Tight WCRT Analysis of Synchronous C Programs

Partha S Roop[*]
Department of Electrical and Computer Engineering
University of Auckland, New Zealand

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

Reinhard von Hanxleden
Department of Computer Science
Christian-Albrechts-University
Kiel, Germany

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

Claus Traulsen
Department of Computer Science
Christian-Albrechts-University
Kiel, Germany

ABSTRACT

Accurate estimation of the tick length of a synchronous program is essential for efficient and predictable implementations that are devoid of timing faults. The techniques to determine the tick length statically are classified as worst case reaction time (WCRT) analysis. While a plethora of techniques exist for worst case execution time (WCET) analysis of procedural programs, there are only a handful of techniques for determining the WCRT value of synchronous programs. Most of these techniques produce overestimates and hence are unsuitable for the design of systems that are predictable while being also efficient. In this paper, we present an approach for the accurate estimation of the exact WCRT value of a synchronous program, called its *tight WCRT value*, using model checking. For our input specifications we have selected a synchronous C based language called PRET-C that is designed for programming Precision Timed (PRET) architectures. We then present an approach for static WCRT analysis of these programs via an intermediate format called TCCFG. This intermediate representation is then compiled to produce the input for the model checker. Experimental results that compare our approach to existing approaches demonstrate the benefits of the proposed approach. The proposed approach, while presented for PRET-C is also applicable for WCRT analysis of Esterel using simple adjustments to the generated model. The proposed approach thus paves the way for a generic approach for determining the tight WCRT value of synchronous programs at compile time.

[*]Roop was supported by research and study leave from Auckland University and a research fellowship for experienced researchers from the Alexander von Humboldt foundation.

1. Introduction

Ubiquitous embedded systems have caught the attention of researchers over the past decade. Typical embedded applications ranging from complex aircraft flight controllers to simple digital cameras require worst case guarantees on their timing performance and hence are called *real-time systems.* The standard approach to real-time system design relies on determining the execution time of code statically, a process known as worst case execution time (WCET) analysis [22]. WCET analysis is used to determine task periods and deadlines. Then an RTOS is employed to emulate the concurrency of tasks and also to achieve task synchronization using mechanisms such as mutexes and monitors. There are several limitations to such an approach including the problem of determining tight WCET bounds. We start this section by first differentiating WCET and WCRT analysis.

WCET analysis is a process of determining the worst delay path in a given program executing on a conventional processor. Estimation of WCET of a given program through static analysis is a very complex process as the tools have to explore all execution paths of a program statically. Speculative processors add to this complexity and hence tools have to also take the underlying architecture into account. Wilhelm et al. [22] give a detailed survey of approaches for such analysis. It is obvious that such analysis is non-trivial and tools often rely on abstraction leading to overestimates.

For a synchronous program which executes in discrete instants, the analysis is a lot simpler. A synchronous program executes using ticks of a global clock. During each instant, inputs from the environments are sampled and latched. Then the *reaction function* is called to do the desired computation for the instant. Finally, the outputs generated by the function are emitted to the environment. Hence, the length of an instant is determined by the length of the longest reaction during any execution of the program. This task is known as WCRT or worst case reaction time analysis [5].

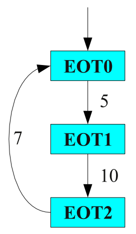
The KEP series of reactive processors [13] execute Esterel programs with predictable timing. They achieve this by fixing the tick length on the processor to the WCRT value obtained by static analysis. Alternatively, in "free running mode" each tick would complete as fast as possible, which can reduce average case reaction times at the expense of predictability. Boldt at al. [5] developed a WCRT analysis approach of single threaded Esterel programs using structural induction over different types of nodes of the KEP Assembler Graph (KAG). Subsequently, they extended this approach to multithreaded Esterel programs executing on the KEP3a processor [5]. This algorithm analyzes the worst tick length by depth first search over the intermediate graph format called CKAG (Concurrent KEP Assembler Graph). The algorithm computes the maximum tick lengths for every thread and the overall tick length is then just the sum of these maximum tick lengths (we call this value as the WCRT$_{max}$ for the program). This approach will thus lead to an overestimation of the WCRT and they establish through experimentation over Estbench [8] programs. The estimated WCRT is on the average about 40% overestimated [5]. An approach to arrive at more accurate values is recently proposed in [10]. Here, Esterel programs are first mapped to C using the CEC compiler, and then, an ILP formulation is developed to eliminate redundant paths in the code, thus yielding more accurate results. Similarly, the approach of [5] has been extended in [16] using an algebraic framework for more accurate WCRT analysis that also eliminates redundant paths.

A synchronous program may be executed using the concept of *variable ticks* [19] to facilitate good average case performance at the expense of predictable execution. However, for real-time systems, we must execute a synchronous program with a fixed tick length such that no timing faults are possible. We term a precise value of this tick length as WCRT$_{tight}$ to indicate the fact that any value less than this may have a possibility of timing faults during the execution of the program. Computing an optimal or tight value for WCRT will pave the way for the design of precision timed machines (PRET) [9] that have been recently proposed as alternative architectures for real-time computing. PRET machines demand that the architecture should guarantee precise timing without sacrificing throughput. Another implicit objective of PRET machines is to facilitate predictable execution of concurrent C code. To this end, we propose a synchronous extension to C, called PRET-C and then develop an approach for tight WCRT analysis of PRET-C programs executing on general purpose processors with minimal customization.

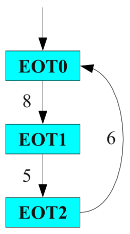
We observe that the overall tight WCRT of a synchronous program (like Esterel) is not necessarily the sum of the maximum over the threads, but the maximum of the sum of the local instants (we call local ticks) over all possible executions of the program. A reactive program has infinite executions and hence computing this by run-time simulation of the program is infeasible. However, we observe that any synchronous program is a collection of strongly connected components (SCCs) [7] and the behaviour of the global programs keeps on branching among these SCCs. Hence, the global behaviour reaches a fixed point. Based on this observation, we propose that the WCRT analysis of a synchronous program is equivalent to the model checking question to compute this fixed point. This is the main hypothesis based on which the current paper is formulated.

The use of model checking for the analysis of real-time systems is not new. Metzner illustrates in [17] the effectiveness of using model checking for WCET analysis using the notion of a basic block automaton to represent a program. A static analysis based formal approach is similarly presented in [15] to compute tight bounds on synchronous programs. Our approach is significantly different from earlier approaches to WCRT analysis in the following ways. Earlier approaches to tighter analysis [10,16] concentrate on the removal of redundant paths but ignore the redundancy present in the overall state-space of the program. We propose a model checking based tight analysis that performs redundant path elimination while also taking the state-based program execution into account. The approach of [15] takes exponential time in the worst case to generate the timed Kripke structure and the timing analysis is done over this model. In contrast, our model is a composition of several automaton and hence the model checker can perform many optimizations or perform compositional reasoning. Most importantly, the complexity of the proposed WCRT analysis is the normal model checking complexity of CTL [7] multiplied by a constant term (see Section 3.6). To our knowledge, our approach is the first model checking based formulation of the tight WCRT analysis problem of synchronous programs. Though developed for PRET-C, the proposed approach can also be applied to other synchronous language like Esterel.

1.1 Motivating Example



(a) TFSM thread 1



(b) TFSM thread 2

Figure 1: Motivating Example

Unlike earlier approaches [5,10], we propose an approach to determine the tight WCRT value of a synchronous program by mapping the WCRT analysis problem to a fixed point computation. We motivate the approach using the following simple example as shown in Figure 1. This example captures a synchronous program with two threads represented as two FSMs. The states of the FSMs correspond to the end of tick (denoted as EOT in PRET-C or pauses for Esterel). The transitions are weighted by integers that represents the length of the computation to move between the two specified EOTs. This representation is generic to any synchronous language.

Given some intermediate representation of a synchronous programs where ticks are marked explicitly in the graph [24] along with explicit cost values for each node (where the cost represents the maximum number of processor clock cycles needed to execute the code corresponding to the node), it is easy to transform such an intermediate representation into a set of FSMs, similar to the ones shown in Figure 1. We call these FSMs as TFSMs (timed FSMs) since transitions are guarded by the actual time instance (processor clock cycles). For example, the TFSM corresponding to thread1 takes 5 clock cycles to move from *EOT0* state to *EOT1.* We will present the intermediate format for PRET-C and the translation process to TFSMs in Section 2.2.

Given the mapping of a synchronous program to a set of TFSMs, it is easy to calculate the $WCRT_{max}$ [5] for this program. For this example, this value would be the sum of the maximum tick length in thread 1 and thread 2 i.e, $10 + 8 = 18$. However, note that synchronous programs execute such that the ticks of the two threads synchronize before moving to the next instant. This execution may be modeled as a barrier synchronization like the one used in [24]. In this execution, in the first instance of the program, thread 1 will first complete its local tick after 5 time units. If thread 2 is scheduled next, then it will take further 8 time units. Hence, the tick length of the first global tick of the program would be 13 time units. Since both programs execute in two fixed cycles ($EOT0 \rightarrow EOT1 \rightarrow EOT2 \rightarrow EOT0$), the actual WCRT of the program, $WCRT_{tight}$ would be then $max(5 + 8, 10 + 5, 7 + 6) = 15$, if we assume that the two threads have no data dependencies between them. We will discuss the issue of data dependencies in Section 3.4.

Note that for this simple example, it is feasible to do this analysis by hand. However, for general programs with many branches, it will be impossible to do this analysis in this manner, as each thread will have many cycles and there will also be arbitrary branches between these cycles. However, we make two key observations in order to be able to do this tight analysis automatically. Firstly, the proposed analysis is like a run-time simulation of the program. In general, this problem looks infeasible for complex programs. However, we observe that any synchronous program is a collection of strongly connected components (SCCs) [7] and the behaviour of the global programs keeps on branching among these SCCs. Hence,the global behaviour reaches a fixed point. A second observation is that TFSMs can be modelled with automaton that have a single integer variable to model the cost of transitions. Then a symbolic model checking technique may be employed to compute the WCRT value of a set of concurrent automaton. We have selected the UPPAAL [1] tool since it offers very efficient algorithms for not only Timed Automata (TA) [2] but also for automata with integer operations. Using these observations, we now propose a mapping of TFSMs to TAs that don't have any clock variables, and then develop a model checking solution to the WCRT analysis question.

The overall goal of the paper is the development and programming of precision timed architectures (PRET). The main contribution are: (1) A novel approach for static timing analysis of synchronous programs is developed. The proposed approach combines state-dependencies with the elimination of redundant paths to yield tighter results. (2) A new synchronous C extension, called PRET-C, is developed for programming PRET machines using a notion of logical time, that is mapped to physical time using a static timing analyser. (3) We propose a new approach for the design of PRET architectures by simple customizations of soft-core processors, called ARPRET. This is unlike the tailored processor approach for PRET proposed in [14].

The organization of the paper is as follows. In Section 2 we present the synchronous C extension for predictable execution called PRET-C. We also present the intermediate format for PRET-C in this section (Section 2.2). In Section 3, we present a model checking based formulation for precise WCRT analysis of PRET-C programs. We illustrate our approach on a running example from Section 2. The proposed PRET architecture that is derived through simple customizations of a general purpose processor (GPP) is presented in Section 4. In Section 5, we present some benchmarking results of our WCRT analysis that compares the proposed approach with existing approaches using some Esbench [8] programs that are written in PRET-C. The final section makes concluding remarks.

2. PRET-C Overview

PRET-C or Precision Timed C is a synchronous extension of the C language similar in spirit to ECL [12] and ReactiveC [6]. However, unlike the earlier synchronous C extensions, it provides only minimal set of extensions and is specially designed for predictable execution on a GPP with simple customizations to achieve PRET. Moreover, unlike the use of signals [6, 12], we used standard shared variables in C for thread synchronization and programs are thread-safe by construction. PRET-C extends C using only three major constructs shown in Table 1. Details of the language and its semantics are reported in [3].

Statement	**Meaning**
PAR(T, U)	synchronous parallel execution of the two threads with higher priority of T over U
EOT	defines the end of a local tick
[weak] abort p when pre c	preemption construct

Table 1: Extensions to C

`PAR()` emulates concurrency by calling a set of C functions synchronously. However, unlike the usual || of Esterel where threads are scheduled in each instant based on their signal dependencies, threads in PRET-C are always scheduled based on a fixed static order based on their textual order in the `PAR` statement.

`EOT` is our extension to provide precise timing to a thread. A thread completes its instant of time, called its *local tick*, when it reaches an `EOT` statement. A *global tick* elapses only when all participating threads of a `PAR()` reach their respective `EOT`. In this sense, the `EOT` is similar to the `pause` statement of Esterel. The `EOT` is used to ensure precise timing of execution of the program by ensuring that the next tick is started only when all threads have reached their barriers (`EOT`) and also the duration of the tick is not less than the WCRT of the program derived by static analysis (to be presented in Section 3). Note that an `EOT` is similar in spirit to the deadline instruction of [14]. However, unlike the *low-level* deadline instruction that manages timing by associating timers, an `EOT` is a *high-level* programming construct. Unlike [14], the task of ensuring precise timing of threads is not left to the programmer but is derived by WCRT analysis and is a compilation task. Moreover, the deadline instruction is also used for achieving mutual exclusion by time-interleaving the access to shared memory. This is achieved by setting precise values to the deadlines. However, this task, if done manually, can be very complex for even simple programs. This is mainly due to arbitrary branching constructs and loops. Also, automating this task is non-trivial and has not been solved in [14]. Our solution to achieve mutual exclusive access to shared memory, on the other hand, is ensured by having static thread priorities and then scheduling threads in this fixed linear order in every instant.

The `abort` construct preempts the body immediately when some condition is true (like immediate aborts in Esterel).

Abortion can be either *strong* (`abort` construct) or *weak* (when the optional `weak` keyword is used). In case of a strong abort, the preemption happens at the beginning of an instant while the weak abort allows its body to execute and then the preemption triggers at the end of the instant. Also note that all preemptions in PRET-C are triggered based on the `pre` value of a Boolean condition i.e., based on the evaluation of the condition in the previous instant. This is needed since the status of variables change during an instant. The use of the pre ensures that preemptions are always taken based on the steady state values of the variables from the previous instant.

A PRET-C programmer just writes a set of normal C functions and spawns concurrent threads using the `PAR()` construct. Threads communicate through global shared variables. The task of ensuring mutually exclusive access is that of ARPRET (pronounced Our-PRET) that is derived by some simple customizations to the Microblaze [23] softcore processor to create a PRET Machine (as discussed in Section 4). It achieves this by ensuring that in every instant of time all threads execute in a fixed linear order by the scheduler. The detailed semantics of PRET-C is available in a companion report [3]. We illustrate these features through the example in the next subsection.

2.1 A Producer Consumer Example

Listing 1: A Producer Consumer in PRET-C

```
#include <pretc.h>
#define N 1000
void sampler(void);
void display(void);
extern sensor; int cnt = 0; float buffer[N];
int main() {
    PAR(sampler, display);
    return 0;
}
void sampler() {
    int i = 0; float sample;
    while (1) {
      sample = read(sensor);
      EOT;
      while (cnt==N) EOT;
      buffer[i] = sample;
      EOT;
      i = (i + 1) % N
      cnt = cnt + 1;
    }
}
void display() {
   int i = 0; float out;
   while(1) {
     while (cnt==0) EOT;
     out = buffer[i];
     EOT;
     i = (i + 1) % N;
     cnt = cnt - 1;
     EOT;
     WriteLCD(out);
   }
}
```

We present a simple producer-consumer adapted from [21] to motivate PRET-C which is shown in Listing 1. The main thread spawns two threads, namely a sampler thread that reads data from a sensor and deposits the data on a global circular buffer and a display thread that reads the deposited data from the buffer and displays this data. Note that the sampler thread and the display thread communicate using shared variables `cnt` and `buffer`. Also, the programmer has assigned a higher priority to the sampler compared to the display thread due to the textual order specified in the call to `PAR()`.

In this program, the sampler thread starts by reading the sensor data in its first local instance of time (local tick). In the next instant, it checks if the data buffer is full, and in this event it just ends its local tick. As long as the buffer is full, it keeps on waiting until the display thread has read some data so that there is empty space. If it successfully comes out of the while loop, then it writes to the next available location of the buffer and ends another local tick. In the next instant of time the index to the buffer and the total number of data in the buffer are incremented (note that this is a circular buffer). Then the sampling loop is restarted.

The display thread starts by first checking if there is any data available to be read ($cnt \neq 0$). If there is no data available, then the thread ends its local tick and keeps on waiting until some data is deposited by the producer. When this happens, it reads the next data from the buffer and ends its local tick. In the next instance, the value of *cnt* is decremented and in the final instance the data read is sent to a display device.

During 7^{th} tick both $cnt = cnt{+}1$ and $cnt = cnt{-}1$ will be executed. However, due to the priority of the sampler, *cnt* will be first incremented by 1 and once the sampler reaches its EOT, the scheduler will (see Section 4) select the display thread that will decrements *cnt* by 1. This repeats every six ticks and the value of *cnt* will always be consistent without the need for enforcing mutual exclusion.

It is easy to see that the execution of this code on any processor and an RTOS to emulate concurrency will lead to race conditions, due to the non-exclusive access to the shared *cnt* variable. It is the responsibility of the programmer to ensure that critical sections are properly implemented using operating system primitives such as semaphores. However, on ARPRET, the execution will always be deterministic. The architecture effectively maps the synchronous parallel into a fixed sequence, data coherency will always be guaranteed [3].

Given a PRET-C program, the next question is how to obtain the tight WCRT analysis of this program. As stated in Section 1, we have mapped this problem to a model checking question. Model checkers require a model of the program to perform analysis. To facilitate the creation of a suitable model for model checking we introduce an intermediate format for PRET-C in the next section.

2.2 Intermediate Format

We propose a new intermediate format for PRET-C programs called Timed Concurrent Control Flow Graph (TCCFG). We generate the intermediate format from the assembler level (rather than the source level) so as to get precise values for each instant in terms of ARPRET clock cycles. Moreover, by working on the assembler level, compiler optimizations need not be turned off.

Since all the PRET-C extensions in Table 1 are standard C macros, using the Microblaze C compiler (gcc) we generate the optimized assembly code. The analysis of this assembly is performed to extract the TCCFG. TCCFG encodes the explicit control-flow of the threads and also has information regarding forking and joining of the threads. The TCCFG corresponding to our example is shown in Figure 2.

TCCFG has the following types of nodes:

- Start/end node: Every TCCFG has a unique start node where the control begins and may have an end

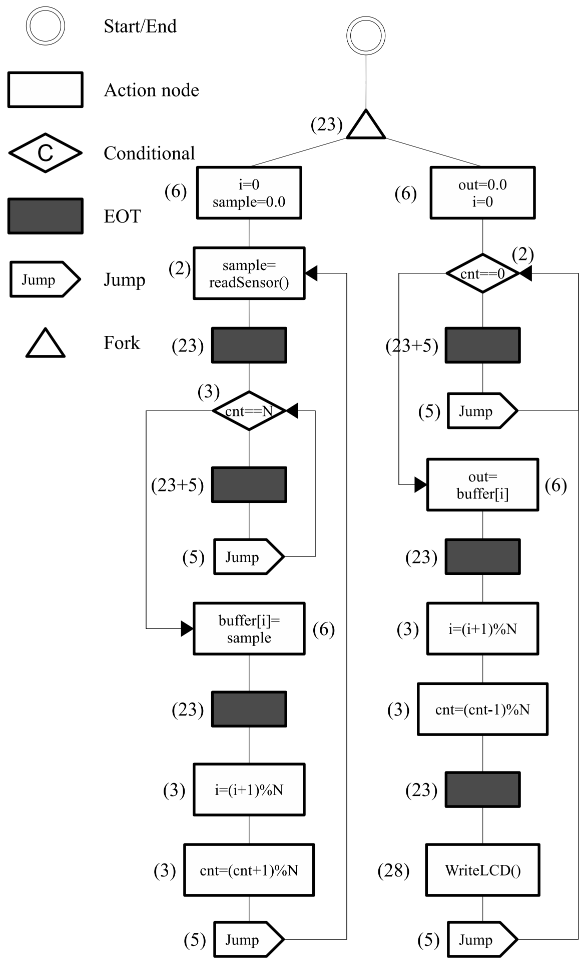


Figure 2: TCCFG of the Producer-Consumer

node, if the program can terminate. These are drawn as concentric circles.

- Fork/join nodes: These are needed to clearly mark concurrent threads of control and where these threads start and end. These are drawn as triangles.
- Action nodes: These are used for any C function call or data computation. We use rectangles to denote these.
- EOT nodes: These nodes indicate a local end of tick and are denoted as filled rectangles.
- Control flow nodes: We have two types of control flow nodes: conditional nodes to implement conditional branching (denoted by a rhombus) and jump nodes for mapping unconditional branches (which are needed to emulate infinite loops).
- Node weights: Beside each node the cost of the node is specified within brackets. This value represents the exact number of clock cycles needed to execute the assembler instructions for that node.

To deal with preemption (abort) we also need additional abort nodes and nodes for checking preemption at tick boundaries called *ckhabort*. These are described in detail in [3] along with the semantics of PRET-C. It is quite easy to spot that the TCCFG is a faithful model of the control flow of the original source and is a one-to-one mapping of the source code into a graph code format. For example, like the program, we start in the TCCFG with the fork node that starts the two threads of control. The first thread then initializes its local variables followed by the sampling of data. It then ends its local instant and hence there is an EOT node and so on.

3. WCRT Analysis using Model Checking

3.1 Preliminaries

A synchronous program executes in discrete instants called *ticks*. A compiler for a synchronous program, usually *compiles away* the logical concurrency to obtain a purely sequential function, which is termed as a *reaction function*. During every instant of execution of the program, inputs from the environment are first read and latched. Then the reaction function is called with these inputs as argument and the generated outputs from the function are finally emitted to the environment. This behaviour is then repeated for each tick. To respect the synchrony hypothesis, any implementation must ensure that the minimum inter-arrival time of external events must be greater than the maximum time of the reaction function. WCRT analysis refers to the process of determining the worst case length of the *tick* of a synchronous program by determining the maximum value for the reaction function such that this tick length will always ensure safe execution of the program (without any possibility of timing faults). We start by defining three different aspects of the WCRT of a program, namely the tight value, the maximum value and the minimum value.

Definition 1. The WCRT value of a synchronous program is equal to the the maximum execution time of the reaction function obtained over all possible execution paths of a program. We will also term this as the tight WCRT value of the program, called $WCRT_{tight}$, to indicate the fact that any value less than this value may cause a timing fault during the execution of the program.

Definition 2. The maximum WCRT value of a synchronous program, termed $WCRT_{max}$, is defined as the sum of the maximum local tick lengths for the participating threads of the program.

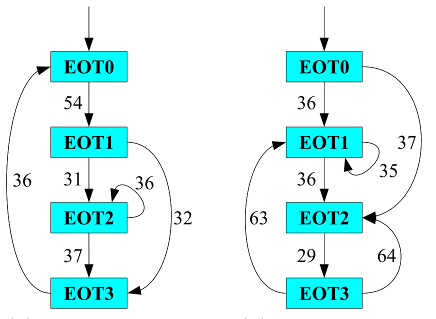
Definition 3. The minimum WCRT value of a synchronous program, termed $WCRT_{min}$, is defined as the sum of minimum local tick lengths for the participating threads of the program.

$WCRT_{tight}$ lies between these two values, i.e, $WCRT_{tight}$ lies in the interval [$WCRT_{min}$,$WCRT_{max}$]. For example, in the producer consumer case, the $WCRT_{min} = 31 + 29$ and $WCRT_{max} = 54+64$ (the numbers can be seen in Figure 3). Hence, $WCRT_{tight}$ has a value in the interval $[60, 118]$.

The overview of the proposed approach is as follows. We convert a TCCFG into an equivalent automata with a single integer variable to capture the cost of transitions. For illustration purposes, we first map TCCFG to TFSMs, which are then mapped to an equivalent TA that has no clocks but a single integer variable. We use a well known model checker for timed automata called UPPAAL [1] to do this mapping. We then model the $WCRT_{tight}$ computation problem to the checking of a CTL property in UPPAAL over *automata with integer variables but no clocks.*

3.2 Mapping to Timed Automata without any Clocks

For illustration, we first map TCCFG to an equivalent TFSM. The TFSM corresponding to the two threads of the producer consumer TCCFG of Figure 2 is shown in the Figure 3. This mapping is done automatically by a depth first search from every EOT node to all EOT nodes that are reachable from this node. During the traversal, the cost of every node is simply added to obtain the total cost between these two EOTs. For example, in Figure 2 the cost of the edge between EOT1 and EOT2 in thread one is 31 clock cycles, which is obtained by adding the costs of all the nodes between these two ticks. The cost of an individual node is obtained by first obtaining the assembly program from the C source and then generating the TCCFG from this assembly program. In our case, this program corresponds to Microblaze assembly and we automatically calculate the processor clock cycles needed for each node by looking at the assembly code corresponding to the node. Note that we generate these costs based on the ARPRET architecture (detailed in Section 4). On ARPRET we have no speculative features enabled. Hence, computing the costs are simple. For example, since we don't use branch prediction, every conditional node's false branch has an extra cost of *five* clock cycles to account for pipeline flushing (see Figure 2 where we have 23+5 to indicate the cost of pipeline flush).



(a) Sampler thread (b) Display thread

Figure 3: TFSMs for the Producer Consumer Example

The next step is the mapping of the TFSM to a timed automata. Note that the composition between TFSMs is strictly synchronous while TA compositions are asynchronous [18]. Hence, the mapping has to be done carefully to preserve synchrony. We illustrate the mapping using the same producer consumer example as shown in Figure 4. The overall mapping is achieved by mapping each TFSM to an equivalent TA. An additional TA, called a barrier, is also introduced to realize the synchronous semantics of PRET-C execution.

3.3 Illustration

The proposed solution for the producer consumer example is shown in Figure 4. We start by describing the conventions used in this figure. We have two kinds of states in the mapped TAs, namely EOT states and barrier states (labelled as EOT$_i$ and b$_{ij}$ respectively). Each transition has two parts. The first part of the transitions represent the transition guard which is the enabling condition of the transition. These appear at the top part of each transition. The bottom part of the transition are actions that are executed when the transition is taken (in UPPAAL this is known as the update part of the transition). For example, the transition from $EOT0$ to $EOT1$ in Figure 4 has $!gtick$ (syntax in UPPAAL to capture $\neg gtick$) as its guard and $x = x + 54, lt1 = true$ as its update parts respectively.

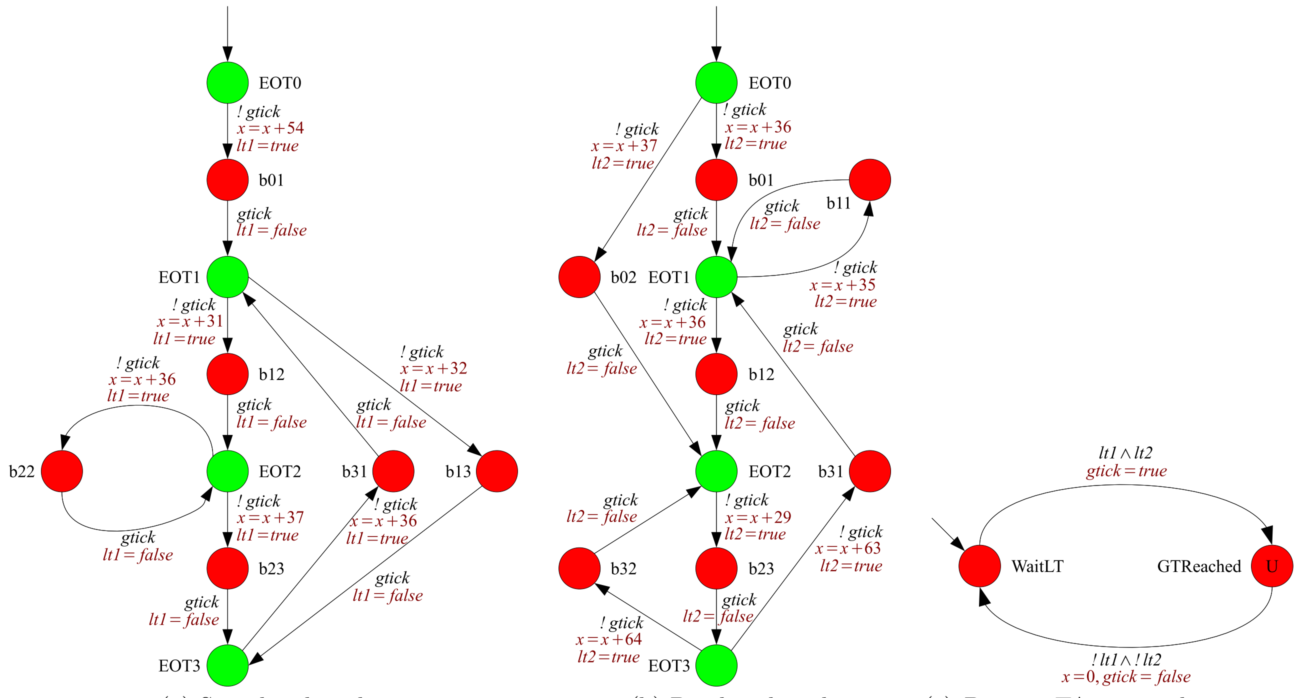
For the proposed mapping a single integer x is used to capture the cost of a global tick. We also use a Boolean variables lt to capture if a given thread has completed its local tick. We use a Boolean variable called $gtick$ that is true when the global tick has happened. These five variables are defined as global.

TAs are composed using an asynchronous parallel operator similar to CCS [18]. To map a given TFSM to an equivalent TA (without clocks) so as to realize synchronous execution semantics, we do the following. For every state EOT_i in the TFSM, we also have an identical state EOT_i in the TA. For every transition from $[EOT_i] \xrightarrow{d} [EOT_j]$ in the TFSM, we introduce two transitions by adding an extra state, called a *barrier state* b_{ij}, in between the two. The barrier state is needed to implement synchronous execution of the TAs in UPPAAL. We then introduce two transitions in the TA of the form $[EOT_i] \xrightarrow[x=x+d,lt=true]{\neg gtick} [b_{ij}]$ and $[b_{ij}] \xrightarrow[lt=false]{gtick} [EOT_j]$. The transition to the barrier node from EOT_i is taken when the global tick hasn't happened (this is the transition guard $\neg$ $gtick$). While taking this transition, the variable x is incremented by the cost of the transition d and the local tick corresponding to the thread (either $lt1$ or $lt2$) is set to true (this is the update part of the transition). Then the automaton reaches a barrier node and stays there until the global tick happens. See, for example, the transition from EOT_0 to b_{01} in the TA corresponding to the sampler thread. This transition happens when $gtick$ is false. During the transition, $lt1$ is set to true, indicating that the local tick for sampler is over and x is incremented by 54 to capture the cost of the transition. The generated TAs for the producer consumer example are shown in the Figure 4.

The task of ensuring that the barrier has been reached is handled by introducing a third automaton called the *barrier* as shown in the Figure 4(c). The barrier has just two states called *WaitLT* and *GTReached*. The barrier remains in the *WaitLT* state until both $lt1, lt2$ have been set to true by the two threads. In this case, both TAs would have taken their respective transition from an *EOT* state to their respective barrier states. The barrier, in response to both local tick variables being true, will set the global tick variable *gtick* to true and will wait in the state *GTReached*. The barrier resets back to the initial state only when both automata have completed their respective barrier transitions, in response to *gtick* becoming true. When this happens, they reset their respective local ticks to false again. Note that when this transition to the initial state is taken by the barrier, the value of the counter x is reset. *Thus, when the barrier is reached (the barrier is in the state GTReached) the value of x captures the cost of a global tick of the program.*

3.4 Taking Data Dependencies into Account

Synchronous programs such as Esterel have signal dependencies. WCRT analysis techniques must take these into consideration to eliminate redundant paths for tighter analysis [10, 16]. Such dependencies are quite easy to model in



(a) Sampler thread (b) Display thread (c) Barrier TA to implement synchronous parallel

Figure 4: Timed Automata (TA) model for the producer consumer example

our approach by just augmenting the transition of the automaton with additional guards (to capture signal test) and assignments (to capture signal emission). Once this is done, model checker will automatically remove redundant paths during the fixed point computation. While PRET-C has no signal dependencies, due to full C, PRET-C programs have data dependencies (variable test and set respectively). The same idea for dealing with signal dependencies is employed by to eliminate redundant paths during model checking.

3.5 WCRT as a CTL Property

We can compute the WCRT of the program by model checking a property of the form $AG(gtick \Rightarrow x \leq val)$, where the value of val is determined as follows. We already know the $WCRT_{max}$ of the program by summing up the maximum local tick value for every thread. Similarly, the minimum WCRT value, $WCRT_{min}$, may be obtained by adding the minimum local tick lengths for each thread. The tight WCRT value, $WCRT_{tight}$ lies between these two values i.e, $WCRT_{tight}$ lies in the interval $[WCRT_{min}, WCRT_{max}]$. For example, in the producer consumer case, the $WCRT_{min}$ $= 31 + 29$ and $WCRT_{max} = 54 + 64$. Hence, $WCRT_{tight}$ has a value in the interval $[60, 118]$. Thus, the value of val is also exactly the same interval. We can use standard binary search to minimize the number of queries. For example, to obtain the tight value for the producer consumer case, we have to write at most 6 queries ($log_2(58)$). In the producer consumer case, the tight value obtained by the above analysis is 101 in comparison to the maximum value of 118.

3.6 Complexity

The complexity of model checking TCTL properties over timed automata has been shown to be PSPACE-Complete. The same complexity also holds for TA with one or two clocks [11]. In our setting clocks are not at all needed, and the cost estimation of the global tick is done by simple increments to an integer variable called x. Hence, the complexity of model checking a single query ($AG(gtick \Rightarrow x \leq val)$) is $O(|val| \times |M| \times |\phi|)$ i.e, is the standard model checking complexity of CTL multiplied by number of possible valuations of the integer x. Note that the value of x ranges between $[WCRT_{min}, WCRT_{max}]$. Hence, the complexity of the proposed WCRT analysis is $O((WCRT_{max} - WCRT_{min}) \times |M| \times |\phi|)$ for checking a single query. Since, we will have $log_2(WCRT_{max} - WCRT_{min})$ queries in the worst case, the overall complexity is $\boldsymbol{O(log_2(WCRT_{max} - WCRT_{min}) \times (WCRT_{max} - WCRT_{min}) \times |M| \times |\phi|)}$.

Note that UPPAAL was used by us to rapidly prototype a solution to the WCRT analysis problem for synchronous programs for two reasons. Firstly, it employs aggressive state-space reduction and symbolic analysis techniques, which will be very useful to make the WCRT analysis technique scale (we present experimental evidence in Section 5). The second reason was that UPPAAL accepts XML based input language and allows both simulation and static deadlock checking that are useful validation aids. However, it may be noted that the WCRT analysis problem developed here is essentially a safety checking problem over synchronous automata with bounded integers. Hence, this analysis may be done using a custom tool in the future to make the analysis both optimal and also to prove that the proposed analysis indeed results in tight WCRT similar in spirit to a recent model checking work on bounded integers for SoC data-exchange protocols [20].

4. The ARPRET Architecture

The overall design philosophy of our PRET design may be summarized using the following three simple concepts:

1. Concurrency: Notion of concurrency is logical but notion of execution is sequential similar in spirit to [6]. This is used to ensure both synchronous execution and thread-safe shared memory communication.

2. Time: Notion of time is logical and the mapping of logical time to physical time is achieved by the compiler and the WCRT analyzer.

3. Design approach: ARPRET achieves PRET by simple customizations of general purpose processors (GPPs). The extensions to the C-language are minimal (notion of concurrency and logical time) and these are realized through C macros.

This section presents the hardware extension to a GPP Microblaze [23] in order to achieve temporal predictability. The basic setup of an ARPRET platform consists of a Microblaze soft-core processor that is connected to a hardware extension called Predictable Functional Unit (PFU) using two unidirectional First In First Out (FIFO) channels. The role of the PFU is to accelerate the scheduling of PRET-C threads using a hardware scheduler.

Microblaze [23] is a customizable RISC based soft-core processor, optimized for implementation on Xilinx FPGA. To maintain predictability its speculative features such as instruction and data caches were disabled. None of the features from the Memory Management Unit were used and no parallel shifters or floating point units were employed. We have opted for a five stage pipeline where the branch delay slot is disabled.

The PFU is somewhat similar in spirit to the Thread Control Block in the STARPro [24] processor that is designed for direct execution of Esterel. For each thread, a thread table stores the program counter and the thread status. Depending on the thread status, the scheduler issues the next program counter when requested.

Microblaze acts as the master by initiating thread creation, termination and suspension. The PFU stores the context of each thread in the thread table and monitors the progress of threads as they execute on the Microblaze. When a given thread completes an EOT instruction on the Microblaze, it sends appropriate control information to the Thread Control Block using an unidirectional FIFO. In response to this, the PFU sets the local tick bit for this thread to 1 and then invokes the scheduler. The scheduler then selects the next highest priority thread for execution by retrieving its program counter value from the thread table and sending it to Microblaze using FIFO. Moreover, when all participating threads have completed their local ticks, the PFU waits for the tick length to expire. Microblaze is blocked whenever it completes a local tick to wait for the next program counter value from the PFU. It also waits when all threads have completed their local ticks but the global tick hasn't happened. The tick length is decided by static WCRT analysis of a PRET-C program as detailed in Section 3.

Communication between Microblaze and any functional unit such as the PFU is done by using the Fast Simplex Link interface [23] provided by Xilinx. This communication is done by the use of hardware FIFOs. It closely couples Microblaze with the PFU using two FIFOs called FIFO_1 and FIFO_2 respectively to provide deterministic and predictable communication. Communication with FIFOs requires exchange of some common control signals such as the clock, reset, buffer status, read, write and also data such as the program counter value. More details of this architecture are provided in [3].

5. Results

In this section we present a set of experimental evaluation of the proposed approach. These include the evaluation of the performance of the model checking based WCRT computation as the number of concurrent processes grow. Following this, we present the results of WCRT analysis over a set of PRET-C programs where we compare the proposed static analysis results to the WCRT_{max} [5] value for the same program. Finally, we present the results of hardware synthesis for the ARPRET architecture.

We started our experiments by evaluating the choice of UPPAAL [1] as a model checker for concurrent automata with integer constraints but no clocks. We created an UPPAAL model consisting of two automata to start with, where each automaton has six states. We then replicated one of the automata to create additional processes from two to twenty-one processes. We wrote a single query for each experiment (two processes to twenty-one processes) that is the exact WCRT value for the program. This value was determined first by using the binary search method described in the paper.

Then, we ran the UPPAAL command called *memtime* to measure the execution time and the number of states explored for each of the experiments separately. During verification, we use the aggressive state-space reduction option. The actual state-space of each process is $2*(6^N)$, where N is any value from 2 to 21. The factor of 2 is due to the size of the barrier process. The result of this experiment is shown in Table 2. These results clearly show that both the execution time (in seconds) and the number of explored states have a linear growth while the input has an exponential growth. This may be due to several factors including symbolic model checking and on the fly generation of the reachable states.

N	Total No of States	No of States Explored	Execution Time
2	72	24	0.1
3	432	48	0.104
4	2592	96	0.101
5	15552	192	0.1
6	93312	384	0.1
7	559872	768	0.101
8	3359232	1536	0.1
9	20155392	3072	0.1
10	120932352	6144	0.1
11	725594112	12288	0.1
12	4353564672	24576	0.201
13	26121388032	49152	0.492
14	1.56728E+11	98304	1.005
15	9.4037E+11	196608	2.098
16	5.64222E+12	393216	4.543
17	3.38533E+13	786432	9.812
18	2.0312E+14	1572864	21.438
19	1.21872E+15	3145728	45.515
20	7.31232E+15	6291456	99.59
21	4.38739E+16	12582912	217.488

Table 2: The complexity growth of WCRT analysis

For comparing the proposed tight WCRT analysis with earlier approaches, we have developed a set of experiments by taking some examples from Estbench [8] and modelling them in PRET-C. We also have some new examples that we have created. These include the producer consumer example from Section 2.1, a standard concurrency example from [4] and the model for a robot performing obstacle avoidance through sonar senors. The `Robot Sonar` example was created to develop a real-time system using PRET-C. Table 3 presents the results of these experiments. The first column shows the name of the example in PRET-C, the remaining columns provide the number of threads, the WCRT$_{max}$, the WCRT$_{tight}$ values that are obtained through UPPAAL. On average the gain (%) from the tight analysis is about 10% greater than the maximum WCRT value.The `ABRO` example had the minimum percentage gain while the `Robot Sonar` example had the maximum gain. Note that the advantage of tighter analysis is very much dependent on the program concerned, the architecture used and which local ticks contribute to this maximum value. Hence, these results will be benchmark dependent. `ABRO` example, has WCRT$_{max}$ equal to WCRT$_{tight}$ since there are very few states in the concurrent threads and no data dependency between threads. In contrast, the `Channel Protocol` example has four threads and signal dependencies which facilitates model checking based optimization. As the weighting of the ticks differ and the amount of data dependency increases, there is more possibility of optimization. This will be illustrated through the experiment below.

Example	**Threads**	**WCRT$_{max}$**	UPPAAL	**Gain (%)**
ABRO	2	89	89	0
Channel Protocol	4	174	152	12.64
Reactor Control	3	121	118	2.47
Producer-Consumer	2	118	101	14.41
Smokers	4	531	449	15.44
Robot Sonar	4	419	346	17.42
Average				10.40

Table 3: Comparing WCRT$_{max}$ and the WCRT$_{tight}$ results obtained from model checking

The relationship between WCRT$_{max}$ and WCRT$_{tight}$ is illustrated using the experiment as shown in Table 4. We selected the `Smokers` program for this experiment. We then identified two local ticks, one in each thread, such that these two local ticks never participate in any global tick simultaneously. We then increased the computation cost of these two local ticks by a factor of N. This loading of these ticks ensured that they became the most significant contributors to the WCRT value of the program. Note that the WCRT$_{max}$ value was equal to the WCRT$_{tight}$ value when $N = 0$. As we kept on increasing N, both the maximum and the tight values grew linearly. However, as N increases, the gap between the two values also increases significantly as shown in the Figure 5. This happened since the WCRT$_{max}$ was proportional to $2 * N$ while the tight value is proportional to N.

This experiment illustrates that by changing the computation costs of some local ticks, the value of WCRT$_{tight}$ could be significantly smaller than that of WCRT$_{max}$. We intend to develop an editor where the programmer can dynamically place EOTs, thus changing the amount of computation between some local ticks. The effect of these changes could be observed to determine effective placement of EOTs.

Load (N)	**WCRT$_{max}$**	**WCRT$_{tight}$**	**Gain (%)**
0	211	211	0
10	483	346	21.00
20	763	486	36.31
30	1043	626	39.98
40	1325	766	42.19
50	1603	906	43.38

Table 4: Comparing WCRT$_{max}$ and the WCRT$_{tight}$ as the load varies for Smoker example

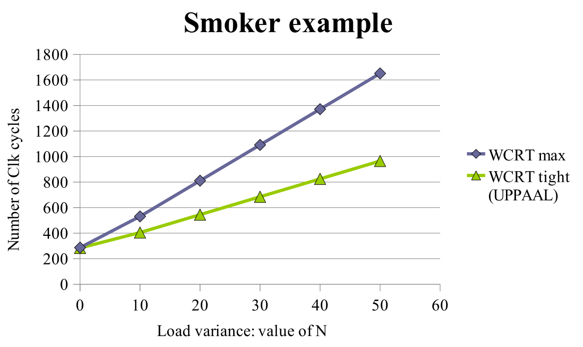


Figure 5: The effect of load variance on the WCRT value of the Smoker example

Finally, the effectiveness of the proposed analysis was determined by comparing the UPPAAL results with experimental results. Using the model checker we first identified the longest path. Suitable test vectors were developed to elicit this longest path during execution. Table 5 compares the actual execution time on the hardware with the WCRT value obtained from UPPAAL. On an average, the actual value is approximately 97% of the value obtained from UPPAAL.

Example	**UPPAAL**	**Measured WCRT**	**Tight (%)**
ABRO	89	87	97.75
Channel Protocol	152	149	98.03
Reactor Control	118	114	96.61
Producer-Consumer	101	99	98.02
Smokers	449	430	95.77
Robot Sonar	365	339	97.40
Average			97.26

Table 5: Comparing the actual worst case execution time and the WCRT value from UPPAAL

6. Conclusions and Future Work

Implementation of real-time systems on speculative processors relies on WCET analysis. However, WCET analysis of C programs on such processors remains a complex and often almost unsolvable task. To alleviate this dilemma, the Precision Timed Architectures (PRET) [9] have been proposed. The goal of PRET is to guarantee precise timing of execution while ensuring that overall throughput does not suffer. Another objective is to make WCET analysis simpler. In this paper, we have proposed an approach of designing PRET machines from general purpose processors (GPP) by simple customizations of the GPP. We have developed a new processor called ARPRET by customizing the Xilinx Microblaze soft-core processor. We also propose a set of simple synchronous extensions to the C-language, called PRET-C, to enable predictable programming of ARPRET like PRET processors. Using ARPRET and PRET-C, we then demonstrate that it is possible to derive the worst case reaction time (WCRT) of a PRET-C program by a simple transformation to facilitate model checking. By this process we have established that we can compute the tight WCRT of a range of PRET-C programs. The proposed method is shown to be about 10% savings on the WCRT value of these benchmarks compared to the maximum WCRT value computed by other researchers. Also, the results obtained through UPPAAL were about 97% tight compared to the measured tight values. To our knowledge, the proposed approach is the first tight WCRT analysis of synchronous C programs.

While the proposed approach is illustrated using PRET-C, it can be applied to any other synchronous language like Esterel. Also, while we have used ARPRET to obtain our results, it would be quite feasible to do similar analysis on any GPP as well. To do this, we have to take speculative features of the processor into account while calculating the cost of the nodes of a TCCFG using techniques similar to [10]. Other possible extensions include combining the proposed approach with orthogonal approaches such as [16] to achieve tight WCRT analysis of Esterel where both stable signal dependencies and transient dependencies are taken into account.

7. References

[1] UPPAAL tool. www.uppaal.com. last accessed on 29.4.09.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] S. Andalam, P. Roop, A. Girault, and C. Traulsen. PRET-C: A new language for programming precision timed architectures. Technical Report 6922, INRIA Grenoble Rhône-Alpes, http://www.ece.auckland.ac.nz/∼roop/pub/2009/andalam09.pdf, 2009.

[4] G. R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, 1991.

[5] M. Boldt, C. Traulsen, and R. von Hanxleden. Worst case reaction time analysis of concurrent reactive programs. *Electronic Notes in Theoretical Computer Science*, 203(4):65–79, June 2008.

[6] F. Boussinot. Reactive C: An extension of C to program reactive systems. In *SOFTWARE PRACTICE AND EXPERIENCE, VOL. 21(4), 401 428*, APRIL 1991.

[7] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.

[8] S. A. Edwards. Estbench Esterel benchmark suite. http://www1.cs.columbia.edu/ sedwards/software/estbench-1.0.tar.gz.

[9] S. A. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. In *Proceedings of the 44th DAC*, pages 264–265, June 2007.

[10] L. Ju, B. K. Huynh, A. Roychoudhury, and S. Chakraborty. Performance debugging of Esterel specifications. In *CODES+ISSS*, pages 173–178, 2008.

[11] F. Laroussinie, N. Markey, and P. Schnoebelen. Model checking timed automata with one or two clocks. In *CONCUR*, pages 387–401, 2004.

[12] L. Lavagno and E. Sentovich. ECL: A specification environment for system-level design. In *Proceedings of Design Automation Conference (DAC)*, New Orleans, June 1999.

[13] X. Li, M. Boldt, and R. von Hanxleden. Mapping Esterel onto a multi-threaded embedded processor. *SIGARCH Comput. Archit. News*, 34(5):303–314, 2006.

[14] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. In *In Proceedings of International Conference on Compilers, Architecture, and Synthesis from Embedded Systems (CASES)*, October 2008.

[15] G. Logothetis, K. Schneider, and C. Metzler. Generating formal models for real-time verification by exact low-level runtime analysis of synchronous programs. In *RTSS*, pages 256–264, Cancun, Mexico, 2003. IEEE Computer Society.

[16] M. Mendler, R. von Hanxleden, and C. Traulsen. WCRT algebra and interfaces for Esterel-style synchronous processing. In *Proceedings of the Design, Automation and Test in Europe (DATE'09)*, Nice, France, April 2009.

[17] A. Metzner. Why model checking can improve WCET analysis. In *CAV*, volume LNCS-3114, pages 334–347, 2004.

[18] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[19] P. S. Roop, Z. Salcic, and M. W. S. Dayaratne. Towards direct execution of Esterel programs on reactive processors. In *4th ACM EMSOFT*, 2004.

[20] R. Sinha, P. S. Roop, S. Basu, and Z. Salcic. Multiclock SoC design using protocol conversion. In *Design Automation and Test in Europe (DATE)*, 2009.

[21] F. Vahid and T. Givargis. *Embedded System Design.* John Wiley and Sons, 2002.

[22] 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. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

[23] Xilinx. *Microblaze Processor Reference Guide*, 2008.

[24] S. Yuan, L. H. Yoong, S. Andalam, P. S. Roop, and Z. Salcic. A new multithreaded architecture supporting direct execution of Esterel. *EURASIP Journal on Embedded Systems*, 2009:Article ID 610891, 19 pages, 2009. doi:10.1155/2009/610891.