

Synchronous Deterministic Parallel Programming for Multicores with ForeC

EUGENE YIP, Department of ECE, The University of Auckland, New Zealand

PARTHA S. ROOP, Department of ECE, The University of Auckland, New Zealand

ALAIN GIRAULT, Inria, France. Université Grenoble Alpes, Lab. LIG, Grenoble, France. CNRS, Lab. LIG, F-38000 Grenoble, France

Morteza Biglari-Abhari, Department of ECE, The University of Auckland, New Zealand

Cyber-physical systems (CPSs) are embedded systems that are tightly integrated with their physical environment. The correctness of a CPS depends on the output of its computations and on the timeliness of completing the computations. The increasing use of high-performing and low-power multi-core processors in embedded systems is pushing embedded programmers to be parallel programming experts. 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 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 allows programmers to express many forms of parallel patterns while ensuring 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 mutual exclusion constructs are not needed. Through benchmarking, we demonstrate that ForeC can achieve better parallel performance than Esterel, a widely used synchronous language for concurrent safety-critical systems, and OpenMP, a popular desktop solution for parallel programming. We 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, multi-core, worst-case execution time, code generation

ACM Reference Format:

Eugene Yip, Partha S. Roop, Alain Girault, and Morteza Biglari-Abhari, 2016. Synchronous Deterministic Parallel Programming for Multicores with ForeC. *ACM Trans. Program. Lang. Syst.* 0, 0, Article 00 (0000), 75 pages.

DOI: 0000001.0000001

1. INTRODUCTION

People interact daily with many *embedded systems*, which are digital systems embedded into a product to add specific functionality, such as those shown in Figure 1. An embedded system is *safety-critical* if its failure to operate correctly may lead to catastrophic consequences [Alemzadeh et al. 2013]. Safety-critical embedded systems [Knight 2002] must be dependable and functionally safe [Buja and Menis 2012; Paolieri and Mariani 2011; Dunn 2003] and certified against safety standards, such

This work was supported in part by the RIPPEs INRIA International Lab.

E. Yip, (Current address) Software Technologies Research Group, University of Bamberg, 96045 Germany. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 0000 ACM. 0164-0925/0000/-ART00 \$15.00

DOI: 0000001.0000001

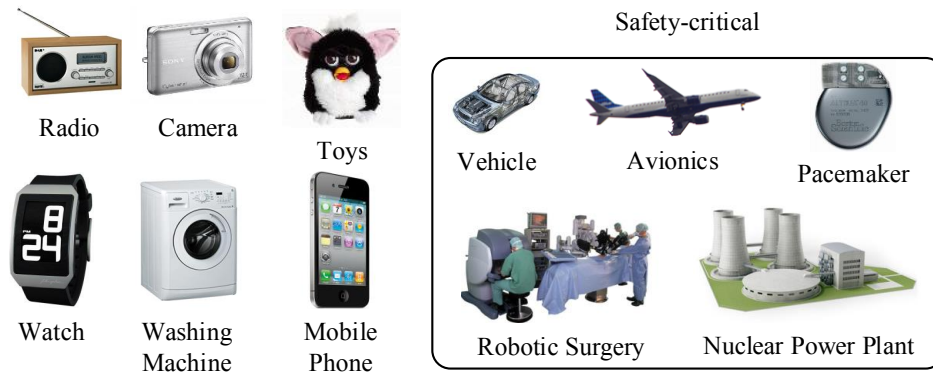


Fig. 1. Examples of embedded systems and those with safety-critical concerns.

as DO-178B [Radio Technical Commission for Aeronautics 1992], IEC 61508 [International Electrotechnical Commission 2010], and ISO 26262 [ISO/TC22/SC3 2011]. Certification is a costly and time consuming exercise and is exacerbated by the use of multi-core processors to create more efficient designs. Safety-critical embedded systems typically monitor and control physical processes in the environment that in turn affects the computations of the embedded systems. Because the computations and physical processes are tightly coupled, these embedded systems need to be real-time and reactive, computing new outputs as soon as new inputs are detected. For example, an unmanned aerial vehicle must react continuously to its surrounding environment to avoid obstacles while it flies to its intended destination. The correctness of an embedded system depends on the output of its computations and on the timeliness of completing the computations [Lee 2009; Wilhelm and Grund 2014].

A key to building successful embedded systems using multi-core processors is the understanding of the timing behaviors of the computations [Ayer et al. 2014] and physical processes. Unfortunately, the timing behavior of computations modeled in the C programming language [ISO/IEC JTC1/SC22/WG14 2011], a popular language for programming embedded systems, is complex because it depends on the underlying architecture. The timing of C programs is typically validated by static worst-case execution time (WCET) analysis [Wilhelm and Grund 2014]. The understanding of the timing behavior of C programs on multi-core processors can be greatly enhanced by the following methods : (1) introducing timing constructs that allow programmers to control time as a first-class resource, e.g., enforcing that the execution time between two programming points must be less than the inter-arrival time of inputs ; and (2) defining a deterministic parallel execution semantics for multi-threaded C programs that communicate over shared memory. This paper tackles these two points by bringing together the formal semantics of synchronous languages [Benveniste et al. 2003] and the benefits of C's control and data structures. The resulting language, called ForeC, is suitable for the deterministic parallel programming of multi-cores. The following sections review the parallel programming of embedded multi-cores and the programming restrictions for easing the certification process.

1.1. Parallel Programming of Embedded Systems

Programs can be executed directly by the hardware (bare-metal) or by a real-time operating system (RTOS) [Wolf et al. 2010]. The bare-metal approach allows all of the system's resources to be used to execute the program, but code must be included to manage the hardware. By contrast, an RTOS manages the hardware and pro-

vides a consistent environment for developing and executing programs, thus, enabling code portability across a range of systems. Hence, the RTOS must be taken into account when analyzing programs. C [ISO/IEC JTC1/SC22/WG14 2011] is a popular language for programming embedded systems with support for multi-threading and parallelism provided by third-party libraries, compilers, and runtime support [Diaz et al. 2012]. Notable examples include Pthreads [The IEEE and The Open Group 2008], OpenMP [OpenMP Architecture Review Board 2013], and MPI [Message Passing Interface Forum 2012]. These multi-threading solutions are inherently *non-deterministic* [Lee 2006] because they allow non-deterministic constructs, such as race conditions over shared variables in the case of Pthreads and OpenMP. The lack of formal semantics for the programming model can also lead to ambiguous behaviors.

Parallel programming is challenging because it requires programmers to have specific skills, experience, and knowledge to avoid the common parallel programming traps and pitfalls [McDowell and Helmbold 1989]. For example, parallel accesses to the same shared variable will interfere and corrupt the value of the shared variable. It is the programmer's responsibility to identify the regions of code that can interfere, called *critical sections*, and ensure that they are executed sequentially at mutually exclusive times. Hence, programmers need to be aware of the data dependencies in their specific program and to choose the appropriate solution to manage the dependencies. Studies have shown that, without careful tuning [Levy and Conte 2009], parallel programs executed on multi-cores can perform worse than their sequential counterparts. The next section describes the use of *synchronous languages* as an alternative to creating concurrent programs that are deterministic.

1.2. Synchronous Languages

Synchronous languages [Benveniste et al. 2003] are based on sound mathematical semantics, which facilitates system verification by formal methods [Benveniste et al. 2003] and the generation of correct-by-construction implementations [Edwards and Zeng 2007; Natale and Zeng 2012]. Figure 2 depicts a synchronous program, defined as a set of concurrent threads, within its physical environment. Synchronous programs react continuously to inputs from the environment by producing corresponding outputs. Each reaction is triggered by a hypothetical (logical) *global clock*. At each global tick, the threads in the program sample the environment, perform their computations, and emit their results to the environment. When a thread completes its computation, we say that the thread has completed its *local tick*. When all threads in the program have completed their local tick, we say that the program has completed its *global tick*. Central to synchronous languages is the *synchrony hypothesis* [Benveniste et al. 2003], which states that the execution of each reaction is considered to be atomic and instantaneous. The sampling of inputs avoids the need to use interrupts which are sources of unpredictable delays that degrade the system's timing predictability. Concurrent threads communicate instantaneously with each other (dashed arrows in Figure 2) due to the synchrony hypothesis. Once the embedded system is implemented, the synchrony hypothesis has to be validated. That is, the worst-case execution time [Wilhelm et al. 2008] of any global tick must not exceed the minimal inter-arrival time of the inputs.

We use the Esterel synchronous language [Berry and Gonthier 1992] to illustrate some features of the synchronous paradigm in Figure 3a. It contains two threads (starting from lines 4 and 7 respectively), scoped between the square brackets and separated by the parallel operator `||` (line 6). The parallel operator is commutative and associative and specifies that both threads are executed concurrently. The execution of a thread can be divided over multiple global ticks with the pause statement (e.g., lines 5, 8, and 11). The pause statement *pauses* the execution of its enclosing thread,

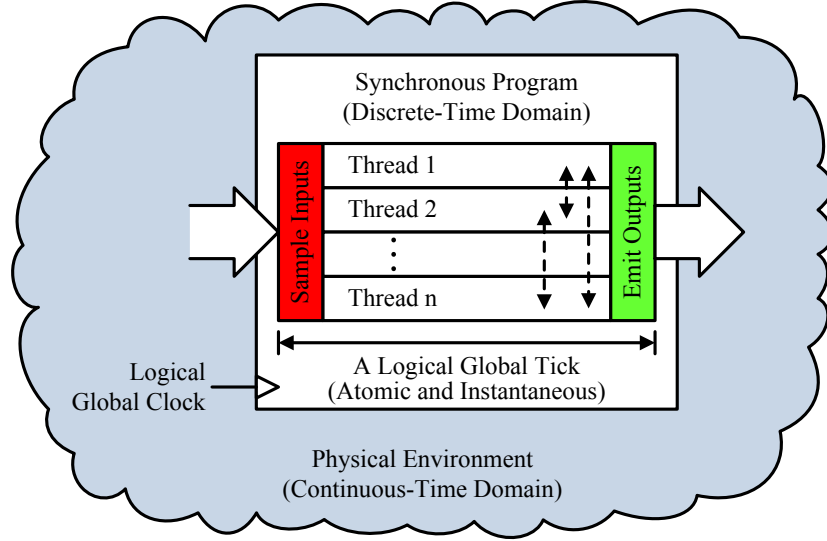


Fig. 2. Synchronous model of computation.

```

1 module main:
2   signal A, B
3   [
4     emit A;
5     pause
6   ]
7   present A then emit B end;
8   pause;
9   abort
10    emit B;
11    pause;
12    emit A; emit B
13    when immediate A
14  ]
15 end module

```

(a) Esterel

```

1 int A = 0, B = 0;
2 thread main(void) {
3   PAR(t0, t1);
4 }
5 thread t0() {
6   A = 1;
7   EOT;
8 }
9 thread t1() {
10  if (A == 1) { B = 1; }
11  EOT;
12  abort {
13    B = 1;
14    EOT;
15    A = 1; B = 1;
16  } when (A == 1);
17 }

```

(b) PRET-C

Fig. 3. Examples of synchronous programs.

demarcating the end of the thread's local tick. All executing threads must pause or terminate to complete the global tick. Thus, the pause acts as a synchronization barrier. At the next global tick, the threads resume from their respective pauses. In Esterel, threads communicate by *emitting* signals and threads can test for their *presence* or *absence*. For example, line 2 declares two signals, A and B, that are emitted by the `emit` statement when execution reaches lines 4, 7, 10, and 12. An emitted signal lasts until the global tick ends, becoming absent in the following global tick unless it is emitted again. Note that an emitted signal is logically present from the start of the global tick to ensure that all the threads see the same signal statuses, even if the `emit` statement occurs later in the global tick. In the first global tick, the first thread emits the signal

A. At the same time, the second thread tests positively for the presence of A and emits B. Using the abort statement, a body of code can be *preempted* by the presence of a signal. Preemption provides a convenient way to model the transitions and states of a state machine. In the second global tick of the example program, the second thread enters an abort (line 9) that preempts its body (lines 10–12) if A is present. Because A is not present, the body is not preempted and B is emitted. Meanwhile, the first thread terminates because it has reached the end of its body.

Synchronous programs are considerably difficult to parallelize [Girault 2005; Ju et al. 2010; Yuan et al. 2011] due to the need to resolve instantaneous thread communication and associated causality issues. At runtime, all potential signal emitters must be executed before all testers of a signal. If this is not possible, then a causality issue arises. Thus, concurrency is typically *compiled away* to produce only sequential code [Edwards and Zeng 2007]. The common approach for parallelizing synchronous programs is to automatically parallelize an intermediate representation of the program [Girault 2005; Ju et al. 2010; Yuan et al. 2011; Baudisch et al. 2010b]. The techniques differ in the heuristics used to partition the program to achieve sufficient parallelism.

Esterel only supports basic data computations and delegates complex data computations to a host language, for instance C. Consequently, C-based synchronous languages have been developed to provide data handling at the language level. These languages extend C with a range of synchronous constructs to support concurrency, preemption, and thread communication. C-based synchronous languages appeal to C programmers because the learning barrier for synchronous languages is reduced. PRET-C [Andalam et al. 2014] is one such example and Figure 3b is the PRET-C version of the Esterel example (Figure 3a). The two threads (t_0 and t_1 , defined on lines 5 and 9) are arguments to the parallel operator PAR (line 3). The EOT statement demarcates the end of a thread's local tick (e.g., lines 7, 11, and 14). Unlike Esterel, threads in the PAR's argument are executed in a left-to-right (static) order. The thread's local tick must be executed entirely before the next thread can be executed. In the example program, t_0 always executes its local tick before t_1 . In PRET-C, threads communicate using globally declared C-variables, not signals. Because threads are always executed in a static order, the local ticks always execute in a mutually exclusive manner. Hence, threads can safely access shared variables without needing to use mutual exclusion constructs; all PRET-C programs are thread-safe by construction. In the first global tick of Figure 3b, t_0 executes first and assigns 1 to the shared variable A (line 6). Then, t_1 executes and checks the condition $A==1$, which is *true*, and assigns 1 to B. PRET-C does not suffer from causality issues because global variables are always present and variables are always accessed sequentially (within a thread and across threads). PRET-C supports preemption with the abort statement but its behavior differs from Esterel's abort. Preemption occurs when the associated C-condition evaluates to *true*. The condition is always checked before the abort body is executed. In the second global tick of the example program, t_0 terminates because it has reached the end of its body. Then, t_1 executes and the condition $A==1$ (line 16) is checked before the abort body (lines 13–15) is executed. The condition is *true*, so the body is preempted. Execution jumps to line 15 and t_1 terminates.

Other C-based synchronous languages exist, such as Synchronous C [von Hanxleden 2009] and Esterel C Language [Lavagno and Sentovich 1999], and are reviewed in Section 2. However, these languages are not designed to take advantage of parallel execution. This paper focuses on developing a C-based, synchronous language for writing parallel programs that perform well on multi-core processors and are amenable to static timing analysis.

1.3. Programming Safety-Critical Embedded Systems

Safety-critical embedded systems need to be certified against stringent safety standards, such as DO-178B [Radio Technical Commission for Aeronautics 1992] or IEC 61508 [International Electrotechnical Commission 2010], before they can be deployed and used in the field. Although the C language is popular for programming safety-critical embedded systems, its semantics [ISO/IEC JTC1/SC22/WG14 2011] includes unspecified and undefined behaviors [Koenig 1988]. Strict coding guidelines [Motor Industry Software Reliability Association 2013; Holzmann 2006; Jet Propulsion Laboratory 2009] are typically used by safety-critical programmers to help write well defined programs that are deterministic, understandable, maintainable, and easier to debug [Gebhard et al. 2011; Hatton 2004]. The coding guidelines can be grouped into three main areas :

Code clarity. These guidelines suggest a style for writing programs free of ambiguous statements and to structure code for readability. For example, the use of braces to clarify the nesting of if–else statements or the forbidding of goto statements. Code clarity helps static analyzers parse the program and attain greater analysis precision.

Defensive programming. These guidelines help minimize the use of unspecified and undefined behaviors, which contribute to non-determinism. For example, the C semantics does not specify the evaluation order of multiple expressions, e.g., in function arguments. Thus, function arguments with side-effects may evaluate to different values depending on the evaluation order used by the implementation. To ensure deterministic evaluation [Norrish 1999], expressions must not contain any assignment operators, e.g., “=”, “+=”, or “++”. Furthermore, the sequencing operator “,” must not be used in expressions.

Runtime reliability. These guidelines help prevent runtime errors from occurring, even when the program is written correctly. For example, a runtime error occurs when a program requests for more memory than is available in the implemented system. To prevent it, memory is always allocated statically at the start of the program. Static verification tools, such as Parasoft [Parasoft 2014], Polyspace [Polyspace 2014], and Parallel Lint [Karpov 2011], can be used to identify possible runtime defects.

1.4. Contributions

We propose the ForeC parallel programming language for simplifying the deterministic parallel programming of embedded multi-core systems. Execution platforms have evolved from single-cores to multi-cores. Hence, all the synchronous languages designed for the single-core era must be reinvented to address the multi-core challenges. 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. A key innovation is ForeC’s shared variable semantics that provides thread isolation and deterministic thread communication. Moreover, many forms of parallel patterns can be expressed in ForeC. We show that ForeC programs are reactive and deterministic by construction. ForeC can be compiled for direct execution on embedded multi-cores or for execution by an OS on desktop multi-cores. Through benchmarking, we demonstrate that ForeC can achieve better parallel performance than Esterel and OpenMP, while also being amenable to static timing analysis.

1.5. Paper Organization

This paper is organized as follows. Section 2 provides a detailed literature review of parallel and synchronous programming languages. Section 3 describes the multi-core architecture considered by this paper. Section 4 introduces the ForeC language, defines the formal semantics, and provides proofs for determinism and reactivity. Section 5 describes our compilation approach for generating code that delivers good parallel performance and that is amenable to static timing analysis. Section 6 presents benchmarking results for ForeC's performance on multi-cores and the time predictability of its execution. Section 7 concludes the paper.

2. RELATED WORK

Designing embedded systems that are time-predictable remains an open challenge [Azer et al. 2014]. Moreover, the growth of embedded multi-cores is pushing more programmers to be parallel programming experts. Table I highlights different approaches for enforcing *mutual exclusion* on shared variables, usually by interleaving the parallel accesses to enforce a sequence of accesses to the *critical sections*. As argued by Lee [Lee 2006], the adoption of parallelism in sequential languages, like C [ISO/IEC JTC1/SC22/WG14 2011], discards important properties, such as determinism, predictability, and understandability. Thus, programmers spend large amounts of time taming the non-determinism in their parallel programs [Lu et al. 2008]. Instruction reordering is regularly employed by compilers and processor cores to maximize execution parallelism, but this can cause wrong values for shared variables to be observed. C provides the programmer with memory fences to enforce a partial ordering on variable accesses between threads : all side-effects of a *releasing* thread are committed before the *acquiring* thread leaves the fence. To help tame non-determinism, runtime environments that enforce deterministic thread scheduling and memory accesses can be used. Such runtime environments have been developed for Linux processes (DPG [Bergan et al. 2010b]), Pthreads (Grace [Berger et al. 2009], Kendo [Olszewski et al. 2009], CoreDet [Bergan et al. 2010a], and Dthreads [Liu et al. 2011]), OpenMP (DOMP [Aviram and Ford 2011]), and MPI (DetMP [Zhang and Ford 2011]). For DPG, Kendo, CoreDet, and Dthreads, all thread interactions are 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) [Valiant 1990] programming model. In the parallel phase, threads execute in parallel until they all reach one of the following synchronization points : a lock, memory access, or statically defined number of executed instructions. Then, in the serial phase, threads take turns to resolve their memory accesses or lock acquisitions. Threads in CoreDet and Dthreads also maintain their own version of the shared memory state, which is resynchronized in every serial phase. This concept is used and formally defined in concurrent revisions [Burckhardt and Leijen 2011]. DOMP and Grace differ in that the resynchronization only occurs when threads reach a synchronization construct. However, understanding the program's behavior at compile time remains difficult because the 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. An alternative is to directly extend and modify the C language with deterministic parallelism, such as SharC [Raman et al. 2010], CAT [Flanagan and Qadeer 2003], SHIM [Tardieu and Edwards 2006], Σ C [Goubier et al. 2011], and ForkLight [Kessler and Seidl 1999]. These solutions allow the asynchronous forking and synchronized joining of threads, but lack a convenient mechanism for preempting groups of threads. However, their timing predictability has not been demonstrated, which is required for programming safety-critical embedded systems.

Table I. Existing solutions for avoiding race conditions.

Programming Constructs : These are constructs written in the host language to provide mechanisms for the programmer to achieve mutual exclusion. Examples include : locks, monitors, memory fences, transactional memory, message passing, and parallel data structures. Using these constructs correctly can be tedious and error prone for large programs and may lead to other errors [Lee 2006; McDowell and Helmbold 1989; Lu et al. 2008], e.g., deadlocks, starvation, or priority inversion.
Language Semantics : The language semantics can have a memory model that defines how threads interact through memory, what value a read can return, and when the value of a write becomes visible to other threads. Although the memory model can prevent race conditions, it may only be suitable for a few types of applications. Examples include : synchronous languages [Benveniste et al. 2003], PRET-C [Andalam et al. 2014], Synchronous C [von Hanxleden 2009], SharC [Anderson et al. 2008], Deterministic Parallel Java [Bocchino 2010], SHIM [Vasudevan and Edwards 2009], Σ C [Goubier et al. 2011], concurrent revisions [Burckhardt and Leijen 2011], and Reactive Shared Variables [Boussinot 1993].
Static Analysis : A compiler or static analyzer can identify and alert the programmer regarding the race conditions in their program (e.g., Parallel Lint [Karpov 2011]) and may try to resolve them by serializing the parallel accesses for the programmer (e.g., Sequentially Constructive Concurrency [von Hanxleden et al. 2013]). However, programmer guidance is needed for race conditions that cannot be resolved.
Runtime Support : Programs are executed on a runtime layer that dynamically enforces deterministic execution and memory accesses. Examples include : dOS [Bergan et al. 2010b], Grace [Berger et al. 2009], Kendo [Olszewski et al. 2009], CoreDet [Bergan et al. 2010a], Dthreads [Liu et al. 2011], DOMP [Aviram and Ford 2011], and DetMP [Zhang and Ford 2011]. However, understanding the program's behavior at compile time remains difficult because the determinism is only enforced at runtime.
Hardware Support : Parallel accesses can be automatically detected and resolved by the hardware, preventing race conditions from happening. Examples include : Ultracomputer's combine hardware [Schwartz 1980] and certain shared bus arbitration (e.g., round-robin, TDMA, and priority). However, the timing of the parallel accesses affects how they are interleaved.

The classic synchronous languages are Esterel [Berry and Gonthier 1992], Lustre [Halbwachs et al. 1991], Signal [Guernic et al. 1991], and the recent extension based on functional programming such as Lucid Synchrone [Colaço and Pouzet 2003], and are well suited to the modeling of control-dominated systems [Caspi and Maler 2005] and safety-critical systems [Benveniste et al. 2003]. To increase their uptake with embedded programmers, C-based synchronous languages have been developed, such as Reactive Shared Variables [Boussinot 1993], Esterel C Language (ECL) [Lavagno and Sentovich 1999], PRET-C [Andalam et al. 2014] and Synchronous C (SC) [von Hanxleden 2009; von Hanxleden et al. 2013]. The inherent sequential execution semantics of SC, Reactive Shared Variables, and PRET-C renders them unsuitable for multi-core execution. Moreover, concurrency in synchronous languages is a logical concept to help the programmer handle concurrent inputs, rather than a specification for parallel execution. Thus, compilers typically generate only sequential code [Edwards and Zeng 2007; Potop-Butucaru et al. 2007], although some generate concurrent tasks [Caspi et al. 2008; Natale et al. 2010; Pagetti et al. 2011; Natale and Zeng 2012] for execution on single-cores. Yuan et al. [Yuan et al. 2011; Yuan 2013] offer a static and dynamic scheduling approach for Esterel on multi-cores. For the static approach, threads are statically load-balanced across the cores and signal statuses are resolved at runtime. For the dynamic approach, threads that need to be scheduled for execution are inserted into a custom hardware queue accessible to all cores. The dynamic approach has been shown to provide better average-case performance compared to the static approach [Yuan 2013]. This is because the static approach uses worst-case execution times to load-balance the threads, even though the actual execution times may be shorter.

The common approach for parallelizing synchronous programs is to parallelize an intermediate representation of the sequentialized code [Girault 2005; Ju et al. 2010; Yuan et al. 2011; Baudisch et al. 2010a; Caspi et al. 1999; Yoong et al. 2006; Papailiopoulos et al. 2011]. Multi-threaded OpenMP programs can be generated from the

Synchronous Guarded Actions intermediate format [Baudisch et al. 2010b]. The techniques differ in the heuristics used to partition and distribute the program to achieve sufficient parallelism. The Synchronized Distributed Executive (SynDEX) [Potop-Butucaru et al. 2010] approach considers the cost of communication when distributing code to each processing element. When distributing a synchronous program, some desynchronization [Benveniste et al. 2000; Girault et al. 2006; Brandt et al. 2009] 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. The use of *futures* has been proposed as a method for desynchronizing long computations in Lustre [Cohen et al. 2012]. A *future* is a proxy for a result that is initially unknown but becomes known at a later time and can be computed in parallel with other computations.

Once a synchronous program is implemented, it is necessary to validate the synchrony hypothesis. That is, the worst-case execution time [Wilhelm and Grund 2014; Wilhelm et al. 2008] (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 [Boldt et al. 2008; Mendler et al. 2009] and various techniques have been developed for single-cores [Mendler et al. 2009; Ju et al. 2012; Wang et al. 2013; Raymond et al. 2013; Closse et al. 2001; Andalam et al. 2011; Boldt et al. 2008; Kuo et al. 2011] and multi-cores [Ju et al. 2010; Yip et al. 2013].

2.1. Discussion

This section has presented a snapshot of the current efforts in the programming of time-predictable CPSs. Many of the attempts at providing deterministic parallelism have used concepts found in synchronous languages. C-based synchronous languages have much to offer to embedded programmers in terms of deterministic concurrency and formally verifiable implementations, but lack support for parallel execution. This paper tackles the lack of a C-based synchronous parallel programming language that offers *both* time-predictability and good parallel execution performance.

3. MULTI-CORE ARCHITECTURE

Embedded systems continue to explode in complexity and functionality [Kopetz 2008]. To meet the size, weight, and power (SWaP) concerns, the advent of affordable embedded multi-core processors [Blake et al. 2009; Sodan et al. 2010] offer designers the opportunity to achieve better performance than single-core processors. Figure 4 illustrates the architecture of a general-purpose multi-core. In pursuit of increasing average-case performance, the cores typically include speculative features [Patterson and Hennessy 1990] such as out-of-order execution, branch prediction, data forwarding, superscalar execution, and on-chip caches. However, such optimizations can cause *timing anomalies* [Lundqvist and Stenstrom 1999] where a local worst-case execution time does not lead to the program's worst-case execution time. Thus, speculation leads to the degradation of time-predictability and is undesirable for embedded systems. The PREcision Timed (PRET) machine [Edwards and Lee 2007; Edwards et al. 2009] and PRedictability Of Multi-Processor Timing (PROMPT) [Cullmann et al. 2010; Kastner et al. 2012] design philosophies aim to tackle this issue by advocating the design of *predictable hardware architectures*, while not sacrificing performance. In particular, the architecture should provide *timing isolation* between the cores, i.e., the actions of the cores must not influence each other's timing behavior. The architecture should be *timing compositional*, i.e., with repeatable timing behavior and free of timing anomalies. The following are examples of unpredictable hardware features with possible predictable alternatives [Wilhelm et al. 2009; Obermaisser et al. 2009; Bekooij et al. 2004] :

- *Replace caches with fast software managed memories, called scratchpads* [Whitham and Audsley 2009]. The selection of data and instructions to be allocated to a scratchpad is determined entirely at compile time [Udayakumaran et al. 2006; Suhendra et al. 2008; Kim et al. 2014; Prakash and Patel 2013; Nguyen et al. 2005]. In the static allocation scheme, the contents of the scratchpad cannot be changed at runtime. In the dynamic allocation scheme, the contents of the scratchpad can be changed at runtime by using compile time decisions. Importantly, the replacement policy of scratchpads is controllable, whereas with caches the replacement policy is controlled by the hardware, sometimes with unpredictable behaviors (e.g., with the PLRU policy). The memory address spaces of scratchpads and global memory are mutually exclusive.
- *Replace out-of-order execution with better code generation from the compiler* [Plazar et al. 2008; Schranzhofer et al. 2011; Ding and Zhang 2011; Ding et al. 2013; Ungerer et al. 2010]. A processor's ability to reorder a group of instructions is limited by the size of its instruction buffer. The compiler does not have this limitation because it has access to the entire program and can make better judgments when reordering instructions. However, it may not have the runtime execution information which may affect the performance.
- *Deactivate high-performance bus features, such as burst transfers or pipelining, and use fair time-sharing arbitration policies, such as round-robin or time division multiple access (TDMA)* [Shah et al. 2011]. The round-robin policy cycles through a static list of cores, granting them access to the bus. If the granted core does not need the bus, then the grant is given to the next core on the list. The TDMA policy cycles through a static list of cores, granting them access for a fixed amount of time (a time slot), whether the core needs it or not. If the granted core does not need the bus, then some policies [Shah et al. 2011; Andrei et al. 2008; Lahiri et al. 2001; Poletti et al. 2003; Hamann and Ernst 2005] will grant the slot to the other cores in a round-robin manner, thus improving the throughput. Fairness of the arbitration is important to ensure that all accesses complete within a bounded length of time.

Embedded systems designed using the PRET [Edwards et al. 2009] or PROMPT [Kastner et al. 2012] philosophies are simpler to understand, model, and analyze. Many predictable single-core processors have been proposed, such as the MACS [Cogswell and Segall 1991], MCGREP [Whitham and Audsley 2006], Patmos [Schoeberl et al. 2011], PTARM [Liu et al. 2012], and FlexPRET [Zimmer et al. 2014] processors. MERASA [Ungerer et al. 2010] is a predictable multi-core processor that supports hard and non-real-time threads. Hard real-time threads access scratchpads for predictability, while non-real-time threads access caches for performance. An analyzable memory controller is used to arbitrate shared bus accesses from the cores. For Java programs, there is the JOP [Schoeberl 2009] processor and its multi-core variant [Puffitsch and Schoeberl 2012] that uses scratchpads and a shared TDMA bus.

The execution of synchronous programs can be accelerated by *reactive processors* [Salcic et al. 2002; Li and von Hanxleden 2012], which have hardware support for signal resolution, concurrency, preemptions, and global tick synchronization. A key feature is their ability to execute programs in a time predictable manner. Single-core multi-threaded reactive processors include KEP [Li and von Hanxleden 2012] and STARPro [Yuan et al. 2009]. Reactive multi-processors include EMPEROR [Dayaratne et al. 2006] and HiDRA [Salcic et al. 2006]. However, these reactive processors and associated compilers do not support the execution of host functions written in a host language, such as C. As a compromise between efficiency and host language support, a general purpose processor can be patched with a reactive functional unit to accelerate the execution of synchronous constructs. The ARPRET [Andalam et al. 2010] processor

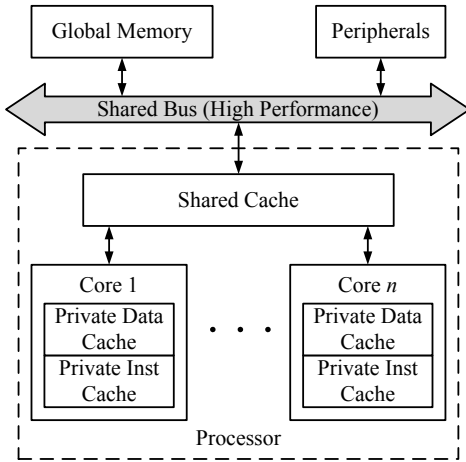


Fig. 4. General multi-core architecture.

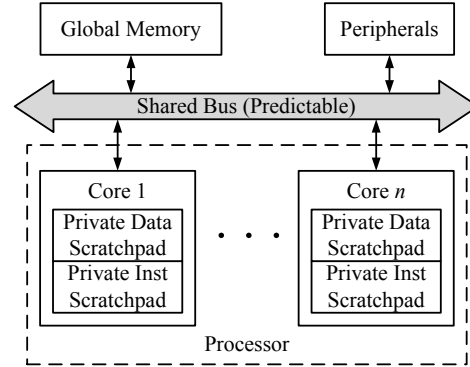


Fig. 5. Example of a predictable multi-core embedded architecture.

is a patched Xilinx MicroBlaze [Xilinx 2012] processor tailored for executing PRET-C. Java-based reactive single-core processors include RJOP [Nadeem et al. 2011] and TPJOP [Li et al. 2014]. GALS-HMP [Salcic and Malik 2013] is a Java-based reactive multi-processor.

3.1. Predictable Embedded Multi-Core Architecture

The architecture of the predictable multi-core used in this paper is representative of existing designs. It is a homogeneous multi-core processor [Cullmann et al. 2010; Schoeberl 2009] that we have designed using identical Xilinx MicroBlaze [Xilinx 2012] cores, illustrated in Figure 5. Each MicroBlaze core has 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. Due to the resource constraints of existing FPGA devices, we developed a multi-core MicroBlaze simulator for benchmarking purposes. We extended an existing MicroBlaze simulator [Whitham 2012] significantly to support cycle-accurate simulation, an arbitrary number of cores, and a shared bus with TDMA arbitration.

4. THE FOREC LANGUAGE

Execution platforms have evolved from single-cores to multi-cores. Hence, all the synchronous languages designed earlier (e.g., Esterel [Berry and Gonthier 1992], Lustre [Halbwachs et al. 1991], Signal [Guernic et al. 1991], Esterel C Language [Lavagno and Sentovich 1999], Reactive Shared Variables [Boussinot 1993], and PRET-C [Andalam et al. 2014]) must be remodeled to address the challenges raised by multi-cores. Over 30 years of synchronous programming languages have demonstrated that they are very well suited to the design of safety-critical real-time systems [Brière et al. 1994; Souyris et al. 2005]. 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 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 in providing deterministic shared

Statements :	st	$:= c_st \mid \text{pause} \mid \text{par}(st, st)$ $\mid \text{weak? abort } st \text{ when immediate?}(exp)$ $\mid st; st$
Type Qualifiers :	tq	$:= c_tq \mid \text{input} \mid \text{output} \mid \text{shared}$

Fig. 6. Syntactic extensions to C.

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. The formal semantics of ForeC is then detailed and important proofs concerning program reactivity and determinism [Maraninchi 1992; Tardieu 2007] are provided.

4.1. Overview and Syntax

ForeC is a synchronous language that extends a safety-critical subset of C [Blazy and Leroy 2009; Jim et al. 2002] (see Section 1.3) with a minimal set of synchronous constructs. We briefly describe the statements, type specifiers, and type qualifiers allowed in the C subset :

C statements (c_st). Expressions in a statement can only be constants, variables, 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 [Buss et al. 2010] because of pointer aliasing. Thus, we assume that pointers are never reassigned to point to other variables. All C control statements, except `goto`, can be used. These are the selection statements (`if-else` and `switch`) and loop statements (`while`, `do-while`, and `for`).

C type specifiers. 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.

C storage class specifiers. The C `typedef`, `extern`, `static`, `auto`, and `register` specifiers can be used.

Figure 6 gives the extended syntax of ForeC and Table II summarizes the informal semantics. A statement (st) in ForeC can be a traditional C statement (c_st), or a barrier (`pause`), fork/join (`par`), or preemption (`abort`) statement. Using the sequence operator (`;`), a statement in ForeC can be an arbitrary composition of other statements. Like C, extra properties can be specified for variables using type qualifiers. A type qualifier (tq) in ForeC is a traditional C type qualifier (c_tq), an environment interface (`input` and `output`), or a shared variable amongst threads (`shared`). The `input`, `output`, and `shared` type qualifiers precede the C type qualifiers in variable declarations.

As a running example to illustrate the ForeC language, we describe the design of an unmanned aerial vehicle (UAV) inspired by the Paparazzi project [Nemer et al. 2006]. A UAV is a remotely controlled aerial vehicle commonly used in surveillance operations. Figure 7 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 paral-

Table II. 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.
pause : Pauses the executing thread until the next global tick.
par (<i>st</i> , <i>st</i>) : Forks two statements <i>st</i> as parallel threads. The par terminates when both threads terminate (join back).
weak ? abort <i>st</i> when immediate ? (<i>exp</i>) : Preempts its body <i>st</i> when the expression <i>exp</i> evaluates to a non-zero value. The optional weak and immediate keywords modify its temporal behavior.

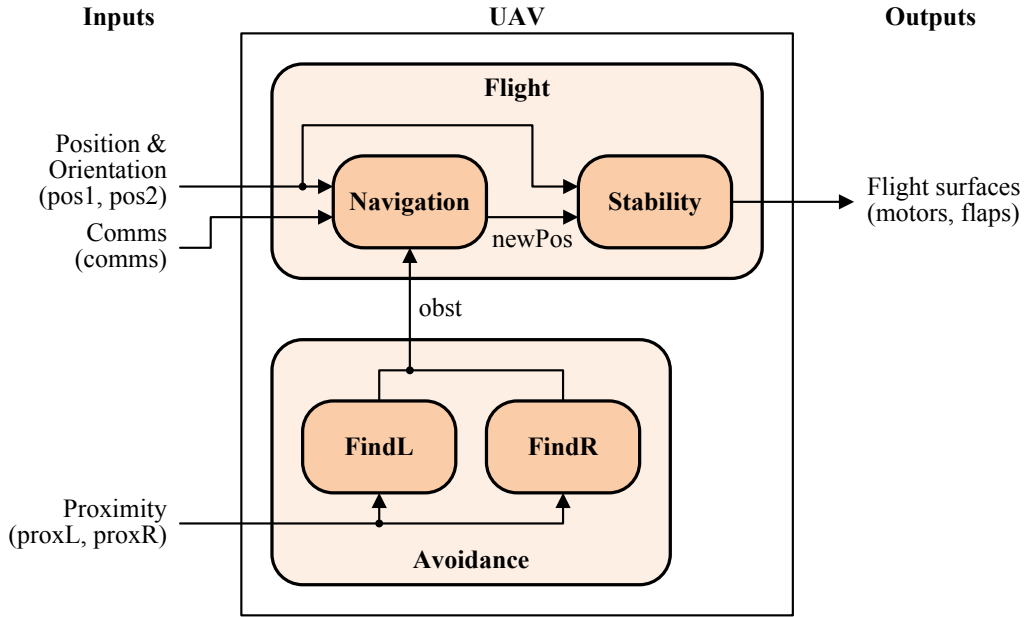


Fig. 7. Tasks of the UAV.

lel 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 8 is a ForeC implementation of the UAV example given in Figure 7. Figure 9 is a possible execution trace of Figure 8 to help illustrate the execution of ForeC programs. Sections 1.2 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 9, the first three global ticks are demarcated along the left-hand side.

In Figure 8, 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

```

1  #include <uav.h>
2  input int pos1, pos2, proxL, proxR; // Inputs.
3  output int motors=0, flaps=0; // Outputs.
4
5  void main(void) {
6      shared int obst=0 combine new with min;
7      shared int newPos=0 combine new with plus;
8      par(Flight(&newPos, &obst), Avoidance(&obst));
9  }
10
11 void Flight(shared int *newPos, shared int *obst) {
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(
33             {*obst=find(proxL);}, // Thread FindL.
34             {*obst=find(proxR);} // Thread FindR.
35         );
36         pause;
37     }
38 }
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. 8. Example ForeC program for the UAV running example.

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 9 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. Lines 6–7 declare variables that can be shared amongst threads (see Section 4.1.3). In Figure 9, the states of the shared

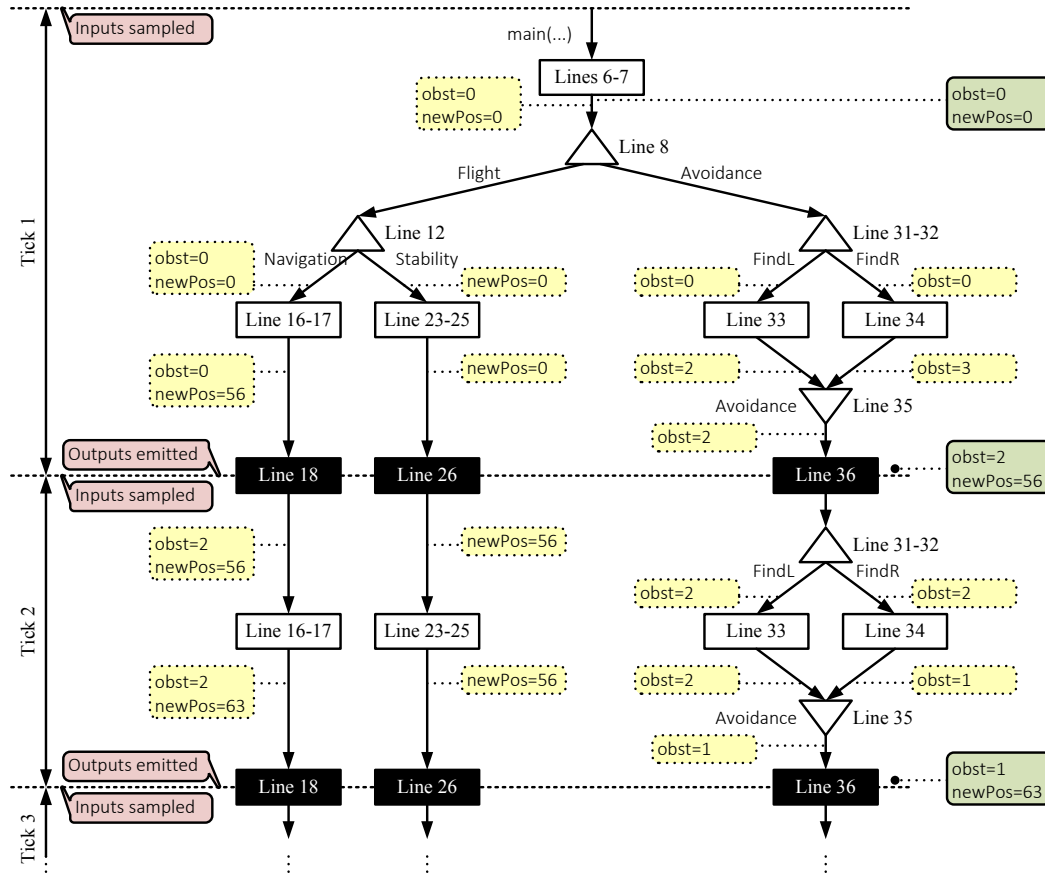


Fig. 9. Possible execution trace for Figure 8.

variables are given inside solid round boxes at specific points in the execution trace. Line 6 declares a shared variable *obst* to store the distance and angle of the closest obstacle as an encoded integer. Line 7 declares a shared variable *newPos* to store the UAV's desired position.

On line 8, the *par* statement forks the *Flight* (line 11) 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 7 as triangles. On 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* statement can also fork blocks of code, e.g., line 32 forks the *FindL* and *FindR* threads. The *par* is a blocking statement and terminates only when both its child threads have terminated and joined together. The joining of threads is represented in Figure 9 as inverted triangles.

After the *Navigation*, *Stability*, *FindL*, and *FindR* threads have forked, they start executing their respective body. For example, the *Navigation* thread enters the while-loop (line 16) and computes a new desired position. Next, the pause statement *pauses* the thread's execution (line 18), acting as a synchronization barrier. In Figure 9, the pause statements are shown as black rectangles and the program completes a global

tick when all the threads pause. This is indicated by the dotted horizontal lines 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 others, yielding mutual exclusion and thread isolation. Moreover, only sequential reasoning is needed within a thread's local tick. In Figure 9, 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 8, the combine function for `obst` (line 6) is `min` (line 40), specified by the `combine` clause, which returns the closest obstacle. 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 9, the `FindL` and `FindR` threads set new values (2 and 3) to their copies of `obst`. When these threads join, the new values are combined to 2 and assigned to their parent thread `Avoidance`. 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 9, the resynchronized values are shown 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 A 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.

4.1.1. Local and Global Ticks. We say that a thread completes its *local tick* when it pauses, terminates, or forks at least one thread that completes its local tick without terminating. For example, in Figure 8, 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, both child threads complete their local tick by terminating. After the child threads join, the `Avoidance` thread reaches a pause (line 36) and completes its first local tick. A program completes its *global tick* when all its threads have completed their respective local ticks. At the next global tick, the paused threads start their next local tick from their respective pauses. For brevity, we shorten “global tick” into “tick” and use “local tick” as before.

4.1.2. 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 their nested child threads. Child threads forked by the same `par` statement are *siblings*. Because the `par` is a blocking statement, threads always execute sequentially with respect to their ancestors. Threads that are not ancestors of each other are *relatives* and can execute in parallel.

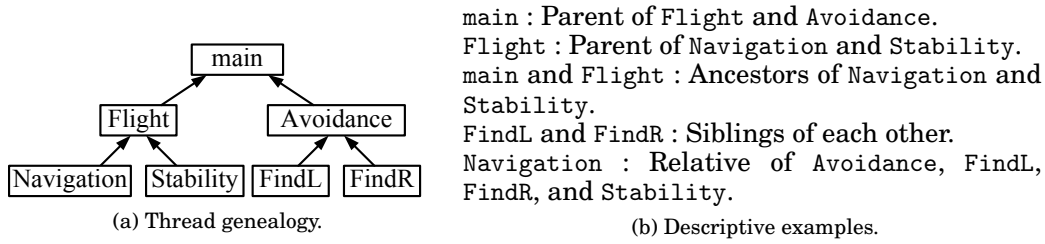


Fig. 10. Thread genealogy for Figure 8.

The thread genealogy of a program can be determined statically by inspecting the program's control-flow. Figure 10a 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 10b exemplifies the thread genealogy.

4.1.3. 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. Thus, any misuse of private variables are easy to detect at compile time. Appendix A.1 describes how shared variables are passed by value and by reference into functions. The semantics we propose for ForeC makes sure that the shared variables can be safely accessed by the parallel threads without the need of mutual exclusion constructs. The goal is to provide a deterministic shared variable semantics that is agnostic to scheduling, which is essential for the design and debug of parallel programs. Within each tick, the accesses to a shared variable from two threads may occur in sequence or in parallel :

Definition 4.1. Accesses from two threads are in *sequence* if both threads are not relatives or if the accesses occur in different ticks.

Definition 4.2. Accesses from two threads are in *parallel* if both threads are relatives and the accesses occur in the same tick.

Improperly managed parallel accesses to a shared variable can cause race conditions, leading to non-deterministic behavior. For example, two parallel writes to a shared variable can non-deterministically and partially overwrite each other's value. 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 I in Section 2 reviewed the solutions that exist for enforcing mutual exclusion on shared variables, usually by interleaving parallel accesses into a sequence. 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 of the model are :

Isolation. 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 [Maraninchi 1992]. Ensure deterministic execution regardless of scheduling decisions. This guarantees that deterministic outputs are always generated at the end of each tick.

Parallelism. 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.

We propose the following mechanisms for achieving our shared memory model : 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.

4.1.4. 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 9, 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) into the child threads (e.g., *obst* on line 8 of Figure 8). When a shared variable is passed by reference into an ordinary function (e.g., *obst* on line 17), the function uses the calling thread's copy of the shared variable.

4.1.5. Resynchronization of Shared Variables. The copies are *resynchronized* every time the program completes its tick (before outputs are emitted). 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. That is, the combine function produces the same outputs from the same inputs, regardless of previous invocations and how the copies are ordered or grouped. The signature of any combine function 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. For example, in Figure 9, the *Avoidance* thread gets a copy of *obst* every time *FindL* and *FindR* terminate. Appendix A 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*. The combine policy of a shared variable is specified during variable declaration 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., which has 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.¹ The default policy is *all* where no copies are ignored. Note that the combine function is not invoked when only one copy remains. Instead, that copy becomes the resynchronized value. Appendix A provides extensive illustrations comparing the behavior of the combine policies.

4.1.6. Hierarchical Preemption. Inspired by Esterel [Berry and Gonthier 1992], the *abort st when (exp)* statement provides preemption [Berry 1993], which is the termination of the *abort* body *st* when the condition *exp* evaluates to *true*. Preemption can be used to model state machines succinctly. The condition *exp* must be a side-effect

1. This differs from the *new* policy because an assignment of the form "*x*=*x*" will be taken into account by the *mod* policy, but not by the *new* policy.

Expressions :	$exp ::= val \mid var \mid ptr[exp] \mid (exp)$	// Constants, variables, and grouping.
	$\mid u_op \ exp \mid exp \ b_op \ exp$	// Unary and binary expressions.
Unary		
Operators :	$u_op ::= * \mid \& \mid ! \mid - \mid \sim$	// Indirection, address, negation, negative, and one's complement.
Binary		
Operators :	$b_op ::= \mid \mid \mid \&\& \mid \sim \mid \mid \mid \& \mid << \mid >>$	// Logical and bitwise operators.
	$\mid == \mid != \mid < \mid > \mid <= \mid >=$	// Relational operators.
	$\mid + \mid - \mid * \mid / \mid \%$	// Arithmetic operators.

Fig. 11. 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      shared int newPos=0 combine new with plus;
7      abort {
8          par( Flight(&newPos,&obst) ,Avoidance(&obst));
9      } when (comms==TERM);
10     safeDescent();
11 }

```

Fig. 12. Figure 8 extended with preemption.

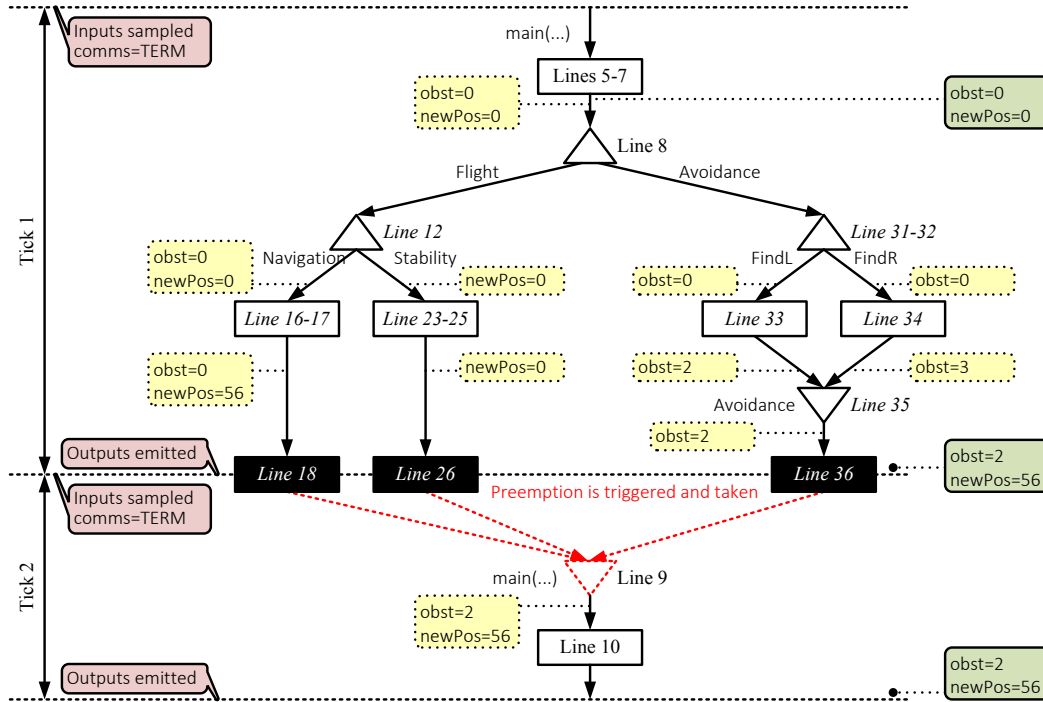


Fig. 13. Possible execution trace for Figure 12.

free expression produced from the syntax shown in Figure 11. In Figure 12, 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 7 preempts the execution of all the UAV tasks when `TERM` is received. A possible execution trace of the program of Figure 12 is given in Figure 13. The *italicized* line numbers in Figure 13 refer to the line numbers in Figure 8, while the non-italicized line numbers refer to the line numbers in Figure 12. We now explain the semantics of the abort statement. 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 13). At the start of each subsequent tick, the condition *exp* is evaluated before the abort body can execute. This allows shared variables in the condition to be evaluated with their resynchronized value. If *exp* 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 13, preemption is triggered because the preemption condition evaluates to *true*. 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 rather than shared variables. As explained in Section 1.2, signals in Esterel are either present or absent in each tick and this information is propagated instantaneously among the threads without delay. Thus, preemptions in Esterel are triggered instantaneously, whereas preemptions in ForeC are triggered with a delay of one tick because the condition *exp* is evaluated using values computed in the previous tick. Like Esterel [Berry 1993], the optional *weak* and *immediate* keywords change the temporal behavior of preemptions. The *weak* keyword delays the termination of the abort body until the body cannot execute any further, e.g., reaches a pause statement. The *immediate* keyword allows preemption to be triggered immediately when execution reaches the abort for the first time. That is, the preemption condition *exp* is evaluated immediately when execution reaches the abort. This is similar to Esterel's *immediate* abort behavior. To illustrate these four different preemption behaviors, Figure 14a presents an abort with the optional keywords commented out.

Non-immediate and strong abort. The *weak* and *immediate* keywords are commented out in Figure 14a. This gives the default preemption behavior, summarized in Figure 14b. 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. Only the *weak* keyword is uncommented in Figure 14a. Figure 14c summarizes the preemption behavior. 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. Only the *immediate* keyword is uncommented in Figure 14a. Figure 14d summarizes the preemption behavior. 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 *exp* is evaluated immediately with the thread's copies of the shared variables. In subsequent ticks, the resynchroni-

```

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” printed. $s = \text{plus}(2,5) = 7$.

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

(b) Non-immediate and strong abort.

Tick 1 : “1” printed. $s = \text{plus}(2,5) = 7$.

Tick 2 : Preemption is triggered.

$s = \text{plus}(3,6) = 9$. The abort body is terminated. “9” printed.

(c) Non-immediate and weak abort.

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

(d) Immediate and strong abort.

Tick 1 : “1” printed. Preemption is triggered. $s = \text{plus}(2,5) = 7$. The abort body is terminated. “7” printed.

(e) Immediate and weak abort.

Fig. 14. 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. 15. Nesting of preemptions.

zed values of the shared variables are used. In tick one of Figure 14d, because the preemption has been triggered, the abort body is terminated without executing.

Immediate and weak abort. Both the weak and immediate keywords are uncommented in Figure 14a. Figure 14e summarizes the preemption behavior. 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 t_0 and t_1 complete their local ticks. This allows t_0 and t_1 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 with the outer abort executing before the inner aborts. Thus, the preemption behavior of the outer abort takes precedence over the inner aborts. Figure 15 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,

Table III. 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}

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.

4.1.7. Bounded Loops. In addition to the strict C-coding guidelines described in Section 1.3, ForeC forbids the use of *unbounded recursion* of function calls and thread forking to ensure static WCRT analyzability. The synchrony hypothesis requires each tick to execute in finite time, which means that all statements need to have bounded execution times. Unfortunately, loop constructs (for and while) 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 possible when pause statements are enclosed by if-statements. 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. The compiler can perform structural induction on the program’s control-flow to conservatively prove whether every loop in the program will always execute a pause in each iteration.

Inspired by PRET-C [Andalam et al. 2014], we have extended the syntax of loops to also allow the programmer to write bounded loops, shown in the first column of Table III. The “#n” after the loop header specifies that only up to n iterations can be executed. The second column of Table III gives the structural translation of bounded loops.

4.2. Semantics of ForeC

This section presents the semantics of ForeC as rewrite rules in the style of structural operational semantics (SOS) [Plotkin 1981]. The semantics is inspired by that of other synchronous programming languages (Esterel [Potop-Butucaru et al. 2007] and PRET-C [Andalam et al. 2014] in particular). The semantics is defined on a set of primitive ForeC constructs (the kernel of Table IV) from which the full ForeC constructs are derived. The kernel constructs are not used for compiling and only consider a subset of the C language : the assignment operator (=), the statement terminator (;) for sequencing, and the if and while statements. Table V shows how the ForeC constructs (Table II) are translated into the kernel constructs (Table IV). This is exemplified by the translation of the ForeC constructs in Figure 16b into the kernel constructs in Figure 16c. The translations for input, output, and pause are straightforward. A shared variable is translated into a 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 semantics to

Table IV. ForeC kernel constructs.

Kernel Construct	Short Description
<code>nop</code>	Empty statement
<code>f; f</code>	Sequence operator
<code>var = exp</code>	Assignment operator
<code>while (exp) f</code>	Loop
<code>if (exp) f else f</code>	Conditional
<code>copy</code>	Creates copies of shared variables
<code>pause</code>	Barrier synchronization
<code>par(t :f, t :f)</code>	Fork/join parallelism
<code>status(a, exp)</code>	Initial preemption status
<code>weak? abort(a, f)</code>	Abort

Note : 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.

Table V. Structural translations of the ForeC constructs (Table II) to kernel constructs (Table IV).

ForeC Construct	ForeC Kernel Constructs
input and output	Translated into global variables.
shared	Translated into global variables and copy kernel statements that are placed at the start of every thread body.
pause	pause
par(f, f)	par($t :f, t :f$)
weak? abort f when (exp)	status($a, 0$); weak? abort(a, f)
weak? abort f when immediate (exp)	status(a, exp); weak? abort(a, f)

distinguish the body of one thread from another. The par kernel statement handles the resynchronization of the shared variables. Traditionally, traps [Potop-Butucaru et al. 2007] are used to translate aborts and other complex preemption statements. In 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 and it takes the unique identifier a and an expression. The expression is 0 (zero) for a non-immediate abort, but is exp (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.

4.2.1. Assumptions. We make the following assumptions about ForeC programs. (1) All programs follow safety-critical coding practices, as discussed in Sections 1.1 and 4.1. Dynamic memory allocation (e.g., malloc) and unstructured jumps (e.g., goto) cannot be used, and loops must be bounded. 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 must not contain any assignment operators. The sequencing operator “,” of C must not be used. These assumptions limit us to a deterministic subset of the C language. (2) All recursive function calls or forking of threads are bounded. This assumption prevents the unbounded execution of functions and threads, leading to unbounded memory use and execution time.

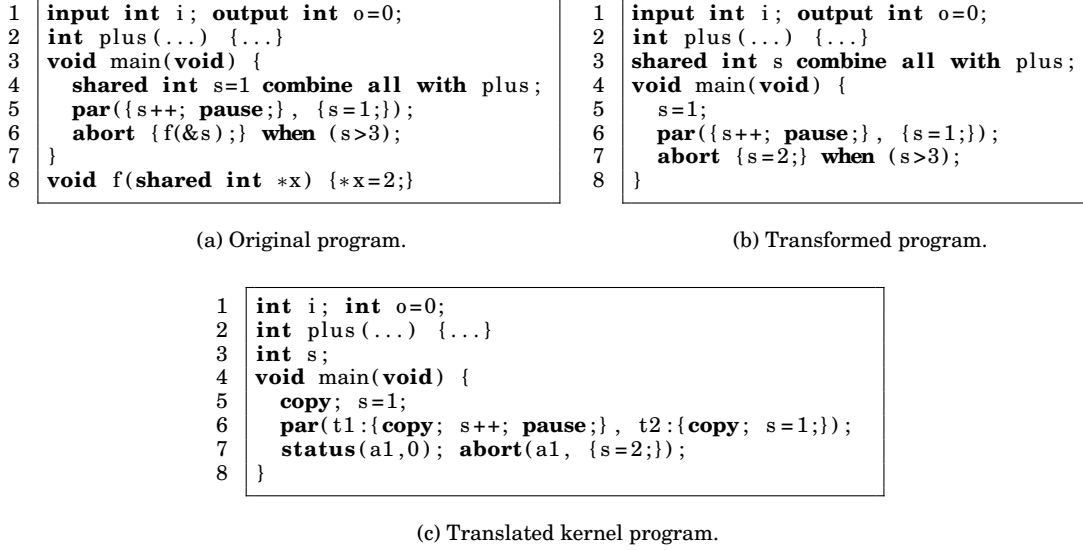


Fig. 16. Example of transforming and translating a ForeC program into the kernel constructs.

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 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 [Buss et al. 2010; Hardekopf and Lin 2011]. Consider the program of Figure 16a that is transformed into the equivalent program of Figure 16b. The shared variable declaration for *s* (line 4 in Figure 16a) is hoisted to the global scope (line 3 in Figure 16b). The function *f* (line 8 in Figure 16a) is inlined into the abort body (line 7 in Figure 16b) and the pointer inside *f* is replaced by the variable *x* it references.

4.2.2. Notation. The rewrite rules have the following form in the style of structural operational semantics (SOS) [Plotkin 1981] :

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

This notation describes a program fragment *f* belonging to thread *t*, in the program state *S* and with inputs *I*, which reacts and modifies the program state to *S'*, 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. Let $\langle S \rangle = \langle E, A \rangle$, 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 (denoted by \mathcal{G}) and threads ($t \in T$) to a store (*Store*) of variables. Let $Id = T \cup \{\mathcal{G}\}$, then $E : Id \hookrightarrow Store$. $E[\mathcal{G}]$ stores all the output, shared, and private variables in the program, which are all globally declared thanks to the program transformations of Section 4.2.1. $E[t]$ stores thread *t*'s copies of shared variables. The store (*Store*)

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. 17. Retrieving statically known information about Figure 16c.

is a partial function that maps variables ($var \in Var$) to values ($v \in Val$) and statuses ($sts \in Sts$), $Store : Var \hookrightarrow (Val, Sts)$. 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, $E = \{\mathcal{G} \rightarrow \{s \rightarrow (1, \text{pre})\}, t1 \rightarrow \{s \rightarrow (3, \text{mod})\}\}$ 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, 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, $A : \mathcal{A} \rightarrow Val$. An abort with a non-zero value means that its preemption condition is *true* and that it has been triggered. For example, $A = \{a1 \rightarrow 1, a2 \rightarrow 0\}$ 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 :

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

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

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

- GETPARENT(t) : Returns the parent of thread t . If $t = \text{main}$, 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 17 exemplifies the use of these functions on the program in Figure 16c.

4.2.5. 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 ($*$, $\&$, $!$, $-$, \sim), a binary arithmetic or Boolean operation ($||$, $\&\&$, \wedge , $|$, $\&$, $<<$, $>>$, $==$, $!=$, $<$, $>$, $<=$, $>=$, $+$, $-$, $*$, $/$, $\%$), a function call with its arguments passed by value and by reference, an array, and so on. For the sake of

simplicity, we do not give the details here [Blazy and Leroy 2009; Norrish 1999] and will just 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 just v), we have assumed that ForeC expressions and functions are side-effect free. During the evaluation, a variable's value is retrieved with the semantic function $\text{GETVAL}(E, I, id, var)$ described by Algorithm 1. The inputs to the 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: $\text{GETVAL}(E, I, id, var)$: 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  $var \in I$  then                                     // If  $var$  is an input.
2   | return  $I[var]$                                      // Return the input value of  $var$ .
3 else if  $var \in E[id]$  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

```

4.2.6. COPY. The semantic function $\text{COPY}(E, t)$ creates in thread t the local copies of each shared variable $var \in \text{GETSHARED}(t)$ 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 the 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 variable¹, 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 .

4.2.7. COMBINE. The semantic function $\text{COMBINE}(E, t_1, t_2, t_0)$ combines all the copies of shared variables from two threads and is described by Algorithm 3. The inputs to the 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 variable¹ var . Line 2 gets the shared variable's pre value (preVal). For the combine policy all, the copies from both threads are combined if they exist. Thus, line 3 gets the set of threads T that have a copy of the shared variable.

1. Recall from Section 4.2.2 that E maps the global and thread scopes to their own store of variables, $E : Id \mapsto \text{Store}$. Variables are mapped to a value and status, $\text{Store} : Var \mapsto (Val, Sts)$ where $Sts = \{\text{pre}, \text{mod}, \text{cmb}, \text{pvt}\}$. 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 $E[t][var]$ looks up the value and status (v, sts) of thread t 's copy of var .

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 the  $var \in \text{GETSHARED}(t)$  do           // For all shared variables needed by thread  $t$ .
2   if  $var \notin E[t]$  then                       // If thread  $t$  does not have a copy.
3     if  $var \in E[\text{GETPARENT}(t)]$  then          // If its parent has a copy.
4        $v := E[\text{GETPARENT}(t)][var].v$           // Value of its parent's copy.
5        $E[t][var \leftarrow (v, \text{pre})]$           // Copy its parent's copy.
6     else                                       // Otherwise, its parent does not have a copy.
7        $E[t][var \leftarrow E[\mathcal{G}][var]]$         // Copy the shared variable from the global scope.
8     end
9   end
10 end
11 return  $E$ 

```

If the combine policy is new, then line 5 keeps only the copies with values that differ from the shared variable's pre value ($E[t][var].v \neq \text{preVal}$) or copies that have been combined ($E[t][var].sts = \text{cmb}$). If the combine 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 gets the shared variable'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 with the status cmb. Line 18 returns the updated environment E restricted to t_0 (i.e., without thread t_1 and t_2 's store).

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 the  $var \in \text{GETSHARED}(\mathcal{G})$  do           // For all shared variables.
2    $\text{preVal} := E[\mathcal{G}][var].v$                      // Get the pre of  $var$ .
3    $T := \{t \mid t \in \{t_1, t_2\}, var \in E[t]\}$  // Set of threads with a copy of  $var$ .
4   if  $\text{GETPOLICY}(var) = \text{new}$  then
5     // Keep only the copies that differ from preVal or have been combined.
6      $T := \{t \mid t \in T, E[t][var].v \neq \text{preVal} \vee E[t][var].sts = \text{cmb}\}$ 
7   else if  $\text{GETPOLICY}(var) = \text{mod}$  then
8     // Keep only the modified or combined copies.
9      $T := \{t \mid t \in T, E[t][var].sts \in \{\text{mod}, \text{cmb}\}\}$ 
10  end
11  if  $|T| = 2$  then                               // If there are two copies to combine.
12     $cf := \text{GETCOMBINE}(var)$                      // Get the combine function of  $var$ .
13     $v := cf(E[t_1][var].v, E[t_2][var].v)$          // Combine the copies.
14     $E[t_0][var \leftarrow (v, \text{cmb})]$              // Assign the combined value to  $t_0$ .
15  else if  $|T| = 1$  then                           // Otherwise, there is only one copy.
16     $E[t_0][var \leftarrow (E[t \in T][var].v, \text{cmb})]$  // Assign the only copy to  $t_0$ .
17  end
18 end
19  $E' = \{(id, store) \mid (id, store) \in E \wedge id \neq t_1 \wedge id \neq t_2\}$ 
20 return  $E'$ 

```

4.2.8. *The Structural Operational Semantics.* This section presents the operational semantics of the kernel constructs presented in Table IV.

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

$$\langle E, A \rangle t : \text{nop} \xrightarrow[I]{0} \langle E, A \rangle t : \quad (\text{nop})$$

4.2.10. *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 :

$$\langle E, A \rangle t : \text{copy} \xrightarrow[I]{0} \langle \text{COPY}(E, t), A \rangle t : \quad (\text{copy})$$

4.2.11. *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) :

$$\langle E, A \rangle t : \text{pause} \xrightarrow[I]{1} \langle E, A \rangle t : \text{copy} \quad (\text{pause})$$

4.2.12. *The status Statement.* Recall that the abort statement is mapped to a status 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 :

$$\langle E, A \rangle t : \text{status}(a, exp) \xrightarrow[I]{0} \langle E, A[a \leftarrow \text{EVAL}(E, I, t, exp)] \rangle t : \quad (\text{status})$$

4.2.13. *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 ($k = \perp$) :

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

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

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

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

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

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

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

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 tick² :

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

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 tick³ :

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

4.2.14. The Assignment Operator (=). The assignment operator evaluates the expression exp into a value $v = \text{EVAL}(E, I, t, exp)$. If var is a shared variable⁴ (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 $E[\mathcal{G}]$:

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

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

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

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

$$\frac{\text{EVAL}(E, I, t, exp) = 0}{\langle E, A \rangle t : \text{if } (exp) f_1 \text{ else } f_2 \xrightarrow[I]{\perp} \langle E, A \rangle t : f_2} \quad (\text{if-else})$$

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

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

2. 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.

3. In addition to footnote 2, the strong preemption prevents the execution of a `copy` statement inside the abort body.

4. Recall from Section 4.2.2 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 $Sts = \{\text{pre}, \text{mod}, \text{cmb}, \text{pvt}\}$. 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 $E[t][var]$ looks up the value and status (v, sts) of thread t 's copy of var .

$$\frac{\text{EVAL}(E, I, t, \text{exp}) = 0}{\langle E, A \rangle t : \text{while}(\text{exp}) f \xrightarrow[I]{0} \langle E, A \rangle t :} \quad (\text{loop-else})$$

4.2.17. *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 :

$$\frac{\langle E, A \rangle t : f_1 \xrightarrow[I]{k \in \{1, \perp\}} \langle E', A' \rangle t : f'_1}{\langle E, A \rangle t : f_1; f_2 \xrightarrow[I]{k} \langle E', A' \rangle t : f'_1; f_2} \quad (\text{seq-left})$$

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

4.2.18. *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 :

$$\frac{\langle E, A \rangle t_1 : f_1 \xrightarrow[I]{\perp} \langle E', A' \rangle t_1 : f'_1 \quad \langle E, A \rangle t_2 : f_2 \xrightarrow[I]{\perp} \langle E'', A'' \rangle t_2 : f'_2}{\langle E, A \rangle t_0 : \text{par}(t_1 : f_1, t_2 : f_2) \xrightarrow[I]{\perp} \langle E^A, A^A \rangle t_0 : \text{par}(t_1 : f'_1, t_2 : f'_2)} \quad (\text{par-1})$$

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 :

$$\frac{\langle E, A \rangle t_1 : f_1 \xrightarrow[I]{k \in \{0, 1\}} \langle E', A' \rangle t_1 : f'_1 \quad \langle E, A \rangle t_2 : f_2 \xrightarrow[I]{\perp} \langle E'', A'' \rangle t_2 : f'_2}{\langle E, A \rangle t_0 : \text{par}(t_1 : f_1, t_2 : f_2) \xrightarrow[I]{\perp} \langle E'', A'' \rangle t_0 : \text{par}(t_1 : f_1, t_2 : f'_2)} \quad (\text{par-2})$$

$$\frac{\langle E, A \rangle t_1 : f_1 \xrightarrow[I]{\perp} \langle E', A' \rangle t_1 : f'_1 \quad \langle E, A \rangle t_2 : f_2 \xrightarrow[I]{k \in \{0, 1\}} \langle E'', A'' \rangle t_2 : f'_2}{\langle E, A \rangle t_0 : \text{par}(t_1 : f_1, t_2 : f_2) \xrightarrow[I]{\perp} \langle E', A' \rangle t_0 : \text{par}(t_1 : f'_1, t_2 : f_2)} \quad (\text{par-3})$$

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` :

$$\frac{\langle E, A \rangle t_1 : f_1 \xrightarrow{1}_I \langle E', A' \rangle t_1 : f'_1 \quad \langle E, A \rangle t_2 : f_2 \xrightarrow{1}_I \langle E'', A'' \rangle t_2 : f'_2}{\langle E, A \rangle t_0 : \text{par}(t_1 : f_1, t_2 : f_2) \xrightarrow{1}_I \langle \text{COMBINE}(E^A, t_1, t_2, t_0), A^A \rangle t_0 : \text{par}(t_1 : f'_1, t_2 : f'_2)} \quad (\text{par-4})$$

Otherwise, the `par` terminates if both of its child threads terminate. The completion code is \perp 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 tick⁵ :

$$\frac{\langle E, A \rangle t_1 : f_1 \xrightarrow{0}_I \langle E', A' \rangle t_1 : \quad \langle E, A \rangle t_2 : f_2 \xrightarrow{0}_I \langle E'', A'' \rangle t_2 :}{\langle E, A \rangle t_0 : \text{par}(t_1 : f_1, t_2 : f_2) \xrightarrow{\perp}_I \langle \text{COMBINE}(E^A, t_1, t_2, t_0), A^A \rangle t_0 : \text{copy}} \quad (\text{par-5})$$

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

$$\frac{\langle E, A \rangle t_1 : f_1 \xrightarrow{0}_I \langle E', A' \rangle t_1 : \quad \langle E, A \rangle t_2 : f_2 \xrightarrow{1}_I \langle E'', A'' \rangle t_2 : f'_2}{\langle E, A \rangle t_0 : \text{par}(t_1 : f_1, t_2 : f_2) \xrightarrow{1}_I \langle \text{COMBINE}(E^A, t_1, t_2, t_0), A^A \rangle t_0 : \text{par}(t_1 : \text{nop}, t_2 : f'_2)} \quad (\text{par-6})$$

$$\frac{\langle E, A \rangle t_1 : f_1 \xrightarrow{1}_I \langle E', A' \rangle t_1 : f'_1 \quad \langle E, A \rangle t_2 : f_2 \xrightarrow{0}_I \langle E'', A'' \rangle t_2 :}{\langle E, A \rangle t_0 : \text{par}(t_1 : f_1, t_2 : f_2) \xrightarrow{1}_I \langle \text{COMBINE}(E^A, t_1, t_2, t_0), A^A \rangle t_0 : \text{par}(t_1 : f'_1, t_2 : \text{nop})} \quad (\text{par-7})$$

4.2.19. 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 :

$$\frac{\langle E, A \rangle \text{main} : f \xrightarrow{k \in \{0,1\}}_I \langle E', A' \rangle \text{main} : f'}{\langle E, A \rangle \text{main} : f \xrightarrow{k}_I \langle E'', A'' \rangle \text{main} : f'} \quad (\text{tick})$$

The rules for the `par` statement ensures that, when the tick completes, `main`'s store in E' has the combined values from all its child threads. The shared variables⁶ are resynchronized by assigning the combined values from $E'[\text{main}]$ 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 $E'[\text{main}]$, we have $E'' = E'[\mathcal{G}][\text{var} \leftarrow (E'[\text{main}][\text{var}].v, \text{pre})] \setminus \{\text{main}\}$. 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 $A'' = A'[a \leftarrow \text{EVAL}(E'', I, \mathcal{G}, \text{GETEXP}(a))]$.

5. The `par` statement may have paused. In this case, when the `par` terminates, the parent thread t_0 will start its local tick.

6. Recall from Section 4.2.2 that E maps the global and thread scopes to their own store of variables, $E : Id \hookrightarrow \text{Store}$. Variables are mapped to a value and status, $\text{Store} : \text{Var} \hookrightarrow (\text{Val}, \text{Sts})$. In $E[\mathcal{G}]$, shared

```

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

```

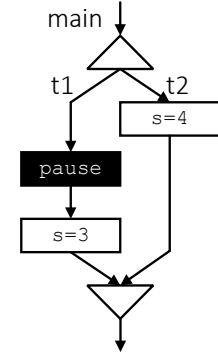
(a) ForeC program.

```

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. 18. Illustrative example one.

4.2.20. Illustrations. The following sections provide two examples of how ForeC programs execute. The executions are given as sequences of rewrites.

4.2.21. Example One. The first program illustrates parallel execution using the `par` statement. Figure 18a presents the ForeC program, and Figure 18c illustrates the program's control-flow. In Figure 18c, the triangle represents the forking of threads while the inverted triangle represents the joining of threads.

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 `plus(0, 4) = 4`. In the second tick, thread `t1` starts its local tick by creating a copy of `s`, assigning 3 to its copy of `s`, and then terminating. The `par` terminates because threads `t1` and `t2` have now joined. Because only thread `t1` has a copy of `s`, that copy is assigned directly to its parent thread `main`. The `main` thread 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 the `main` thread has a copy of `s`.

Before we apply the rewrite rules to the program, it is structurally translated into Figure 18b (see the start of Section 4.2). Note that 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 19.

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

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

variables have the status `pre`. The notation $E[t][var]$ looks up the value and status (v, sts) of thread t 's copy of var .

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

Fig. 19. Initial program environment and its derivatives.

Step 2 : Both threads of the par execute sequential statements. 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 .

$$\begin{array}{c}
\text{(copy)} \frac{}{\langle E^1, A \rangle \text{ t1:copy} \xrightarrow{0} \langle E^2, A \rangle \text{ t1:}} \quad \frac{}{\langle E^1, A \rangle \text{ t2:copy} \xrightarrow{0} \langle E^3, A \rangle \text{ t2:}} \\
\text{(seq-right)} \frac{}{\langle E^1, A \rangle \text{ t1:copy; } \frac{\perp}{I} \rightarrow \langle E^2, A \rangle \text{ t1:}} \quad \frac{}{\langle E^1, A \rangle \text{ t2: } \frac{\perp}{I} \rightarrow \langle E^3, A \rangle \text{ t2:}} \\
\text{(par-1)} \frac{}{\langle E^1, A \rangle \text{ main:par(t1:\{copy; } \frac{\perp}{I} \rightarrow \langle E^4, A \rangle \text{ main:par(t1:\{pause; } \\
\text{pause;s=3;\}, t2:\{copy;s=4;\})} \quad \text{copy;s=4;\} \frac{\perp}{I} \rightarrow \text{pause;s=3;\}, t2:\{s=4;\})}
\end{array}$$

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 aggregated environment is the same as E^5 , which is then combined to be E^6 . When the (tick) rule is applied, E^6 is resynchronized to be E^7 .

$$\begin{array}{c}
\text{(pause)} \frac{}{\langle E^4, A \rangle \text{ t1: } \frac{1}{I} \rightarrow \langle E^4, A \rangle \text{ t1:}} \\
\text{(seq-left)} \frac{}{\langle E^4, A \rangle \text{ t1: } \frac{1}{I} \rightarrow \langle E^4, A \rangle \text{ t1:}} \quad \text{(assign-shared)} \frac{s \in \{s\}}{\langle E^4, A \rangle \text{ t2: } \frac{0}{I} \rightarrow \langle E^5, A \rangle \text{ t2:}} \\
\text{(par-7)} \frac{}{\langle E^4, A \rangle \text{ main:par(t1:\{pause; } \frac{1}{I} \rightarrow \langle E^6, A \rangle \text{ main:par(t1:\{copy; } \\
\text{s=3;\}, t2:\{s=4;\})} \quad \text{s=3;\}, t2:nop)} \\
\text{(tick)} \frac{}{\langle E^4, A \rangle \text{ main:par(t1:\{pause; } \frac{1}{I} \rightarrow \langle E^7, A \rangle \text{ main:par(t1:\{copy; } \\
\text{s=3;\}, t2:\{s=4;\})} \quad \text{s=3;\}, t2:nop)}
\end{array}$$

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.

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

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 aggregated environment is the same as E^9 , which is then combined to be E^{10} .

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

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.**

$$\begin{array}{c}
\text{(copy)} \frac{}{\langle E^{10}, A \rangle \text{ main:copy} \xrightarrow[I]{0} \langle E^{10}, A \rangle \text{ main:}} \\
\text{(tick)} \frac{}{\langle E^{10}, A \rangle \text{ main:copy} \xrightarrow[I]{0} \langle E^{11}, A \rangle \text{ main:}}
\end{array}$$

4.2.22. Example Two. The second program illustrates the preemption by using an immediate weak abort statement. Figure 20a presents the ForeC program and Figure 20c illustrates the program's control-flow. In Figure 20c, the pair of decorated diamonds represents the scope of the abort body.

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

Before we apply the rewrite rules to the program, it is structurally translated into Figure 20b (see the start of Section 4.2). The copy kernel statement is not inserted into the program because no shared variables are used. Note that the semantic functions $\text{GETSHARED}(\text{main})$ and $\text{GETSHARED}(\mathcal{G})$ all return \emptyset . The program's environment E , preemption statuses A , and their derivatives are defined in Figure 21.

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.

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

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

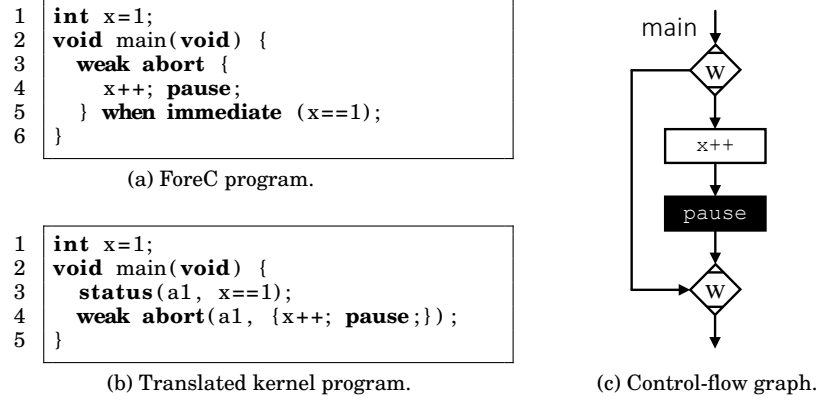


Fig. 20. Illustrative example two.

$$\begin{array}{ll}
 E &= \{\mathcal{G} \rightarrow \{x \rightarrow (1, \text{pvt})\}\} & A &= \{a1\} \\
 E^1 &= \{\mathcal{G} \rightarrow \{x \rightarrow (2, \text{pvt})\}\} & A^1 &= \{a1 \rightarrow 1\} \\
 & & A^2 &= \{a1 \rightarrow 0\}
 \end{array}$$

Fig. 21. Initial program state and its derivatives.

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

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

$$\begin{array}{l}
 \text{(pause)} \frac{}{\langle E^1, A^1 \rangle \text{ main: } \text{pause} \xrightarrow{1} \langle E^1, A^1 \rangle \text{ main: } \text{copy}} \\
 \text{(abort-5)} \frac{\langle E^1, A^1 \rangle \text{ main: } \text{weak abort} \xrightarrow{1} \langle E^1, A^1 \rangle \text{ main: } \text{weak abort} \quad (A^1[a1] \neq 0)}{\langle E^1, A^1 \rangle \text{ main: } \text{weak abort} \xrightarrow{1} \langle E^1, A^1 \rangle \text{ main: } \text{copy}}
 \end{array}$$

Step 4 : Apply the (tick) and (copy) rules. The preemption statuses in A^1 are updated to be A^2 . **The tick ends and the program terminates.**

$$\begin{array}{l}
 \text{(copy)} \frac{}{\langle E^1, A^1 \rangle \text{ main: } \text{copy} \xrightarrow{0} \langle E^1, A^1 \rangle \text{ main:}} \\
 \text{(tick)} \frac{}{\langle E^1, A^1 \rangle \text{ main: } \text{copy} \xrightarrow{0} \langle E^1, A^2 \rangle \text{ main:}}
 \end{array}$$

4.3. Definitions and Proofs

The semantics of the ForeC kernel constructs (Section 4.2.8) can be used to formally prove two desirable properties of safety-critical programs, called *reactivity* and *determinism* [Maraninchi 1992; Tardieu 2007]. A program is reactive if it always responds

to changes in the environment, i.e., does not deadlock and produces outputs. 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 programs. In terms of semantic derivation rules, a program is deterministic if there is at most one derivation tree in response to the environment. The definitions for reactivity and determinism are normally based on a program's tick, which is a sequence of transitions. Because the state of a ForeC program depends on the initial valuations of its variables, we define a stronger notion of reactivity and determinism based on program transitions.

Definition 4.3. 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) :

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

THEOREM 4.4. All ForeC programs are **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 :

$$\begin{aligned} \langle S \rangle t : \text{nop} & \xrightarrow[I]{0} \langle S \rangle t : \\ \langle S \rangle t : \text{copy} & \xrightarrow[I]{0} \langle S' \rangle t : \\ \langle S \rangle t : \text{pause} & \xrightarrow[I]{1} \langle S \rangle t : \text{copy} \\ \langle S \rangle t : \text{status}(a, \text{exp}) & \xrightarrow[I]{0} \langle S' \rangle t : \\ \langle S \rangle t : \text{var} = \text{exp} & \xrightarrow[I]{0} \langle S' \rangle t : \\ \langle S \rangle t : \text{if } (\text{exp}) f_1 \text{ else } f_2 & \xrightarrow[I]{\perp} \langle S \rangle t : f_1 \quad \text{or} \quad \langle S \rangle t : \text{if } (\text{exp}) f_1 \text{ else } f_2 \xrightarrow[I]{\perp} \langle S \rangle t : f_2 \\ \langle S \rangle t : \text{while } (\text{exp}) f & \xrightarrow[I]{\perp} \langle S \rangle t : f; \text{ while } (\text{exp}) f \quad \text{or} \quad \langle S \rangle t : \text{while } (\text{exp}) f \xrightarrow[I]{0} \langle S \rangle t : \end{aligned}$$

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 least one transition :

$$\exists S'_1, S'_2, f'_1, f'_2, k_1, k_2 \text{ such that } \langle S_1 \rangle t_1 : f_1 \xrightarrow[I]{k_1} \langle S'_1 \rangle t_1 : f'_1 \quad (\text{H1})$$

$$\langle S_2 \rangle t_2 : f_2 \xrightarrow[I]{k_2} \langle S'_2 \rangle t_2 : f'_2 \quad (\text{H2})$$

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

- (1) Consider $t_1 : f_1; f_2$. Due to the induction hypotheses, 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 the induction hypothesis (H1), then from the (seq-right) rule we have :

$$(\text{seq-right}) \frac{\langle S_1 \rangle \ t_1 : f_1 \xrightarrow[I]{k_1=0} \langle S'_1 \rangle \ t_1 :}{\langle S_1 \rangle \ t_1 : f_1; f_2 \xrightarrow[I]{\perp} \langle S'_1 \rangle \ t_1 : f_2}$$

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

$$(\text{seq-left}) \frac{\langle S_1 \rangle \ t_1 : f_1 \xrightarrow[I]{k_1 \in \{1, \perp\}} \langle S'_1 \rangle \ t_1 : f'_1}{\langle S_1 \rangle \ t_1 : f_1; f_2 \xrightarrow[I]{k_1} \langle S'_1 \rangle \ t_1 : f'_1; f_2}$$

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

- (2) Consider $t_1 : \text{weak? abort}(a_1, f_1)$. Due to the induction hypotheses, 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	\perp	0	1	\perp
$A[a_1]$	$= 0$	(abort-2)	(abort-1)		(abort-2)	(abort-1)	
	$\neq 0$	(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 the induction hypothesis (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 $t : \text{par}(t_1 : f_1, t_2 : f_2)$. Due to the induction hypotheses, 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	\perp
k_1	0	(par-5)	(par-6)	(par-2)
	1	(par-7)	(par-4)	
	\perp	(par-3)		(par-1)

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

$$(\text{par-5}) \frac{\langle S \rangle t_1 : f_1 \xrightarrow[k_1=0]{I} \langle S'_1 \rangle t_1 : \quad \langle S \rangle t_2 : f_2 \xrightarrow[k_2=0]{I} \langle S'_2 \rangle t_2 :}{\langle S \rangle t : \text{par}(t_1 : f_1, t_2 : f_2) \xrightarrow[\perp]{I} \langle S'' \rangle t : \text{copy}}$$

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

□

Definition 4.5. A program $t : f$ is **deterministic** if, in any state S , for any input configuration I , there exists at most one transition such that :

$$\begin{aligned} \forall S, I : \quad & \text{if } \langle S \rangle t : f \xrightarrow[k']{I} \langle S' \rangle t : f' \\ & \text{and } \langle S \rangle t : f \xrightarrow[k'']{I} \langle S'' \rangle t : f'' \quad \text{then } S' = S'', f' = f'', k' = k'' \end{aligned}$$

Only the rewrite rules of the par statement allow state S to be changed in parallel by multiple transitions. The (par-1) rule aggregates the changes into a single state. The (par-4), (par-5), (par-6), and (par-7) rules use the semantic function COMBINE (Algorithm 3) to combine the copies in the aggregated state. The (par-2) and (par-3) rules only allow one of the changed states to take effect. Before proving that all ForeC programs are deterministic, we prove that the aggregation of states and the semantic function COMBINE are both deterministic. This is captured by Lemmas 4.8 and 4.9 below with the assumption that all the combine functions are deterministic.

Definition 4.6. A combine function cf is any C function with two input values v_1 and v_2 of identical type, which returns a value of the same type.

Hypothesis 4.7. Each combine function cf always returns the same value regardless of the current state, provided that the input values, v_1 and v_2 , are identical :

$$\begin{aligned} \forall S, S', I, t, v_1, v_2 : \\ \text{EVAL}(S.E, I, t, cf(v_1, v_2)) = \text{EVAL}(S'.E, I, t, cf(v_1, v_2)) \end{aligned}$$

Because the combine functions are defined in C, we require that the combine functions always terminate without error.

LEMMA 4.8. For any initial state⁷ $S = \langle E, A \rangle$, let $S' = \langle E', A' \rangle$ and $S'' = \langle E'', A'' \rangle$ be the states of two threads after their transition. If the threads can only change their own private variables and copies of shared variables, then the aggregation of S' and S'' is **deterministic** if there exists only one aggregated state :

$$\begin{aligned} \forall S = \langle E, A \rangle, S' = \langle E', A' \rangle, S'' = \langle E'', A'' \rangle : \\ \text{if } E_1^A = (E' \setminus (E' \cap E)) \cup (E'' \setminus (E'' \cap E)) \cup (E' \cap E'') \\ \text{and } E_2^A = (E' \setminus (E' \cap E)) \cup (E'' \setminus (E'' \cap E)) \cup (E' \cap E'') \quad \text{then } E_1^A = E_2^A \\ \text{if } A_1^A = (A' \setminus (A' \cap A)) \cup (A'' \setminus (A'' \cap A)) \cup (A' \cap A'') \\ \text{and } A_2^A = (A' \setminus (A' \cap A)) \cup (A'' \setminus (A'' \cap A)) \cup (A' \cap A'') \quad \text{then } A_1^A = A_2^A \end{aligned}$$

⁷ Recall from Section 4.2.2 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, $Store : Var \hookrightarrow (Val, Sts)$. The notation $E[t][var]$ looks up the value and status (v, sts) of thread t 's copy of var . Recall that A maps the abort identifiers to their preemption statuses, $A : \mathcal{A} \rightarrow Val$. The notation $A[a]$ looks up the preemption status v of abort a .

PROOF. We begin by proving that the aggregation of environments E' and E'' is deterministic. If the threads can only change their private variables and copies of shared variables, then their changes to E are always mutually exclusive. That is, for any two threads t' and t'' , where $t' \neq t''$, the threads never access each other's store because $E[t'] \neq E[t'']$. Moreover, by definition, the threads never access each other's private variables in $E[\mathcal{G}]$. Intersecting two environments, e.g., $E' \cap E$, always gives a new environment containing the variables that have the same values and statuses in E' and E , i.e., have not changed. $E' \setminus (E' \cap E)$ always gives a new environment containing the variables that have changed in E' . Operations on sets are deterministic because two variables are either identical or not. The aggregation always takes 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 unchanged variables in E' and E'' (i.e., $E' \cap E''$). Because the changes in E' and E'' are always mutually exclusive, the aggregation always takes the union of three disjoint environments.

We now prove that the aggregation of two sets of preemption statuses A' and A'' is deterministic. Threads can only change A by executing a status statement (the (status) rule). By construction, each status statement has a unique abort identifier a . Thus, changes to A are always mutually exclusive. Intersecting two sets of preemption statuses, e.g., $A' \cap A$, always gives a new set containing the statuses that have the same values in A' and A , i.e., have not changed. $A' \setminus (A' \cap A)$ always gives a new set containing the statuses that have changed in A' . Operations on sets are deterministic because two statuses are either identical or not. The aggregation always takes the union of the changes in A' (i.e., $A' \setminus (A' \cap A)$) and in A'' (i.e., $A'' \setminus (A'' \cap A)$) with the unchanged statuses in A' and A'' (i.e., $A' \cap A''$). Because the changes in A' and A'' are always mutually exclusive, the aggregation always takes the union of three disjoint sets. \square

LEMMA 4.9. *If all combine functions are deterministic, then the semantic function COMBINE is **deterministic** if, in any state $S = \langle E, A \rangle$, for any three threads t_1 , t_2 , and t_0 , there exists only one environment that can be returned :*

$$\begin{aligned} \forall S = \langle E, A \rangle, \forall t_1, t_2, t_0 : \quad & \text{if } E' = \text{COMBINE}(E, t_1, t_2, t_0) \\ & \text{and } E'' = \text{COMBINE}(E, t_1, t_2, t_0) \quad \text{then } E' = E'' \end{aligned}$$

PROOF. The semantic function COMBINE is an algorithm that initializes all its local variables (preVal , T , cf , v), that is side-effect-free, and that uses only deterministic instructions. In particular, line 11 in Algorithm 3 is deterministic due to the hypothesis that all combine functions cf are deterministic (Hypothesis 4.7). Hence, the semantic function COMBINE is deterministic. \square

THEOREM 4.10. *If all combine functions are deterministic, then all ForeC programs are **deterministic**.*

PROOF. The proof can be shown by a 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 :

$$\begin{aligned}
\langle S \rangle t : \text{nop} & \xrightarrow[I]{0} \langle S \rangle t : \\
\langle S \rangle t : \text{copy} & \xrightarrow[I]{0} \langle S' \rangle t : \text{nop} \\
\langle S \rangle t : \text{pause} & \xrightarrow[I]{1} \langle S \rangle t : \text{copy} \\
\langle S \rangle t : \text{status}(a, \text{exp}) & \xrightarrow[I]{0} \langle S' \rangle t : \text{nop}
\end{aligned}$$

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 : (assign-shared) and (assign-private), (if-then) and (if-else), and (loop-then) and (loop-else). The premises are complementary in the sense that, if the premise of one rule is *true*, then the premise of the other rule must be *false*, and vice versa. This implies that these kernel constructs have at most one transition :

$$\begin{aligned}
& \text{if } \text{var} \in \text{GETSHARED}(t) \text{ then } \langle S \rangle t : \text{var} = \text{exp} \xrightarrow[I]{0} \langle S' \rangle t : \\
& \quad \text{otherwise } \langle S \rangle t : \text{var} = \text{exp} \xrightarrow[I]{0} \langle S'' \rangle t : \\
& \text{if } \text{EVAL}(S.E, I, t, \text{exp}) \neq 0 \text{ then } \langle S \rangle t : \text{if } (\text{exp}) f_1 \text{ else } f_2 \xrightarrow[I]{\perp} \langle S \rangle t : f_1 \\
& \quad \text{otherwise } \langle S \rangle t : \text{if } (\text{exp}) f_1 \text{ else } f_2 \xrightarrow[I]{\perp} \langle S \rangle t : f_2 \\
& \text{if } \text{EVAL}(S.E, I, t, \text{exp}) \neq 0 \text{ then } \langle S \rangle t : \text{while } (\text{exp}) f \xrightarrow[I]{\perp} \langle S \rangle t : f; \text{ while } (\text{exp}) f \\
& \quad \text{otherwise } \langle S \rangle t : \text{while } (\text{exp}) f \xrightarrow[I]{0} \langle S \rangle t :
\end{aligned}$$

Of the rewrite rules considered in the base case, only the (copy), (status), (assign-shared), and (assign-private) rules make direct changes to 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 the same abort identifier. 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 :

$$\begin{aligned}
& \text{If } \exists S'_1, S''_1, f'_1, f''_1, k'_1, k''_1 \text{ such that } \langle S_1 \rangle t_1 : f_1 \xrightarrow[I]{k'_1} \langle S'_1 \rangle t_1 : f'_1 \quad (\text{H3}) \\
& \quad \text{and } \langle S_1 \rangle t_1 : f_1 \xrightarrow[I]{k''_1} \langle S''_1 \rangle t_1 : f''_1 \\
& \quad \text{then } S'_1 = S''_1, f'_1 = f''_1, k'_1 = k''_1
\end{aligned}$$

$$\begin{aligned}
\text{If } \exists S'_2, S''_2, f'_2, f''_2, k'_2, k''_2 \quad \text{such that } \langle S_2 \rangle t_2 : f_2 \xrightarrow[I]{k'_2} \langle S'_2 \rangle t_2 : f'_2 \quad (\text{H4}) \\
\text{and } \langle S_2 \rangle t_2 : f_2 \xrightarrow[I]{k''_2} \langle S''_2 \rangle t_2 : f''_2 \\
\text{then } S'_2 = S''_2, f'_2 = f''_2, k'_2 = k''_2
\end{aligned}$$

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

- (1) Consider the fragment $t_1 : f_1; f_2$. Due to the induction hypothesis (H3), there is only one possible transition for the fragment $t_1 : f_1$,

$$\begin{aligned}
\text{which is either } \langle S_1 \rangle t_1 : f_1 \xrightarrow[I]{k_1=0} \langle S'_1 \rangle t_1 : \\
\text{or } \langle S_1 \rangle t_1 : f_1 \xrightarrow[I]{k_1 \in \{1, \perp\}} \langle S'_1 \rangle t_1 : f'_1
\end{aligned}$$

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

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

So, thanks to the induction hypothesis (H3), the sequence operator “;” is deterministic.

- (2) Consider the abort kernel statement in the fragment $t_1 : \text{weak? abort}(a_1, f_1)$. Due to the induction hypothesis (H3), there is only one possible transition for the program fragment $t_1 : f_1$,

$$\begin{aligned}
\text{which is either } \langle S_1 \rangle t_1 : f_1 \xrightarrow[I]{k_1=0} \langle S'_1 \rangle t_1 : \\
\text{or } \langle S_1 \rangle t_1 : f_1 \xrightarrow[I]{k_1 \in \{1, \perp\}} \langle S'_1 \rangle t_1 : f'_1
\end{aligned}$$

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	\perp	0	1	\perp
$A[a_1]$	$= 0$	(abort-2)	(abort-1)		(abort-2)	(abort-1)	
	$\neq 0$	(abort-6)			(abort-3)	(abort-5)	(abort-4)

So, thanks to the induction hypothesis (H3), the abort kernel statement is deterministic.

- (3) Consider the par kernel statement in the fragment $t : \text{par}(t_1 : f_1, t_2 : f_2)$. Due to the induction hypotheses (H3) and (H4), there is only one possible transition for the program fragment $t_1 : f_1$,

$$\begin{aligned}
\text{which is either } \langle S_1 \rangle t_1 : f_1 \xrightarrow[I]{k_1=0} \langle S'_1 \rangle t_1 : \\
\text{or } \langle S_1 \rangle t_1 : f_1 \xrightarrow[I]{k_1 \in \{1, \perp\}} \langle S'_1 \rangle t_1 : f'_1
\end{aligned}$$

Table VI. Comparing ForeC with Esterel, PRET-C, and Concurrent revisions.

Property	Esterel	PRET-C	ForeC	Concurrent revisions
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
Parallelism is Dynamic	No			Yes
Thread Communication Method	Pure and valued signals	Shared Variables		
Thread Communication Speed	Instantaneous	Instantaneous (sequential)	Delayed to the end of each tick	Delayed to thread termination
Resynchronization of shared variables or valued signals	Combine functions (mod values)	Not required	Combine functions with policies	Merge functions (all values)
Parallelism is Commutative and Associative	Yes	No (Sequential)	Depends on combine functions	Depends on merge functions

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 be applied depending on the completion codes k_1 and k_2 :

		k_2		
		0	1	\perp
k_1	0	(par-5)	(par-6)	(par-2)
	1	(par-7)	(par-4)	
	\perp	(par-3)		(par-1)

So, thanks to the induction hypotheses (H3) and (H4), the par kernel statement is deterministic.

□

4.4. Comparison with Esterel, PRET-C, and Concurrent Revisions

This section compares ForeC with Esterel [Berry and Gonthier 1992], PRET-C [Andalam et al. 2014], and Concurrent revisions [Burckhardt and Leijen 2011] and Table VI summarizes this qualitative comparison. 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 its snapshot. This ensures thread isolation during execution. When two threads join, their snapshots are merged together using a deterministic *merge function*.

ForeC and PRET-C are intended for applications that have control and data parallelism. Control parallelism is not a strength of concurrent revisions because its semantics does not consider (reactive) inputs and outputs. In Concurrent revisions, threads are forked asynchronously, allowing a parent thread to execute alongside its children. Hence, the parent thread can vary the amount of parallelism that is needed at runtime (e.g., fork more threads when there are more input data). The `rjoin` construct

can be used to force the parent thread to wait for its children to terminate. In contrast, threads are forked synchronously in ForeC, Esterel, and PRET-C, meaning that the parent thread blocks until all its children have terminated. Hence, the parent thread cannot vary the amount of parallelism at runtime.

Threads in PRET-C are executed in a strict sequential order, which is unsuitable for multi-core execution. However, the 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 in the sequential order (instantaneous in the synchronous model of computation, that is, occurring in the same global tick), but delayed by one tick in the reverse order. Similar to ForeC, threads in Concurrent revisions communicate over shared variables. When a child thread is forked, it creates a snapshot of the shared variables from its parent thread. When the child thread joins back with its parent, the snapshots of both threads are merged with a programmer-specified *merge function*. The merge function always considers both copies, i.e., equivalent to ForeC's combine policy `all`. Thus, thread communication is always delayed until the child thread terminates. In contrast, ForeC threads may execute over several ticks and thread communication is only delayed to the end of each tick. Esterel threads communicate instantaneously by emitting and receiving pure or valued signals during each tick. Pure signals are either present or absent and carry no value. All potential signal emissions must be performed by the concurrent threads before the signal status (present or absent) can be read. Compilers [Girault 2005; Edwards and Zeng 2007; Yuan et al. 2011] typically compile away the parallelism to create a sequential program that resolves the causality, but this restricts the execution onto single cores. Valued signals are like pure signals except that each emission has an associated value. Before a valued signal can be read, all the emitted values are combined using a programmer-specified commutative and associative combine function. The combine function only considers the emitted values, i.e., equivalent to ForeC's combine policy `mod`.

Esterel'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 not commutative or associative because threads communicate in a strict sequential order. For ForeC and Concurrent revisions, the commutativity and associativity of their parallel construct depends on their combine and merge functions, respectively.

Preemptions in ForeC and PRET-C are inspired by Esterel, but behave slightly differently. Preemptions in Esterel are triggered instantaneously, whereas preemptions in ForeC are triggered with a delay of one tick. Preemptions in PRET-C are triggered instantaneously, but the non-immediate behavior is not supported. Concurrent revisions do not support preemptions. Esterel programs may be non-causal [Benveniste et al. 2003] because of instantaneous feedback cycles. Thanks to delayed communication, ForeC and Concurrent revisions programs are always causal by construction. PRET-C programs are always causal by construction because threads communicate in a strict sequential order.

4.5. Discussion

This section has introduced the ForeC language that enables the deterministic parallel programming of multi-cores. The language features of ForeC help bridge the differences between synchronous-reactive programming and general-purpose parallel programming. ForeC makes deterministic parallelism accessible to traditional embedded C programmers. ForeC offers shared variable semantics that removes the burden of ensuring mutual exclusion from the programmer and guarantees deadlock freedom. Thread isolation is guaranteed by stipulating that threads work on local copies of the

shared variables. Resynchronizing the shared variables when the threads have finished their respective local ticks makes program behavior agnostic to scheduling decisions. These features simplify the understanding and debugging of ForeC programs. Important definitions and proofs for ForeC were given for reactivity and determinism. Finally, a critical comparison showed that ForeC merges the benefits offered by synchronous languages, such as Esterel [Berry and Gonthier 1992], with those offered by deterministic runtime solutions such, as Concurrent revisions [Burckhardt and Leijen 2011].

Traditional synchronous programming languages [Benveniste et al. 2003] are notoriously difficult to distribute or parallelize [Girault 2005] due to their signal communication model. The key advantage of the synchronous model of computation is that it removes the need to use thread synchronization mechanisms such as mutual exclusion. However, the need to maintain monotonic signal values [Yuan et al. 2011; Baudisch et al. 2010b] makes it very difficult to parallelize these programs. Moreover, static analysis is needed to ensure that the presence or absence of all signals can be determined exactly in each tick of the program (a corollary is that programs can react to the *absence* of signals). In contrast, communication in ForeC is delayed using shared variables, the values which are only resolved when threads complete their local ticks, hence allowing threads to execute in parallel and in isolation (but preventing the reaction to absence).

Section 5 presents a straightforward compilation approach for ForeC. Benchmarking in Section 6 reveals that 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.

ForeC's combine functions are inspired by Esterel [Potop-Butucaru et al. 2007] but similar solutions can be found in other parallel programming frameworks, e.g., OpenMP's reduction operators [OpenMP Architecture Review Board 2013], MPI's MPI_Reduce and MPI_Gather functions [Message Passing Interface Forum 2012], Intel Thread Building Blocks' `tbb::parallel_reduce` function and `tbb::combinable` data type [Intel 2012b], Intel Cilk Plus' reducer data types [Intel 2012a], and Unified Parallel C's collective functions [The UPC Consortium 2013]. Solutions developed for these frameworks could be reworked into ForeC combine functions. Appendix A provides more extensive examples of combine functions. A description of how the combine policies and combine functions work together to combine more than two copies of a shared variable is given.

Determinism in ForeC is guaranteed by the formal semantics, and preserved by the compiler. This is unlike the deterministic runtime solutions developed for Pthreads [Berger et al. 2009; Olszewski et al. 2009; Bergan et al. 2010a; Liu et al. 2011], OpenMP [Aviram and Ford 2011], and MPI [Zhang and Ford 2011], where determinism is only enforced at runtime and can be sensitive to changes in the program code. However, the behaviors of the deterministic runtimes are not described with formal semantics. Moreover, program execution in Pthreads, OpenMP, and MPI is not portable across the runtime solutions because each deterministic runtime enforces its own notion of determinism.

5. COMPILING FOREC FOR PARALLEL EXECUTION

The previous section described the language. To be useful, the ForeC program must be compiled appropriately to exploit the parallelism of the target hardware architecture. This section describes how the ForeC compiler generates code for direct execution on a predictable parallel architecture described in Section 3.1. The chosen compilation strategy generates code that is amenable to static timing analysis and achieves good execution performance, as benchmarking results in Section 6 reveal.

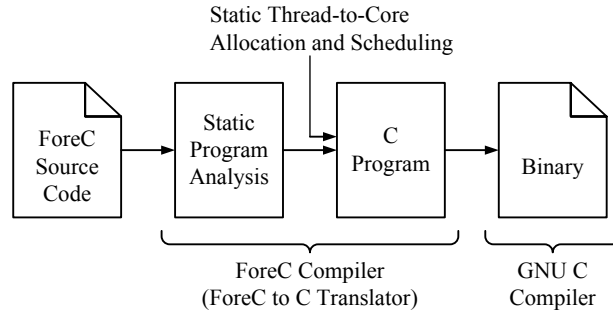


Fig. 22. Overview of compiling ForeC programs.

5.1. Overview

The ForeC compiler can generate code for direct (bare metal) execution on the Xilinx MicroBlaze embedded multi-core (Section 3.1) processor. Later in Section 5.9, we extend the compiler to generate code for execution on an operating system. Figure 22 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. The 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 [Potop-Butucaru et al. 2007] for creating the reactive interface.

5.2. Static Thread Scheduling

This section deals exclusively with ForeC threads. We illustrate the static thread scheduling with the example of Figure 23a. The programmer statically allocates the threads to the cores and passes the allocations into the compiler. The scheduling is *static* and *non-preemptive* (cooperative). 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 4.1.3) 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 23b depicts the thread hierarchy of the ForeC program from Figure 23a, where numbers indicate the total order. A lower number means higher execution priority. Figure 23c shows a possible thread allocation chosen by the programmer for two cores, in their thread scheduling order. When a thread reaches a par statement, its child threads are forked for execution on their allocated cores. The core that executes the parent thread is called the *master* core. The cores that execute the child threads are called the *slave* cores. Depending on the thread allocations, a core could be the master core of one thread and be the slave core of another thread. For the par statement on line 6 of Figure 23a, core 1 is the master core and core 2 is the slave core.

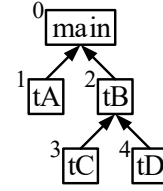
Based on the the thread allocation and scheduling order shown in Figure 23c, Figure 23d is a possible execution trace. The trace for both cores ("Core 1" and "Core 2") progresses downwards from the top of Figure 23d. Thread executions are shown as white segments in the trace and each one has the thread's name and the executed

```

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 }

```

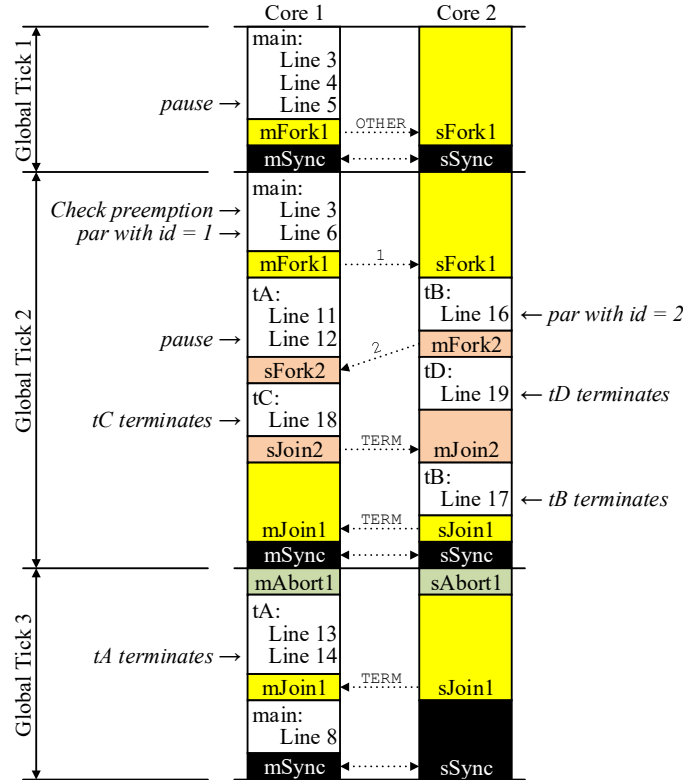
(a) Example ForeC program.



(b) Total order.

Core 1	Core 2
main	tB
tA	tD
tC	

(c) Thread allocation.



(d) Possible execution trace of the compiled program.

Fig. 23. Example ForeC program to be compiled.

Table VII. 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. 24. Definition of a linked list node and its operations in node.h.

lines of code from Figure 23a. 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 one has the routine's name. The names are prefixed with "m" or "s" to identify whether a 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 23d all manage the threads forked by thread main. Table VII summarizes the behavior of the routines. The mFork and sFork routines manage the forking of child threads (Section 5.4). The mJoin and sJoin routines manage the joining of child threads (Section 5.4). The mSync and sSync routines manage the global tick synchronization of all the cores (Section 5.8). In Figure 23d, the synchronization between the routines are shown as arrows marked with the information that is sent. The information is an integer value that encodes the following execution states of a thread : 0 (TERM) for thread termination, 1 and greater for the unique id of the par statement that is executing, and -1 (OTHER) for executing a pause statement or for not executing a par statement.

The threads and synchronization routines are statically scheduled on each core with *doubly linked lists*. Each node (defined in Figure 24) 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.

5.3. Structure of the Generated Program

Figure 25 shows a simplified version of the C program generated for the ForeC program in Figure 23a. All line numbers in section refer to Figure 25. The generated C program contains :

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

Although the scheduling routines dominate the generated code in Figure 25, their code remains constant whatever the size of the user-defined threads (which could be arbitrarily large). 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 VIII. The threads and routines are inlined into the main function because fast context-switching is implemented by jumping between C labels with GNU's computed goto extension. Jumping with goto is restricted to C labels located in the same function scope. 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.

5.4. 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 scheduling 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, a slave core uses an `sFork` routine to *block* 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 23d, `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 scheduling 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 schedules 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


```

1  #include "node.h" // Figure 24
2
3  // Programmer-defined
4  int x=0; // Shared variable
5  int a_tC; // tC's local variable
6  int plus(int th1,int th2) {return th1+th2;}
7
8  // Compiler-defined
9  enum State {OTHER=-1,TERM=0};
10 int mainState=OTHER, tAState=OTHER,
11    tBState=OTHER, tCState=OTHER,
12    tDState=OTHER;
13 int x_main, x_tA, x_tB, x_tC, x_tD;
14
15 // Entry point
16 void main(void) {
17     // Nodes for the linked lists
18     Node main={.pc=&main},
19     tA={.pc=&tA}, tB={.pc=&tB},
20     tC={.pc=&tC}, tD={.pc=&tD};
21     Node mFork1={.pc=&mFork1},
22     sFork1={.pc=&sFork1},
23     mJoin1={.pc=&mJoin1},
24     sJoin1={.pc=&sJoin1};
25     Node mFork2={.pc=&mFork2},
26     sFork2={.pc=&sFork2},
27     mJoin2={.pc=&mJoin2},
28     sJoin2={.pc=&sJoin2};
29     Node mAbort1={.pc=&mAbort1},
30     sAbort1={.pc=&sAbort1};
31     Node mSync={.pc=&mSync},
32     sSync={.pc=&sSync};
33     // Create initial linked lists
34     if (core == 1) {
35         main.prev=main.next=&main;
36         insert(main,mFork1); insert(mFork1,mSync);
37         goto *main.pc;
38     } else if (core == 2) {
39         sFork1.prev=sFork1.next=&sFork1;
40         insert(sFork1,sSync); goto *sFork1.pc;
41     } else { while(1); }
42
43     // Forking
44     mFork1: {
45         send(mainState);
46         if (mainState == 1) {
47             Insert the nodes mAbort1,tA,sFork2, and mJoin1
48             after mFork1.
49             remove(main); remove(mFork1); goto *tA.pc;
50         } else { goto *mFork1.next->pc; }
51     }
52     sFork1: {
53         receive(mainState);
54         if (mainState == 1) {
55             Insert the nodes sAbort1,tB,mFork2, and sJoin1
56             after sFork1.
57             remove(sFork1); goto *sFork2.pc;
58         } else { goto *sFork1.next->pc; }
59     }
60
61     mFork2: {
62         send(tBState);
63         if (tBState == 2) {
64             Insert the nodes tD and mJoin2 after mFork2.
65             remove(tB); remove(mFork2); goto *tD.pc;
66         } else { goto *mFork2.next->pc; }
67     }
68     sFork2: {
69         receive(tBState);
70         if (tBState == 2) {
71             Insert the nodes tC and sJoin2 after sFork2.
72             remove(sFork2); goto *tC.pc;
73         } else { goto *sFork2.next->pc; }
74     }
75
76     // Joining
77     mJoin2: {
78         receive(tCState);
79         x_tB=plus(x_tC,x_tD,x); // Combine
80         if (tCState == TERM
81             && tDState == TERM) {
82             tBState=OTHER; send(tBState);
83             insert(mJoin2,tB); remove(mJoin2);
84             goto *tB.pc;
85         } else {
86             send(tBState); goto *mJoin2.next->pc;
87         }
88     }
89     sJoin2: {
90         send(tCState); receive(tBState);
91         if (tBState == OTHER) { remove(sJoin2); }
92         goto *sJoin2.next->pc;
93     }
94     mJoin1: {
95         receive(tBState);
96         x_main=plus(x_tA,x_tB,x); // Combine
97         if (tAState == TERM
98             && tBState == TERM) {
99             mainState=OTHER; send(mainState);
100            insert(mJoin1,main); remove(mAbort1);
101            remove(mJoin1); goto *main.pc;
102        } else {
103            send(mainState); goto *mJoin1.next->pc;
104        }
105    }
106    sJoin1: {
107        send(tBState);
108        receive(mainState);
109        if (mainState == OTHER) { remove(sJoin1); }
110        goto *sJoin1.next->pc;
111    }
112
113    // Preempting
114    mAbort1: {
115        if (x > 1) {
116            Remove the linked nodes between mAbort1
117            and mJoin1 inclusive.
118            main.pc = &mAbort1; goto *main.pc;
119        } else { goto *mAbort1.next->pc; }
120    }

```

Fig. 25. Example of the C program generated for Figure 23a.

```

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
156         goto *main.next->pc;
157         join1:;
158         copy(x_main,x);
159         // when (x_main > 1);
160         abort1: exit(0);
161     }
162     tA: {
163         copy(x_tA,x_main);
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 } // End of main()

```

Fig. 25. (Continued.) Example of the C program generated for Figure 23a.

Table VIII. 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

slave core have terminated. In the second and third global ticks of Figure 23d, sJoin1 notifies mJoin1 that thread tB has terminated (TERM is sent).

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 VIII visualizes core 1 and 2's linked lists when threads main and tB fork (lines 153 and 177 respectively). Each par statement is assigned a unique positive integer id by the compiler. Lines 153–156 in Figure 25 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. 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 in-

sert the allocated child threads and an `mJoin` routine into the linked list. The parent thread and `mFork` routine are removed from the linked list. If a child thread can fork its own threads, then further `mFork` and `sFork` routines need to be inserted into the linked lists. This ensures that the nested threads can be forked. 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.

Recall that the slave cores have an `sFork` routine in their initial linked list. Lines 52–59 is 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 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 25 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 that state to the 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 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. Line 110 is a context-switch to the next node in sequence.

5.5. The `pause` Statement

The `pause` statement is a context-switching point and lines 146–149 in Figure 25 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 statement immediately after the `pause` statement.

5.6. 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 25). 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., 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 4.2.8, 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

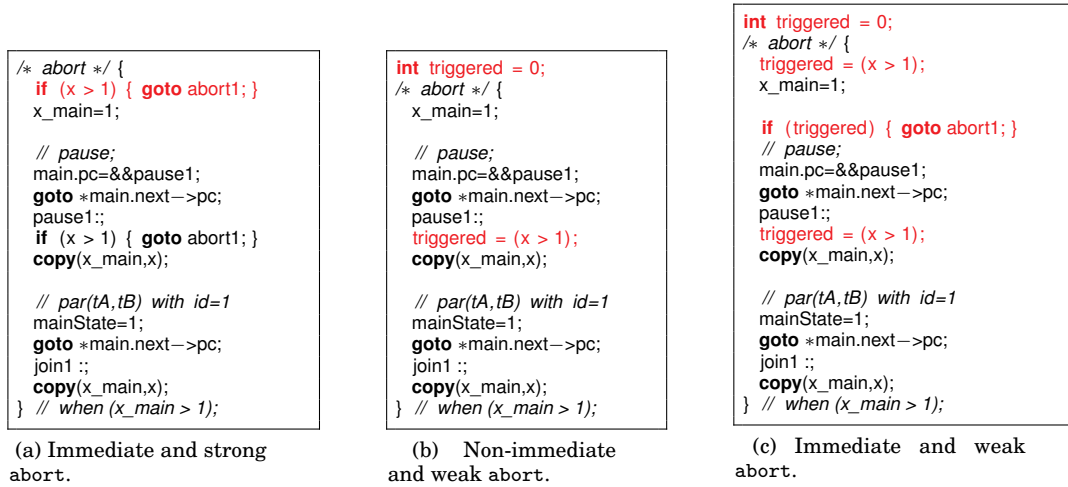


Fig. 26. C code for the immediate and weak variants of the abort on lines 143–158 of Figure 25.

respective local ticks. The final values of the shared variables are computed by the `mJoin` routine of thread `main`.

5.7. 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 25 is an abort and 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 23d, the cores use the `mAbort` and `sAbort` routines to evaluate the preemption condition on line 7 of Figure 23a. It is safe to evaluate the preemption conditions in parallel because they are side-effect free by definition (Section 4.1.6). Thus, when a fork occurs, an `Abort` routine is inserted before the child threads in the linked lists. For a master core, lines 114–120 in Figure 25 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 26a is an example of the C code that would be generated if the abort on lines 143–158

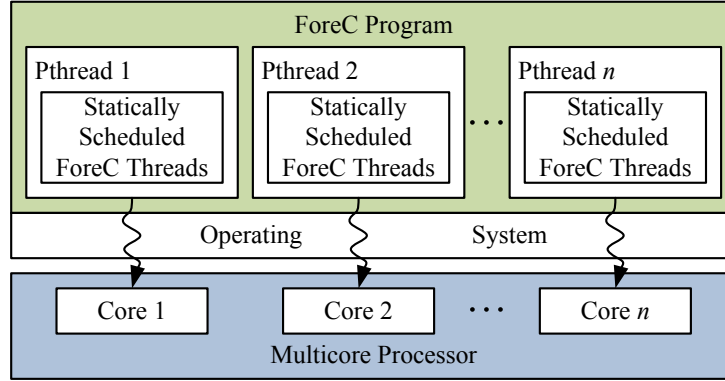


Fig. 27. Using Pthreads to adapt the generated code for multi-cores.

in Figure 25 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 separated 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 the weak abort, then the `mAbort` and `sAbort` routines are inserted after the child threads in the linked lists. Figure 26b is an example of the C code that would be generated if the abort on lines 143–158 in Figure 25 was a weak abort. Figure 26c is an example of the C code generated if it was an immediate and weak abort.

5.8. Global Tick Synchronization

The notion of a global tick is preserved by ending each linked list with a `Sync` routine that implements *barrier synchronization*. This synchronization is shown at the end of each global tick in Figure 23d. For the master core that executes the main thread, lines 130–135 is an example of the C code that is generated for an `mSync` routine. 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 an example of the C code that is generated for an `sSync` routine. 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.

5.9. Generating Programs for Execution on Operating Systems

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. We modify the ForeC compiler to generate a `Pthread` [The IEEE and The Open Group 2008] 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 27. 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 a GNU C compiler.

```

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 25
11 void *forecMain(void *args) { ... }
```

Fig. 28. Example Pthreads program.

For the ForeC program of Figure 23a, Figure 28 is a simplified extract of the generated Pthreads program. In addition to the global declarations shown in Figure 25, 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 25 (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.

5.10. 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 [Edwards and Zeng 2007]. 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 with our static scheduling approach include : (1) a light-weight scheduling of ForeC threads, and (2) analysis is easier 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` [ISO/IEC JTC1/SC22/WG14 2011]) 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 the 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.

For future work, the dynamic scheduling of ForeC threads can be developed to improve the average-case performance on desktop computers. In future versions of the compiler, we also wish to implement proper thread stacks to allow the execution of functions that pause.

The distribution of traditional synchronous programs over multiple processors is not new [Girault 2005; Baudisch et al. 2010a; Caspi et al. 1999; Ju et al. 2010; Yuan et al. 2011; Yoong et al. 2006; Papailiopoulou et al. 2011]. It is motivated by the desire to execute computations closer to their inputs and outputs, which may be distributed over a geographical area. Unfortunately, the use of signals for instantaneous communication makes compilation notoriously difficult. First, *causality analysis* [Potop-Butucaru et al. 2007] is needed to ensure that the presence or absence of all signals can be determined exactly in each global tick. Second, the compiler must generate code for resolving si-

gnal statuses at runtime. A common approach is to compile away the parallelism and to generate a sequential program [Edwards and Zeng 2007; Potop-Butucaru et al. 2007]. Third, the sequential program is partitioned into subprograms and distributed to execute on their allocated processors. Desynchronization techniques [Benveniste et al. 2000; Girault et al. 2006; Brandt et al. 2009] can be used when the processors execute and communicate at different speeds. SynDEx [Potop-Butucaru et al. 2010] is a tool that automatically distributes synchronous programs and considers the cost of communication between the processors. In contrast, ForeC is significantly easier to compile because thread communication is delayed with shared variables (Section 4.1.3). Causality analysis is not required and ForeC threads can be distributed directly to the available cores. The parallelism specified by the programmer is preserved by the ForeC compiler and a sequential intermediate code is not required.

With the advent of multi-cores, the distribution of synchronous programs is motivated by the desire to improve their execution performance. The distribution of synchronous programs over multi-threaded and multi-core reactive processors has been studied extensively [Li and von Hanxleden 2012; Yuan et al. 2009; Dayaratne et al. 2006; Salcic et al. 2006; Yoong et al. 2006]. Reactive processors handle the scheduling of threads in hardware, thereby simplifying the code generation. However, causality analysis is still required and signal statuses still need to be resolved at runtime. Signal resolution may reduce a program's parallel performance because a thread must wait for a signal's status to be resolved before it can be read. There have been studies on the parallelization of synchronous programs on general-purpose multi-cores [Ju et al. 2010; Yuan et al. 2011; Baudisch et al. 2010b]. These approaches extract a parallel program from a sequential representation of the original synchronous program. Due to control and signal dependencies, the opportunities for extracting parallelism from a sequential program is limited. In contrast, execution dependencies only exist at the local tick boundaries of ForeC threads (recall that each thread operates on a local copy of each shared variable). As a result, ForeC threads have more opportunity to execute in parallel.

6. FOREC BENCHMARKING

This section quantitatively assesses ForeC's parallel execution performance on a mixture of data and control dominated benchmark programs. ForeC's execution performance is compared with that of Esterel, a widely used synchronous language for concurrent safety-critical systems that has inspired some features of ForeC, and that of OpenMP, a popular desktop solution for parallel programming. The static timing analysis of ForeC using the reachability technique is described in a previous paper [Yip et al. 2013]. The benchmark results [Yip et al. 2013] showed that the worst-case reaction time [Boldt et al. 2008] (WCRT) of ForeC programs could be estimated to a high degree of precision, which is very useful for implementing real-time embedded systems in general, and time-predictable systems in particular. We highlight some of the key findings in this section.

6.1. Benchmark Programs

This section describes the benchmark programs used in the evaluations :

- FlyByWire is based on the real-time UAV benchmark called PapaBench [Nemer et al. 2006]. FlyByWire is a control dominated program with several tasks managing the UAV's motors, navigation, timer, and operation mode.
- FmRadio [Pop and Cohen 2011] is based on the GNU Radio Package [GNU 2013], which transforms a fixed stream of radio signals into audio. The history of the radio signals is used to determine how the remaining stream of signals should be transformed. FmRadio is data orientated.

- Life simulates Conway’s Game of Life [Gardner 1970] for a fixed number of iterations and a given grid of cells. In each iteration of the simulation, the outcome of each cell can be computed independently. Life has a good mixture of data and control dominated computations.
- Lzss uses the Lempel-Ziv-Storer-Szymanski (LZSS [Storer and Szymanski 1982]) 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 be compressed. Lzss has 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 a data-parallel program.
- MatrixMultiply computes the matrix multiplication of two equally sized square matrices. Each element in the resulting matrix can be computed independently, making MatrixMultiply a data-parallel program.

6.2. Performance Evaluation

The performance of ForeC is evaluated against that of Esterel, a traditional synchronous language, and OpenMP, a general-purpose parallel programming extension to C. 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 :

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

where P is either the ForeC, Esterel, or OpenMP version of the benchmark program being tested. Hence, the speedups of ForeC, Esterel, and OpenMP are always with respect to the execution time of the C version. The higher the speedup the better.

6.2.1. Comparison with Esterel. The approaches by Yuan et al. [Yuan et al. 2011; Yuan 2013] for parallelizing the execution of Esterel programs has been shown to perform well on an Intel multi-core and on a (simulated) Xilinx MicroBlaze multi-core. However, only compiler support is available for Yuan et al.’s dynamic scheduling approach on MicroBlaze multi-core. Thus, we evaluate ForeC against Esterel on a MicroBlaze multi-core and use the dynamic scheduling approach to parallelize the Esterel programs. The static scheduling approach presented in Section 5 is used to parallelize the ForeC programs. Yuan et al.’s dynamic scheduling approach uses a special hardware FIFO queue to allocate the threads to the cores. 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 added to the queue when they are forked by other threads. 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 3.1 is extended with a hardware queue to support the dynamic scheduling. The configuration of the simulator is shown in Figure 29.

Table IX shows the implementation details of the ForeC and Esterel versions of the benchmark programs. Esterel is suited for specifying control concurrency, but is not so for specifying data dominated computations. Esterel allows data computations to be delegated to external host functions, defined in a host language such as C. Hence, for the “Lines of Code” column in Table IX, the first number is the lines of Esterel code and the second number is the lines of host C-code (excluding header files). The “Number

Xilinx MicroBlaze, 4 physical 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.

Fig. 29. MicroBlaze multi-core configuration.

Table IX. ForeC versus Esterel benchmarks.

Benchmark	Lines of Code		Number of Threads	
	ForeC	Esterel	ForeC	Esterel
Life	212	139+111	4 (4)	7 (4)
Lzss	485	42+421	4 (4)	4 (4)
Mandelbrot	381	220+337	8 (8)	18 (9)
MatrixMultiply	162	51+53	16 (8)	16 (8)

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 [Yuan 2013] uses an intermediate format called GGraph Code (GRC) [Potop-Butucaru et al. 2007], which transforms the program into an acyclic execution graph. The GRC helps schedule the resolution of signals, executing the GRC from the top to the bottom 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. Compared to GRC, for most programs, the ForeC compiler (see Section 5) can generate more efficient code that require less context-switching. The same input vector is given to the ForeC and Esterel versions of a benchmark program to ensure that the same computations are performed. When a program terminates, the simulator returns the execution time in clock cycles.

Figure 30 shows the speedups achieved by ForeC and Esterel when the benchmark programs execute on four cores. Apart from MatrixMultiply, ForeC shows superior performance compared to Esterel, 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 execute before any signal consumers can execute and this invariant is achieved using a signal locking protocol [Yuan 2013] 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 $1.2\times$ on four cores. In fact, when Mandelbrot is executed on one 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. For MatrixMultiply, the fork-join pattern was used by ForeC and Esterel. 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 very similar speedup values.

6.2.2. Comparison with OpenMP. Table X shows the implementation details of the ForeC and OpenMP versions of the benchmark programs. Intel VTune Amplifier XE 2013 [Intel 2014] software was used during the development and testing of the parallelized benchmarks. The software is useful in providing insight into the regions of code which are the most time consuming and therefore top candidates for parallelization. It is also used to calculate the total runtimes and to show the number of cores being used

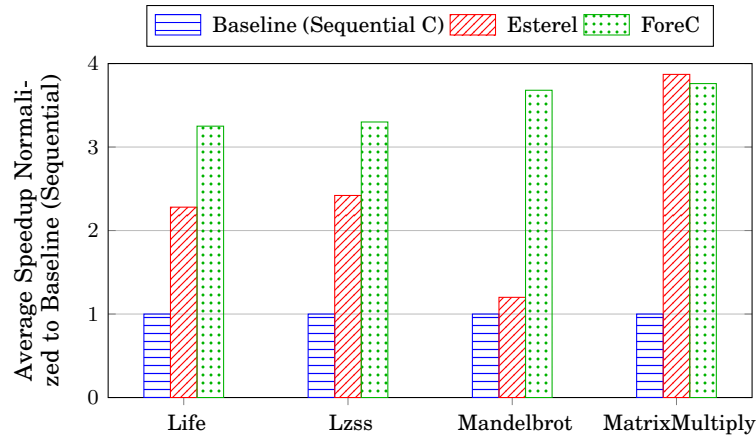


Fig. 30. Average speedup results for ForeC and Esterel on four cores normalized to sequential runtime. Platform details in Figure 29.

Intel Core-i5 3570 at 3.4 Ghz, 4 physical cores, Hyper-Threading disabled, Turbo Boost disabled, SpeedStep disabled, 3 MB L3 data-cache, Linux 3.6, 8 GB of RAM, Benchmarks compiled with GCC-4.8 -O2.

Fig. 31. Intel multi-core configuration.

Table X. ForeC versus OpenMP benchmarks.

Benchmark	Lines of Code		Number of Threads
	ForeC	OpenMP	ForeC
FlyByWire	241	227	8 (7)
FmRadio	481	382	12 (6)
Life	325	268	10 (8)
Lzss	593	552	4 (4)
Mandelbrot	111	89	4 (4)
MatrixMultiply	156	121	7 (4)

during the execution of each benchmark. Figure 31 shows the specifications of the desktop computer on which all testing is carried out.

Figure 32 shows the speedups achieved by ForeC and OpenMP when the benchmark programs are executed over four cores. The speedups are averaged over 200 executions of each program to take into account the potential effects of long term use, e.g., filling and flushing of the cache, and background kernel processes on the computer. ForeC and OpenMP are able to achieve a speedup factor of between two and four over four cores. However, from the results it is clear that in most cases ForeC produces a greater speedup factor, barring two exceptions. Firstly, OpenMP is more suited and delivered a much higher speedup factor in FlyByWire. Secondly, for Mandelbrot, the OpenMP version is unable to utilize all four available cores.

We would like to mention that we used dynamic and static thread scheduling pragmas in OpenMP. Static scheduling is used in benchmarks (e.g., FlyByWire) when we could determine at compile time the so called *chunk size* (the amount of work and number of loop iterations that each thread needs to perform). Dynamic scheduling is used in benchmarks (e.g., MatrixMultiply and Mandelbrot) when the chunk size of each

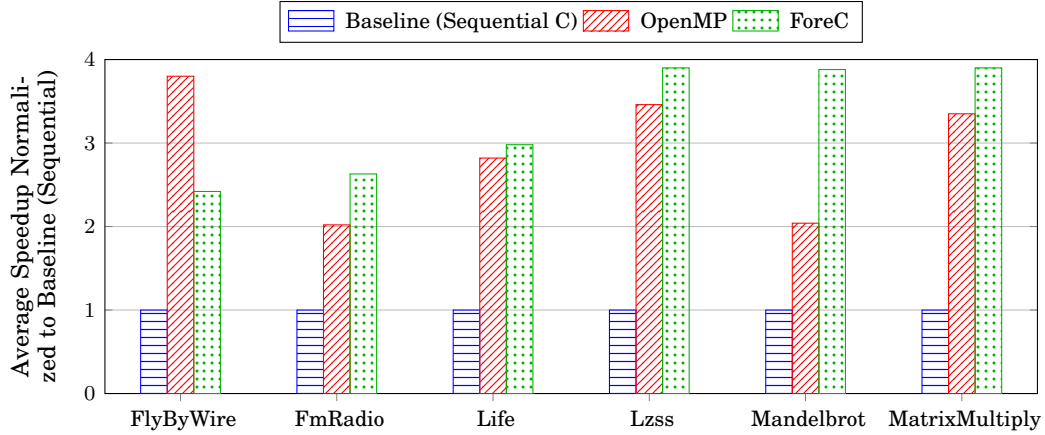


Fig. 32. Average speedup results for ForeC and OpenMP on four cores normalized to sequential runtime. Platform details in Figure 31.

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 over the overall run of the benchmarks. This OpenMP scheduling approach is in stark contrast to the ForeC approach, where all work scheduling is static and determined automatically by the ForeC compiler, whereas in OpenMP all work scheduling is the programmer's burden.

6.3. Time Predictability

We developed a C++ static timing analysis tool [Yip et al. 2013], called ForeCast, that statically analyzes the WCRT of ForeC programs on embedded multi-core processors. We highlight the key findings of our previous paper [Yip et al. 2013] on the static WCRT analysis of ForeC programs executed on embedded multi-cores. Benchmarking is performed on the MicroBlaze multi-core simulator with the configuration shown in Figure 33 and ForeCast itself is executed on a 2.20 GHz Intel Core 2 Duo computer with 3 GB RAM and Linux 2.6.38. We highlight the results of the benchmark program called 802.11a [Pop and Cohen 2011]. 802.11a is production code from Nokia that tests various signal processing algorithms needed to decode 802.11a data transmissions. 802.11a has both complex data and control dominated computations. 802.11a has 2147 lines of ForeC code and forks up to 26 threads, of which 10 can execute in parallel. 802.11a is distributed on up to 10 cores and ForeCast is used to compute the WCRT of each possible distribution. The WCRT computed by ForeCast is taken as the *computed* WCRT. To evaluate the tightness of the computed WCRTs, 802.11a is executed on the MicroBlaze simulator for one million global ticks or until the program terminates. Test vectors are generated to elicit the worst-case program state by studying the program's control-flow. The simulator returns the execution time of each global tick and the longest is taken as the *observed* WCRT.

The observed and computed WCRTs of 802.11a (in clock cycles) are plotted as a line graph in Figure 34. This graph shows that the static timing analysis is very precise, even when the number of cores increases. The *over-estimation* of the computed WCRT

Xilinx MicroBlaze, 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 $5 \times (\text{number of cores})$ cycles long bus schedule), Benchmarks compiled with MB-GCC-4.1.2 -O0 and decompiled with MB-OBJDUMP-4.1.2.

Fig. 33. MicroBlaze multi-core configuration.

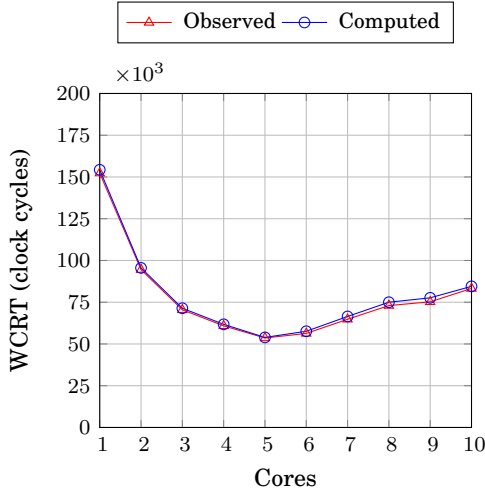


Fig. 34. WCRT results for 802.11a in clock cycles.

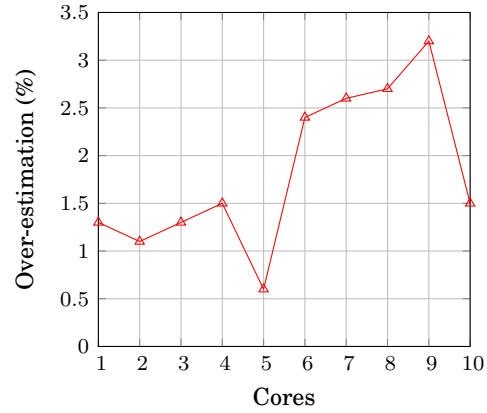


Fig. 35. WCRT over-estimations for 802.11a.

can be calculated as follows :

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

Figure 35 exemplifies 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 shows that extremely time predictable systems can be designed with ForeC.

The computed WCRTs of 802.11a in Figure 34 reflect the benefit of multi-core execution. The computed WCRT decreases when the number of cores is increased from one to five cores. The computed WCRT at five cores corresponded to the execution time of one thread which is already allocated to its own core. Thus, the WCRT could not be improved by distributing the remaining threads. The WCRT increases after five cores because of the increasing scheduling overheads and cost of accessing global memory. These costs reduce the benefit of multi-core execution.

We perform 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. For our experiments, the WCET of a program is the total time that it takes for the entire program to execute from start to finish.² We limit the Life program to simulate 10,000 iterations of the game. The same input vector is given to the ForeC and Esterel versions of a benchmark program to ensure that the same computations are performed. The Esterel programs are

2. The entire execution of a benchmark program occurs over multiple ticks.

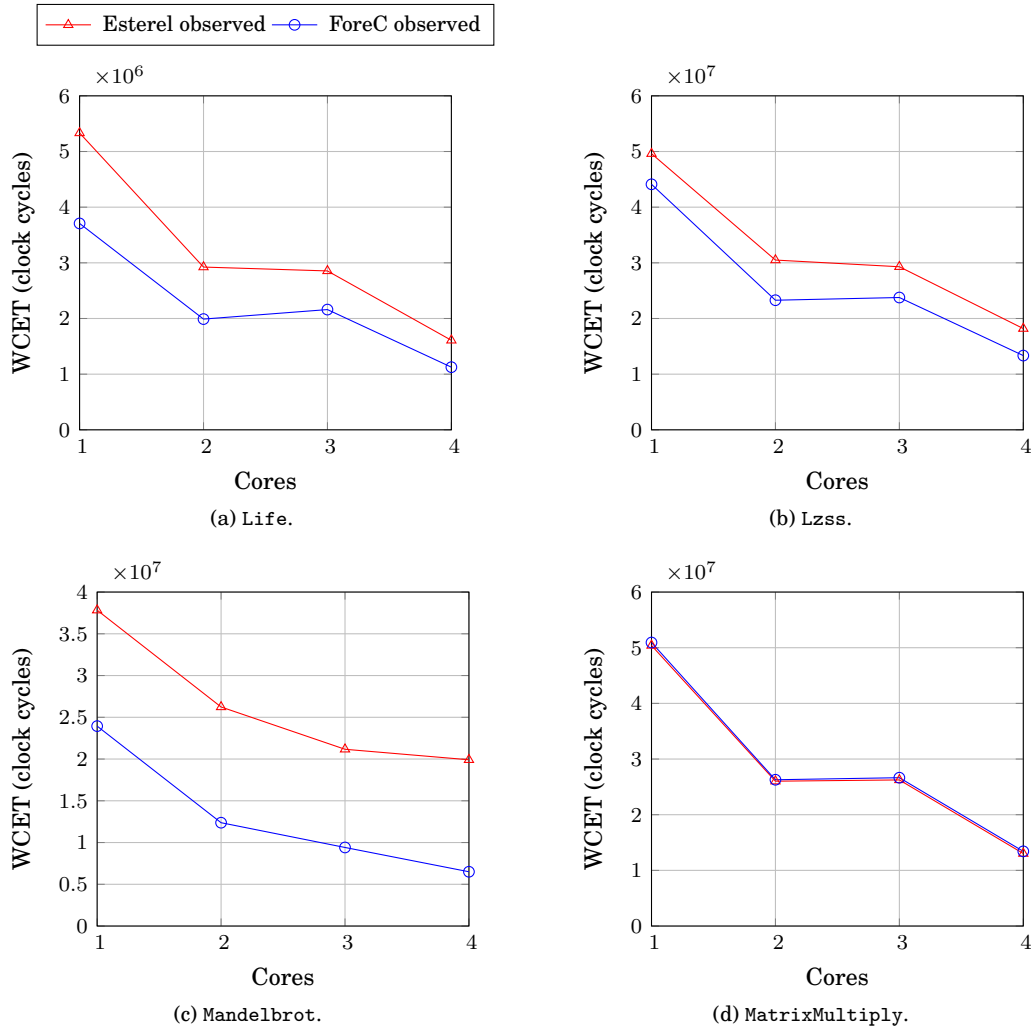


Fig. 36. Observed WCETs for ForeC and Esterel. Platform details in Figure 29.

compiled using Yuan et al.'s approach [Yuan 2013]. For each benchmark program in Figure 36, the WCETs for ForeC and Esterel are plotted. Apart from MatrixMultiply, the observed WCETs for ForeC are much shorter than those for Esterel. Unfortunately, the static timing analysis of such multi-core Esterel programs has not been developed, preventing an objective comparison of time-predictability. To compute WCRTs for Esterel that are as tight as ForeCast, the dynamic resolution of signal statuses will need to be analyzed extremely carefully to rule out the infeasible runtime decisions.

6.4. Discussion

This section assessed the performance of ForeC on an embedded multi-core and desktop multi-core. The static timing analysis of ForeC programs was presented in an earlier paper [Yip et al. 2013]. In this section, on an embedded multi-core, most of the statically scheduled ForeC programs performed better than Yuan et al.'s [Yuan 2013] dynamically scheduled Esterel programs. This is because it is easier to extract

parallelism from ForeC programs, largely thanks to its shared variable semantics (see Section 4.1.3). Serializing Esterel programs into GRC can obfuscate the parallelism and the need to update internal state variables can add unnecessary overhead. Run-time resolution is also needed to resolve Esterel's instantaneous signal communication. Ju et al. [Ju et al. 2010] 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.

On a desktop multi-core, ForeC's static scheduling approach proved to be competitive against OpenMP, a dynamic runtime solution. These are encouraging results for the use of ForeC to develop high performing parallel programs. Moreover, determinism is enforced by ForeC's formal semantics, not by a particular runtime environment. There is much scope to improve the ForeC compiler to generate more efficient code. For example, thread allocations could be refined automatically by feeding the WCRT results of the ForeCast analyzer into the ForeC compiler until the WCRT cannot be reduced.

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 4 provided an in-depth description of ForeC and, unlike existing C-based synchronous languages, it is designed specifically for parallel programming. The semantics of ForeC is designed to give programmers the ability to express many forms of parallel patterns 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 5 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 ForeC semantics.

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 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.

```

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. 37. Example of passing a shared variable by value and by reference.

A. 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.

A.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. E.g., in Figure 37, line 4 makes a call to function `f` with the arguments `x` (a shared variable) and `y` (a private variable). Function `f` (line 8) declares two variables `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. Changes made to the dereferenced parameter are made to the argument. The syntax for a function parameter that passes a shared variable by reference should begin with “shared *data_type** *p*”. This follows the C convention for other type qualifiers such as “const”, where “const int* *p*” declares a pointer *p* to a constant int variable. E.g., in Figure 37 on line 11, the parameter of function `g` declares a pointer *p* to a shared int variable. On line 5, the “&” unary operator is used to pass the shared variable `x` by reference into `g`.

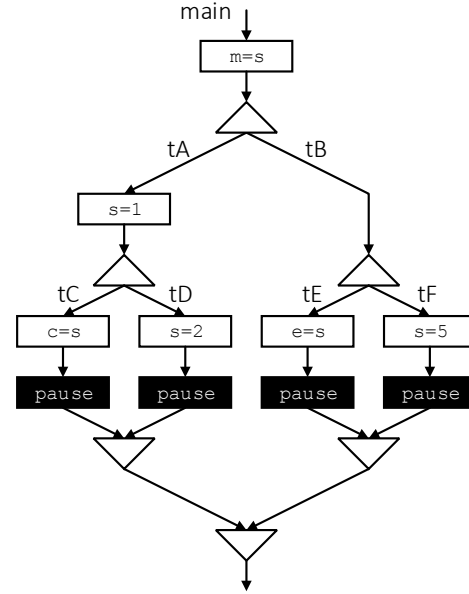
Just like “int const* *p*” declares a constant pointer *p* to an int variable, “int shared* *p*” declares a shared pointer *p* to an int variable. This means that the address stored in *p* is shared among multiple threads. The final possibility is “shared int shared* *p*” which declares a shared pointer *p* to a shared int variable.

```

1  shared int s=3 combine 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. 38. Example ForeC program.

A.2. Combining More Than Two Copies

The ForeC program shown in Figure 38a is used to explain how multiple copies of a shared variable are combined. The program's control-flow graph is shown in Figure 38b. The program has a shared variable called s that uses the combine function `plus`. The initial value of s is 3 for the program's first tick. Figure 39a 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 \bullet 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 4.1.4). 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 4.2) 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 combined 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 39b illustrates this for 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` ignores the copies that have the same value as their shared variable, which is not changed during the tick. For the copies shown in Figure 39a, thread `main`, tB , and tE 's copies of s would be ignored. Figure 39c illustrates how the copies are combined for 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.

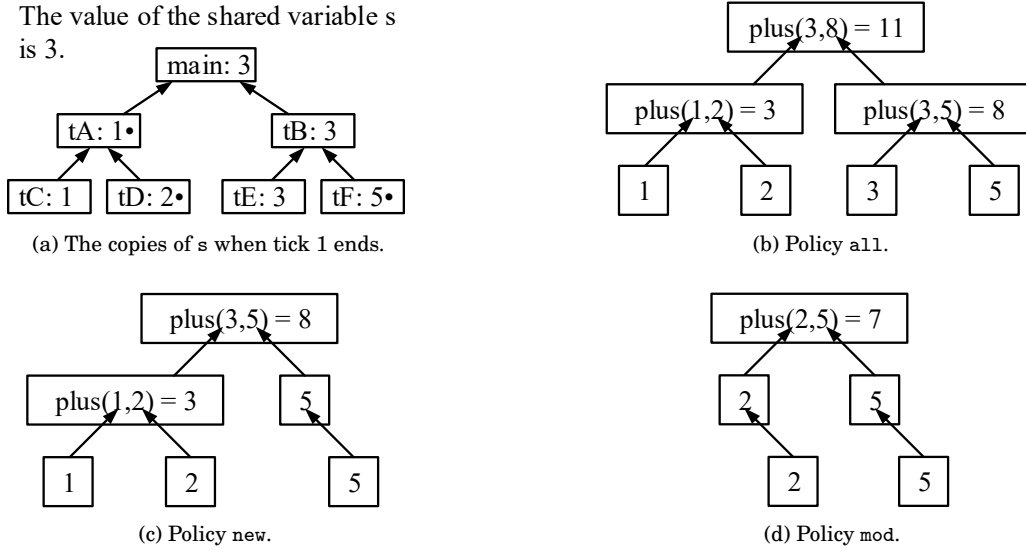


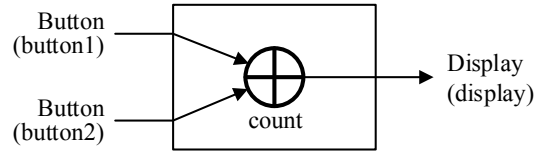
Fig. 39. Effects of the combine policies.

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 39a, thread main, tB, tC, and tE's copies of s would be ignored. Figure 39d illustrates how the copies are combined for the combine policy mod. The final combined value is 7.

A.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 40. Figure 40a shows a block diagram of the ForeC program of Figure 40b. The program outputs the number of times button1 and button2 are pressed in each tick of the program. On line 6 in Figure 40b, threads t_1 and t_2 are forked to check which buttons have been pressed. The results are assigned to the shared variable count. Line 6 also forks thread t_3 to read the value of count and to output it to display. 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 41a provides possible input values for five ticks of the program. For example, only button1 is pressed in tick 2. Figure 41b 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 = 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 t_1 and t_2 's modified copies are combined.

Figure 41c illustrates the behavior of the combine policy new over several ticks. The values of threads t_1 , t_2 , and t_3 's local copies are ignored when they have the same value as count. Figure 41d illustrates the behavior of the combine policy all over several ticks. In this case, threads t_1 , t_2 , and t_3 '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.



(a) Button counter.

```

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); }

```

(b) ForeC program of the button counter.

Fig. 40. 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	3	4	4

(d) Combine policy all.

Fig. 41. The value of count under each combine policy.

A.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.

```

1  typedef struct {int prod; int sum;} ProdSum;
2
3  ProdSum prodsum(ProdSum th1,ProdSum th2) {
4      ProdSum res = {.prod=(th1.prod*th2.prod), .sum=(th1.sum+th2.sum)};
5      return res;
6  }
7  // th1={.prod=2,.sum=2} and th2={.prod=5,.sum=5}
8  // prodsum(th1,th2)={.prod=10,.sum=7}

```

(a) Product and sum of two values.

```

1  typedef struct {int value; int min;} Min;
2
3  Min min(Min th1,Min th2) {
4      Min res;
5      if (th1.value<th2.value) res.min=th1.value; else res.min=th2.value;
6      res.value = res.min;
7      return res;
8  }
9  // th1={.value=2,.min=0} and th2={.value=5,.min=0}
10 // min(th1,th2)={.value=2,.min=0}

```

(b) Minimum of two values.

Fig. 42. Examples of combine functions for C-structs.

The combine functions should be associative and commutative so as to make the par statement commutative and associative, relying only on the nesting of binary par statements to fork more than two child threads at the same time (e.g., $\text{par}(\text{par}(t_1, t_2), t_3) = \text{par}(t_1, \text{par}(t_2, t_3))$). The programmer is free to write combine functions based on non-commutative or non-associative binary operators (e.g., $-$, $/$, or $\%$), but this will violate the hypothesis of the theorem for determinism (Theorem 4.10).

Combine functions can also be defined for user-defined data structures. For example, Figure 42a 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, line 1 of Figure 42b defines a C-struct called `Min` that stores an assigned value (`value`) and tracks the minimum value that has been assigned (`min`). The combine function (line 3) reads the values and assigns the minimum value to `res.min`. The minimum value is also assigned to `res.value`, so that it will be read when it is combined with another copy. An example of its behavior is provided after the function as comments. Indeed, the combine function will only behave in an associative and commutative manner if the threads only write to `value` and do not read from it.

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 A.4 is an example ForeC program that calculates the average of three input values `in[i]`, declared on line 1 in Figure A.4. 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).

```

1  input double in[3];
2  typedef 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. 43. Averaging two or more values.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

We wish to acknowledge Pascal Fradet for his discussions on an earlier version of the ForeC semantics, and Jean-Bernard Stefani for his discussions on program equivalence in the context of ForeC.

Références

- Homa Alemzadeh, Ravishankar K. Iyer, Zbigniew Kalbarczyk, and Jai Raman. 2013. Analysis of Safety-Critical Computer Failures in Medical Devices. *IEEE Security Privacy* 11, 4 (2013), 14 – 26.
- Sidharta Andalam, Partha S. Roop, and Alain Girault. 2010. Predictable Multithreading of Embedded Applications Using PRET-C. In *8th IEEE/ACM International Conference on Formal Methods and Models for Code Design (MEMOCODE)*. 159 – 168.
- 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.
- 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.
- 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.
- Alexandru Andrei, Petru Eles, Zebo Peng, and Jakob Rosen. 2008. Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *Proceedings of the 21st International Conference on VLSI Design (VLSID)*. IEEE Computer Society, 103 – 110.
- Amittai Aviram and Bryan Ford. 2011. Deterministic OpenMP for Race-Free Parallelism. In *Proceedings of the 3rd USENIX conference on Hot topic in parallelism (HotPar)*. USENIX Association.
- 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.
- Daniel Baudisch, Jens Brandt, and Klaus Schneider. 2010a. 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.
- Daniel Baudisch, Jens Brandt, and Klaus Schneider. 2010b. Multithreaded Code from Synchronous Programs : Extracting Independent Threads for OpenMP. In *Design, Automation and Test in Europe (DATE)*. EDA Consortium, 949 – 952.
- Marco Bekooij, Orlando Moreira, Peter Poplavko, Bart Mesman, Milan Pastrnak, and Jef Meerbergen. 2004. Predictable Embedded Multiprocessor System Design. In *Software and Compilers for Embedded Systems*, Henk Schepers (Ed.). Lecture Notes in Computer Science, Vol. 3199. Springer Berlin Heidelberg, 77 – 91.
- 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.
- 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.
- Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 2010a. 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 (ASPLOS)*. ACM, 53 – 64.
- Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. 2010b. Deterministic Process Groups in dOS. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation (OSDI)*. 1 – 16.
- Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. 2009. Grace : Safe Multithreaded Programming for C/C++. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 81 – 96.
- Gérard Berry. 1993. Preemption in Concurrent Systems. In *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, 72 – 93.

- G rard Berry and Georges Gonthier. 1992. The Esterel Synchronous Programming Language : Design, Semantics and Implementation. *Science of Computer Programming* 19, 2 (1992), 87 – 152.
- Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. 2009. A Survey of Multicore Processors. *IEEE Signal Processing Magazine* 26, 6 (Nov. 2009), 26 – 37. DOI: <http://dx.doi.org/10.1109/MSP.2009.934110>
- 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.
- Robert L. Bocchino. 2010. *An Effect System and Language for Deterministic-by-Default Parallel Programming*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.
- 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.
- Fr d ric Bousinot. 1993. *Reactive Shared Variables Based Systems*. Technical Report 1849. INRIA.
- 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.
- Dominique Bri re, 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.
- Giuseppe Buja and Roberto Menis. 2012. Dependability and Functional Safety : Applications in Industrial Electronics Systems. *IEEE Industrial Electronics Magazine* 6, 3 (2012), 4 – 12.
- 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 (ESOP/ETAPS)*. Springer-Verlag, 116 – 135.
- 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.
- 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.
- 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.
- 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.
- 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.
- Bryce Cogswell and Zary Segall. 1991. MACS : a predictable architecture for real time systems. In *Proceedings of the 12th Real-Time Systems Symposium*. 296 – 305.
- Albert Cohen, L onard G rard, and Marc Pouzet. 2012. Programming Parallelism with Futures in Lustre. In *Proceedings of the 10th ACM International Conference on Embedded Software (EMSOFT)*. ACM, 197 – 206.
- Jean-Louis Cola o 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.
- Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza Burgui re, 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).
- 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.
- 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.
- 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*.

- 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.
- William R. Dunn. 2003. Designing Safety-Critical Computer Systems. *Computer* 36, 11 (2003), 40 – 46.
- 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)*. IEEE.
- 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)*. ACM, 264 – 265.
- Stephen A. Edwards and Jia Zeng. 2007. Code Generation in the Columbia Esterel Compiler. *EURASIP Journal on Embedded Systems* 2007 (2007).
- Cormac Flanagan and Shaz Qadeer. 2003. Types for Atomicity. In *Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*. ACM, 1 – 12.
- Martin Gardner. 1970. Mathematical Games : The Fantastic Combinations of John Conway’s new Solitaire Game “Life”. *Scientific American* (Oct. 1970), 120 – 123.
- 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))*, Philipp Lucas, Lothar Thiele, Benoit Triquet, Theo Ungerer, and Reinhard Wilhelm (Eds.), Vol. 18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 1 – 10.
- 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.
- 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.
- GNU. 2013. GNU Radio. (2013). [Online] <http://gnuradio.org>.
- 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.
- 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.
- Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The Synchronous Data Flow Programming Language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305 – 1320.
- Arne Hamann and Rolf Ernst. 2005. TDMA Time Slot and Turn Optimization with Evolutionary Search Techniques. In *Proceedings of the Design, Automation and Test in Europe*. 312 – 317.
- 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.
- Les Hatton. 2004. Safer Language Subsets : An Overview and a Case History, MISRA C. *Information and Software Technology* 46, 7 (2004), 465 – 472.
- Gerard J. Holzmann. 2006. The Power of 10 : Rules for Developing Safety-Critical Code. *IEEE Computer* 39, 6 (2006), 95 – 97.
- Intel. 2012a. Intel Cilk Plus. (3 Nov. 2012). [Online] <http://software.intel.com/en-us/intel-cilk-plus>.
- Intel. 2012b. Intel Thread Building Blocks. (3 Nov. 2012). [Online] <http://threadingbuildingblocks.org>.
- Intel. 2014. Intel® VTune™ Amplifier. (2014). [Online] <https://software.intel.com/en-us/intel-vtune-amplifier-xe>.
- International Electrotechnical Commission. 2010. IEC 61508 : Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems. (April 2010). Standard.
- ISO/IEC JTC1/SC22/WG14. 2011. ISO/IEC 9899 :2011. (2011). Standard.
- ISO/TC22/SC3. 2011. ISO 26262-1 :2011. (2011). Standard.
- Jet Propulsion Laboratory. 2009. JPL Institutional Coding Standard for the C Programming Language. (3 March 2009). Standard 1.0. [Online] http://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_C.pdf.
- 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.
- Lei Ju, BachKhoa Huynh, Abhik Roychoudhury, and Samarjit Chakraborty. 2012. Performance Debugging of Esterel Specifications. *Real-Time Systems* 48, 5 (2012), 570 – 600.

- 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)*. ACM, 48 – 51.
- Andrey Karpov. 2011. Parallel Lint. (2 Nov. 2011). [Online] <http://software.intel.com/en-us/articles/parallel-lint>.
- 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.
- 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 Hoeks-tra, and Bob Hertzberger (Eds.). Lecture Notes in Computer Science, Vol. 1593. Springer Berlin Heidelberg, 525 – 534.
- Yooseong Kim, David Broman, Jian Cai, and Aviral Shrivastava. 2014. WCET-Aware Dynamic Code Management on Scratchpads for Software-Managed Multicores. In *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS)*.
- John C. Knight. 2002. Safety Critical Systems : Challenges and Directions. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*. 547 – 550.
- Andrew Koenig. 1988. *C Traps and Pitfalls*. Addison-Wesley. I–XI, 1–147 pages.
- Hermann Kopetz. 2008. The Complexity Challenge in Embedded System Design. In *Proceedings of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*. IEEE Computer Society, 3 – 12.
- 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)*. 480 – 485.
- Kanishka Lahiri, Anand Raghunathan, and Ganesh Lakshminarayana. 2001. LOTTERYBUS : A New High-Performance Communication Architecture for System-on-Chip Designs. In *Proceedings of the 38th Annual Design Automation Conference (DAC)*. ACM, 15 – 20.
- 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)*.
- Edward A. Lee. 2006. The Problem with Threads. *Computer* 39 (2006), 33 – 42.
- Edward A. Lee. 2009. Computing Needs Time. *Commun. ACM* 52, 5 (May 2009), 70 – 79.
- Markus Levy and Thomas M. Conte. 2009. Embedded Multicore Processors and Systems. *IEEE Micro* 29, 3 (2009), 7 – 9.
- Xin Li and Reinhard von Hanxleden. 2012. Multithreaded Reactive Programming - the Kiel Esterel Processor. *IEEE Trans. Comput.* 61, 3 (March 2012), 337 – 349.
- Zhenmin Li, Avinash Malik, and Zoran Salcic. 2014. TACO : A Scalable Framework for Timing Analysis and Code Optimization of Synchronous Programs. In *IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 1 – 8.
- Isaac Liu, Jan Reineke, David Broman, Michael Zimmer, and Edward A. Lee. 2012. A PRET Microarchitecture Implementation with Repeatable Timing and Competitive Performance. In *Proceedings of the 30th IEEE International Conference on Computer Design (ICCD)*. 87 – 93.
- Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads : Efficient Deterministic Multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 327 – 336.
- 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 (ASPLOS XIII)*. ACM, 329 – 339.
- Thomas Lundqvist and Per Stenstrom. 1999. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*. 12 – 21.
- 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.
- Charles E. McDowell and David P. Helmbold. 1989. Debugging Concurrent Programs. *ACM Comput. Surv.* 21, 4 (Dec. 1989), 593 – 622.
- 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.

- Message Passing Interface Forum. 2012. MPI : A Message-Passing Interface Standard. (21 Sept. 2012). Standard 3.0.
- Motor Industry Software Reliability Association. 2013. MISRA-C : 2012 : Guidelines for the Use of the C Language in Critical Systems. (2013). Standard.
- Muhammad Nadeem, Morteza Biglari-Abhari, and Zoran Salcic. 2011. RJOP - A Customized Java Processor for Reactive Embedded Systems. In *48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1038 – 1043.
- 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.
- 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.
- Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. 2006. Papa-Bench : A Free Real-Time Benchmark. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*.
- Nghi Nguyen, Angel Dominguez, and Rajeev Barua. 2005. Memory Allocation for Embedded Systems with a Compile-Time-Unknown Scratch-Pad Size. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. ACM, 115 – 125.
- Michael Norrish. 1999. Deterministic Expressions in C. In *Proceedings of the 8th European Symposium on Programming Languages and Systems (ESOP)*. Springer-Verlag, 147 – 161.
- R. Obermaisser, C. El Salloum, B. Huber, and H. Kopetz. 2009. From a Federated to an Integrated Automotive Architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 7 (July 2009), 956 – 965.
- 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 (ASPLOS)*. ACM, 97 – 108.
- OpenMP Architecture Review Board. 2013. OpenMP Application Program Interface. (July 2013). Standard 4.0.
- 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.
- Marco Paolieri and Riccardo Mariani. 2011. Towards Functional-Safe Timing-Dependable Real-Time Architectures. In *17th IEEE International On-Line Testing Symposium (IOLTS)*. 31 – 36.
- Virginia Papailiopolou, 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.
- Parasoft. 2014. Parasoft : Service Virtualization, API Testing, Development Testing. (18 June 2014). [Online] <http://www.parasoft.com>.
- David A. Patterson and John L. Hennessy. 1990. *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- 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.
- Gordon D. Plotkin. 1981. *A Structural Approach to Operational Semantics*. Technical Report DAIMI FN-19. Aarhus University, Computer Science Department.
- Francesco Poletti, Davide Bertozzi, Alessandro Bogliolo, and Luca Benini. 2003. Performance Analysis of Arbitration Policies for SoC Communication Architectures. *Design Automation for Embedded Systems* 8 (2003), 189 – 210.
- Polyspace. 2014. Static Analysis Tools for C/C++ and Ada - Polyspace. (18 June 2014). [Online] <http://www.mathworks.com/products/polyspace>.
- Antoni 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 (HiPEAC)*. ACM, 5 – 14.
- 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.
- Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. 2007. *Compiling Esterel*. Springer. I–XXI, 1–335 pages.

- Aayush Prakash and Hiren D. Patel. 2013. An Instruction Scratchpad Memory Allocation for the Precision Timed Architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 11 (2013), 1819 – 1823.
- Wolfgang Puffitsch and Martin Schoeberl. 2012. On the Scalability of Time-Predictable Chip-Multiprocessing. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*. ACM, 98 – 104.
- Radio Technical Commission for Aeronautics. 1992. Software Considerations in Airborne Systems and Equipment Certification. (April 1992). Standard DO-178B.
- 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 (RV)*. Springer-Verlag, 368 – 383.
- 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 (RTNS)*. ACM, 141 – 150.
- Zoran Salcic, Morteza Biglari-Abhari, and Abbas Bigdeli. 2002. REFLIX : A Processor Core for Reactive Embedded Applications. In *Field-Programmable Logic and Applications : Reconfigurable Computing Is Going Mainstream*, Manfred Glesner, Peter Zipf, and Michel Renovell (Eds.). Lecture Notes in Computer Science, Vol. 2438. Springer Berlin Heidelberg, 945 – 954.
- Zoran Salcic and Avinash Malik. 2013. GALS-HMP : A Heterogeneous Multiprocessor for Embedded Applications. *ACM Trans. Embed. Comput. Syst.* 12, 1s, Article 58 (March 2013), 26 pages.
- 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.
- Martin Schoeberl. 2009. Time-Predictable Computer Architecture. *EURASIP Journal on Embedded Systems* 2009, Article ID 758480 (2009).
- Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, and Christian W. Probst. 2011. Towards a Time-predictable Dual-Issue Microprocessor : The Patmos Approach. In *Bringing Theory to Practice : Predictability and Performance in Embedded Systems (OpenAccess Series in Informatics (OASISs))*, Philipp Lucas, Lothar Thiele, Benoit Triquet, Theo Ungerer, and Reinhard Wilhelm (Eds.), Vol. 18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 11 – 21.
- 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.
- Jacob T. Schwartz. 1980. Ultracomputers. *ACM Trans. Program. Lang. Syst.* 2, 4 (Oct. 1980), 484 – 521.
- 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.
- Angela Sodan, Jacob Machina, Arash Deshmeh, Kevin Macnaughton, and Bryan Esbaugh. 2010. Parallelism via Multithreaded and Multicore CPUs. *Computer* 43, 3 (March 2010), 24 – 32.
- 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.
- James A. Storer and Thomas G. Szymanski. 1982. Data Compression via Textual Substitution. *J. ACM* 29, 4 (Oct. 1982), 928 – 951.
- Vivy Suhendra, Abhik Roychoudhury, and Tulika Mitra. 2008. Scratchpad Allocation for Concurrent Embedded Software. In *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS)*. ACM, 37 – 42.
- Olivier Tardieu. 2007. A Deterministic Logical Semantics for Pure Esterel. *ACM Trans. Program. Lang. Syst.* 29, 2, Article 8 (April 2007).
- 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 (EMSOFT)*. ACM, 142 – 151.
- The IEEE and The Open Group. 2008. POSIX.1-2008. (2008). Standard Issue 7.
- The UPC Consortium. 2013. UPC Language Specifications. (2013). Standard 1.3.
- Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. 2006. Dynamic Allocation for Scratch-pad Memory using Compile-Time Decisions. *ACM Trans. Embed. Comput. Syst.* 5, 2 (2006), 472 – 511.
- 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 Gulashvili, Michael Houston, Florian Kluge, Stefan Metzlaß, and Jorg Mische. 2010. MERASA : Multicore

- Execution of Hard Real-Time Applications Supporting Analyzability. *IEEE Micro* 30, 5 (Sept. 2010), 66 – 75.
- Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (Aug. 1990), 103 – 111.
- 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 (SAC)*. ACM, 1626 – 1631.
- 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*. 225 – 234.
- 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)*.
- 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.
- Jack Whitham. 2012. Scratchpad Memory Management Unit. (2012). [Online] <http://www.jwhitham.org/c/smmu.html>.
- Jack Whitham and Neil Audsley. 2006. MCGREP - A Predictable Architecture for Embedded Real-Time Systems. In *27th IEEE International Conference on Real-Time Systems Symposium (RTSS)*. 13 – 24.
- 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)*. ACM, 265 – 274.
- 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.
- Reinhard Wilhelm and Daniel Grund. 2014. Computation Takes Time, but How Much? *Commun. ACM* 57, 2 (Feb. 2014), 94 – 103.
- Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. 2009. Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 7 (July 2009), 966 – 978. DOI: <http://dx.doi.org/10.1109/TCAD.2009.2013287>
- Julian Wolf, Mike Gerdes, Florian Kluge, Sascha Uhrig, Jorg Mische, Stefan Metzloff, Christine Rochange, Hugues Casse, Pascal Sainrat, and Theo Ungerer. 2010. RTOS Support for Parallel Execution of Hard Real-Time Applications on the MERASA Multi-core Processor. In *13th IEEE International Symposium on Object / Component / Service-Oriented Real-Time Distributed Computing (ISORC)*. 193 – 201.
- Xilinx. 2012. MicroBlaze Processor Reference Guide. (2012). [Online] http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/mb_ref_guide.pdf.
- 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)*. 10.
- 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.
- Simon Yuan. 2013. *Architectures Specific Compilation for Efficient Execution of Esterel*. Ph.D. Dissertation. Electrical and Electronic Engineering, The University of Auckland.
- 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.
- 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.
- Yu Zhang and Bryan Ford. 2011. A Virtual Memory Foundation for Scalable Deterministic Parallelism. In *Proceedings of the Second Asia-Pacific Workshop on Systems (APSys)*. ACM, 5.
- 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)*.

Received July 2016; revised July 2016; accepted July 2016