. Plotkin. 1981. *A Structural Approach to Operational Semantics.* Technical Report DAIMI FN-19. Aarhus University, Computer Science Department.
[98] Antoniu Pop and Albert Cohen. 2011. A Stream-Computing Extension to OpenMP. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers* (Heraklion, Greece) *(HiPEAC).* ACM, 5 – 14.
[99] Dumitru Potop-Butucaru, Akramul Azim, and Sebastian Fischmeister. 2010. Semantics-Preserving Implementation of Synchronous Specifications Over Dynamic TDMA Distributed Architectures. In *International Conference on Embedded Software (EMSOFT).* ACM, 199 – 208.
[100] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. 2007. *Compiling Esterel.* Springer. I–XXI, 1–335 pages.
[101] Dumitru Potop-Butucaru and Isabelle Puaut. 2013. Integrated Worst-Case Execution Time Estimation of Multicore Applications. In *13th International Workshop on Worst-Case Execution Time Analysis (OpenAccess Series in Informatics (OASIcs), Vol. 30).* Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 21 – 31.
[102] Radio Technical Commission for Aeronautics. 1992. Software Considerations in Airborne Systems and Equipment Certification. Standard DO-178B.
[103] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2010. Efficient Data Race Detection for Async-Finish Parallelism. In *Proceedings of the 1st International Conference on Runtime Verification* (St. Julians, Malta) *(RV).* Springer-Verlag, 368 – 383.
[104] Pascal Raymond, Claire Maiza, Catherine Parent-Vigouroux, and Fabienne Carrier. 2013. Timing Analysis Enhancement for Synchronous Program. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems* (Sophia Antipolis, France) *(RTNS).* ACM, 141 – 150.
[105] Zoran A. Salcic, Dong Hui, Partha S. Roop, and Morteza Biglari-Abhari. 2006. HiDRA - A Reactive Multiprocessor Architecture for Heterogeneous Embedded Systems. *Microprocessors and Microsystems* 30, 2 (2006), 72 – 85.
[106] Andreas Schranzhofer, Rodolfo Pellizzoni, Jian-Jia Chen, Lothar Thiele, and Marco Caccamo. 2011. Timing Analysis for Resource Access Interference on Adaptive Resource Arbiters. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).* 213–222.
[107] Jacob T. Schwartz. 1980. Ultracomputers. *ACM Trans. Program. Lang. Syst.* 2, 4 (Oct. 1980), 484 – 521.
[108] H. Shah, A. Raabe, and A. Knoll. 2011. Priority Division: A High-Speed Shared-Memory Bus Arbitration with Bounded Latency. In *Design, Automation Test in Europe Conference Exhibition (DATE).* 1 – 4.
[109] Steven Smyth, Christian Motika, Karsten Rathlev, Reinhard von Hanxleden, and Michael Mendler. 2018. SCEst: Sequentially Constructive Esterel. *ACM Trans. Embed. Comput. Syst.* 17, 2 (2018), 33:1–33:26.
[110] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. 2019. Practical Causality Handling for Synchronous Languages. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, Jürgen Teich and Franco Fummi (Eds.). IEEE, 1281–1284. https://doi.org/10.23919/DATE.2019.8715081
[111] Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Victor Jégu, Guillaume Borios, and Reinhold Heckmann. 2005. Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation. In *International Workshop on Worst-case Execution Time, WCET'05.* Mallorca, Spain, 21 – 24.
[112] James A. Storer and Thomas G. Szymanski. 1982. Data Compression via Textual Substitution. *J. ACM* 29, 4 (Oct. 1982), 928 – 951.
[113] Olivier Tardieu. 2007. A Deterministic Logical Semantics for Pure Esterel. *ACM Trans. Program. Lang. Syst.* 29, 2, Article 8 (April 2007).
[114] Olivier Tardieu and Stephen A. Edwards. 2006. Scheduling-Independent Threads and Exceptions in SHIM. In *Proceedings of the 6th ACM/IEEE International conference on Embedded software* (Seoul, Korea) *(EMSOFT).* ACM, 142 – 151.
[115] The IEEE and The Open Group. 2008. POSIX.1-2008. Standard Issue 7.
[116] The UPC Consortium. 2013. UPC Language Specifications. Standard 1.3.
[117] Theo Ungerer, Francisco J. Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quinones, Mike Gerdes, Marco Paolieri, Julian Wolf, Hugues Casse, Sascha Uhrig, Irakli Guliashvili, Michael Houston, Florian Kluge, Stefan Metzlaff, and Jorg Mische. 2010. MERASA: Multicore Execution of Hard Real-Time Applications Supporting Analyzability. *IEEE Micro* 30, 5 (Sept. 2010), 66 – 75.

[118] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (Aug. 1990), 103 – 111.
[119] Nalini Vasudevan and Stephen A. Edwards. 2009. Celling SHIM: Compiling Deterministic Concurrency to a Heterogeneous Multicore. In *Proceedings of the ACM symposium on Applied Computing* (Honolulu, Hawaii) *(SAC)*. ACM, 1626 – 1631.
[120] Reinhard von Hanxleden. 2009. SyncCharts in C - A Proposal for Light-Weight, Deterministic Concurrency. In *Proceedings of the 9th ACM/IEEE International conference on Embedded software* (Grenoble, France). 225 – 234.
[121] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. 2014. SCCharts: sequentially constructive statecharts for safety-critical applications: HW/SW-synthesis for a conservative extension of synchronous statecharts. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 372–383. https://doi.org/10.1145/2594291.2594310
[122] Reinhard von Hanxleden, Michael Mendler, Joaquin Aguado, Bjorn Duderstadt, Insa Fuhrmann, Christain Motika, Stephen Mercer, and Owen O'Brien. 2013. Sequentially Constructive Concurrency: A Conservative Extension of the Synchronous Model of Computation. In *Design, Automation and Test in Europe (DATE)* (Grenoble, France).
[123] Jia Jie Wang, Partha S. Roop, and Sidharta Andalam. 2013. ILPc: A Novel Approach for Scalable Timing Analysis of Synchronous Programs. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. 1 – 10.
[124] Jack Whitham. 2012. Scratchpad Memory Management Unit. [Online] http://www.jwhitham.org/c/smmu.html.
[125] Jack Whitham and Neil Audsley. 2009. Implementing Time-Predictable Load and Store Operations. In *Proceedings of the 7th ACM International Conference on Embedded software (EMSOFT)* (Grenoble, France). ACM, 265 – 274.
[126] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.* 7, 3 (2008), 1 – 53.
[127] Reinhard Wilhelm and Daniel Grund. 2014. Computation Takes Time, but How Much? *Commun. ACM* 57, 2 (Feb. 2014), 94 – 103.
[128] Xilinx. 2012. MicroBlaze Processor Reference Guide. [Online] http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/mb_ref_guide.pdf.
[129] Eugene Yip, Alain Girault, Partha S. Roop, and Morteza Biglari-Abhari. 2016. The ForeC Synchronous Deterministic Parallel Programming Language for Multicores. In *International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)* (Lyon, France). IEEE.
[130] Eugene Yip, Partha S. Roop, Morteza Biglari-Abhari, and Alain Girault. 2013. Programming and Timing Analysis of Parallel Programs on Multicores. In *13th International Conference on Application of Concurrency to System Design (ACSD)* (Barcelona, Spain). 10 pages.
[131] Li Hsien Yoong, Partha S. Roop, Zoran Salcic, and Flavius Gruian. 2006. Compiling Esterel for Distributed Execution. In *International Workshop on Synchronous Languages, Applications, and Programming (SLAP)*. Vienna.
[132] Simon Yuan. 2013. *Architectures Specific Compilation for Efficient Execution of Esterel.* Ph. D. Dissertation. Electrical and Electronic Engineering, The University of Auckland.
[133] Simon Yuan, Sidharta Andalam, Li Hsien Yoong, Partha S. Roop, and Zoran Salcic. 2009. STARPro - A New Multithreaded Direct Execution Platform for Esterel. *Electron. Notes Theor. Comput. Sci.* 238, 1 (June 2009), 37 – 55.
[134] Simon Yuan, Li Hsien Yoong, and Partha S. Roop. 2011. Compiling Esterel for Multi-core Execution. In *14th Euromicro Conference on Digital System Design (DSD)*. 727 – 735.
[135] Yu Zhang and Bryan Ford. 2011. A Virtual Memory Foundation for Scalable Deterministic Parallelism. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (Shanghai, China) *(APSys)*. ACM, 5 pages.
[136] Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. 2014. FlexPRET: A Processor Platform for Mixed-Criticality Systems. In *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS)*.

A PROOFS

A.1 Reactivity of ForeC Kernel Programs

Definition 3. *A program $t : f$ is* **reactive** *if, in any state S, for any input configuration I, there exists* at least *one transition (i.e., the program never deadlocks):*

$$\forall S, I: \quad \exists S', f', k \quad \textit{such that} \quad \langle S\rangle\ t : f \xrightarrow[I]{k} \langle S'\rangle\ t : f'$$

Theorem 4. *Let $t : f$ be a ForeC kernel program that is correct with respect to syntax and type checking. Then, any such $t : f$ is* **reactive**.

Proof. The proof can be shown by structural induction on $t : f$.

Base cases: The (nop), (copy), (pause), (status), (assign-shared), (assign-private), (if-then), (if-else), (loop-then), and (loop-else) rules imply that the following kernel constructs have at least one transition:

$$\langle S\rangle\ t : \texttt{nop} \xrightarrow[I]{0} \langle S\rangle\ t :$$

$$\langle S\rangle\ t : \texttt{copy} \xrightarrow[I]{0} \langle S'\rangle\ t :$$

$$\langle S\rangle\ t : \texttt{pause} \xrightarrow[I]{1} \langle S\rangle\ t : \texttt{copy}$$

$$\langle S\rangle\ t : \texttt{status}(a, exp) \xrightarrow[I]{0} \langle S'\rangle\ t :$$

$$\langle S\rangle\ t : var = exp \xrightarrow[I]{0} \langle S'\rangle\ t :$$

$$\langle S\rangle\ t : \texttt{if}\ (exp)\ f_1\ \texttt{else}\ f_2 \xrightarrow[I]{\perp} \langle S\rangle\ t : f_1 \quad \text{or} \quad \langle S\rangle\ t : \texttt{if}\ (exp)\ f_1\ \texttt{else}\ f_2 \xrightarrow[I]{\perp} \langle S\rangle\ t : f_2$$

$$\langle S\rangle\ t : \texttt{while}\ (exp)\ f \xrightarrow[I]{\perp} \langle S\rangle\ t : f;\ \texttt{while}\ (exp)\ f \quad \text{or} \quad \langle S\rangle\ t : \texttt{while}\ (exp)\ f \xrightarrow[I]{0} \langle S\rangle\ t :$$

Induction step: The sequence operator (;), `abort`, and `par` kernel statements allow the composition of kernel constructs. Let $t_1 : f_1$ and $t_2 : f_2$ be arbitrary compositions of kernel constructs. We assume the induction hypotheses that they each have at least one transition:

$$\exists S_1', S_2', f_1', f_2', k_1, k_2 \quad \text{such that} \quad \langle S_1\rangle\ t_1 : f_1 \xrightarrow[I]{k_1} \langle S_1'\rangle\ t_1 : f_1' \tag{H1}$$

$$\text{and} \quad \langle S_2\rangle\ t_2 : f_2 \xrightarrow[I]{k_2} \langle S_2'\rangle\ t_2 : f_2' \tag{H2}$$

Next, we show that the remaining sequence operator (;), `abort`, and `par` kernel statements have at least one transition.

(1) Consider $t_1 : f_1; f_2$. Thanks to (H1) and (H2), the table below shows that at least one sequence rule can be applied to all possible completion codes k_1 of the first program fragment f_1. Note that the sequence rules do not consider the completion code k_2 of the second program fragment f_2:

k_1		
0	**1**	$\perp$
(seq-right)	(seq-left)	

If $k_1 = 0$ and the premise is true by (H1), then from the (seq-right) rule we have:

$$\text{(seq-right)}\ \frac{\langle S_1\rangle\ t_1 : f_1 \xrightarrow[I]{k_1=0} \langle S_1'\rangle\ t_1 :}{\langle S_1\rangle\ t_1 : f_1; f_2 \xrightarrow[I]{\bot} \langle S_1'\rangle\ t_1 : f_2}$$

If $k_1 \in \{1, \bot\}$ and the premise is true by (H1), then from the (seq-left) rule we have:

$$\text{(seq-left)}\ \frac{\langle S_1\rangle\ t_1 : f_1 \xrightarrow[I]{k_1 \in \{1,\bot\}} \langle S_1'\rangle\ t_1 : f_1'}{\langle S_1\rangle\ t_1 : f_1; f_2 \xrightarrow[I]{k_1} \langle S_1'\rangle\ t_1 : f_1'; f_2}$$

Thus, any sequential composition of reactive programs has at least one transition and is, therefore, reactive.

(2) Consider t_1 : weak? abort(a_1, f_1). Thanks to (H1) and (H2), the table below shows that at least one abort rule can be applied to every combination of k_1 and preemption status $A[a_1]$:

		Strong abort, k_1			**Weak abort, k_1**		
		0	**1**	**⊥**	**0**	**1**	**⊥**
$A[a_1]$	**= 0**	(abort-2)	(abort-1)		(abort-2)	(abort-1)	
	≠ 0	(abort-6)			(abort-3)	(abort-5)	(abort-4)

For example, if $k_1 = 0$ and $A[a_1] = 0$ and the premise is true by (H1), then from the (abort-2) rule we have:

$$\text{(abort-2)}\ \frac{\langle S_1\rangle\ t_1 : f_1 \xrightarrow[I]{k_1=0} \langle S_1'\rangle\ t_1 :}{\langle S_1\rangle\ t_1 : \text{weak? abort}(a_1, f_1) \xrightarrow[I]{0} \langle S_1'\rangle\ t_1 :}\ (A[a_1] = 0)$$

The other cases are similar. Thus, any preemptive composition of reactive programs has at least one transition and is, therefore, reactive.

(3) Consider t : par$(t_1 : f_1,\ t_2 : f_2)$. Thanks to (H1) and (H2), the table below shows that at least one par rule can be applied to every combination of k_1 and k_2:

		k_2		
		0	**1**	**⊥**
k_1	**0**	(par-5)	(par-6)	(par-2)
	1	(par-7)	(par-4)	
	⊥	(par-3)		(par-1)

For example, if $k_1 = 0$ and $k_2 = 0$ and the premise is true by (H1) and (H2), then from the (par-5) rule we have:

$$\text{(par-5)}\ \frac{\langle S\rangle\ t_1 : f_1 \xrightarrow[I]{k_1=0} \langle S_1'\rangle\ t_1 : \qquad \langle S\rangle\ t_2 : f_2 \xrightarrow[I]{k_2=0} \langle S_2'\rangle\ t_2 :}{\langle S\rangle\ t : \text{par}(t_1 : f_1,\ t_2 : f_2) \xrightarrow[I]{\bot} \langle S''\rangle\ t : \text{copy}}$$

The other cases are similar. Thus, any parallel composition of reactive programs has at least one transition and is, therefore, reactive.

□

A.2 Parallel Execution and Shared Variable Resynchronization

Before proving that all ForeC kernel programs are deterministic, we prove that threads execute their local ticks in isolation, and that the aggregation of states and the semantic function Combine are both deterministic. These are captured by Lemmas 8, 10, and 11 below, with the assumption that all the combine functions are deterministic. We further prove in Lemmas 13 and 15 that the associative and commutative property of the `par` statement is dependent on combine functions having the same respective properties.

Definition 5. *A* **combine function** $cf : D \times D \rightarrow D$ *is any C function with two input values* v_1 *and* v_2 *of identical type, which returns a value of the same type.*

Definition 6. *A combine function* cf *is* **deterministic** *if there exists only one value that can be returned for a given set of input values:*

$$\forall v_1, v_2 \in D \times D, \exists! v_3 \in D \text{ such that } cf(v_1, v_2) = v_3$$

From now on, we assume that every combine function cf always terminates and is deterministic, regardless of the program's current state.

Assumption 7.

$$\forall S, S', I, t, v_1, v_2, cf: \quad \textsc{Eval}(S.E, I, t, cf(v_1, v_2)) = \textsc{Eval}(S'.E, I, t, cf(v_1, v_2))$$

Lemma 8. *Every thread executes their local ticks in* **isolation***, i.e., when a thread executes its local tick, it cannot modify the variables of any other thread and it cannot observe the modifications of any other thread.*

Proof. Recall from Section 3.2 that $E[\mathcal{G}]$ stores all the output, shared, and private variables of the program, while $E[t]$ stores the copies of the shared variables of thread t. The requirement for a private variable not to appear in the body of more than one thread can be ensured via static type checking. Recall that threads always create copies of shared variables at the start of their local tick via the `copy` kernel construct and the Copy semantic function (Section 3.3.3).

Let t be a thread that writes value v to variable var during its local tick. This is only possible via the (assign-private) and (assign-shared) rules:

Private variable: In the (assign-private) rule, v is assigned directly to the private variable, i.e., $E[\mathcal{G}][var]$. Static type checking ensures that no other thread can also write to variable var.

Shared variable: In the (assign-shared) rule, v is only assigned to the thread's copy of the shared variable, i.e., $E[t][var]$. It is not possible to write to another thread's copy.

Let t' be a thread that reads value v' from variable var' during its local tick. This is only possible via the (status), (assign-shared), (assign-private), (if-then), (if-else), (loop-then), and (loop-else) rules, which rely on the GetVal semantic function (Section 3.3.2). The value that GetVal returns depends on whether var' is a shared or private variable (see Algorithm 1):

Shared variable: In Algorithm 1, the value is returned from the thread's copy of the shared variable, i.e., $E[t'][var']$. It is not possible to read from another thread's copy.

Private variable: In Algorithm 1, the value is returned directly from the private variable, i.e., $E[\mathcal{G}][var']$, because the Copy semantic function (Section 3.3.3) never makes copies of private variables.

In conclusion, each thread executes its local tick in isolation because (1) no thread can access the private variables belonging to other threads, and (2) a thread can only access (read and write) its own copies of the shared variables. □

Definition 9. *The aggregation of states S' and S'' into a single state S^A is* **deterministic** *if there exists only one possible S^A. Let $\odot$ be the binary operator denoting the aggregation of states, then*

$$\forall S', S'', \exists! S^A \text{ such that } S^A = S' \odot S''$$

Lemma 10. *Let $t_1{:}f_1$ and $t_2{:}f_2$ be two ForeC kernel threads that are correct with respect to syntax and type checking. Let the initial state be $S = \langle E, A\rangle$ and the resulting states be $S' = \langle E', A'\rangle$ and $S'' = \langle E'', A''\rangle$ such that*

$$\langle S\rangle\ t_1 : f_1 \xrightarrow[I]{k'} \langle S'\rangle\ t_1 : f_1' \quad \text{and} \quad \langle S\rangle\ t_2 : f_2 \xrightarrow[I]{k''} \langle S''\rangle\ t_2 : f_2''$$

The aggregation of states $S^A = S' \odot S''$ is deterministic.

Recall from Section 3.4.10 that the aggregation $S^A = \langle E^A, A^A\rangle$ is defined as:

$$E^A = (E' \setminus (E' \cap E)) \cup (E'' \setminus (E'' \cap E)) \cup (E' \cap E'') \quad \text{and}$$
$$A^A = (A' \setminus (A' \cap A)) \cup (A'' \setminus (A'' \cap A)) \cup (A' \cap A'')$$

Proof. We begin by proving that the aggregation of environments E' and E'' is deterministic. The intersection $(E' \cap E'')$ contains all variables that have not changed from E. The two complements are the subsets $E_1 = (E' \setminus (E' \cap E))$ and $E_2 = (E'' \setminus (E'' \cap E))$ that contain all the variables that have changed from E. To prove that the aggregation is deterministic, it is sufficient to prove that $E_1 \cap E_2 = \emptyset$:

Private variables of threads t_1 and t_2 that have changed are in $E_1[\mathcal{G}]$ and $E_2[\mathcal{G}]$, respectively. Thanks to Lemma 8 we know that t_1 and t_2 cannot access the same private variables, thus, $E_1[\mathcal{G}] \cap E_2[\mathcal{G}] = \emptyset$.

Copies of shared variables for threads t_1 and t_2 belong in $E_1[t_1]$ and $E_2[t_2]$, respectively. Thanks to Lemma 8 we know that t_1 and t_2 can only access their own copies of shared variables and never the copies of other threads, thus, $\forall t \in T : E_1[t] \cap E_2[t] = \emptyset$.

We now prove that the aggregation of preemption statuses A' and A'' is deterministic. The intersection $(A' \cap A)$ contains all preemption statuses that have not changed from A. The two complements are the subsets $A_1 = (A' \setminus (A' \cap A))$ and $A_2 = (A'' \setminus (A'' \cap A))$ that contain all the preemption statuses that have changed from A. To prove that the aggregation is deterministic, it is sufficient to prove that $A_1 \cap A_2 = \emptyset$: A thread can only change the preemption status of an `abort` identifier by executing a `status` statement (see the (status) rule). By construction, a unique `abort` identifier is passed into every `status` statement, thus, $A_1 \cap A_2 = \emptyset$.

In conclusion the aggregation of S' and S'' into S^A is deterministic because only the unions of disjoint sets are taken. □

Lemma 11. *The semantic function Combine(E, t_1, t_2, t_0) is deterministic: there exists only one environment that it can return for a given set of inputs.*

Proof. Algorithm 3 of the semantic function Combine intializes all its local variables (`preVal`, T, cf, v), is side-effect-free, and uses only deterministic instructions. In particular, line 11 in Algorithm 3 is deterministic thanks to Assumption 7 (all combine functions cf are deterministic), and because values are passed into cf in a fixed order. Hence, the semantic function Combine is deterministic. □

Definition 12. *The* par *statement is associative if, in any state S, for any thread bodies f_a, f_b, and f_c, the following holds:*

$$\langle S\rangle\ t_0 : \mathsf{par}(t_1 : f_a,\ t_2 : \mathsf{par}(t_3 : f_b,\ t_4 : f_c)) \xrightarrow[I]{k'} \langle S'\rangle\ t_0 : \mathsf{par}(t_1 : f'_a,\ t_2 : \mathsf{par}(t_3 : f'_b,\ t_4 : f'_c)),$$

$$\langle S\rangle\ t_0 : \mathsf{par}(t_1 : \mathsf{par}(t_2 : f_a,\ t_3 : f_b),\ t_4 : f_c) \xrightarrow[I]{k'} \langle S'\rangle\ t_0 : \mathsf{par}(t_1 : \mathsf{par}(t_2 : f'_a,\ t_3 : f'_b),\ t_4 : f'_c).$$

Lemma 13. *The* $\mathsf{par}(t_1{:}f_1,\ t_2{:}f_2)$ *statement is associative if and only if the combine functions cf involved in combining shared variables are also associative.*

Proof. The (par-1) and (par-4)–(par-7) rules require the states of threads t_1 and t_2 to be aggregated before the shared variables are combined. From Lemma 10, we know that the aggregation is simply the union of disjunct changes to the states, which does not affect the associativity of the par.

By inspecting the (par-1) and (par-4)–(par-7) rules, the semantic function Combine is only applied to threads t_1 and t_2's copies of shared variables in the aggregated environment E^A. In Algorithm 3, the combine policy is applied to each copy individually (lines 4–8), and the combine function *cf* is invoked (line 11) if the combine policy allows both copies to be combined. Because the copies are passed to *cf* in the same textual order that their threads appear in the par statement, the par statement is associative if and only if *cf* is associative. □

Definition 14. *The* par *statement is commutative if, in any state S, for any thread bodies f_a and f_b, the following holds:*

$$\langle S\rangle\ t_0 : \mathsf{par}(t_1 : f_a,\ t_2 : f_b) \xrightarrow[I]{k'} \langle S'\rangle\ t_0 : \mathsf{par}(t_1 : f'_a,\ t_2 : f'_b),$$

$$\langle S\rangle\ t_0 : \mathsf{par}(t_2 : f_b,\ t_1 : f_a) \xrightarrow[I]{k'} \langle S'\rangle\ t_0 : \mathsf{par}(t_2 : f'_b,\ t_1 : f'_a).$$

Lemma 15. *The* $\mathsf{par}(t_1{:}f_1,\ t_2{:}f_2)$ *statement is commutative if and only if the combine functions cf involved in combining shared variables are also commutative.*

Proof. Similar to the proof of associativity (Lemma 13), we know that the aggregation of thread states is simply the union of disjunct changes, which does not affect the commutativity of the par. In Algorithm 3 for the semantic function Combine, the combine function *cf* is invoked (line 11) with both copies passed in the same textual order as their threads in the par statement. Hence, the par statement is commutative if and only if *cf* is commutative. □

A.3 Determinism of ForeC Kernel Programs

Equipped with the proof that shared variables are always combined deterministically (Lemma 11), we are ready to prove that ForeC kernel programs are deterministic.

Definition 16. *A program $t : f$ is* **deterministic** *if, in any state S, for any input configuration I, there exists* at most *one transition such that:*

$$\forall S, I: \quad \textit{if} \quad \langle S\rangle\ t : f \xrightarrow[I]{k'} \langle S'\rangle\ t : f' \quad \textit{and} \quad \langle S\rangle\ t : f \xrightarrow[I]{k''} \langle S''\rangle\ t : f''$$
$$\textit{then} \quad S' = S'',\ f' = f'',\ k' = k''$$

Theorem 17. *Let Assumption 7 be satisfied and let $t : f$ be a ForeC kernel program that is correct with respect to syntax and type checking. Then, any such $t : f$ is* **deterministic**.

Proof. The proof can be shown by structural induction on $t : f$.

Base cases: The (nop), (copy), (pause), and (status) rules imply that the following kernel statements have at most one transition:

$$\langle S\rangle\ t : \texttt{nop} \xrightarrow[I]{0} \langle S\rangle\ t :$$

$$\langle S\rangle\ t : \texttt{copy} \xrightarrow[I]{0} \langle S'\rangle\ t : \texttt{nop}$$

$$\langle S\rangle\ t : \texttt{pause} \xrightarrow[I]{1} \langle S\rangle\ t : \texttt{copy}$$

$$\langle S\rangle\ t : \texttt{status}(a, exp) \xrightarrow[I]{0} \langle S'\rangle\ t : \texttt{nop}$$

The assignment, `if–else`, and `while` kernel constructs are each described by a pair of rewrite rules with complementary premises that do not depend on other transitions: respectively (assign-shared) and (assign-private), (if-then) and (if-else), and (loop-then) and (loop-else). This implies that these kernel constructs have at most one transition:

$$\text{if } var \in \textsc{GetShared}(t) \text{ then} \quad \langle S\rangle\ t : var = exp \xrightarrow[I]{0} \langle S'\rangle\ t :$$

$$\text{otherwise} \quad \langle S\rangle\ t : var = exp \xrightarrow[I]{0} \langle S''\rangle\ t :$$

$$\text{if } \textsc{Eval}(S.E, I, t, exp) \neq 0 \text{ then} \quad \langle S\rangle\ t : \texttt{if}\ (exp)\ f_1\ \texttt{else}\ f_2 \xrightarrow[I]{\perp} \langle S\rangle\ t : f_1$$

$$\text{otherwise} \quad \langle S\rangle\ t : \texttt{if}\ (exp)\ f_1\ \texttt{else}\ f_2 \xrightarrow[I]{\perp} \langle S\rangle\ t : f_2$$

$$\text{if } \textsc{Eval}(S.E, I, t, exp) \neq 0 \text{ then} \quad \langle S\rangle\ t : \texttt{while}\ (exp)\ f \xrightarrow[I]{\perp} \langle S\rangle\ t : f;\ \texttt{while}\ (exp)\ f$$

$$\text{otherwise} \quad \langle S\rangle\ t : \texttt{while}\ (exp)\ f \xrightarrow[I]{0} \langle S\rangle\ t :$$

Among the rewrite rules considered in the base case, only the (copy), (status), (assign-shared), and (assign-private) rules make direct changes to the state S. The (copy) rule changes only the store $E[t]$ of the executing thread t. This can be verified by inspecting Algorithm 2 of the semantic function Copy. By construction, each `status` statement has a unique `abort` identifier a. Thus, the (status) rule never changes the status of an `abort` identifier more than once. The (assign-shared) rule changes only the store $E[t]$ of the executing thread t. The (assign-private) rule changes only the private variables in $E[\mathcal{G}]$ of the executing thread.

Induction step: The sequence operator (`;`), `abort`, and `par` kernel statements allow the composition of kernel constructs. For some $t_1 : f_1$ and $t_2 : f_2$ that are arbitrary compositions of kernel constructs, assume the induction hypotheses that they each have at most one transition:

$$\begin{aligned} \text{If} \quad \exists S_1', S_1'', f_1', f_1'', k_1', k_1'' \quad \text{such that} \quad & \langle S_1\rangle\ t_1 : f_1 \xrightarrow[I]{k_1'} \langle S_1'\rangle\ t_1 : f_1' \\ \text{and} \quad & \langle S_1\rangle\ t_1 : f_1 \xrightarrow[I]{k_1''} \langle S_1''\rangle\ t_1 : f_1'' \\ \text{then} \quad & S_1' = S_1'',\ f_1' = f_1'',\ k_1' = k_1'' \end{aligned} \tag{H3}$$

$$\begin{aligned} \text{If} \quad \exists S_2', S_2'', f_2', f_2'', k_2', k_2'' \quad \text{such that} \quad & \langle S_2\rangle\ t_2 : f_2 \xrightarrow[I]{k_2'} \langle S_2'\rangle\ t_2 : f_2' \\ \text{and} \quad & \langle S_2\rangle\ t_2 : f_2 \xrightarrow[I]{k_2''} \langle S_2''\rangle\ t_2 : f_2'' \\ \text{then} \quad & S_2' = S_2'',\ f_2' = f_2'',\ k_2' = k_2'' \end{aligned} \tag{H4}$$

Next, we show that the sequence operator (;), and the `abort` and `par` kernel statements have at most one transition.

(1) Consider the fragment $t_1 : f_1;\ f_2$. By induction hypothesis (H3), there is only one possible transition for the fragment $t_1 : f_1$, which is either:

$$\langle S_1 \rangle\ t_1 : f_1 \xrightarrow[I]{k_1=0} \langle S_1' \rangle\ t_1 :$$

or:

$$\langle S_1 \rangle\ t_1 : f_1 \xrightarrow[I]{k_1 \in \{1,\bot\}} \langle S_1' \rangle\ t_1 : f_1'$$

The table below shows that at most one sequence rule can be applied depending on the completion code k_1:

k_1		
0	**1**	**⊥**
(seq-right)	(seq-left)	

It follows that the sequence operator ";" is deterministic.

(2) Consider the `abort` kernel statement in the fragment $t_1 :$ `weak?` `abort`(a_1, f_1). By induction hypothesis (H3), there is only one possible transition for the program fragment $t_1 : f_1$, which is either:

$$\langle S_1 \rangle\ t_1 : f_1 \xrightarrow[I]{k_1=0} \langle S_1' \rangle\ t_1 :$$

or:

$$\langle S_1 \rangle\ t_1 : f_1 \xrightarrow[I]{k_1 \in \{1,\bot\}} \langle S_1' \rangle\ t_1 : f_1'$$

The table below shows that at most one `abort` rule can be applied depending on the completion code k_1 and the preemption status $A[a_1]$:

		Strong abort, k_1			**Weak abort, k_1**		
		0	**1**	**⊥**	**0**	**1**	**⊥**
$A[a_1]$	**= 0**	(abort-2)	(abort-1)		(abort-2)	(abort-1)	
	≠ 0	(abort-6)			(abort-3)	(abort-5)	(abort-4)

It follows that the `abort` kernel statement is deterministic.

(3) Consider the `par` kernel statement in the fragment $t :$ `par`$(t_1 : f_1,\ t_2 : f_2)$. By induction hypotheses (H3) and (H4), there is only one possible transition for the program fragment $t_1 : f_1$, which is either:

$$\langle S_1 \rangle\ t_1 : f_1 \xrightarrow[I]{k_1=0} \langle S_1' \rangle\ t_1 :$$

or:

$$\langle S_1 \rangle\ t_1 : f_1 \xrightarrow[I]{k_1 \in \{1,\bot\}} \langle S_1' \rangle\ t_1 : f_1'$$

and there is only one possible transition for the program fragment $t_2 : f_2$, which is either:

$$\langle S_2 \rangle\ t_2 : f_2 \xrightarrow[I]{k_2=0} \langle S_2' \rangle\ t_2 :$$

or:

$$\langle S_2 \rangle \ t_2 : f_2 \xrightarrow[I]{k_2 \in \{1,\perp\}} \langle S_2' \rangle \ t_2 : f_2'$$

The table below shows that at most one `par` rule can applied depending on the completion codes k_1 and k_2:

		k_2		
		0	**1**	**⊥**
k_1	**0**	(par-5)	(par-6)	(par-2)
	1	(par-7)	(par-4)	
	⊥	(par-3)		(par-1)

It follows that the `par` kernel statement is deterministic.

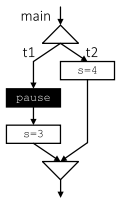
In conclusion, all ForeC kernel programs that are correct with respect to syntax and type checking are deterministic. □

```
1 shared int s=0 combine all with plus;
2 void main(void) {
3   par({pause; s=3;},{s=4;});
4 }
```

(a) ForeC program.

```
1 int s=0;
2 void main(void) {
3   copy;
4   par(t1:{copy; pause; s=3;},
5       t2:{copy; s=4;});
6 }
```

(b) Translated kernel program.



(c) Control-flow graph.

Fig. 25. Illustrative example one.

B ILLUSTRATIONS

This section illustrates how ForeC programs execute under the formal semantics presented in Section 3.4. Two example ForeC programs are used, and their executions are given as a sequence of rewrites.

B.1 Example One

The first program illustrates parallel execution using the `par` statement. Figure 25a presents the ForeC program, and Figure 25c illustrates the program's control-flow. Recall that an upright triangle represents the forking of threads while an inverted triangle represents the joining of threads. The corresponding kernel program is shown in Figure 25b.

In the program's first tick, the parent thread `main` begins its local tick by forking two child threads, `t1` and `t2`. The child threads start their local ticks by copying the shared variable `s`. Thread `t1` pauses while thread `t2` assigns 4 to its local copy of `s` and terminates. The first tick ends and the shared variable `s` is resynchronized. Using the combine policy `all`, the new value (or the resynchronized value) of `s` becomes $\texttt{plus}(0, 4) = 4$. In the second tick, only thread `t1` is active and it starts its local tick by creating a copy of `s`, assigning 3 to its copy of `s`, and then terminating. Now that both child threads have terminated and joined, the `par` combines its children's copies of `s` before also terminating. Because only thread `t1` was active in the second tick, its copy of `s` is assigned directly to its parent thread `main`. Thread `main` starts its local tick which results in the program terminating. The second tick ends and the shared variable `s` is resynchronized with the value 3 because only thread `main` has a copy of `s`.

Before we apply the rewrite rules to the program, it is structurally translated into the kernel program of Figure 25b (see the start of Section 3). The semantic functions GetShared(`main`), GetShared(`t1`), GetShared(`t2`) and GetShared($\mathcal{G}$) all return {`s`}. The set of preemption statuses A is initially $\emptyset$. The program's environment E and its derivatives are defined in Figure 26.

Step 1: Start the tick by applying the (seq-right) and (copy) rules.

$$\text{(seq-right)}\ \frac{\text{(copy)}\ \dfrac{}{\langle E, A\rangle\ \texttt{main:copy} \xrightarrow[I]{0} \langle E^1, A\rangle\ \texttt{main:}}}{\langle E, A\rangle\ \texttt{main:copy;par(t1:\{copy;}\ \texttt{pause;s=3;\},t2:\{copy;s=4;\})} \xrightarrow[I]{\perp} \langle E^1, A\rangle\ \texttt{main:par(t1:\{copy;}\ \texttt{pause;s=3;\},t2:\{copy;s=4;\})}}$$

$$
\begin{array}{lcl}
E & = & \{\mathcal{G} \to \{s \to (0, \texttt{pre})\}\} \\
E^1 & = & \{\mathcal{G} \to \{s \to (0, \texttt{pre})\}, \mathit{main} \to \{s \to (0, \texttt{pre})\}\} \\
E^2 & = & \{\mathcal{G} \to \{s \to (0, \texttt{pre})\}, \mathit{main} \to \{s \to (0, \texttt{pre})\}, t1 \to \{s \to (0, \texttt{pre})\}\} \\
E^3 & = & \{\mathcal{G} \to \{s \to (0, \texttt{pre})\}, \mathit{main} \to \{s \to (0, \texttt{pre})\}, t2 \to \{s \to (0, \texttt{pre})\}\} \\
E^4 & = & \{\mathcal{G} \to \{s \to (0, \texttt{pre})\}, \mathit{main} \to \{s \to (0, \texttt{pre})\}, t1 \to \{s \to (0, \texttt{pre})\}, \\
 & & t2 \to \{s \to (0, \texttt{pre})\}\} \\
E^5 & = & \{\mathcal{G} \to \{s \to (0, \texttt{pre})\}, \mathit{main} \to \{s \to (0, \texttt{pre})\}, t1 \to \{s \to (0, \texttt{pre})\}, \\
 & & t2 \to \{s \to (4, \texttt{mod})\}\} \\
E^6 & = & \{\mathcal{G} \to \{s \to (0, \texttt{pre})\}, \mathit{main} \to \{s \to (4, \texttt{cmb})\}\} \\
E^7 & = & \{\mathcal{G} \to \{s \to (4, \texttt{pre})\}\} \\
E^8 & = & \{\mathcal{G} \to \{s \to (4, \texttt{pre})\}, t1 \to \{s \to (4, \texttt{pre})\}\} \\
E^9 & = & \{\mathcal{G} \to \{s \to (4, \texttt{pre})\}, t1 \to \{s \to (3, \texttt{mod})\}\} \\
E^{10} & = & \{\mathcal{G} \to \{s \to (4, \texttt{pre})\}, \mathit{main} \to \{s \to (3, \texttt{cmb})\}\} \\
E^{11} & = & \{\mathcal{G} \to \{s \to (3, \texttt{pre})\}\}
\end{array}
$$

Fig. 26. Initial program environment and its derivatives.

Step 2: Both threads of the `par` execute sequential statements, both starting with a `copy` statement. Apply the (par-1) rule. Additionally, apply the (seq-right) and (copy) rules to both threads. The environments of both threads, E^2 and E^3, are aggregated into E^4.

$$
\dfrac{\dfrac{(\text{copy})\ \dfrac{}{\langle E^1, A\rangle\ \texttt{t1:copy} \xrightarrow[I]{0} \langle E^2, A\rangle\ \texttt{t1:pause;s=3}}}{\langle E^1, A\rangle\ \texttt{t1:copy;pause;s=3} \xrightarrow[I]{\perp} \langle E^2, A\rangle\ \texttt{t1:pause;s=3}}\ (\text{seq-right}) \qquad \dfrac{(\text{copy})\ \dfrac{}{\langle E^1, A\rangle\ \texttt{t2:copy} \xrightarrow[I]{0} \langle E^3, A\rangle\ \texttt{t2:copy;s=4;}}}{\langle E^1, A\rangle\ \texttt{t2:copy;s=4;} \xrightarrow[I]{\perp} \langle E^3, A\rangle\ \texttt{t2:s=4;}}\ (\text{seq-right})}{\langle E^1, A\rangle\ \texttt{main:par(t1:\{copy;pause;s=3;\},t2:\{copy;s=4;\})} \xrightarrow[I]{\perp} \langle E^4, A\rangle\ \texttt{main:par(t1:\{pause;s=3;\},t2:\{s=4;\})}}\ (\text{par-1})
$$

Step 3: Apply the (tick) and (par-7) rules. Additionally, apply the (seq-left) and (pause) rules to the first thread and the (assign-shared) rule to the second thread. **The program completes the tick.** Note that when the (par-7) rule is applied, the aggregation of environments E^4 and E^5 is simply E^5, whose stores for threads `t1` and `t2` are combined into thread `main`'s store, resulting in E^6. When the (tick) rule is applied, E^6 is resynchronized to be E^7.

$$
\dfrac{\dfrac{\dfrac{(\text{pause})\ \dfrac{}{\langle E^4, A\rangle\ \texttt{t1:pause} \xrightarrow[I]{1} \langle E^4, A\rangle\ \texttt{t1:copy}}}{\langle E^4, A\rangle\ \texttt{t1:pause;s=3} \xrightarrow[I]{1} \langle E^4, A\rangle\ \texttt{t1:copy;s=3}}\ (\text{seq-left}) \qquad \dfrac{s \in \{s\}}{\langle E^4, A\rangle\ \texttt{t2:s=4;} \xrightarrow[I]{0} \langle E^5, A\rangle\ \texttt{t2:}}\ (\text{assign-shared})}{\langle E^4, A\rangle\ \texttt{main:par(t1:\{pause;s=3;\},t2:\{s=4;\})} \xrightarrow[I]{1} \langle E^6, A\rangle\ \texttt{main:par(t1:\{copy;s=3;\},t2:nop)}}\ (\text{par-7})}{\langle E^4, A\rangle\ \texttt{main:par(t1:\{pause;s=3;\},t2:\{s=4;\})} \xrightarrow[I]{1} \langle E^7, A\rangle\ \texttt{main:par(t1:\{copy;s=3;\},t2:nop)}}\ (\text{tick})
$$

Step 4: Start the next tick by applying the (par-3) rule. Additionally, apply the (seq-right) and (copy) rules to the first thread and the (nop) rule to the second thread.

$$\dfrac{\dfrac{\text{(copy)}\ \dfrac{}{\langle E^7, A\rangle\ \texttt{t1:copy} \xrightarrow[I]{0} \langle E^8, A\rangle\ \texttt{t1:}}}{\text{(seq-right)}\ \dfrac{\langle E^7, A\rangle\ \texttt{t1:}\ \texttt{copy;s=3} \xrightarrow[I]{\perp} \langle E^8, A\rangle\ \texttt{t1:s=3}}{}} \quad \text{(nop)}\ \dfrac{}{\langle E^7, A\rangle\ \texttt{t2:}\ \texttt{nop} \xrightarrow[I]{0} \langle E^7, A\rangle\ \texttt{t2:}}}{\text{(par-3)}\ \dfrac{}{\langle E^7, A\rangle\ \texttt{main:par(t1:\{copy;}\ \texttt{s=3;\},t2:nop)} \xrightarrow[I]{\perp} \langle E^8, A\rangle\ \texttt{main:par(t1:\{s=3;\},t2:nop)}}}$$

Step 5: Apply the (par-5) rule. Additionally, apply the (assign-shared) rule to the first thread and the (nop) rule to the second thread. Note that when the (par-5) rule is applied, the aggregation of environments E^8 and E^9 is simply E^9, whose stores for `t1` and `t2` are combined into `main`'s store, which results in E^{10}.

$$\dfrac{\dfrac{\text{(assign-shared)}\ \dfrac{s \in \{s\}}{\langle E^8, A\rangle\ \texttt{t1:}\ \texttt{s=3} \xrightarrow[I]{0} \langle E^9, A\rangle\ \texttt{t1:}} \quad \text{(nop)}\ \dfrac{}{\langle E^8, A\rangle\ \texttt{t2:}\ \texttt{nop} \xrightarrow[I]{0} \langle E^8, A\rangle\ \texttt{t2:}}}{\text{(par-5)}\ \dfrac{}{\langle E^8, A\rangle\ \texttt{main:par(t1:\{s=3;\},t2:nop)} \xrightarrow[I]{\perp} \langle E^{10}, A\rangle\ \texttt{main:copy}}}$$

Step 6: Apply the (tick) and (copy) rules. The environment E^{10} is resynchronized to be E^{11}. The tick ends and the program terminates.

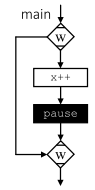
$$\dfrac{\dfrac{\text{(copy)}\ \dfrac{}{\langle E^{10}, A\rangle\ \texttt{main:copy} \xrightarrow[I]{0} \langle E^{10}, A\rangle\ \texttt{main:}}}{\text{(tick)}\ \dfrac{}{\langle E^{10}, A\rangle\ \texttt{main:copy} \xrightarrow[I]{0} \langle E^{11}, A\rangle\ \texttt{main:}}}$$

```
1 int x=1;
2 void main(void) {
3   weak abort {
4     x++; pause;
5   } when immediate (x==1);
6 }
```

(a) ForeC program.

```
1 int x=1;
2 void main(void) {
3   status(a1, x==1);
4   weak abort(a1, {x++; pause;});
5 }
```

(b) Translated kernel program.



(c) Control-flow graph.

Fig. 27. Illustrative example two.

$$E = \{\mathcal{G} \to \{x \to (1, \mathsf{pvt})\}\} \qquad A = \{a1\}$$
$$E^1 = \{\mathcal{G} \to \{x \to (2, \mathsf{pvt})\}\} \qquad A^1 = \{a1 \to 1\}$$
$$A^2 = \{a1 \to 0\}$$

Fig. 28. Initial program state and its derivatives.

B.2 Example Two

The second program illustrates the preemption by using an `immediate weak abort` statement. Figure 27a presents the ForeC program and Figure 27c illustrates the program's control-flow. Recall that a pair of decorated diamonds represents the scope of an `abort` body. The corresponding kernel program is shown in Figure 27b.

In the program's first tick, thread `main` reaches the `immediate and weak abort` and immediately evaluates the preemption condition `(x==1)`. The condition evaluates to *true* so the preemption is triggered. Because the `abort` is weak, the preemption is taken only when the execution of the encapsulated thread reaches the `pause`, hence, after variable `x` has been incremented. The `abort` terminates and, as a result, thread `main` terminates. At this point, the first tick ends.

Before we apply the rewrite rules to the program, it is structurally translated into the kernel program of Figure 27b (see the start of Section 3). No `copy` kernel statement is inserted into the program because no shared variables are used. The semantic functions GetShared(`main`) and GetShared($\mathcal{G}$) all return $\emptyset$. The program's environment E, preemption statuses A, and their derivatives are defined in Figure 28.

Step 1: Start the tick by applying the (seq-right) and (status) rules. Note that the `abort`'s preemption is triggered because the condition `x==1` evaluates to 1.

$$\text{(seq-right)}\ \frac{\text{(status)}\ \dfrac{}{\langle E, A\rangle\ \texttt{main: status(a1,x==1)} \xrightarrow[I]{0} \langle E, A^1\rangle\ \texttt{main:}} \qquad \langle E, A\rangle\ \texttt{main:status(a1,x==1);}\ \texttt{weak abort(a1,\{x++;pause;\})} \xrightarrow[I]{\perp} \langle E, A^1\rangle\ \texttt{main:weak abort}\ \texttt{(a1,\{x++;pause;\})}}{}$$

Step 2: Apply the (abort-4), (seq-right), and (assign-private) rules.

$$\text{(abort-4)}\ \frac{\text{(seq-right)}\ \dfrac{\text{(assign-private)}\ \dfrac{x \notin \emptyset}{\langle E, A^1\rangle\ \texttt{main:x++} \xrightarrow[I]{0} \langle E^1, A^1\rangle\ \texttt{main:}}}{\langle E, A^1\rangle\ \texttt{main:x++;pause} \xrightarrow[I]{\perp} \langle E^1, A^1\rangle\ \texttt{main:pause}}}{\begin{array}{l}\langle E, A^1\rangle\ \texttt{main:weak abort} \\ \texttt{(a1,\{x++;pause;\})}\end{array} \xrightarrow[I]{\perp} \begin{array}{l}\langle E^1, A^1\rangle\ \texttt{main:weak abort} \\ \texttt{(a1,\{pause;\})}\end{array}}\ (A^1[a1] \neq 0)$$

Step 3: Apply the (abort-5) and (pause) rules. Note that the preemption is taken because the `abort`'s body has reached a `pause`.

$$\text{(abort-5)}\ \frac{\text{(pause)}\ \dfrac{}{\langle E^1, A^1\rangle\ \texttt{main:pause} \xrightarrow[I]{1} \langle E^1, A^1\rangle\ \texttt{main:copy}}}{\langle E^1, A^1\rangle\ \texttt{main:weak abort(a1,\{pause;\})} \xrightarrow[I]{\perp} \langle E^1, A^1\rangle\ \texttt{main:copy}}\ (A^1[a1] \neq 0)$$

Step 4: Apply the (tick) and (copy) rules. The tick ends, the preemption statuses are reevaluated to A^2, and the program terminates.

$$\text{(tick)}\ \frac{\text{(copy)}\ \dfrac{}{\langle E^1, A^1\rangle\ \texttt{main:copy} \xrightarrow[I]{0} \langle E^1, A^1\rangle\ \texttt{main:}}}{\langle E^1, A^1\rangle\ \texttt{main:copy} \xrightarrow[I]{0} \langle E^1, A^2\rangle\ \texttt{main:}}$$

```
1  void main(void) {
2    shared int x=3 combine all with plus;
3    int y=5;
4    f(x,y);
5    g(&x);
6  }
7
8  void f(int d, shared int e) {
9    ...
10 }
11 void g(shared int* p) {
12   // p points to the shared variable x declared on line 2.
13   ...
14 }
```

Fig. 29. Example of passing a shared variable by value and by reference.

C SHARED VARIABLES

This appendix describes how shared variables are passed by value or by reference into functions, and how the combine policies and combine functions work together to combine more than two copies of a shared variable. We compare the behaviors of the combine policies, `all`, `new`, and `mod`, using an illustrative example. We also provide additional examples of combine functions for primitive C data types and for programmer-specified data structures.

C.1 Passing Shared Variables by Value and by Reference

Following the C convention, a function argument in ForeC can be passed by value or by reference. An argument passed by value can either be a shared or a private variable. For example, in Figure 29, line 4 makes a call to function `f` with the arguments `x` (a shared variable) and `y` (a private variable). The definition of function `f` (line 8) has parameters `d` (a private variable) and `e` (a shared variable) that are initialized with the values 3 (from `x`) and 5 (from `y`), respectively.

When passed by reference, the address of the function's argument is copied into the function's parameter. Thus, changes made to the dereferenced parameter are made to the argument. The following examples demonstrate how the `shared` type qualifier can be used in the declaration of references:

`int* p` declares an ordinary C pointer to an `int` variable. This means that pointer `p` and the referenced `int` variable can only be accessed by at most one thread.

`shared int* p` declares a pointer `p`, which can only be accessed by at most one thread, to a shared `int` variable. In Figure 29, line 11, the parameter of function `g` declares pointer `p` to a shared `int` variable. On line 5, the "`&`" unary operator is used to pass shared variable `x` into `g` by reference.

`shared int shared* p` declares a shared pointer `p`, which can be accessed by multiple threads, to a shared `int` variable.

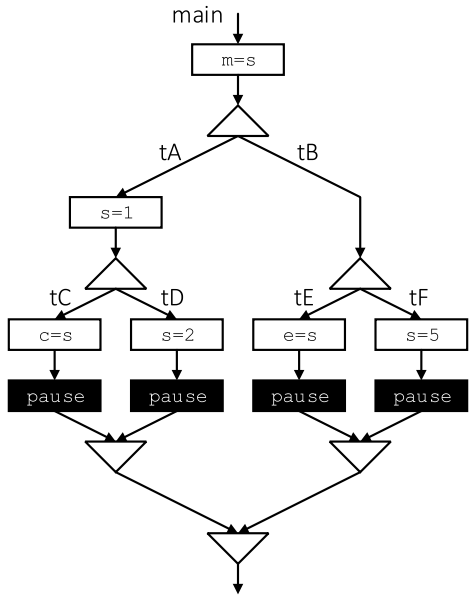
`int shared* p` shorthand of "`shared int shared* p`" when only one thread can write to the referenced `int` variable.

C.2 Combining More Than Two Copies

The ForeC program shown in Figure 30a is used to explain how multiple copies of a shared variable are combined. The program's control-flow graph is shown in Figure 30b. The program has a shared

```
1  shared int s=3 combine all with plus;
2  void main(void) {
3    int m=s;
4    par(tA(),tB());
5  }
6
7  void tA(void) {
8    s=1;
9    par(tC(),tD());
10 }
11 void tC(void) { int c=s; pause; }
12 void tD(void) { s=2; pause; }
13
14 void tB(void) {
15   par(tE(),tF());
16 }
17 void tE(void) { int e=s; pause; }
18 void tF(void) { s=5; pause; }
19
20 int plus(int th1,int th2) {
21   return (th1+th2);
22 }
```

(a) ForeC program.



(b) Control-flow graph.

Fig. 30. Example ForeC program.

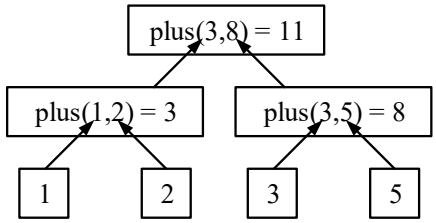
variable called `s` that uses the combine function `plus`. The initial value of `s` is 3 for the program's first tick. Figure 31a shows the copies of `s` at the end of the first tick, organized by the thread genealogy. Each node represents a thread and the current value of its local copy, e.g., main: 3 means that the `main` thread has a local copy of `s` with the value 3. Copies that were assigned a value during the tick have the • symbol, e.g., tA: 1• means that thread `tA`'s copy has been assigned the value 1. Arrows are drawn from the child threads to their parents to show the thread genealogy. Threads `tC` and `tD` create their copies from `tA`'s copy (see Section 2.5). Hence, the value of thread `tC`'s copy is 1. Threads `tE` and `tF` create their copies from `tB`'s copy.

The formal semantics of ForeC (Section 3) defines how more than two copies of a shared variable are combined. The copies from sibling threads (i.e., threads forked by the same `par` statement) are combined and the resulting value is assigned to their parent thread. Then, the copies of the parent and its sibling are combined together and assigned to their parent. This continues until the `main` thread is reached. Figure 31b illustrates the combine policy `all`, where all the copies are combined. The final combined value is 11 and it is assigned to shared variable `s` to complete the global tick.

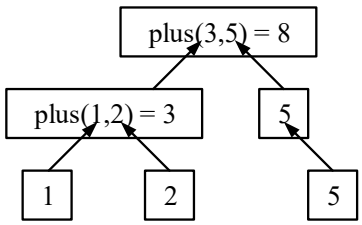
The combine policy `new` only takes into account the copies for which the value has changed since the previous global tick. For the copies shown in Figure 31a, thread `main`, `tB`, and `tE`'s copies of `s` would be ignored. Although thread `tC` did not assign any value to its copy, the value was taken from thread `tA` who had assigned the new value 1 to its copy. Thus, thread `tC`'s copy does have a new value. Figure 31c illustrates the combine policy `new`. Note that thread `tF`'s copy is assigned directly to `tB` because its sibling's copy is ignored. The final combined value is 8.

For the combine policy `mod`, the copies that have not been assigned a value during the tick are ignored. For the copies shown in Figure 31a, thread `main`, `tB`, `tC`, and `tE`'s copies of `s` would be ignored. Figure 31d illustrates the combine policy `mod`. The final combined value is 7.

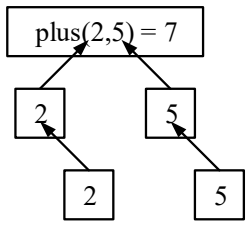
(a) The copies of `s` when tick 1 ends.



(b) Policy `all`.



(c) Policy `new`.



(d) Policy `mod`.

Fig. 31. Effects of the combine policies.

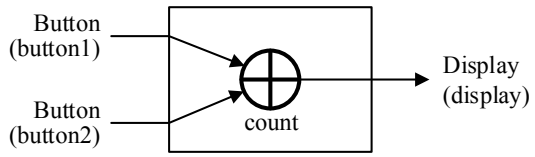
C.3 Combine Policies Illustrated

This section illustrates the behavior of the combine policies `all`, `new`, and `mod` over several ticks by using the example of Figure 32. Figure 32a is a ForeC program that outputs the number of times that `button1` and `button2` were pressed in each tick. The program and its interface with the environment is illustrated in Figure 32b. On line 6 in Figure 32a, threads `t1` and `t2` are forked to check which buttons have been pressed. The results are assigned to the shared variable `count`. Line 6 also forks thread `t3` to read the value of `count` and to assign it to `display` output. Hence, three copies of `count` will be created in each tick. The copies of `count` are combined with the function `plus` (line 18) with the combine policy `mod`. Figure 33a provides possible input values for five ticks of the program. For example, only `button1` is pressed in tick 2. Figure 33b shows the value of the shared variable `count` and the value of each thread's local copy of `count`. The copies that were assigned a value during the tick have the • symbol. For tick 1, `count` is equal to 0; its initial value. From tick 2 onwards, the value of `count` corresponds to the number of button presses in the previous tick, because only threads `t1` and `t2`'s modified copies are combined.

Figure 33c illustrates the behavior of the combine policy `new` over several ticks. The values of threads `t1`, `t2`, and `t3`'s local copies are ignored when they have the same value as `count`. Figure 33d illustrates the behavior of the combine policy `all` over several ticks. In this case, threads `t1`, `t2`, and `t3`'s local copies are always used to compute the value of `count`. The value of `count` corresponds to the running total of button presses, i.e., in tick 6 a total of four button presses have occurred in previous ticks.

```
1  input int button1,button2;
2  output int display=0;
3  shared int count=0 combine mod with plus;
4
5  void main(void) {
6    par( par(t1(),t2()), t3() );
7  }
8  void t1(void) {
9    while (1) { count = (button1==1); pause; }
10 }
11 void t2(void) {
12   while (1) { count = (button2==1); pause; }
13 }
14 void t3(void) {
15   while (1) { display = count; pause; }
16 }
17
18 int plus(int th1,int th2) { return (th1+th2); }
```

(a) ForeC program of the button counter.



(b) Button counter.

Fig. 32. Example of counting the number of button inputs.

	Tick					
	1	**2**	**3**	**4**	**5**	**6**
button1	0	1	1	1	0	···
button2	0	0	1	0	0	···

(a) Possible input values.

	Tick					
count	**1**	**2**	**3**	**4**	**5**	**6**
t1's copy	0•	1•	1•	1•	0•	···
t2's copy	0•	0•	1•	0•	0•	···
t3's copy	0	0	1	2	1	···
count	0	0	1	2	1	0

(b) Combine policy mod.

	Tick					
count	**1**	**2**	**3**	**4**	**5**	**6**
t1's copy	0•	1•	1•	1•	0•	···
t2's copy	0•	0•	1•	0•	0•	···
t3's copy	0	0	1	1	0	···
count	0	0	1	1	0	0

(c) Combine policy new.

	Tick					
count	**1**	**2**	**3**	**4**	**5**	**6**
t1's copy	0•	1•	1•	1•	0•	···
t2's copy	0•	0•	1•	0•	0•	···
t3's copy	0	0	1	3	4	···
count	0	0	1	2	1	0

(d) Combine policy all.

Fig. 33. The value of count under each combine policy.

C.4 Examples of Combine Functions

The simplest combine functions are those based on the associative and commutative binary mathematical operators:

- Arithmetic addition (+) and multiplication (*);
- Logical AND (&&) and OR (||);
- Bitwise AND (&) and OR (|);
- Minimum and maximum using `if-else` statement.

The combine functions should be associative and commutative so as to make the `par` statement associative (Lemma 13) and commutative (Lemma 15), relying only on the nesting of binary `par` statements to fork more than two child threads at the same time (e.g., $\texttt{par(par(t}_1\texttt{,t}_2\texttt{),t}_3\texttt{)} = \texttt{par(t}_1\texttt{,par(t}_2\texttt{,t}_3\texttt{))}$). The programmer is free to write combine functions based on non-associative or non-commutative binary operators (e.g., -, /, or %), but will negate the associative or commutative property, respectively, of the `par` statement.

Combine functions can also be defined for user-defined data structures. For example, Figure 34a defines a C-struct called `ProdSum` that stores the product (`prod`) and sum (`sum`) of the numbers assigned to it. The combine function `prodsum` multiplies all the values in `prod` and sums all the values in `sum`. An example of its behavior is provided after the function as comments.

For another example, Figure 34b defines a combine function that returns the minimum value that has been assigned since the start of the program. To achieve this, line 1 of Figure 34b defines a C-struct called `Min` that stores the value (`val`) that a thread assigns in the current tick and tracks the minimum value (`min`) that has been assigned since the start of the program. The combine function (line 3) stores in `res.val` the minimum value between `th1.val` and `th2.val`. Then, `res.min` stores the minimum value between itself and `res.val`.

The behavior of combine functions can be extended with dedicated threads that perform additional computations on the combined values of one or more shared variables. Figure 35 is an example ForeC program that calculates the average of three input values `in[i]`, declared on line 1 in Figure 35. Line 6 forks three threads `f` (line 9) to check the validity of each input. An input is valid if its value is greater than zero (line 11). In each tick, the threads assign valid inputs to their copy of the shared variable `val`. The modified copies of `val` are combined with the combine function `sum` (line 34), which sums the input values and the number of valid inputs. The `average` thread (line 22) reads the resulting combined value of `val` to calculate the average input value (line 29).

```c
typedef struct {int prod; int sum;} ProdSum;

ProdSum prodsum(ProdSum th1,ProdSum th2) {
  ProdSum res = {.prod=(th1.prod*th2.prod), .sum=(th1.sum+th2.sum)};
  return res;
}

// th1={. prod=2, . sum=2} and th2={. prod=5, . sum=5}
// prodsum(th1, th2)={. prod=10, . sum=7}
```

(a) Product and sum of two or more values.

```c
typedef struct {int val; int min;} Min;

Min min(Min th1,Min th2) {
  Min res;
  // Calculate the min for the current tick
  if (th1.val>th2.val) {
    res.val=th2.val;
  } else {
    res.val=th1.val;
  }
  // Calculate the min since the start of the program
  if (th1.min>res.val) {
    res.min=res.val;
  } else {
    res.min=th1.min;
  }
  return res;
}

// Initial value: res={. value=0, . min=0}
//
// Tick 1:
// th1={. value=1, . min=0} and th2={. value=2, . min=0}
// min(th1, th2)={. value=1, . min=0}
//
// Tick 2:
// th1={. value=-1, . min=0} and th2={. value=2, . min=0}
// min(th1, th2)={. value=-1, . min=-1}
```

(b) Minimum of two or more values.

Fig. 34. Examples of combine functions for C-structs.

```
1  input double in[3];
2  typdef struct {double value; int valid;} ValidValue;
3  shared ValidValue val={.value=0,.valid=0} combine mod with sum;
4
5  void main(void) {
6    par( par( par(f(0),f(1)), f(2) ), average() );
7  }
8
9  void f(int i) {
10   while (1) {
11     if (in[i] > 0) {
12       val.value = in[i];
13       val.valid = 1;
14     } else {
15       val.value = 0;
16       val.valid = 0;
17     }
18     pause;
19   }
20 }
21
22 void average(void) {
23   double result = 0;
24   while (1) {
25     pause;
26     if (val.valid == 0) {
27       result = 0;
28     } else {
29       result = val.value/val.valid;
30     }
31   }
32 }
33
34 ValidValue sum(ValidValue th1,ValidValue th2) {
35   ValidValue res;
36   res.value = th1.val+th2.val;
37   res.valid = th1.valid+th2.valid;
38   return res;
39 }
```

Fig. 35. Averaging two or more values.

Received 25 March 2022; revised 13 January 2023; accepted 8 March 2023