The ForeC Synchronous Deterministic Parallel Programming Language for Multicores

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

The ForeC Synchronous Deterministic Parallel Programming Language for Multicores

(Invited Paper)

Eugene Yip*, Alain Girault†, Partha S. Roop‡ and Morteza Biglari-Abhari‡
*Software Technologies Research Group, University of Bamberg, 96045 Germany. Email: eugene.yip@uni-bamberg.de
†Inria, France. Université Grenoble Alpes, Lab. LIG, Grenoble, France.
CNRS, Lab. LIG, F-38000 Grenoble, France. Email: alain.girault@inria.fr
‡Department of ECE, The University of Auckland, New Zealand. Email: {p.roop, m.abhari}@auckland.ac.nz

***Abstract*—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. This paper proposes the ForeC language for the deterministic parallel programming of CPS applications on multi-core execution platforms. ForeC's synchronous semantics 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 programs are amenable to static timing analysis. One of ForeC's main innovation is its shared variable semantics that provides thread isolation and deterministic thread communication. 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 precisely.**

1. Introduction

Safety-critical embedded systems must be dependable and functionally safe [1] and certified against safety standards, such as DO-178B [2]. Certification is a costly and time consuming exercise and is exacerbated by the use of multi-core processors. The correctness of a safety-critical embedded systems depends on the output of its computations and on the timeliness of completing the computations [3] in response to inputs. Thus, a key to building successful embedded systems using multi-cores is the understanding of the timing behaviors of the computations [4]. This is typically achieved with static worst-case execution time (WCET) analysis [5] and is a complex process because it depends on the underlying execution platform.

C [6] is a popular language for programming embedded systems with support for multi-threading and parallelism provided by third-party libraries, such as Pthreads [7] and OpenMP [8]. Unfortunately, these multi-threading solutions are inherently *non-deterministic* [9] and programmers fall into common parallel programming traps and pitfalls [10]. To help alleviate this issue, runtime environments that enforce deterministic thread scheduling and memory accesses can be used. CoreDet [11] maps all thread interactions onto a logical timeline. The program's execution alternates between parallel and serial phases. However, understanding the program's behavior at compile time remains difficult because the determinism is enforced by the runtime. Threads in CoreDet maintain their own snapshot of the shared memory state, which is resynchronized in every serial phase. This concept is used and formally defined in Concurrent revisions [12]. An alternative is to extend the C language with deterministic parallelism, such as SharC [13], but its time predictability is unknown.

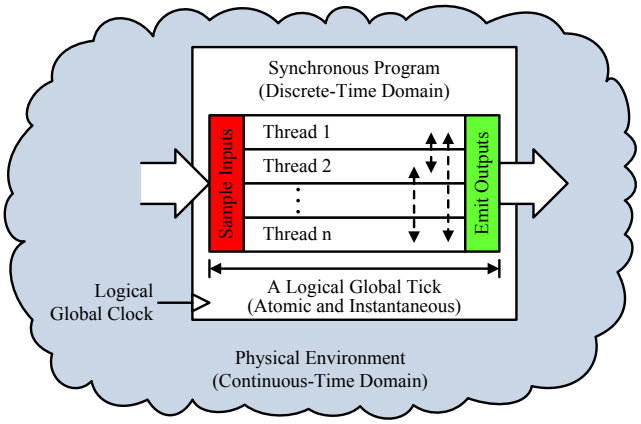

Figure 1. Synchronous model of computation.

Synchronous languages [14], such as Esterel [15], provide an alternative by offering deterministic concurrency that is based on sound mathematical semantics, which facilitates system verification by formal methods [16]. Figure 1 depicts a synchronous program, defined as a set of concurrent threads, within its physical environment. A synchronous program reacts to inputs by producing corresponding outputs. Each reaction is triggered by a hypothetical (logical) *global clock*. Central to synchronous languages is the *synchrony hypothesis* [14], which states that the execution of

each reaction is considered to be atomic and instantaneous. Concurrent threads communicate instantaneously with each other (dashed arrows in Figure 1) using signals. Once the embedded system is implemented, the synchrony hypothesis is validated by ensuring that the WCET of any global tick does not exceed the minimal inter-arrival time of the inputs. This is known as worst-case reaction time analysis and techniques have been developed for multi-cores [17], [18]. C-based synchronous languages, such as PRET-C [19] and SyncCharts in C [20], appeal to C programmers because the learning barrier for synchronous languages is reduced. However, their inherent sequential execution semantics render them unsuitable for multi-core execution.

Synchronous programs are considerably difficult to parallelize [21] 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. Hence, concurrency is typically *compiled away* to produce only sequential code [22] that is then parallelized [21], [23]. The Synchronized Distributed Executive (SynDEx) [24] approach considers communication costs when distributing code to each processing element. Yuan et al. [25] offer static and dynamic scheduling approaches for Esterel on multi-cores.

Contributions. The synchronous languages designed for the single-core era must be redesigned to address the multi-core challenges. Section 2 describes the multi-core architecture considered by this paper. We propose a C-based synchronous parallel programming language, called ForeC, for simplifying the deterministic parallel programming of embedded multi-cores. ForeC brings together the formal deterministic semantics of synchronous languages and the benefits of C's control and data structures (Section 3). A key innovation is ForeC's shared variable semantics that provides thread isolation and deterministic thread communication. Moreover, many forms of parallel programming patterns can be expressed in ForeC, such as the software pipeline design pattern in Section 4. ForeC can be compiled for direct execution on embedded multi-cores or for execution on desktop multi-cores using an operating system (Section 5). Through benchmarking in Section 6, we demonstrate that ForeC can achieve better parallel performance than Esterel and OpenMP, while being amenable to static timing analysis. Section 7 concludes the paper.

2. Multi-Core Architecture

The architecture of the predictable multi-core used in this paper is representative of existing predictable designs [26]. It is a homogeneous multi-core processor that we have designed using Xilinx MicroBlaze [27] cores. Each MicroBlaze core has a three-stage in-order pipeline that is free of timing anomalies and is 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. We developed a multi-core MicroBlaze simulator for benchmarking purposes by extending an existing MicroBlaze simulator [28] to support cycle-accurate simulation, an arbitrary number of cores, and a shared bus with TDMA arbitration. We observe that the focus of the paper is not on architectural innovations for time predictability, but rather language theoretic innovations (Section 3).

3. The ForeC Language

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 variable semantics that is agnostic to scheduling. This is essential for the reasoning and debugging of parallel programs.

3.1. Overview of ForeC

ForeC extends a safety-critical subset of C [29] with a minimal set of synchronous constructs. Although C [6] is popular for programming safety-critical embedded systems, it has unspecified and undefined behaviors. Safety-critical programmers follow strict coding guidelines [30] to help write deterministic programs that are understandable, maintainable, and easier to debug. We describe the statements, specifiers, and qualifiers allowed in the C subset:

C statements (*c_st*): Expressions can only be constants, variables, pointers, and array elements that are composed with the logical, bitwise, relational, and arithmetic operators of C. Although the use of pointers and arrays is allowed, pointer aliasing makes static analysis difficult [31]. Thus, we assume that pointers are only assigned once. All C control statements, except `goto`, can be used.

C type specifiers and qualifiers: All the C primitives and qualifiers can be used. Custom data types can be defined using `struct`, `union`, and `enum`.

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

Figure 2 gives the extended syntax of ForeC and Table 1 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`).

3.1.1. I/O, Threads, and Pausing. Like traditional C programs, the function `main` is the program's entry point and serves as the initial thread of execution. To recap, the threads of a synchronous program execute in lock-step to the ticking of a *global clock*. During each global tick, the threads sample the environment, perform their computations,

Statements:
st ::= *c_st* | `pause` | `par` (*st*, *st*)
| `weak?` `abort` *st* `when` `immediate?` (*exp*) | *st*; *st*

Type Qualifiers:
tq ::= *c_tq* | `input` | `output` | `shared`

Figure 2. Syntactic extensions to C.

TABLE 1. FOREC CONSTRUCTS AND THEIR SEMANTICS.

`input`: Type qualifier to declare an input, the value of which is updated by the environment at the start of every global tick.
`output`: Type qualifier to declare an output, the value of which is emitted to the environment at the end of every global tick.
`shared`: Type qualifier to declare a shared variable, which can be accessed by multiple threads.
`pause`: Pauses the executing thread until the next global tick.
`par`(*st*, *st*): Forks two statements *st* as parallel threads. The `par` terminates when both threads terminate (join back).
`weak?` `abort` *st* `when` `immediate?` (*exp*): Preempts its body *st* when the expression *exp* evaluates to a non-zero value. The optional `weak` and `immediate` keywords modify its temporal behavior.

and emit their results to the environment. When a thread completes its computation, we say that it completes its *local tick*. When all threads complete their local ticks, we say that the program completes its *global tick*. The program below declares two input and two output variables, and a `main` thread that forks the execution of two threads that each have three sequential statements:

```
input int X, Y;    output int A=0, B=0;
void main(void) { par(t1(), t2()); }
void t1(void) {
  int a = 1+X; pause; A = a*X;
}
void t2(void) {
  int b = 1+Y; pause; B = b*Y;
}
```

Inputs are read-only and their values are updated by the environment at the start of each global tick. Outputs emit their values to the environment at the end of each global tick. The program starts its first global tick from the `main` thread. The `main` thread executes the `par` statement that forks its arguments (the functions `t1` and `t2`) into two parallel *child threads*. The `par` is a blocking statement and terminates only when both its child threads terminate, i.e., join together. The child threads `t1` and `t2` initialize their local variables by incrementing the input values by `1`. Let the values of the inputs be `X=1` and `Y=2` during the first global tick. Hence, thread `t1` assigns `a=2` and thread `t2` assigns `b=3`. Both child threads execute a `pause` statement, which *pauses* their execution and acts as a synchronization barrier. We say that both threads have completed their *local ticks*. Next, the program completes its first global tick and the outputs `A=0` and `B=0` are emitted.

The program starts its second global tick by resuming the child threads `t1` and `t2` from their respective `pause` statements. That is, both threads begin their next local ticks. Let the values of the inputs be `X=3` and `Y=4` during the second global tick. Thread `t1` assigns `A=6` and thread `t2` assigns `B=12` to the output variables. Both child threads terminate, causing the `par` statement in the `main` thread to terminate. The `main` thread resumes its execution by reaching the end of its body and terminating. The second global tick ends and the outputs `A=6` and `B=12` are emitted.

3.1.2. 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. The semantics for shared variables permit them to be accessed deterministically in parallel, without needing the programmer to explicitly use mutual exclusion. We modify our program by making the child threads `t1` and `t2` share the variable `x`:

```
input  int X, Y;   output int A=0;
void main(void) {
  shared int x=1 combine all with plus;
  par(t1(&x), t2(&x));
  A = x;
}
void t1(shared int *x) {
  x = 1+X; pause; x = x*X;
}
void t2(shared int *x) {
  x = 1+Y; pause; x = x*Y;
}
int plus(int th1, int th2) {
  return (th1+th2);
}
```

The `main` thread now declares a shared variable called `x`. C's call by reference is used to pass `x` to the child threads. When the child threads start their local tick, they each create a *local copy* of `x`. When the child threads need to access `x`, they access their copy of `x` instead. Hence, their copies of `x` remain distinct from the shared variable `x` declared in the `main` thread. The changes made by one thread cannot be observed by any other, yielding mutual exclusion and thread isolation. Thread isolation minimizes the need to serialize parallel accesses to shared variables, thereby maximizing runtime parallelism. This is key to enhancing execution performance and is unlike conventional synchronous-reactive languages [21]. Moreover, only sequential reasoning is needed within the thread's local tick. Next, let the values of the inputs be `X=1` and `Y=2` during the first global tick. Threads `t1` and `t2` assign `2` and `3`, respectively, to their local copies of `x` before pausing. The first global tick ends and the local copies of `x` are *automatically combined* into a single value by a programmer-specified *combine function*. The combine function for the shared variable `x` is `plus`, specified in the `combine` clause of its declaration. Thus, the combined value of both copies is `plus(2,3)=5` and is assigned to the shared variable `x`. We call the combined value that is assigned to the shared variable the *resynchronized value* and call the process of updating the shared variable as *resynchronizing*. Resynchronizing shared variables at the end of each global tick ensures deterministic outputs at the

Expressions:
exp ::= *val* | *var* | *ptr*`[`*exp*`]` | `(`*exp*`)` | *u_op exp* | *exp b_op exp*

Unary Operators:
u_op ::= `*` | `&` | `!` | `-` | `~`

Binary Operators:
b_op ::= `||` | `&&` | `^` | `|` | `&` | `<<` | `>>` | `==` | `!=` | `<` | `>` | `<=` | `>=` | `+` | `-` | `*` | `/` | `%`

Figure 3. Syntax of preemption conditions.

end of each global tick. Finally, the first global tick ends and the output `A=0` is emitted.

When the program starts its second global tick, the child threads start their next local ticks by creating a local copy of `x`. Their copies are initialized with `x`'s resynchronized value of `5`. Let the values of the inputs be `X=3` and `Y=4` during the second global tick. Threads `t1` and `t2` assign `15` and `20`, respectively, to their local copies of `x` and terminate. When all the child threads of a `par` terminate, their local copies are automatically combined and assigned to their parent thread. In this case, the combined value of both copies is `plus(15,20)=35`. The `main` thread resumes and assigns the combined value of `x` to the output `A` and then terminates. The second global ends and the output `A=35` is emitted.

3.1.3. Combine Functions and Policies. The signature of any combine function is $C : Val \times Val \rightarrow Val$. The two input parameters are the two copies to be combined. A combine function is invoked multiple times when more than two copies need to be combined, e.g., $c(v_1, c(v_2, \cdots c(v_{n-1}, v_n))))$. Combine functions must be deterministic, associative, and commutative. That is, they produce the same outputs from the same inputs, regardless of previous invocations and how the copies are ordered or grouped. ForeC's combine functions are inspired by Esterel [15] but similar solutions can be found in other parallel programming frameworks, e.g., OpenMP's `reduction` operators [8]. Solutions developed for these frameworks could be reworked into ForeC combine functions and policies.

It may 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` and they are used in the `combine` clause during variable declaration, e.g., `combine new with`. The `new` policy ignores the copies that have the same value as their shared variable's resynchronized value. The `mod` policy ignores the copies that were not assigned a value during the global tick. 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 is resynchronized value.

3.1.4. Preemption. Inspired by Esterel [15], the `abort` *st* `when` (*exp*) statement provides preemption, which is the termination of the abort body *st* when the condition *exp* evaluates to *true*. Preemption can be used to model hierarchical state machines succinctly. The condition *exp* must be a side-effect free expression produced from the syntax shown in Figure 3. The program below is an example of a *non-immediate and strong* `abort`:

```
void main(void) {
  int x = 1;
  abort {
    x = 2; pause; x = 3; pause; x = 4;
  } when (x>2);
}
```

After initializing variable `x` to `1`, execution reaches the `abort`. `x=2` is executed and then the first global tick ends. At the start of the second global tick (and at each subsequent global tick), the preemption condition is evaluated before the `abort` body can execute. This allows shared variables in the condition to be evaluated with their resynchronized value. If the preemption condition evaluates to *true* (any non-zero value following the C convention), then the `abort` terminates without executing its body. For this `abort` example, the preemption condition is *true* at the start of the third global tick, as `x=3`. An `abort` will also terminate when execution reaches the end of its body.

Like Esterel, the optional `weak` and `immediate` keywords change the temporal behavior of the 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 following is an example of a *non-immediate and weak* `abort`:

```
void main(void) {
  int x = 1;
  weak abort {
    x = 2; pause; x = 3; pause; x = 4;
    pause;
  } when (x>2);
}
```

Here, although the preemption condition is *true* at the start of the third global tick, the termination of the `abort` is delayed until the third `pause` is reached.

The `immediate` keyword allows the `abort` to terminate immediately as soon as execution reaches it for the first time. The following is an example of an *immediate and strong* `abort`:

```
void main(void) {
  int x = 3;
  abort {
    x = 2; pause; x = 3; pause; x = 4;
  } when immediate (x>2);
}
```

Here, the initial value of `x` is `3`, meaning that the preemption condition is *true* when execution first reaches the `abort`. Hence, the `abort` body is not executed at all.

Lastly, both the `weak` and `immediate` keywords can be used together to define an *immediate and weak* `abort`:

```
void main(void) {
  int x = 3;
  weak abort {
    x = 2; pause; x = 3; pause; x = 4;
  } when immediate (x>2);
}
```

Here, the preemption condition is *true* when execution first reaches the `abort`. However, the termination of the `abort` is delayed until the first `pause` is reached.

3.2. Comparison with Esterel and Concurrent Revisions

This section compares ForeC with Esterel [15] and Concurrent revisions [12]. Concurrent revisions is a programming model that supports the forking and joining of asynchronous threads. When a thread is forked, it creates a snapshot of the shared variables. Changes performed by the thread are applied to its snapshot, thus, ensuring thread isolation. The snapshots are merged together using a deterministic *merge function* when the threads join. The merge function always considers all the copies, i.e., equivalent to ForeC's combine policy `all`. Thread communication is, therefore, always delayed until the child threads join. In contrast, ForeC threads may execute over several global ticks and thread communication is only delayed to the end of each global tick. Esterel threads communicate instantaneously by emitting and receiving signals during each global tick. Signal emissions may have associated values that must be combined using a programmer-specified combine function before the signal can be read. Esterel's combine function only considers the emitted values, i.e., equivalent to ForeC's combine policy `mod`.

In Concurrent revisions, the parent thread can execute alongside its children and, e.g., fork more threads in response to higher input workloads. This is not the case with ForeC and Esterel because the parent thread blocks until of all its children have joined. The commutativity and associativity of ForeC, Esterel, and Concurrent revisions' parallel construct depends on the commutativity and associativity of their combine and merge functions, respectively.

Preemptions in Esterel are triggered instantaneously by instantaneous signal communication. However, preemptions in ForeC are triggered after a delay of one global tick because preemption conditions are evaluated using values computed in the previous global tick. Concurrent revisions does not support preemptions. Esterel programs may be non-causal [14] because of instantaneous feedback cycles. Thanks to delayed communication, programs in ForeC and Concurrent revisions are always causal by construction.

4. Software Pipeline Design Pattern in ForeC

Design patterns [32] are reusable templates that programmers can use to solve recurring problems of any size and achieve high execution performance. For example, the software pipeline pattern [32] solves the problem of needing to process an audio or video data stream in stages. An instance of this pattern in ForeC is given in Figure 4. The data processing is broken down into pipeline stages that work in parallel on different chunks of the data stream. The pattern forks a thread for each pipeline stage. Each stage gets a chunk of data, processes the data, and passes the result to the next stage. This is repeated until the data stream ends. Data is passed from each stage using shared variables.

```
input int in;    output int out;
shared int s1=0, s2=0;
void main(void) {
  par(stage1(), par(stage2(),stage3()));
}
void stage1(void) {
  while(1) {  s1=process1(in); pause; }
}
void stage2(void) { pause;
  while(1) {  s2=process2(s1); pause; }
}
void stage3(void) { pause; pause;
  while(1) { out=process3(s2); pause; }
}
```

Figure 4. Software pipeline.

The explicit use of buffers is not needed because threads always work on local copies of the shared variables. The pipeline is synchronous because the stages pause before processing their next chunk of data. Hence, the throughput is determined by the slowest pipeline stage. To initialize the pipeline correctly, each stage must wait for their initial chunk of data. The waiting is achieved by placing, at the start of the threads, a number of `pause` statements equal to the number of preceding stages. The stages execute in parallel after all have waited for their initial chunk of data.

5. Compiling ForeC for Parallel Execution

The ForeC compiler generates statically scheduled code for direct execution on a predictable parallel architecture described in Section 2. The aim is to generate performant code that is amenable to static timing analysis. The compiler translates the ForeC statements into equivalent C code and generates thread scheduling routines for each core. The programmer statically allocates the threads to the cores and passes the allocations into the compiler. Based on the allocations, the compiler defines a total order for all the threads, using on a depth-first traversal of the program's control-flow graph. A *static* and *non-preemptive* (cooperative) schedule is created for each core, encoded as a *doubly linked list*, similar to the approach of the Columbia Esterel Compiler [22]. Each thread is represented as a node in the linked list of its allocated core. The node stores the thread's continuation point (a program counter) and links to the threads (nodes) that are scheduled before and after it. The initial continuation point of a thread is the start of its body. Each core begins its execution by jumping to the continuation point of the first thread in its linked list. The thread executes without interruption until it reaches a *context-switching point*: a `par` or `pause` statement, or the end of its body. At this point, the thread stores its next continuation point into its node and jumps to the continuation point in the next node. Thus, inserting or removing a thread or routine from the list controls whether it is included or excluded, respectively, from execution.

A problem with encoding the static schedule as a linked list is that the global tick in which threads fork and join can only be determined at runtime. This means that threads need to be inserted or removed from the linked lists whenever they are forked or whenever they terminate. This is managed by inserting *synchronization routines* into the linked lists that send and receive scheduling information between cores at runtime. For example, when a parent thread on one core reaches a `par`, the other cores receive this information via their synchronization routines and will insert the newly forked threads into their linked lists. The notion of a global tick is preserved by ending each linked list with a global tick synchronization routine that implements barrier synchronization. One core is nominated to perform the following housekeeping tasks: resynchronizing the shared variables, emitting the outputs, and sampling the inputs.

Shared variables are hoisted up to the program's global scope to allow all cores to access them. The copies of shared variables are implemented as unique global variables. In each thread, all shared variable accesses are replaced by accesses to their copies.

6. Benchmarks

This section begins by evaluating the static worst-case reaction time (WCRT) analysis of ForeC programs. We have previously shown [18] that the WCRT of ForeC programs could be estimated to a high degree of precision, which is very useful for implementing real-time embedded systems. Further, we provide a performance comparison between ForeC and Esterel. Although ForeC has been designed with embedded multi-cores in mind, we believe that the ForeC language can be applied to the deterministic parallel programming of desktop multi-cores. Performant parallel programs can be achieved by using parallel design patterns within a synchronous language, like ForeC. We evaluate this hypothesis with a performance comparison to OpenMP, a popular desktop solution for parallel programming that primarily exploits loop data parallelism.

6.1. Time Predictability

ForeC is amenable to static timing analysis because we implement all language features using static scheduling. Additionally, bounded loops are used to ensure bounded execution times, synchronous preemption is used instead of (asynchronous) hardware interrupts, and threads execute in isolation thanks to the shared variable semantics. Our C++ ForeCast tool [18] statically analyzes the WCRT of ForeC programs on embedded multi-cores. This section highlights our key findings [18] for the MicroBlaze multi-core simulator described in Section 2, with the configuration shown in Figure 5. The **802.11a** [33] benchmark is production code from Nokia, that tests various signal processing algorithms needed to decode 802.11a data transmissions. 802.11a has complex data and control dominated computations, comprised of 2147 lines of ForeC code that forks 26 threads, of which up to 10 can execute in parallel. 802.11a was distributed on up to 10 cores and ForeCast *computed* the WCRT of each distribution. To evaluate the precision of the computed WCRTs, 802.11a was executed by the multi-core simulator for one million reactions unless the program terminated. Test vectors were generated to elicit the worst-case execution path by studying the program's control-flow. The simulator reported the execution time of each global tick and the longest was taken as the *observed* WCRT.

Xilinx MicroBlaze, 3-stage pipeline, no branch prediction, no caches, no out-of-order execution, 8 KB private data and instruction scratchpads on each core (1 cycle access time), 32 KB global memory (5 cycle access time), TDMA shared bus (5 cycle time slots per core and, thus, a 5×(number of cores) cycle long bus schedule), programs compiled with MB-GCC-4.1.2 -O0 and decompiled with MB-OBJDUMP-4.1.2.

Figure 5. MicroBlaze multi-core configuration.

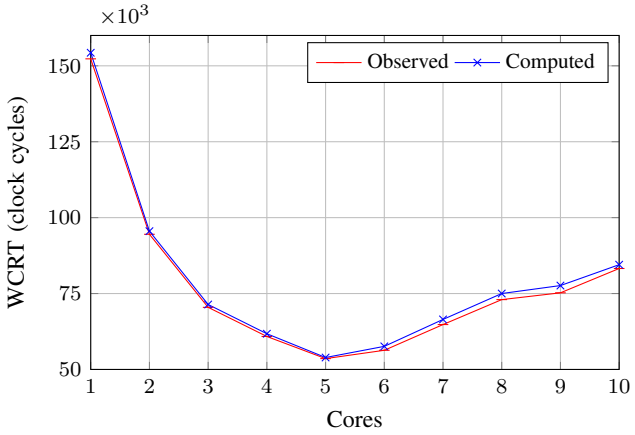

Figure 6. WCRT results for 802.11a in clock cycles.

The observed and computed WCRTs of 802.11a are plotted as a line graph in Figure 6. This graph shows that ForeCast is very precise, even as the number of cores increased. ForeCast computed WCRTs that were only 3.2% longer than the observed WCRTs. Similar levels of precision have been demonstrated with other benchmark programs [18]. The computed WCRTs of 802.11a in Figure 6 also reflect the benefit of multi-core execution. The computed WCRT decreased when the number of cores increased from one to five. The computed WCRT at five cores corresponded to the execution time of one thread that was already allocated to its own core. Thus, the WCRT could not be improved by distributing the remaining threads. The WCRT increased after five cores because of increased scheduling overheads and global memory access times. Ju et al. [17] offer the only other known static WCRT analysis approach for synchronous programs on multi-cores. Unfortunately, we cannot compare with that work because their results are only for a four core system with no precision results reported.

6.2. Comparison with Esterel

The dynamic scheduling approach by Yuan et al. [25] for Esterel programs has been shown to perform well on a

TABLE 2. AVERAGE WCRT SPEEDUP RESULTS FOR FORE C AND ESTEREL ON FOUR CORES NORMALIZED TO SINGLE-THREADED C.

Version	Life	Lzss	Mandelbrot	MatrixMultiply
Esterel	2.28	2.42	1.20	3.87
ForeC	3.25	3.30	3.68	3.76

TABLE 3. AVERAGE SPEEDUP RESULTS FOR FORE C AND OPENMP ON FOUR CORES NORMALIZED TO SINGLE-THREADED C.

Version	FmRadio	Life	Lzss
OpenMP	2.02	2.82	3.46
ForeC	2.63	2.98	3.90

MicroBlaze multi-core platform similar to ours. Yuan's approach requires a special hardware FIFO queue to help allocate threads to cores for resolving signal statuses. For benchmarking, the MicroBlaze multi-core simulator described in Section 2 was extended with a hardware queue to support the dynamic scheduling. Because a WCRT analysis technique for Yuan et al.'s approach does not exist, we compare instead the average WCRT speedup achieved by ForeC and Esterel for the following programs: **Life** simulates Conway's Game of Life for a fixed number of iterations and a given grid of cells. **Lzss** uses the Lempel-Ziv-Storer-Szymanski algorithm to compress a fixed amount of text. **Mandelbrot** computes the Mandelbrot set for a square region of the complex number plane. **MatrixMultiply** computes the matrix multiplication of two equally sized square matrices. Single-threaded C, ForeC, and Esterel versions of each benchmark program were created and handcrafted for best performance. The same input vector was given to each version to ensure that the same computations were performed. The MicroBlaze simulator returns the execution time when a program terminates.

Table 2 shows the speedups achieved by ForeC and Esterel when the benchmark programs were executed on four cores. Speedup is calculated as:

$$Speedup(P) = \frac{\text{Execution time of single-threaded C}}{\text{Execution time of } P} \quad (1)$$

where P is the ForeC or Esterel version of the benchmark program being tested. Except for MatrixMultiply, ForeC shows superior performance than Esterel, even though Esterel uses dynamic scheduling with hardware acceleration. For Esterel, all possible signal emitters must execute before any signal consumers and this invariant is achieved using a signal locking protocol [25] that is costly. In comparison, shared variables in ForeC only need to be resolved at the end of each global tick. The significance of the overhead is evident in the Mandelbrot results, where the Esterel version has 24 unique signals and only achieved a speedup of 1.2. Because of minimal data dependencies in MatrixMultiply, the scheduling overheads of the ForeC and Esterel versions were minimal, resulting in similar speedups.

6.3. Comparison with OpenMP

We extended the ForeC compiler to target desktop multi-cores running an operating system. For each core, the compiler creates a Pthread [7] to run its static thread schedule. Hence, a fixed pool of Pthreads executes the ForeC threads, so the cost of creating each Pthread is only incurred once. Although the Pthreads are dynamically scheduled by the operating system, the original ForeC threads follow their static schedule. Because execution is inherently speculative on desktop multi-cores, we present benchmarking results for the average execution time speedup for the following programs: **FmRadio** [33] transforms a fixed stream of radio signals into audio. The history of the radio signals is used to guide the transformation of the remaining signals. **Life** and **Lzss** are desktop versions of those used in Section 6.2.

Single-threaded C, ForeC, and OpenMP versions of each benchmark program were created and handcrafted for best performance. For the ForeC versions, FmRadio used the software pipeline and fork-join patterns and Lzss used the fork-join pattern. The Intel VTune Amplifier XE [34] software helped identify areas of code that could be parallelized for the OpenMP versions of the benchmarks. Benchmarking was carried out on a 3.4 Ghz four-core Intel Core-i5 3570 desktop running Linux 3.6 with 8 GB of RAM. Hyper-Threading, Turbo Boost, and SpeedStep were disabled. The benchmarks were compiled with GCC-4.8 -O2.

Table 3 shows the average speedups achieved by ForeC and OpenMP. Speedups were calculated using equation (1). Although ForeC and OpenMP achieved a speedup of between two and four, ForeC demonstrates greater speedup than OpenMP in these preliminary results. Dynamic and static thread scheduling pragmas were used in the OpenMP versions and dynamic scheduling does introduce slight overheads, especially thread locking, but these overheads should be amortized over the overall run of the benchmarks. This scheduling approach of OpenMP contrasts with the ForeC approach, where all work scheduling is static and determined automatically by the ForeC compiler.

7. Conclusions

This paper 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. The local copying of shared variables ensures thread isolation and determinism, while minimizing the need to serialize parallel accesses to shared variables. The behavior of shared variables can be tailored to the application at hand by specifying suitable *combine functions* and *policies*. A critical comparison showed that ForeC combines the benefits offered by synchronous languages with those offered by deterministic runtime solutions. To the best of our knowledge, no other synchronous language achieves parallel execution and time predictability similar to ForeC. For future work, the ForeC compiler could be improved to generate more efficient code that remains amenable to static timing analysis. In particular, different static scheduling strategies could be explored for different

parallel programming patterns. The allocation of ForeC threads could be refined automatically by feeding ForeCast's WCRT analysis results back into the ForeC compiler.

Acknowledgments

This work was supported in part by the RIPPES INRIA International Lab, and the PRETSY2 project under DFG Funding No. ME 1427/6-2. The authors would like to thank Simon Yuan for setting up the Esterel benchmarks and Avinash Malik for setting up the OpenMP benchmarks.

References

[1] M. Paolieri and R. Mariani, "Towards Functional-Safe Timing-Dependable Real-Time Architectures," in *17th IEEE International On-Line Testing Symposium (IOLTS)*, 2011, pp. 31 – 36.

[2] Radio Technical Commission for Aeronautics, "Software Considerations in Airborne Systems and Equipment Certification," Apr. 1992, standard DO-178B.

[3] R. Wilhelm and D. Grund, "Computation Takes Time, but How Much?" *Commun. ACM*, vol. 57, no. 2, pp. 94 – 103, Feb. 2014.

[4] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. V. Hanxleden, R. Wilhelm, and W. Yi, "Building Timing Predictable Embedded Systems," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 4, pp. 82:1–82:37, Mar. 2014.

[5] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 1 – 53, 2008.

[6] ISO/IEC JTC1/SC22/WG14, "ISO/IEC 9899:2011," 2011.

[7] The IEEE and The Open Group, "POSIX.1-2008," 2008, standard Issue 7.

[8] OpenMP Architecture Review Board, "OpenMP Application Program Interface," Jul. 2013, standard 4.0.

[9] E. A. Lee, "The Problem with Threads," *Computer*, vol. 39, pp. 33 – 42, 2006.

[10] S. Lu, S. Park, E. Seo, and Y. Zhou, "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*, ser. ASPLOS XIII. ACM, 2008, pp. 329 – 339.

[11] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman, "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*, ser. ASPLOS. ACM, 2010, pp. 53 – 64.

[12] S. Burckhardt and D. Leijen, "Semantics of Concurrent Revisions," in *Proceedings of the 20th European Conference on Programming Languages and Systems*, ser. ESOP/ETAPS, 2011, pp. 116 – 135.

[13] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Efficient Data Race Detection for Async-Finish Parallelism," in *Proceedings of the 1st International Conference on Runtime Verification*, ser. RV. Springer-Verlag, 2010, pp. 368 – 383.

[14] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, "The Synchronous Languages 12 Years Later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64 – 83, Jan. 2003.

[15] G. Berry and G. Gonthier, "The Esterel Synchronous Programming Language: Design, Semantics and Implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87 – 152, 1992.

[16] J. Souyris, E. L. Pavec, G. Himbert, V. Jégu, G. Borios, and R. Heckmann, "Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation," in *International Workshop on Worst-case Execution Time*, Mallorca, Spain, Jul. 2005, pp. 21 – 24.

[17] L. Ju, B. K. Huynh, A. Roychoudhury, and S. Chakraborty, "Timing Analysis of Esterel Programs on General-Purpose Multiprocessors," in *Proceedings of the 47th Design Automation Conference (DAC)*. ACM, 2010, pp. 48 – 51.

[18] E. Yip, P. S. Roop, M. Biglari-Abhari, and A. Girault, "Programming and Timing Analysis of Parallel Programs on Multicores," in *13th International Conference on Application of Concurrency to System Design (ACSD)*, Jul. 2013.

[19] S. Andalam, P. S. Roop, A. Girault, and C. Traulsen, "Predictable Framework for Safety-Critical Embedded Systems," *IEEE Transactions on Computers*, vol. 63, no. 7, pp. 1600 – 1612, Jul. 2014.

[20] R. von Hanxleden, "SyncCharts in C - A Proposal for Light-Weight, Deterministic Concurrency," in *Proceedings of the 9th ACM/IEEE International conference on Embedded software*, Oct. 2009.

[21] A. Girault, "A Survey of Automatic Distribution Method for Synchronous Programs," in *International Workshop on Synchronous Languages, Applications and Programs, SLAP'05*, ser. ENTCS, F. Maraninchi, M. Pouzet, and V. Roy, Eds. Elsevier Science, Apr. 2005.

[22] S. A. Edwards and J. Zeng, "Code Generation in the Columbia Esterel Compiler," *EURASIP Journal on Embedded Systems*, vol. 2007, 2007.

[23] D. Baudisch, J. Brandt, and K. Schneider, "Multithreaded Code from Synchronous Programs: Extracting Independent Threads for OpenMP," in *Design, Automation and Test in Europe (DATE)*. EDA Consortium, 2010, pp. 949 – 952.

[24] D. Potop-Butucaru, A. Azim, and S. Fischmeister, "Semantics-Preserving Implementation of Synchronous Specifications Over Dynamic TDMA Distributed Architectures," in *International Conference on Embedded Software (EMSOFT)*. ACM, Nov. 2010, pp. 199 – 208.

[25] S. Yuan, "Architectures Specific Compilation for Efficient Execution of Esterel," Ph.D. dissertation, Electrical and Electronic Engineering, The University of Auckland, Jul. 2013.

[26] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. M. Burguiere, J. Reineke, B. Triquet, and R. Wilhelm, "Predictability Considerations in the Design of Multi-Core Embedded Systems," *Embedded Real Time Software and Systems (ERTS)*, 2010.

[27] Xilinx, "MicroBlaze Processor Reference Guide," 2012, [Online] http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/mb_ref_guide.pdf.

[28] J. Whitham, "Scratchpad Memory Management Unit," 2012, [Online] http://www.jwhitham.org/c/smmu.html.

[29] S. Blazy and X. Leroy, "Mechanized Semantics for the Clight Subset of the C Language," *Journal of Automated Reasoning*, vol. 43, no. 3, pp. 263 – 288, 2009.

[30] Motor Industry Software Reliability Association, "MISRA-C: 2012: Guidelines for the Use of the C Language in Critical Systems," p. 226, 2013, standard.

[31] M. Buss, D. Brand, V. Sreedhar, and S. A. Edwards, "A Novel Analysis Space for Pointer Analysis and Its Application for Bug Finding," *Sci. Comput. Program.*, vol. 75, no. 11, Nov. 2010.

[32] M. McCool, A. D. Robison, and J. Reinders, *Structured Parallel Programming*. Morgan Kaufmann, Jun. 2012.

[33] A. Pop and A. Cohen, "A Stream-Computing Extension to OpenMP," in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, ser. HiPEAC. ACM, 2011, pp. 5 – 14.

[34] Intel, "Intel® VTune™ Amplifier," 2014, [Online] https://software.intel.com/en-us/intel-vtune-amplifier-xe.