HAL
open science

Programming and Timing Analysis of Parallel Programs on Multicores

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

To cite this version:

Eugene Yip, Partha Roop, Morteza Biglari-Abhari, Alain Girault. Programming and Timing Analysis of Parallel Programs on Multicores. International Conference on Application of Concurrency to System Design, ACSD'13, Jul 2013, Barcelona, Spain. pp.167–176. hal-00842402

HAL Id: hal-00842402

https://inria.hal.science/hal-00842402v1

Submitted on 9 Jul 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Programming and Timing Analysis of Parallel Programs on Multicores

Eugene Yip, Partha S Roop, Morteza Biglari-Abhari
Department of Electrical and Computer Engineering
University of Auckland
Auckland, New Zealand
eyip002@aucklanduni.ac.nz, {p.roop, m.abhari}@auckland.ac.nz

Alain Girault
INRIA
Grenoble, France
alain.girault@inria.fr

***Abstract*—Multicore processors provide better power-performance trade-offs compared to single-core processors. Consequently, they are rapidly penetrating market segments which are both safety critical and hard real-time in nature. However, designing time-predictable embedded applications over multicores remains a considerable challenge. This paper proposes the ForeC language for the deterministic parallel programming of embedded applications on multicores. ForeC extends C with a minimal set of constructs adopted from synchronous languages. To guarantee the worst-case performance of ForeC programs, we offer a very precise reachability-based timing analyzer. To the best of our knowledge, this is the first attempt at the efficient and deterministic parallel programming of multicores using a synchronous C-variant. Experimentation with large multicore programs revealed an average over-estimation of only 2% for the computed worst-case execution times (WCETs). By reducing our representation of the programs state-space, we reduced the analysis time for the largest program (with $43,695$ reachable states) by a factor of 342, to only 7 seconds.**

***Keywords*-parallel programming; synchronous languages; WCET analysis**

I. Introduction

Embedded systems have recently exploded in complexity and functionality, motivating the use of high-performance but low-power multicore processors. Consequently, multicores are penetrating many embedded market segments including those with considerable safety concerns, such as automotive engine control units (ECUs) [1]. These systems have hard real-time constraints and a key requirement is the need to always behave in a functionally-correct and time-predictable manner [2].

C is the programming language of choice for embedded systems. Multithreading libraries using the shared memory model, like OpenMP [3] and Pthreads [4], are popular for parallel programming. As highlighted in [5], such multithreading is inherently non-deterministic and requires the programmer to manage shared memory. This makes the understanding and debugging of parallel programs very difficult and time consuming [6]. Such drawbacks are undesirable when programming safety-critical applications.

Synchronous languages [7] offer an alternate approach for deterministic concurrency. All concurrent threads execute in lock-step to the ticks of a global clock (hence the term *global tick*). The *synchrony hypothesis* makes the simplifying assumption that the program reacts *instantaneously* to the changing environment. This abstraction separates the time of the executing machine from the physical environment and enables formal analysis [7]. Hence, synchronous languages are widely used to program safety-critical applications. However, synchronous programs are notoriously difficult to parallelize [8]–[10] due to the need to resolve: (1) signal statuses, and (2) causality issues. Thus, concurrency is typically *compiled away* to produce sequential code. The common approach for parallelizing synchronous programs is to automatically parallelize an intermediate representation of the program [8]–[11]. The techniques differ in the heuristics used to partition the program to achieve sufficient parallelism. SynDEx [12] also considers the cost of communication when partitioning and allocating code to specific processing elements. C-based lightweight multithreading libraries with synchronous semantics, such as PRET-C [13] and SC [14], are recent developments. They offer deterministic concurrency but their semantics prescribes a sequential order for executing concurrent threads, which is unsuitable for multicore execution. Unlike these, ForeC, as proposed here, supports a truly parallel synchronous semantics tailored for multicore execution. Using the shared memory model [3], threads communicate via shared variables, which eliminates the need to resolve signal statuses. Thread-safe access to shared variables is achieved by providing threads with local copies of shared variables in each global tick. Thus, threads execute in isolation as changes made by other threads cannot be observed, which also simplifies the execution behavior. After each global tick, the local copies are written back to the shared memory, by first combining the copies of each shared variable with a *shared memory combinator* (see Section III). This behavior ensures that ForeC programs are causal and deterministic [7] by construction.

To validate the synchrony hypothesis of a synchronous program, the longest time needed to complete a global tick has to be determined. This is known as the worst-case reaction time (WCRT) analysis [15]. The techniques used by existing approaches can be classified broadly as Max-Plus algebra [15], model checking [13], reachability [16], or Integer Linear Programming (ILP) [9], [17]. The time

Table I
APPROACHES TO WCRT ANALYSIS.

Approach	Technique	Time Complexity
Boldt et al. [15]	Max-Plus	Sum of thread states
Roop et al. [13]	Model checking	Product of thread states + binary search
Kuo et al. [16]	Reachability	Product of thread states
Ju et al. [9], [17]	ILP	NP-hard
Proposed	Reachability	Product of thread states

complexity for each approach is summarized in Table I. In [15], the maximum WCRT of each thread is summed together for the program's WCRT. The technique is fast but assumes that the worst-case paths of all threads will occur in the same global tick, which usually leads to large over-estimations. In [9], [17], an objective function, describing the program's execution costs, is maximized to determine the WCRT. Solving ILP is known to be NP-complete. In [13], a model checking based formulation was developed to take the state-dependencies between threads into account for tighter analysis. Subsequently, a similar formulation was developed using ILP [17]. The model checking approach [13] requires a binary search to find the program's WCRT. In [16], the binary search is avoided by computing the execution time of all *reachable* global ticks and reporting the largest computed time as the WCRT. This ensures reachability has a lower complexity than model checking and ILP.

Apart from [9], all approaches are developed for single core execution and, therefore, do not analyze the inter-core interactions that exist for multicores. To the best of our knowledge, only [9] is developed for multicore execution. In [9], the program is sequentialized on each core and ILP is used for timing analysis (see Table I). Instead, we propose the timing analysis of parallel programs using reachability. Our approach is significantly different from [16] because we need to compute the execution time of multiple cores and take complex inter-core dependencies and bus schedules into account.

The main contributions of this paper are:

1) ForeC is the first known C-based synchronous language for the deterministic parallel programming of multicore applications. To the best of our knowledge, ForeC excels compared to traditional synchronous programs, such as Esterel [7], and associated parallel execution because: (1) causality analysis is not needed, (2) a new shared memory combinator provides thread-safe access to shared memory, and (3) signal resolution is not needed.
2) The proposed reachability-based timing analysis is more efficient than the only other known approach [9] for synchronous programs on multicores. Benchmarking reveals that the proposed approach computes very tight WCRTs in a fast and scalable manner, even when programs are distributed over additional cores. This demonstrates the efficacy of the proposed approach for the design of safety-critical embedded systems using multicore processors.

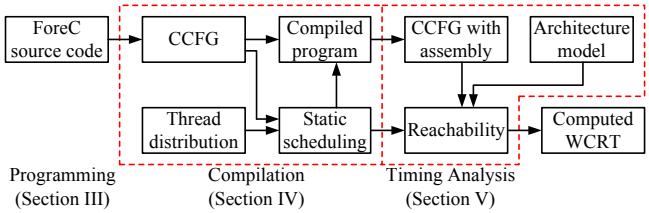


Figure 1. Overview of the proposed framework.

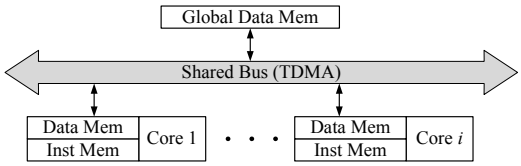


Figure 2. The multicore architecture.

A. Overview

Fig. 1 provides an overview of the proposed parallel programming and timing analysis framework, and the layout of the paper. The architecture of the multicore processor that the framework targets is outlined in Section II. The programmer begins by writing the parallel program in the ForeC language, described in Section III. Using an intermediate representation of the program, called Concurrent Control-Flow Graph (CCFG), Section IV describes how the program threads are distributed and scheduled over the available cores. The proposed timing analysis using reachability is presented in Section V and is evaluated in Section VII. The paper is concluded in Section VIII.

II. ARCHITECTURE OF THE MULTICORE PROCESSOR

Fig. 2 illustrates the multicore architecture for executing ForeC programs, similar to existing predictable multicores [1], [18]. In this paper, we do not focus on the processor micro-architecture or caches during the timing analysis. Thus, we assume a homogeneous set of cores with in-order pipelines free from timing anomalies [1]. Only local (data and instruction) and global (data) memories are used. The data coherence between the local copies of shared variables is managed by the ForeC runtime support. A shared bus, using Time Division Multiple Access (TDMA) arbitration, connects the cores to the global data memory. For experimentation, an existing single-core Xilinx MicroBlaze simulator [19] was used. We extended the simulator to be cycle-accurate, to support multiple cores, and a shared TDMA bus, and to use the stated architectural assumptions.

III. THE FOREC LANGUAGE

ForeC is a C-based, multi-threaded, synchronous language that enables the deterministic parallel programming of multicores with minimal extensions to C. A ForeC program

Table II
SUMMARY OF FOREC EXTENSIONS TO C.

Statement and Semantics
input *i* Declares an input variable *i*, the value of which is updated by the environment at the start of every global tick.
output *o* Declares an output variable *o*, the value of which is emitted to the environment at the end of every global tick.
shared *s* combine with *c* Declares a shared variable *s* that is accessible by multiple threads. In each global tick, threads are provided with a local copy of the variable. At the end of each global tick, the modified copies are *combined* into a single value with function *c* and then assigned back to the shared variable.
pause Pauses the execution until the next global tick.
par $(f_0, ..., f_n)$ Forks the functions f_0 to f_n to execute as parallel threads and then waits until all the threads have terminated (joined back).
[weak] abort $\{b\}$ **when [immediate]** (c) Executes the body *b* when execution reaches the abort statement. In subsequent global ticks, the condition *c* is checked before the body is executed. If *c* is true, then the body is preempted. The *weak* variant allows the body to execute one last time when preemption is triggered. The *immediate* variant checks *c* when execution reaches the abort statement.

executes in discrete steps governed by a *global clock*. In each *global tick*, the `input` variables are sampled, the threads are executed, and the `output` variables are emitted. The semantics of the extensions are described in Table II. In the design of safety-critical systems, the use of C is typically restricted [20], [21] so as to ensure deterministic execution. These restrictions concern the use of pointers, dynamic memory allocation, recursion, unbounded loops, and expressions with side-effects. Polyspace [22] and Parasoft [23] are two examples of tools that can check the absence of such constructs in C programs.

In ForeC, fork-join parallelism is captured by the `par` statement and parallel threads can execute in any order. A thread executes until it terminates or completes its *local tick* by reaching a `pause` statement. The global tick is reached when all threads have reached their local tick. A `pause` acts as a state boundary by pausing the thread's execution until the next global tick. We illustrate the ForeC execution semantics with the following example.

A. Motivating Example

Consider a robot that inspects hazardous tunnels for cracks using a pair of cameras. The robot has a speed of one meter/second and reports the total number of cracks found. To ensure all sections of the tunnel are inspected, the pictures must overlap and need to be taken and processed every 0.25 seconds. To complete these tasks within one global tick, the robot's WCRT cannot be longer than 0.25 seconds. We refer to this example as the *robot example*.

Fig. 3 is the ForeC program for the robot example. Line 1 defines the functions used in the program. Lines 2 and 3 declares inputs from the pair of cameras, a data link and

```
1  #include <functions.h>
2  input int camLeft[SIZE],camRight[SIZE];
3  input int linkInput,stop;
4  output int linkOutput,motorL,motorR;
5  shared int total=0 combine with plus;
6  void main(void) {
7    abort {
8      while(1) {
9        par(camL(),camR(),move(),link());
10     }
11   } when immediate(stop==1);
12 }
13 void camL(void) {
14   total=cracks(camLeft); pause;
15 }
16 void camR(void) {
17   total=cracks(camRight); pause;
18 }
19 void move(void) {
20   motorL=moveMotorL(); motorR=moveMotorR();
21   pause; pathPlanning();
22 }
23 void link(void) {
24   par(linkIn(),linkOut()); pause;
25 }
26 void linkIn(void) {
27   receiveLinkInput(linkInput);
28 }
29 void linkOut(void) {
30   linkOutput=total;
31 }
32 int plus(int copy1,int copy2) {
33   return (copy1+copy2);
34 }
```

Figure 3. The ForeC source code for the inspection robot example.

stop button. Line 4 declares outputs for the number of cracks found in each global tick, and to drive the robot's motors. Line 5 declares a shared variable to store the number of cracks found (`total`). The `main` function defines the program's entry point (line 6). The functions `camL` (line 13) and `camR` (line 16) analyze the pictures from each camera for cracks. The function `move` (line 19) decides the robot's movement and path planning. The functions `link` (line 23), `linkIn` (line 26), and `linkOut` (line 31) handle communication coming in and out of the robot.

At the program's first global tick, the inputs are sampled before `main` is executed. Execution reaches an `abort` (line 7), which supports the preemption of its body (lines 8 - 10). Before the body is executed, the preemption condition (line 11) is checked. If the condition is *false*, the body is executed. Otherwise, the body is preempted and execution jumps to line 12.

Line 9 forks the child threads `camL`, `camR`, `move` and `link` to execute in parallel. The parent thread (`main`) is suspended while it waits for its child threads to terminate. As an example of nested parallelism, `link` forks (line 24) two more child threads, `linkIn` and `linkOut`. The global tick ends when all threads complete their respective local ticks. Then, the outputs are emitted.

The threads `camL`, `camR` and `linkOut` communicate

via the `shared` variable `total`. Only shared variables can be used for thread communication. That is, all non-shared variables can only be accessed by one thread. Thread-safe communication is ensured by providing threads with local copies of shared variables at the start of their local tick. During their local tick, threads only access their local copies. Thus, all parallel accesses to shared variables are mutually exclusive. This isolates the thread execution as changes made in one thread cannot be observed by others. On lines 14 and 17, new local values are assigned to `total`. When the global tick ends, the modified local copies are combined automatically by a *shared memory combinator*. The combinator is a programmer defined commutative and associative function that specifies the computation needed to combine two local copies. To combine n-copies of a shared variable `v` with a combine function c, the computation is $c(v_1, c(v_2, \ldots c(v_{n-1}, v_n)))$. Line 32 is the combinator for the shared variable `total`, with input parameters for two local copies. The combined value is written back to `total`.

IV. Parallel Execution

In this section, we describe the scheduling of ForeC threads for parallel execution. The threads are scheduled statically and non-preemptively over the available cores, which results in one possible thread schedule for each global tick. This simplifies timing analysis by avoiding the need to analyze multiple thread schedules for each global tick. Currently, the programmer provides the thread distribution over the cores and the compiler defines an arbitrary thread scheduling order for each core. The order is based on the textual order of the threads in the `par` statements. For the robot example, the programmer may define the following thread distribution over two cores:

- Core 1: {main, camL, link, linkIn}
- Core 2: {camR, move, linkOut}

The following total order is used to decide the scheduling order for threads on the same core: `main`, `camL`, `camR`, `move`, `link`, `linkIn`, `linkOut`.

The distribution of program code is performed on an intermediate representation, called Concurrent Control-Flow Graph (CCFG), similar in spirit to the CCFG of [13]. The robot example's CCFG is shown in Fig. 4. The CCFG is constructed by creating a node for each statement and connecting them according to the program's control-flow. For reference, the nodes are labeled *n* followed by an integer. A `par` statement is a pair of *fork/join* nodes (e.g., n3 and n16), with the child threads appearing between them. An `abort` statement is a pair of *abort* nodes (e.g., n1 and n17) with a directed edge representing the scope. On preemption, control jumps to the end of the `abort` scope (n17). Using the programmer-defined thread distribution, the compiler partitions the CCFG over the cores. To preserve the execution semantics of `par`, the forking/joining of child threads is synchronized among the participating cores. The

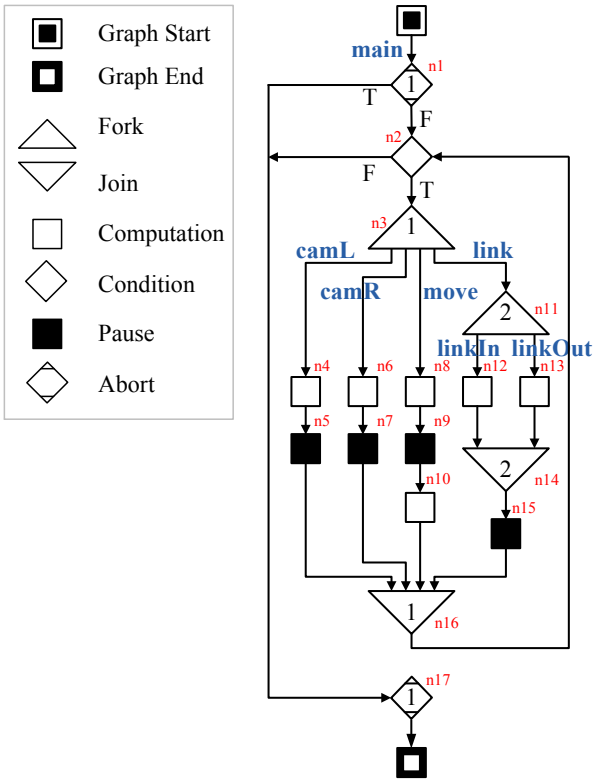


Figure 4. The CCFG for the inspection robot example.

synchronization occurs over global variables, representing thread execution states, by using (blocking) receive and (non-blocking) send routines. When a thread completes its local tick, by forking, terminating, or pausing, the thread uses the send routine to send its new execution state to global memory. A core uses the receive routine when it needs to receive new thread execution states from the global memory. Hence, the receive routine lets the core to wait for a parent thread to fork, or for all child threads to terminate.

If an `abort` body contains threads that are distributed over different cores, then the preemption behavior needs to be preserved across those cores. Currently, we replicate the abort checking on each participating core. As the abort condition is evaluated in parallel, this technique requires the abort conditions to be side-effect free. The abort checking is performed before the participating threads are executed. When a strong (resp. weak) preemption occurs, the threads are terminated before (after) they are executed. Their execution states are then updated with a send routine.

To preserve the notion of a global tick, the cores synchronize after executing their allocated threads. The compiler chooses one core to perform the following housekeeping tasks when the global tick ends: emitting outputs, combining the shared variables, and sampling inputs. The compiler generates an executable program for each core, containing its partition of the CCFG and a light-weight static scheduler. All synchronizations are implemented using global variables.

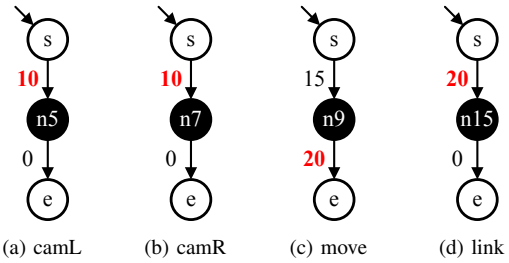


Figure 5. FSMs for threads camL, camR, move, link from the robot example.

The next section describes our timing analysis approach for validating the synchrony hypothesis.

V. Static WCRT Analysis

WCRT analysis is needed to validate that the implementation of a synchronous program adheres to the synchrony hypothesis. Such static analysis needs the model of the underlying hardware architecture, such as the (1) processor pipeline and micro-architecture, (2) underlying memory hierarchy, and (3) buses or inter-connection networks used in connecting the cores in a multicore system. In this work, we have modeled the Microblaze-based multicore architecture that was described in Section II. The analysis begins by annotating the nodes of the CCFG (Fig. 4) with the assembly instructions in each core's executable. Using the model of the multicore architecture, the execution time of each instruction is computed. The CCFG is analyzed to find the execution time needed to complete each global tick. The longest such time is the program's WCRT. The WCRT analysis for multicores must consider: (1) the overlapping of thread execution times from parallelism, (2) the inter-core synchronizations, (3) the scheduling overheads, and (4) the variable delays in accessing the shared bus. These four factors must be considered simultaneously when computing the WCRT to capture the parallel nature of execution. This makes the timing analysis of multicores very challenging compared to single cores. To the best of our knowledge, our proposed timing analysis approach is the first to consider all mentioned factors for synchronous programs. The approach of [9] considers inter-core synchronization arising from signal resolution, and the overlapping of execution times. However, the variability in bus delay is not discussed, and their approach does not require any scheduler to ensure the correct execution of Esterel programs.

In Max-Plus [15], the WCRT of each thread is computed and summed together for the program's overall WCRT. The WCRT of a thread is the longest time needed to complete a local tick. Thus, Max-Plus makes the assumption that all threads will execute their longest local tick in the same global tick. This results in very efficient analysis but trades off precision as the above assumption is not always true. For complex programs, the imprecision can lead to large over-

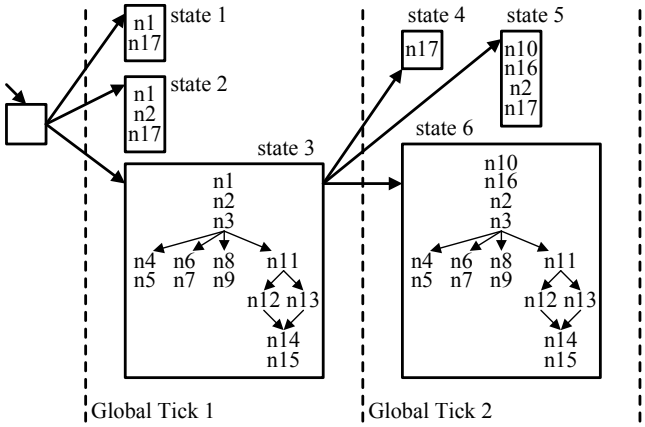


Figure 6. Illustration of reachability for the robot example.

estimations. We illustrate this observation with Fig. 5 and use it to motivate our reachability approach.

Each sub-figure in Fig. 5 is a finite state machine (FSM) representing one of `main`'s child threads (`camL`, `camR`, `move`, and `link`) from the robot example. Each state in an FSM corresponds to the start (s) or end (e) of the thread body, or a `pause` in the thread (black circle with its corresponding label from Fig. 4). Each edge corresponds to a local tick and is annotated with its execution time on the target architecture in clock cycles. The longest local tick of each thread is in bold font. If the threads were statically scheduled to execute sequentially on the same processor, then Max-Plus would compute a maximum execution time of $10+10+20+20 = 60$ clock cycles, assuming no scheduling overheads or bus delays (addressed later in Section V-C). However, threads in a synchronous program execute in lock-step, relative to the global clock. Following this execution semantics, the possible *alignments* [13] of the local ticks in Fig. 5 are (`camL` = 10, `camR` = 10, `move` = 15, `link` = 20) and (`camL` = 0, `camR` = 0, `move` = 20, `link` = 0). The WCRT of all threads never align together and the maximum execution time should be $10 + 10 + 15 + 20 = 55$ clock cycles instead of 60. Thus, the tight WCRT computation of a program must consider the alignment of local ticks. By sacrificing some efficiency in the overall WCRT analysis, much higher precision can be gained. In the following, we describe the use of reachability to explore the local ticks alignments.

A. Reachability

The intuition for using reachability is based on the observation that threads in a synchronous program execute in lock-step, relative to the global clock. A combination (alignment) of local ticks is executed in each global tick, resulting in a new *program state*. To find all the global ticks with feasible local tick alignments, the program's CCFG is traversed according to the ForeC semantics.

Fig. 6 is a graph showing all the reachable global ticks

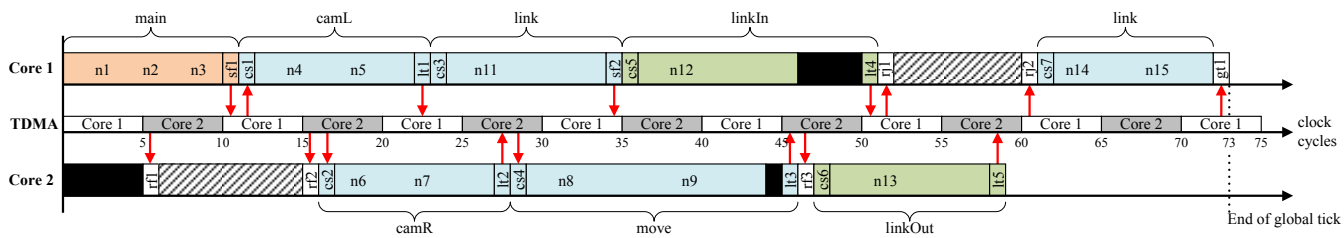


Figure 7. Computing the execution time of state 3 from Fig. 6 for the robot example.

for the robot example. Each state in the figure is a unique alignment of local ticks, annotated with the traversed CCFG nodes. An edge connects a state to its next reachable state(s). Reachability begins from the CCFG's *start* node (n1) using ForeC execution semantics. Whenever an *abort* (n1) or *condition* node (n2) is reached, the state is duplicated to explore each outgoing edge. Hence, a state only explores a single execution path through each thread. States 1 and 2 are the result of branching from the *abort* and *condition* nodes. Whenever a *fork* node (n3) is reached, each child thread is traversed. For state 3, the traversal of each child thread ends at their *pauses* (n5, n7, n9, n15). For the second global tick, the program can only proceed from state 3. Before continuing to traverse the child threads, the enclosing `abort` is checked. On preemption, n17 is reached (state 4). Otherwise, the *join* node (n16) is reached and the traversal of `main` is resumed (states 5 and 6). Note that state 6 ends at the same *pauses* as state 3 (n5, n7, n9, n15). Hence, we ignore the successors of state 6 as they would be the same as state 3's. The traversal of the CCFG is guaranteed to reach a fix-point because the number of nodes in the CCFG is finite and previously visited states are ignored. This ensures the termination of reachability.

B. Computing the Reaction Time

The execution time of each program state can be computed during reachability because the CCFG is traversed using ForeC semantics and the thread scheduling is statically known. Fig. 7 is a timeline showing how the execution time of state 3, from Fig. 6, is computed. The computation starts from the left of the timeline. The computed execution time for Cores 1 and 2 are tracked separately by their own integer counter, initially set to 0. This allows us to handle thread execution times that overlap due to parallel execution. As threads are traversed, the counters are incremented by the computed execution times of their allocated threads, scheduling overheads, TDMA bus delays, and inter-core synchronization costs. Threads in the CCFG are traversed in the total order defined for static scheduling to preserve the scheduling order on each core. Thus, we compute the execution time of `main`, followed by `camL`, `camR`, and so forth. Each core's execution times are shown in Fig. 7 as a row of blocks annotated with the executed CCFG nodes (from Fig. 4) or scheduling overheads (listed in Table III). The black blocks represent bus delays and the patterned blocks represent inter-core synchronization costs. The remainder of this section describes how the scheduling overheads, TDMA bus delays, and inter-core synchronizations are resolved during timing analysis.

Table III
TYPES OF SCHEDULING OVERHEADS.

Type	Description	Type	Description
sf	Send a fork.	cs	Context-switch.
rf	Receive a fork.	lt	End the local tick.
rj	Receive a join.	gt	End the global tick.

C. Scheduling Overhead, Bus Delay, and Synchronization

For each type of scheduling overhead (Table III), its execution time is computed by analyzing the assembly instructions and control-flow of the scheduling code. The use of non-preemptive thread scheduling means that the overheads only occur between local ticks. For example, in Fig. 7, `main`'s execution time is computed first and assigned to core 1. Since `main` completed its local tick by forking, the overhead for synchronizing the fork across the participating cores needs to be computed. For core 1, the overhead is the execution time of the (non-blocking) send routine, labeled *sf1* in Fig. 7. However, the send routine requires access to global memory which must go through the shared TDMA bus. The TDMA bus allocates fixed-length time slots to each core for accessing the bus. The slots are statically scheduled in a round-robin manner, called a bus schedule. In Fig. 7, the row labeled TDMA shows the bus schedule for cores 1 and 2 repeated over time. Each slot is 5 clock cycles long. Arrows between the scheduling overheads and bus schedule indicate read/write accesses to global memory. If an access occurs within the core's slot, then the access is granted immediately. If the access does not finish within the core's remaining slot time, then the access is aborted and retried at the core's next slot. If the access occurs outside of the core's slot, then the access is delayed until the core's next slot. The bus delays need to be computed precisely for tight WCRT analysis. The starting times of each slot in the TDMA bus schedule can be modelled by the following recurrence relation:

$$S_n^i = S_n^{i-1} + D; \quad S_n^0 = I_n \tag{1}$$

where S_n^i is the starting time of core n's slot in the i-th repetition of the TDMA bus schedule, D is the duration of the bus schedule, and I_n is the initial start time of core n's

slot. For Fig. 7, the bus schedule is 10 clock cycles long and core 1 and 2's slot starts from 0 and 5 clock cycles, respectively. Using equation 1, core 1's slots start from 0, 10, 20, ... clock cycles. The bus delays for core 1's accesses can be computed from the slot start times by taking into account each slot is 5 clock cycles long.

We continue with the computation of the synchronization overheads for `main`'s fork. The send routine *sf1* in Fig. 7 completes without any bus delay. For core 2, the overhead is the execution time of the (blocking) receive routine. The routine is delayed until core 2's TDMA slot is scheduled because it accesses global memory. The bus delay is shown as a black block. The routine prepares to block and this initial overhead is computed, labeled *rf1* in Fig. 7. The blocking can only finish after *sf1* has sent the fork. Thus, we advance core 2's execution time to core 1's execution time computed so far (11 clock cycles). Note that, core 2's execution time would not be advanced if it was greater than core 1's. The routine stops blocking the next time it accesses the global memory to receive the fork. This final overhead is computed and labeled *rf2* in Fig. 7. The patterned block between *rf1* and *rf2* is the inter-core synchronization time.

The traversal of the CCFG continues for the child threads `camL`, `camR`, `move`, followed by `link`. For each thread, the context-switching overhead is computed first (labeled, for example, *cs1* in Fig. 7). For `camL`, `camR`, and `move`, the overhead also includes the cost to copy the shared variable `total`, which requires global memory access. Next, the execution time of the thread's local tick is computed. Then, we compute the overhead for updating the thread's execution state and, if applicable, the overhead for writing its modified copy of `total` to global memory. This overhead is labeled, for example, *lt1* in Fig. 7. Since `link` completes its local tick by forking, the synchronization overhead for cores 1 and 2 are computed (labeled *sf2* and *rf3*). The traversal of the CCFG continues for the child thread `linkIn` and then `linkOut`. After completing these traversals, we need to compute the synchronization overhead for joining the child threads back to its parent, `link`. As send and receive routines are used for synchronization, we follow the same approach used to compute the synchronization overhead for a fork. In Fig. 7, *rj1* is the initial overhead to block for `linkOut`'s execution state. The patterned block after *rj1* is core 1's inter-core synchronization time and *rj2* is the final overhead to stop blocking.

When all the threads in the program state have been traversed, the overhead for ending the global tick is computed. The overhead is the execution time needed to perform the housekeeping tasks and it is assigned to core 1, the nominated core (labeled *gt1* in Fig. 7). Finally, the execution time of the global tick is computed as $max(Core\ 1,\ Core\ 2) =$ 73 clock cycles.

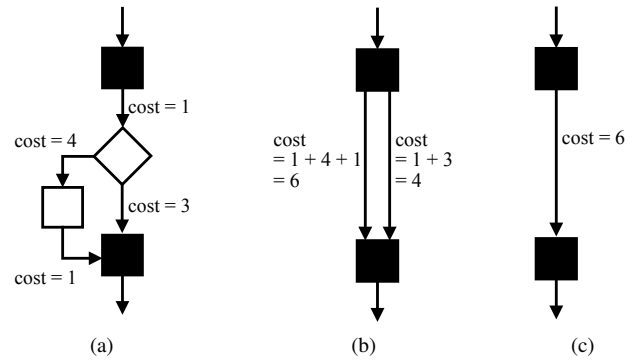


Figure 8. CCFG optimizations. (a) The original CCFG, (b) merging of computation and condition nodes, and (c) merging of edges.

D. Time Complexity

The time complexity of reachability is equal to the program's total state-space. Let t_i denote the number of reachable *pauses* in thread i. Then, the program's total state-space is the product of the reachable *pauses* of each thread, or $t_1 \times \ldots \times t_n$ for n number of threads. The complexity of computing the synchronization costs is $O(c)$, where c is the number of cores. The complexity of computing the context-switching costs and the sending of thread execution state is constant. The complexity of computing the cost to copy the shared variables for each local tick is $O(s)$, where s is the number of shared variables. The complexity of computing the cost to complete the housekeeping tasks at the end of the global tick is also $O(s)$. The complexity of computing the TDMA bus delays is constant. Therefore, the overall time complexity of the approach is $O(t_1 \times \ldots \times t_n \times s \times c)$.

E. Optimizations

Our reachability approach (Section V-A) does not use value analysis to prune away infeasible paths arising from conditional branches. Thus, reachability explores a superset of all feasible global ticks. However, this allows us to reduce the size and complexity of the CCFG [16] to improve the performance of reachability without losing precision. We illustrate two optimizations on the CCFG sub-graph shown in Fig. 8a. The cost to execute a node is placed on its incoming edge.

1) Merging Computation and Condition Nodes: This reduces the number of nodes that have to be traversed in the CCFG. We take an outgoing edge from a *computation* or *condition* node and merge it with the outgoing edges of its destination node. The cost on the outgoing edge is added to the destination edges. Applying this optimization to Fig. 8a produces Fig. 8b.

2) Merging Edges: This reduces the program's reachable state-space by reducing the number of paths in the CCFG. When multiple edges exist between two nodes (Fig 8b), reachability explores all edges separately. We observe that the state containing the edge with the highest cost always

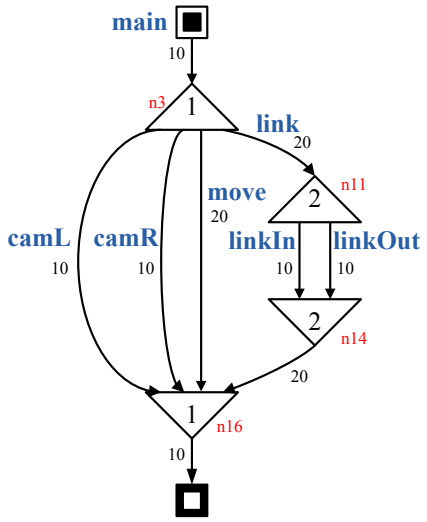


Figure 9. Simplified CCFG of Fig. 4 for Max-Plus.

has the highest execution time. Thus, only the edge with the highest cost is kept (Fig. 8c).

VI. Max-Plus for Multicore Execution

The existing Max-Plus approach [15] only targets the single-core execution of synchronous programs. To allow for a comparison between Max-Plus and Reachability, we modify the Max-Plus approach to target multicores by adding support for the computation of the fork/join, inter-core synchronization costs. We start by computing the WCRT of each individual thread and then simplifying the threads in the CCFG into single edges annotated with their WCRT. Applying this simplification to Fig. 4 produces Fig. 9. The simplified CCFG has only one reachable global tick because each thread is an instantaneous path. For the parent threads (`main` and `link`), their computed WCRT appears twice on the incoming and outgoing edges of the *fork* and *join* nodes, respectively. This is needed to consider the possibility that their child threads could fork or join at the start or end of a local tick. Using the simplified CCFG, the program's WCRT is computed according to Sections V-B and V-C.

VII. Results

In this section, we compare the precision and performance of the proposed Reachability approach (Section V-A) against the Max-Plus approach (Section VI) for timing analysis. We implemented a C++ timing analysis tool that targets programs compiled for execution on the multicore simulator, presented in Section II. The simulator had the following memory configuration. Each core had 16KB of local data and instruction memory with 1 cycle access time. The global memory was 32KB in size with 5 cycle access time. The shared TDMA bus had a fixed-length time slot of 5 cycles for each core. Thus, the TDMA bus schedule was 5×*(number of cores)* cycles long.

Table IV
Benchmark programs.

Program	Lines	Threads	Levels
802.11a	2147	26 (10)	2
FmRadio	808	13 (4)	1
Fly by Wire	204	9 (7)	2
Life	1020	11 (8)	2
Matrix	232	9 (8)	1

We chose moderate to large sized ForeC programs with varying levels of parallelism for benchmarking. In Table IV, the first and second column gives the name of each program and the number of lines of ForeC source code. The third column gives the total number of threads in the program. The maximum number of threads that can be forked in parallel is given in brackets. The fourth column gives the deepest level of thread nesting. `802.11a` and `FmRadio` are large signal processing applications taken from [24]. The original C programs were modified to ForeC programs, suitable for multicore execution. Both programs contain complex control and data-dominated computations and operate on large data sets in global memory. `802.11a` is the largest program and its threads have unbalanced workloads. `Fly by Wire` is based on the PapaBench [25] and chosen for its control-dominated computations and threads with unbalanced workloads. Lastly, `Life` and `Matrix`, were chosen for their data-dominated computation of matrices.

We analyzed each program using Reachability and Max-Plus, and recorded the *computed* WCRTs. The programs were distributed up to their maximum number of parallel threads and the analysis was repeated for each possible distribution. The timing analyzer produced a timing trace for the WCRT, which was used to (manually) refine the thread distributions. To evaluate the tightness of the computed WCRTs, we executed the programs on the multicore simulator (Section II) for one million global ticks or until the program terminated. The timing analyzer also reported the program state with the computed WCRT. Test vectors were generated to elicit this program state 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. All experiments were performed on a 2.20 GHz Intel Core 2 Duo computer with 3GB RAM, with the Linux kernel 2.6.38.

We begin by comparing the observed and computed WCRTs of the programs (in clock cycles), plotted as bar graphs in Fig. 10. Apart from `Matrix`, Reachability computes much tighter WCRTs than Max-Plus, even when the programs are executed on one core. Max-Plus has large over-estimations because the assumption that all thread WCRTs occur in the same global tick is not valid. However, the assumption is valid for `Matrix` because it completes its execution in one global tick. Hence, Max-Plus computes a tight WCRT for `Matrix`. For Max-Plus, the amount of over-estimation depends on the program's structure and

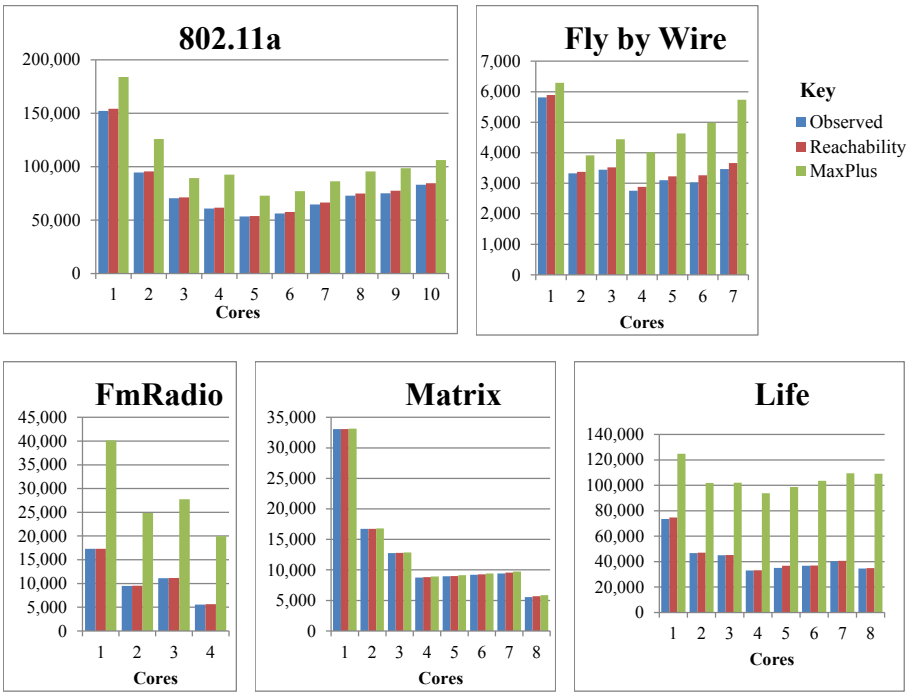
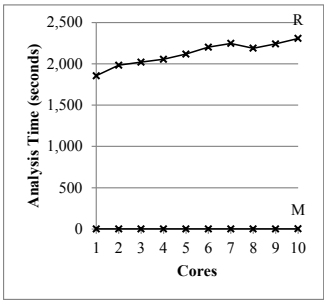
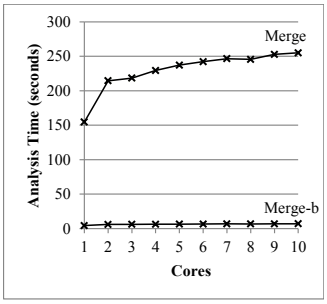


Figure 10. WCRT results for the benchmark programs in clock cycles.



(a) Reachability and Max-Plus without optimizations.



(b) Reachability with optimizations.

Figure 11. Analysis times for 802.11a.

thread distribution. For example, the WCRTs of sequentially executed threads get summed together. Sequential execution can result from control-flow dependencies, such as the sequencing or nesting of `par` statements. Threads allocated to the same core will also execute sequentially. Also, the parent thread's WCRT is summed again after each join for a sound WCRT computation. Such program structure contributes towards the high over-estimations for `FmRadio` and `Life`. For Reachability and Max-Plus, the inter-core synchronization time is over-estimated by assuming the worst-case scenario. The worst-case is when the receive routine reads from the global variable just before it is updated by the send routine.

For `802.11a`, Fig. 11a plots the analysis time for Reachability (*R*) and Max-Plus (*M*) without CCFG optimizations. Reachability has much higher analysis times because the analysis complexity (Section V-D) is much higher than Max-Plus. Fig. 11b plots the analysis time for Reachability with CCFG optimizations (Section V-E). *Merge* denotes the merging of CCFG nodes, and *Merge-b* denotes the merging of CCFG nodes and edges. The average speed up in analysis time was a factor of 9.34 for *R-merge* and a factor of 342 for *Merge-b*. For *Merge-b*, the maximum analysis time was only 6.93 seconds. This dramatic speed up is due to the reduction in the program's reachable state-space (Section V-E2). Table V gives the number of reachable states for the programs, with (*Merge*, *Merge-b*) and without (*None*) the optimizations. The slow increase in analysis time for *Merge-b* demonstrates its scalability over increasing number of cores.

Table V
REACHABLE PROGRAM STATES.

Program	None	Merge	Merge-b
802.11a	43,695	43,695	515
FmRadio	4,296	4,296	36
Fly by Wire	50,032	50,032	1,032
Life	2,053	2,053	1,029
Matrix	1	1	1

The observed WCRTs in Fig. 10 show that the programs benefit from multicore execution. `Matrix` had the greatest speedup of 5.94 times on 8 cores. Although its 8 parallel threads are symmetrical, they cannot be distributed evenly over 5 to 7 cores. Thus, some cores are always allocated with 2 threads and the WCRT cannot improve. `Fly by Wire` had the least speed up of 2.12 times on 4 cores. This was because the thread workloads could not be balanced over the cores, which prevented the full utilization of the cores. For `802.11a`, its WCRT at 5 cores corresponded to the execution time of one thread which was already allocated to its own core. Thus, the WCRT could not be improved by distributing the remaining threads. The WCRT increases after 5 cores because of the increasing scheduling overheads and cost of accessing global memory. These costs reduce the benefit of multicore execution.

Overall, the results show that Reachability computes far tighter WCRTs than Max-Plus. Reachability with CCFG optimizations demonstrated a large reduction in analysis

time without trading-off precision. In addition, our implementation of the Reachability approach provides a timing trace for the program's WCRT. This feedback allows for an effective approach to design space exploration. The efficiency of the timing analysis enables it to be used early on in the exploration. The speed up in WCRTs also demonstrates ForeC's ability to take advantage of multicore execution. The only other known approach to the timing analysis of synchronous programs over multicores is [9]. Unfortunately, we cannot compare with [9] as their results are only for a four-core system with no precision results reported by them.

VIII. Conclusions

The ForeC language and associated timing analysis are presented to address the need for a predictable design framework over embedded multicores. ForeC augments the popular C-language with a small set of synchronous constructs. Determinism and parallelism are key properties of ForeC that aids in simplifying the understanding and debugging of parallel programs. A static timing analysis framework was presented for guaranteeing the worst-case performance of ForeC programs. Through benchmarking, we demonstrated the precision, scalability and efficiency of the analysis. For future work, we plan to incorporate instruction cache analysis, prune more infeasible paths by using value analysis, and use heuristics to decide thread distributions that best reduce the WCRT.

References

[1] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Burguiere, J. Reineke, B. Triquet, and R. Wilhelm, "Predictability Considerations in the Design of Multi-Core Embedded Systems," in *Proceedings of Embedded Real Time Software and Systems*, 2010, pp. 36 – 42.

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

[3] OpenMP Architecture Review Board, "OpenMP," http://openmp.org/wp/.

[4] B. Barney, "POSIX Thread Programming," https://computing.llnl.gov/tutorials/pthreads/.

[5] E. A. Lee, "The Problem with Threads," *Computer*, vol. 39, no. 5, pp. 33 – 42, 2006.

[6] C. E. McDowell and D. P. Helmbold, "Debugging Concurrent Programs," *ACM Comput. Surv.*, vol. 21, no. 4, pp. 593 – 622, Dec. 1989.

[7] 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, 2003.

[8] 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., 2005.

[9] 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*, Anaheim, California, 2010, pp. 48 – 51.

[10] S. Yuan, L. H. Yoong, and P. S. Roop, "Compiling Esterel for Multi-core Execution," in *Digital System Design (DSD), 2011 14th Euromicro Conference on*, Oulu, Finland, Sep. 2011, pp. 727 – 735.

[11] 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)*, Dresden, Germany, 2010, pp. 949 – 952.

[12] 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.

[13] P. S. Roop, S. Andalam, R. von Hanxleden, S. Yuan, and C. Traulsen, "Tight WCRT Analysis of Synchronous C Programs," in *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, Grenoble, France, 2009, pp. 205 – 214.

[14] R. von Hanxleden, "SyncCharts in C - A Proposal for Light-Weight, Deterministic Concurrency," in *Proceedings of the International Conference on Embedded Software (EMSOFT'09)*, Grenoble, France, Oct. 2009, pp. 225 – 234.

[15] M. Boldt, C. Traulsen, and R. von Hanxleden, "Worst Case Reaction Time Analysis of Concurrent Reactive Programs," *Electronic Notes in Theoretical Computer Science*, vol. 203, no. 4, 2008.

[16] M. Kuo, R. Sinha, and P. Roop, "Efficient WCRT Analysis of Synchronous Programs using Reachability," in *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, San Diego, USA, 2011, pp. 480 – 485.

[17] L. Ju, B. K. Huynh, S. Chakraborty, and A. Roychoudhury, "Context-Sensitive Timing Analysis of Esterel Programs," in *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, San Francisco, USA, 2009, pp. 870 – 873.

[18] M. Schoeberl, "Time-Predictable Computer Architecture," *EURASIP J. Embedded Syst.*, vol. 2009, pp. 2:1–2:17, 2009.

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

[20] G. Gebhard, C. Cullmann, and R. Heckmann, "Software Structure and WCET Predictability," in *Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES)*, Grenoble, France, 2011, pp. 1 – 10.

[21] G. J. Holzmann, "The Power of 10: Rules for Developing Safety-Critical Code," *IEEE Computer*, vol. 39, no. 6, pp. 95 – 97, 2006.

[22] Polyspace, http://www.mathworks.com/products/polyspace/.

[23] Parasoft, http://www.parasoft.com.

[24] 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*, Heraklion, Greece, 2011, pp. 5 – 14.

[25] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. D. Michiel, "PapaBench: A Free Real-Time Benchmark," in *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dresden, Germany, 2006.