Worst-Case Reaction Time analysis of GALS Programs

Authors Name/s per 1st Affiliation (*Author*)

line 1 (of *Affiliation*): dept. name of organization

line 2-name of organization, acronyms acceptable

line 3-City, Country

line 4-e-mail address if desired

Authors Name/s per 2nd Affiliation (*Author*)

line 1 (of *Affiliation*): dept. name of organization

line 2-name of organization, acronyms acceptable

line 3-City, Country

line 4-e-mail address if desired

*Abstract*—

Keywords—

# Introduction

Real-time systems consist of a number of input and output devices connected to one or more computer systems. A real-time program reads inputs from these devices, processes these inputs, and produces outputs to control these devices. Two characteristics distinguish a real-time program from other programs: (1) the real-time program should not miss any external input event, and (2) the response time from the arrival of the input to the output should be bounded [[Lehoczky et al. 1989](#_ENREF_13" \o "Lehoczky, 1989 #278); [Mok 1983](#_ENREF_16" \o "Mok, 1983 #277)]. The standard approach [Liu and Layland 1970] to *guaranteeing* these real-time properties requires determining a scheduling approach, e.g., earliest deadline first, primarily dependent on the *Worst Case Execution Time* (WCET) of the tasks in the real-time program using static code analysis [The worst-case execution time problem, TECS]. Once the scheduling technique is determined, a real-time operating system (RTOS) is employed to achieve the objectives. There are a number of problems with this approach; first of all, since the program is usually written in general purpose languages like ‘C’, communication and synchronization between tasks is achieved via low level constructs such as mutexes, which makes guaranteeing functional correctness, let alone timing correctness, hard [E.lee problem with threads]. Furthermore, general purpose languages executing on general purpose processors built with speculative units such as branch predictors, complex memory hierarchy, etc, makes estimating the WCET of tasks very challenging [TECS paper again] as all possible execution paths need to be explored.

Synchronous languages such as Esterel [berry] and its many variants such as ForeC [ref], PretC [ref], etc running on time predictable processors such as PretARM [ref] ameliorate this problem to a great extent and hence, have been proposed as a viable alternative in research literature. These synchronous programming techniques calculate the so called *Worst Case Reaction Time* (WCRT) rather than WCET of tasks, being based on a discrete global logical clock model. The incoming input events are latched at the start of the global clock tick, a *reaction function* is called to process these inputs and finally the outputs are generated at the end of this clock tick. The WCRT is the maximum execution time amongst all these global clock ticks. As long as the WCRT of a synchronous program is less than the inter arrival time of any input event from the environment the program is guaranteed to not miss any input event. This global clock model is a double edged sword. On the one hand determining *tight* (close to optimal) WCRT bounds is possible due to the global clock model, but on the other hand a synchronous program cannot be easily distributed on a multi-core or a multi-processor distributed memory system, since all tasks are very tightly coupled thanks to the synchronization required with the global clock tick. Furthermore, all synchronous programs make an implicit assumption, that all output event(s) are generated instantaneously after the reaction function. This might not be the case at all. If an output event is generated after multiple global ticks and their associated reactions, the second characteristic of a real-time system, bounded response times, cannot be guaranteed by the WCRT analysis. In this paper we propose the WCRT analysis of *Globally Asynchronous Locally Synchronous* (GALS) programs. The GALS model of computation (MoC) enhances the synchronous model with asynchrony and is perfectly suited for design of multi-core and distributed memory real-time systems. Introduction of asynchrony allows exploiting multi-core and multi-processor systems, but makes WCRT analysis challenging. In fact, WCRT or WCET of a GALS program is not yet defined in literature. In this paper our main contributions are:

1. We give a precise definition of worst case execution/reaction time in a GALS setting.
2. We propose a new framework for static timing analysis for GALS systems including micro-architectural analysis of a time predictable execution platform.
3. Our approach uses model-checking tools Our approach produces tight (as good as the best proposed timing analysis techniques in literature) timing estimates.
4. Finally, we stick to the *What You Prove Is What You Execute* (WYPIWYE) [berry’ref] paradigm proposed by G. Berry. In that, our timing analysis is performed directly on the intermediate format produced by the compiler. We also use the result of the timing analysis for code optimization leading to large savings in memory footprint in the generated object code size.

# Related Work

Plethora of WCRT analysis techniques have been proposed for synchronous programs, which can be classified as Max-Plus algebra [23], model checking [24, 25, 26], reachability [27], and Integer Linear Programming (ILP) [28, 29, 30]. In Max-Plus algebra, the maximum WCRT of each thread is summed up to obtain total WCRT. This technique has polynomial complexity but the WCRT is usually overestimated [23], i.e., it is far from the optimal. In Taxys [24], a timed model of an Esterel Program is extracted and timed automaton with real-valued clocks is used. Therefore, this approach has exponential complexity. The use of model checking for timing analysis of real-time systems is not new [TODO]. As an extension, Model checking based WCRT analysis for synchronous programs has been proposed in [TODO]. The model checking based approach in [25] takes the state-dependencies between threads into account for tighter analysis. Another model checking based approach in [26] has the ability for more complex infeasible path pruning than the ILP approach, and can often provide tighter WCRT estimates. An ILP based formulation has been presented in [28]. The synchronous program is first compiled into a sequential C program; ILP constraints are then derived to compute the WCRT. In [29] the ILP formulation was further refined to prune redundant states by using an additional automaton. The approach in [27] avoids the binary search, which is required in [25] by computing the execution time of all reachable ticks and recording the largest computed time as the WCRT. In [30], a WCRT computation approach for Esterel programs executing on multiprocessor platforms is proposed. This approach pre-computes feasible control states and removes infeasible execution paths at compile time. The time complexity for each approach is listed in TABLE I.

1. Comparison of timing analysis techniques for synchronous programs

|  |  |  |
| --- | --- | --- |
| Approach | Technique | Time Complexity |
| Boldt et al. [23] | Max-Plus | Square of node number |
| Bertin et al. [24] | Model checking | Exponential |
| Roop et al. [25-26] | Model checking | Product of thread sizes  × binary search |
| Kuo et al. [27] | Reachability | Product of thread sizes |
| Ju et al. [28-30] | ILP | NP-hard |

All the aforementioned approaches are developed for synchronous program and single-core execution platform, except for [30] which handles multi-core case by sequentializing the program into each core and using ILP to compute the WCRT. The feasibility of WCRT analysis critically depends on the hardware execution platform. The approach presented in [23] is targeting Kiel Esterel Processor (KEP) [39] for Esterel language, while approaches in [25-27] are targeting Microblaze processor running ForeC program. Analysis in [28, 29] are based on single-core general purpose processor. The approach in [30] is aiming at general-purpose multiprocessors. As compared to all the others, our approach is targeted at timing analysis of GALS programs along with code optimization using a model checking approach.

# background – the systemj language

SystemJ is a novel system-level design language based on *Globally Asynchronous Locally Synchronous* (GALS) Model of Computation (MoC) [13, 14]. SystemJ extends the Java language by introducing both synchronous and asynchronous concurrency [22]. SystemJ integrates the synchronous essence of Esterel language [15] and the asynchronous concurrency of CSP [16]. SystemJ is grounded on delicate and rigorous mathematical semantics and thus is amenable to formal verification [14].

On the top level, a SystemJ program comprises a set of clock-domains executing asynchronously. , Reactions can themselves be composed of more synchronous parallel reactions, thereby forming a hierarchy. Finally, everyreaction in the clock-domaininput and output At the beginning of each clock-domain tick the input signal are captured, a reaction function processes these input signals and emits outputs signals. These output signals are presented to the environment at the end of the clock-domain tick. Inter clock-domain communication is achieved by implementing CSP style rendezvous mechanism through point-to-point unidirectional channels.

A complete list of SystemJ kernel statements is shown in TABLE II.

1. SystemJ kernel statements

|  |  |
| --- | --- |
| **Statement** | **Description** |
| p1;p2 | p1 and p2 in sequence |
| pause | Consumes a logical instant of time (a tick boundary) |
| [input] [output] [type] signal S | Declaring a pure or valued signal |
| emit S [(exp)] | Emitting a signal with an optional value |
| while(true) p | Temporal loop |
| present(S){p1}else{p2} | If signal S is present do p1 else do p2 |
| [weak] abort ([immediate] S) {p} | Preempt if S is present |
| [weak] suspend ([immediate] S) {p} | Suspend for 1 tick if S is present |
| trap (T) {p} | Software exception |
| exit T | Throw a software execption |
| p1 || p2 | Run p1 and p2 in lock-step parallel |
| [input] [output] [type] channel C | Declaring input or output channel |
| send C[(exp)] | Sending data over the channel |
| receive C | Receiving data over the channel |
| #C | Retrieving data from a valued signal or channel |

In SystemJ, Inter clock-domain communications through channels are implemented by the send and receive statements. Inside a clock-domain, fork-join parallelism of reactions is captured by the synchronous parallel operator (||). All the parallel reactions forked out by the || operator are executed in lock-steps with clock-domain’s tick. Sequential SystemJ statements are separated by a semicolon ; within a reaction. The tick boundaries of a clock-domain are marked by the pause statements, which explicitly specify the end of the current tick and also the beginning of the next tick. Similar to Esterel, there are two types of signals in SystemJ, pure signals containing only a Boolean status and valued signals consisting of both a status and a value. The synchronous broadcast mechanism on signals is implemented by the emit statement which makes a signal present by setting its status to true and updating its value for a valued signal. Control-flow can be described using statements such as present, abort and suspend depending on the status of signals. Note that SystemJ uses delayed semantics, i.e., all aforementioned control-flow constructs check the status of the signals from the previous tick, not the current instant. A temporal loop in SystemJ must contain at least one pause statement inside its body and is constructed using the while statement. The trap and exit statements are used to implement software exception.



# motivating example and the problem definition

We use the motivating example of a Fruit Sorter to describe the semantics of the SystemJ language, as shown in Fig. 1. The Fruit Sorter combines several sensors and two mechatronic components, a Conveyor and a Mechanical Arm (Arm), respectively, to form a simple system for sorting different types of fruits. Fruit items are loaded at the loading end (BEG) of the Conveyor and get sorted into corresponding bins when they reach the sorting end (END) of the Conveyor.



1. Fruit Sorter Example with SystemJ Program Structure

The SystemJ program describing Fruit Sorter is shown in Fig. 2. There are four reactions enclosed in ConveyorController CD, namely R1-4 as shown in Fig. 2 where R3 and R4 are child reactions forked by parent reaction R2. Reaction R1 continuously monitors the current item loads of the Conveyor and controls the movement of Conveyor and item loader. Reaction R2 waits until an item is ready to be picked and recognizes the type of the item by invoking the image recognition procedure. There are two reactions (R5, R6) in ArmController CD which control the sorting operation of Arm based on item type and display the corresponding item type on LCD. ArmController CD sends back ITEM\_PICKED signal through another channel upon the completion of a sorting operation. Note that in Reaction R2 the operation of the image recognition module is performed in separate synchronous parallel reactions (R3 and R4) so that their executions can be parallelized, and pipe-lined for efficiency. Reaction R2 can be duplicated to achieve image recognition using multiple cameras for better precision and speed.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39**  **40**  **41**  **42**  **43**  **44**  **45**  **46**  **47**  **48**  **49**  **50**  **51**  **52**  **53**  **54**  **55**  **56**  **57** | import camera.\*;  ItemRecognition( //ItemRecognition CD  output String channel ITEM\_TYPE;  input signal NEW\_ITEM;  )->{//R1  Image signal picture; String signal itemtype;  while(true) {  //wait until new item arrives  await (NEW\_ITEM);  {//R2: take a picture of item using Camera  emit picture(Camera.takepicture());  }  ||  {//R3: recognize and emit image type  await (picture);  emit itemtype(Process\_Image.get\_item\_type(#picture));  }  // wait for item type and send it through channel  await(itemtype);  send ITEM\_TYPE(#itemtype);  }  }  ArmController( //ArmController CD  input String channel ITEM\_TYPE;  input signal ITEM\_READY;  input signal REACHED\_LEFT, REACHED\_RIGHT, REACHED\_HOME;  output signal PICK\_TO\_LEFT, PICK\_TO\_RIGHT, MOVE\_HOME;  )->{  {//R4: perform item sorting  while(true){  receive ITEM\_TYPE;  await(ITEM\_READY);  if (#ITEM\_TYPE.equals("apple"))  {//put item into apple bin  emit PICK\_TO\_LEFT;  await(REACHED\_LEFT);  emit MOVE\_HOME;  } else {  //put item into pear bin  emit PICK\_TO\_RIGHT;  await(REACHED\_RIGHT);  emit MOVE\_HOME;  }  // wait until arm is returned to default position  await(REACHED\_HOME);  pause;  }  }  ||  {//R5: display sorted item count using LCD  while(true){  present(REACHED\_LEFT) {LCD.display\_apple\_count();}  present(REACHED\_RIGHT) {LCD.display\_pear\_count();}  pause;  }  }  } |

1. Fruit Sorter described in SystemJ

In SystemJ, all the input events (signals) are captured from the environment and channels at the beginning of each tick. Then each reaction of the clock-domain is executed synchronously in lock-steps with a global logic clock named tick, and during the execution all the output events (signals) are either emitted or sent through the channels. The output signals emitted in the current tick to the environment or other reactions become available at the beginning of next tick and last for that tick. The output signals sent through channels to other clock-domains become available to the receiving clock-domain after a certain delay of channel communication. For instance, a segment of execution trace for ConveyorController CD is illustrated in Fig. 3. All the input and output events are shown during the execution time of eight consecutive logic ticks starting from a random time (tick N). The ITEM\_ READYTOPICK signal captured at tick N+1 indicate that a fruit item is ready to be sorted, and this signal is present for every tick until the item is picked away from the conveyor. The image of the fruit is taken upon the presence of ITEM\_READYTOPICK signal, followed by the start of image recognition procedure. The STOP\_LOADER and STOP\_ CONVEYOR signal is emitted to the environment from Tick N+2 to N+7 because the maximum loads capacity of the conveyor is reached and the sorting of the fruit item is in progress during those ticks. The signal of item type is generated then and sent to ArmController CD through output channel ITEM\_TYPE at tick N+4. The input signal captured from ITEM\_PICKED channel means that the item has been picked by the Arm and sorted properly. Hence the STOP\_LOADING and STOP\_CONVEYOR signals are cleared and MOVE\_CONVEYOR signal become available at tick N+8.



*Execution Trace for ItemRecognition CD*



*Exectution Trace for ArmController CD*

1. A Segment of Execution Trace for Fruit Sorter Example

Effects of Channel Communication on WCTT

SystemJ exploits the same channel communication style as shown in Fig. 5(b), therefore the CSP style rendezvous is compartmentalized into separate logic ticks [TODO]. This sound implementation fundamentally enables WCTT analysis of a clock domain to be independent from inter-clock-domain communications, thus making the WCRT analysis of GALS system tractable.



*Timing diagram for complete blocking style rendezvous*



*Timing diagram for partial blocking style rendezvous*

1. Illustration of effects on WCTT for different rendezvous styles

# static timing analysis for gals system

## WCRT Analysis Framework



1. WCRT Analysis Framework
2. Conversion from AGRC to CDUM

## The timing analyzable execution platform

SystemJ language extends Java with synchronous and asynchronous concurrency; hence SystemJ programs need to be compiled before running on target hardware platforms. The whole compilation procedure can be divided into front-end stage and back-end stage. After some internal steps in the front-end compilation, SystemJ program is translated into an intermediate format named Asynchronous GRaph Code (AGRC) which captures the formal semantics of SystemJ language and guarantees equivalence between the designed system and its implementation [14, 22]. AGRC representation of SystemJ program is then sent to back-end compilation for code generation. The platform-dependent information is used by compiler back-end to generate the final back-end code, which can be directly executed on different hardware platforms.

Since SystemJ programs integrate both control-driven and data-driven operations, which are tightly coupled, implementing both types of operations in a single control processor is inefficient and results in bulky target code size and slow execution speed [32]. However, if we can separate control-driven and data-driven operations and execute them separately, not only can we greatly improve the execution efficiency, but the intricacy of control flow analysis could be substantially simplified. The separation of control-driven and data-driven operations for SystemJ programs has emerged in [32-34]. The main idea is to separate control-driven and data-driven operations at the back-end compilation stage and then mapping them onto two processing units, namely Control Processor (CP) and Data Processor (DP), respectively. Both processors work in tandem, hence the combination of CP and DP is also called the tandem processor [33].

The control flow representing control-driven operations is named Concurrency and Reactivity Control Flow (CRCF) [36]. Besides control flow of the reactions, CRCF also includes scheduling of all reactions and clock domains, communication between reactions and clock domains, and communication with the environment. CRCF code is executed on the CP. On the other hand, the control flow of data-driven operations is called Java Control Flow (JCF), since all data computations of SystemJ programs are compiled to Java bytecodes running on the DP.

With this separation, SystemJ program execution is initiated and directed by the CRCF; whenever a Java data computation is required the execution is switched to JCF and returns to CRCF again upon completion.

### TP-JOP Multi-core Execution Platform

Inspired by the separation of SystemJ control/data flow, several SystemJ execution platforms have been proposed [32-37]. Among these solutions we choose a multi-core platform named Tandem Processor – Java Optimized Processor (TP-JOP) as our target execution platform. TP-JOP execution platform uses Reactivity and COncurrency Processor (ReCOP) [37] as CP and Java Optimized Processor (JOP) [38] as DP. TP-JOP is based on the idea of Tandem Processor (TP) proposed in [33] with different implementation for DP. Both ReCOP and JOP are designed with time-predictable features, which make WCTT analysis of SystemJ programs feasible on TP-JOP. The block diagram of TP-JOP is shown in I.

### Time-Predictable Features of TP-JOP

The reason for choosing TP-JOP as the target platform of SystemJ WCTT analysis is not only because it matches the separation of control flow into CRCF and JCF, but also in respect that the two processors composing TP-JOP, respectively JOP and ReCOP, are provided with time-predictable features and hence are highly suitable for timing analysis.

JOP is an open-sourced hardware (soft-core) implementation of the Java Virtual Machine (JVM) targeting embedded real-time systems, it is a low cost processor in terms of gate count and power consumption, and it is also highly customizable and extendable [38]. JOP is much more efficient in running Java programs compared to software implementations of the JVM. JOP is a time-predictable processor, which is ideally suitable for timing analysis, because of its unique architecture for time-predictable execution of Java bytecodes and Java methods. The time-predictable features of JOP include: (1) WCET of each bytecode can be predicted in terms of the number of JOP clock cycles. (2) The pipeline of JOP core is designed without prefetching and queuing in order to eliminate pipeline dependencies that are hard to analyze. In addition, the pipeline stalls, which can be caused by interrupts or memory subsystem, are also avoided to simplify the timing analysis. (3) JOP contains two time-predictable caches: a stack cache, which acts as a substitute for the data cache and a method cache, which acts as a substitute for the program cache available in standard processors.



1. Block diagram of TP-JOP execution platform

A WCET analysis tool called JOP WCET Analyzer (WCA) is provided with JOP. JOP WCA exploits all the time-predictable features of JOP and performs static WCET analysis based on Implicit Path Enumeration (IPET) approach [38]. With the help of data flow analysis which detects loop bounds, WCA calculates the WCET estimates of a Java program which runs on JOP. WCA also integrates the method cache analysis to generate more accurate results.

ReCOP is designed exclusively to deal with the execution of CRCF of SystemJ programs. It also handles the communication with DP (namely JOP) and with the environment. ReCOP executes the CRCF assembly code generated by the back-end of SystemJ compiler. The instructions are carried out as sequences of micro-operations. Every instruction is executed in exactly three ReCOP cycles. The three execution cycles of ReCOP instructions are instruction fetching cycle, decoding cycle and execution cycle.

### Execution Pattern on TP-JOP

There are two types of execution pattern on TP-JOP execution platform, interleaving and sequential execution pattern, respectively. The original TP-JOP with the FIFO buffer features an interleaving execution pattern, whereby the control on ReCOP and data processing on JOP can proceed in tandem. In order to facilitate the WCTT timing analysis, we simplify the execution model by removing FIFO and forcing two processors to execute exclusively and sequentially. Sequential execution pattern indicates that there is no interleaving between the execution of ReCOP and JOP. ReCOP perform the initialization at the beginning, followed by the execution of control-driven operations. Whenever a data computation request is encountered, the control of execution pattern is transferred to JOP. ReCOP halts until JOP finishes performing Java data computation and return the results. In interleaving execution pattern, ReCOP does not halt when JOP is performing Java computations. Instead, the execution of ReCOP jumps to the CRCF code of other reactions to execute them until there is no reaction that can be executed further without the results of data computation from JOP.



In this paper we proposed a complete WCRT analysis framework for GALS system with code optimization capability. An overview of the framework is shown in Fig. 6. SystemJ program source code is firstly compiled to AGRC intermediate format using the front-end of SystemJ compiler presented in [TODO] (Step 1). At this step, the generated AGRC does not contain any timing information. Then SystemJ compiler back-end generates target code for TP-JOP execution platform for both CRCF code executed on ReCOP and JCF code executed on JOP (Step 2). The tracing information is also kept at Step 2 in order to trace the origin AGRC node of the target code. Both CRCF assembly code and JCF Java code is analyzed by our customized WCET analyzer to calculate WCET information for each AGRC node (Step 3). The customized WCET Analyzer used in Step 3 comprises of JOP Worst-Case Analyzer [TODO] and ReCOP WCET Analyzer, the latter performs WCET analysis by counting the number of assembly instructions for an AGRC node and deriving the WCET result based on the Micro-architecture level analysis of ReCOP. The original AGRC is then back-annotated with the node-level WCET information acquired from Step 3. Afterwards the back-annotated AGRC is partitioned into a set of graph code for each clock-domain which is named Clock-Domain GRaph Code (CDGRC). All CDGRCs are later converted to equivalent Timed Automaton (TA). Instead of using any clocks we use a single bounded integer to capture the total time cost of a complete transition path from the beginning to the end of a logic tick. Then we model the tight WCTT analysis problem as verifying a CTL property in UPPAAL model checker (Step 5), as detailed in Section V-C. The WCTT results for all the clock domains acquired from Step 5 are provided to clock-domain scheduler to generate a proper scheduling targeting a specific scheduling objective. The final WCRT result can only be analyzed with all the WCTT information together with the clock-domain scheduling (Step 6). However, clock-domain scheduling is out of the scope of this paper. Thus Step 6 is not discussed in detailed here except for a trivial case where a cyclic scheduling of clock domains is applied. Furthermore, we perform an optimization for the size of generated target code size in Step 7. We achieve reduced target code size by pruning the infeasible branches found in Step 5 from the original AGRC and repeating Step 2 based on the optimized AGRC. Step 7 is discussed in details in Section D. Note that in our proposed framework, all the time costs are measured in clock cycles of ReCOP and JOP, and we assume they are both running at the same clock frequency, for the sake of simplicity.



1. Experimental restuls

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Testbench** | **LOC** | **AGRC node numbers** | **Max-plus** | **State encoding only** | **States stored** | **States explored** | **Execution time(s)** | **Tracking all signals** | **States stored** | **States explored** | **Execution time(s)** |
| abrodata |  | 49 | 320 | 320 | 118 | 111 | 0.8 | 320 | 118 | 111 | 0.8 |
| abro |  | 50 | 342 | 342 | 119 | 111 | 0.8 | 342 | 119 | 111 | 0.8 |
| ex1 |  | 54 | 318 | 309 | 54 | 52 | 0.8 | 309 | 54 | 52 | 0.8 |
| ex2 |  | 77 | 544 | 544 | 182 | 173 | 0.9 | 511 | 320 | 320 | 1 |
| pcabro cd2 |  | 96 | 440 | 440 | 286 | 261 | 0.8 | 440 | 290 | 262 | 0.8 |
| ex3 |  | 105 | 636 | 582 | 183 | 180 | 0.9 | 555 | 89 | 87 | 0.9 |
| demoloop |  | 122 | 612 | 612 | 3864 | 3850 | 0.9 | 568 | 406 | 400 | 0.9 |
| asproto cd1 |  | 171 | 423 | 423 | 279 | 274 | 0.8 | 423 | 279 | 274 | 0.8 |
| pcabro cd1 |  | 172 | 653 | 653 | 12929 | 12910 | 0.9 | 649 | 14751 | 14751 | 1 |
| freq cd1 |  | 206 | 890 | 890 | 28716 | 28307 | 1.8 | 617 | 581 | 580 | 1 |
| sorter 1pic |  | 210 | 773 | 773 | 4884 | 4753 | 0.9 | 706 | 2093 | 2055 | 1 |
| sorter2pics |  | 241 | 1262 | 1230 | 1278556 | 1261188 | 38.5 | 1152 | 12293 | 12072 | 1.1 |
| asprotocd2 |  | 267 | 1106 | 1106 | 303608 | 298175 | 7 | 849 | 12797 | 12797 | 1.1 |
| pcabro cd3 |  | 305 | 948 | 948 | 63593 | 63537 | 2.7 | 586 | 124 | 116 | 0.9 |

## Model Checking Based Tight WCTT Analysis

In our proposed WCRT analysis framework, a model checking based analysis approach is exploited in order to find the tight WCTT for each clock-domain of a GALS system, as shown in Step 4 and 5 in Fig. 6. The back-annotated AGRC is partitioned to individual CDGRCs which are then converted into a set of equivalent Clock-Domain Timed Automaton (CDTA), which is carried out at Step 4 in Fig. 6. CDTA has no clocks but utilizes a single integer variable to model the time cost of transitions. We use a well-known model checker for timed automata called UPPAAL [43] to model our CDTA since it not only offers efficient algorithms for both TA and automata with integer operations but also provides remarkable user interface. The CDTA modeled in UPPAAL model checker is named Clock-Domain UPPAAL Model (CDUM). Nonetheless, our proposed approach is applicable to any other model checker with support for bounded integer counters.

An example of conversion from CDGRC to CDUM is illustrated in Fig. 7. The graph code shown in Fig. 7(a) is the CDGRC for the ArmController CD of Fruit Sorter Example described in Section IV. This CDGRC is obtained by partitioning the back-annotated AGRC. The corresponding CDUM converted from the CDGRC is shown in Fig. 7(b). To convert CDGRC to CDUM, we firstly perform one-to-one mapping for each node in the CDGRC to a location in the CDUM. For example, *Afork Node* AF0 in the CDGRC is mapped to the initial location AF0 in CDUM. Similarly *Switch Node* S1 is mapped to a location S1, and so on. After that we map each control flow dependency in the CDGRC to a transition between two locations in the CDUM. The conditional branching is modeled by augmenting the transitions with additional guards. For instance, the transition from location S1 to E3 in Fig. 7(b) has the expression S15==26 as its guard to capture the conditional branching semantics based on state decoding of Switch Node S1 in Fig. 7(a). The signal emission and state encoding process are captured by the assignments of the transition. In Fig. 7(b), assignments of transitions from A15 and A16 model the emissions of signal A\_L and A\_R, respectively; while assignments of transitions from E3, E12 and E13 model three distinct state encoding. As described in Section V-B, a bounded integer (*wctt*) is used to count the total execution time within a tick. The bounded integer wctt is incremented by the WCET cost of each node in CDGRC by augmenting every transition targeting the corresponding location in CDUM with a self-increment assignment.

Due to the semantics of AGRC nodes related with control flow, modifications should be done to the CDUM to model the same execution flow. Because the parallelism of the branches forked out by a *Fork Node* is compiled away in a way that every branch of *Fork Node* should be executed one after another. Thus the *Join Node* should have a transition to its corresponding *Fork Node* to model the semantics of Fork and Join Node. Furthermore, since a tick starts at *Afork Node* and finishes at *Ajoin Node*, there should be a transition from *Ajoin Node* back to *Afork Node* to model the repetitive execution of ticks. More importantly, house-keeping procedure described in Section III should be modeled by assignments of transition from *Ajoin Node*. This bounded integer should also be cleared to zero during the transition from *Ajoin Node* to *Afork Node* (from AJ5 to AF0)

The tight WCTT, named *WCTTtight*, is the objective of WCTT analysis. *WCTTtight* lies in the bounded range denoted by [*WCTTmin, WCTTmax*]. WCTTmax is a safe upper-bound of WCTT obtained by applying Max-Plus algebra to the CDGRC [23], which is essentially summing up the maximum tick execution time of every reaction in the clock-domain. Similarly, we calculate a lower-bound of WCTT, termed *WCTTmin*, by summing up the minimum tick execution time of every reaction in the clock-domain. After the construction of CDUM, we can check the validity of a WCTT estimate of the clock-domain, termed *WCTTest*, by verifying a CTL property upon the CDUM using UPPAAL model checker. The CTL property is written as *A[](wctt ≤ WCTTest).* In order to minimize the number of queries, we use standard binary search algorithm to find WCTTtight. In each iteration of binary search, we select the mid-value of the search range as *WCTTest* and perform verification of CTL property. For the CDUM example shown in Fig. 7(b), we have to write TODO queries in order to obtain the *WCTTtight.* The final result of *WCTTtight* is TODO in comparison with *WCTTmax* value of TODO .

//TODO should we add another sub-section here to describe how to assemble WCTTs into WCRT of a GALS system?

## Code Optimization through Infeasible Branch Pruning

In this subsection we describe the techniques we use to perform code optimization as described in Step 7 in Fig. 6 through infeasible branch pruning. Due to the semantics of AGRC nodes, infeasible branch only exists in the branches forked out by either Switch Node or Test Node [22], because only these two types of node perform conditional branching. In order to identify the infeasible branches, we attach a unique flag on every transition forked out by a location which corresponds to either Switch Node or Test Node in CDGRC, we term this type of locations as *branching locations*. Specifically, we argument a unique flag (boolean variable) assignment to every transition fired from a branching location. We assign those flags which is initialized as true to false upon transitions, thus a flag will become false only if the corresponding transition take place. Then we identify infeasible branches by verifying the following CTL property for every flag, *A[](flag = true).*A branch is considered to be infeasible if the corresponding query is not satisfied.

For example, in CDUM shown in Fig. 7(b) the transition from location S1 to E1 is augmented with a boolean variable assignment *S1\_E1=false*, where *S1\_B1* is an infeasible branch flag initialized as true. We augment an assignment of a unique boolean variable to every transition originated from a branching location. Then we check whether the branch started by transition from S1 to E1 is infeasible by verifying a CTL property *A[](S1\_E1 = true).* This branch is actually feasible in this example because this property is not satisfied. We collect the infeasible paths after verifying all the queries related with infeasible path flags. Then we remove the corresponding branches in the AGRC and repeat the back-end code generation step in order to generate optimized target code.

## Complexity Analysis

The value of WCTT tracing variable *wctt* is bounded by the range [*WCTTmin, WCTTmax*]. Therefore, the complexity of our proposed WCTT analysis is *O((WCTTmax − WCTTmin ) × |M| × |φ|)* for checking a single query, where *M* is the CDTA and *φ* is the complexity of the query. The complexity of standard binary search used to find *WCTTtight* determines there will be *log2(WCTTmax−WCTTmin )* queries in the worst case, thus the overall complexity of our proposed WCTT analysis method is given by *O(log2(WCTTmax−WCTTmin ) × (WCTTmax − WCTTmin ) × |M| × |φ|).* Comparing with earlier model checking based approach of finding WCRT for synchronous programs, our approach excels in the complexity of *|M|.* Due to the elimination of barrier synchronization method used in [40-42] and the integration of reaction automata in CDTA, the complexity induced by parallel automata is mitigated. Instead of creating one automaton for each reaction in the clock domain plus a barrier automaton for synchronization globally, there is only one automaton which is actually CDTA in our proposed approach. Therefore the overall complexity of *|M|* is reduced.

# results

In this section, we present a set of experimental evaluations of our proposed WCRT analysis framework. We implement the entire framework including code optimization part in Java language. The performance of WCRT analysis is evaluated over a set of GALS system benchmarks described in SystemJ. These benchmarks belong to the class of real-time GALS system which model realistic applications, including The Fruit Sorter example described in Section IV. We extend the sorter example by introducing more cameras in ConveyorController CD to increase the speed of fruit item type recognition, which will effective increase the number of reactions and the complexity of the example. The number of clock domain in each benchmark ranges from 1 to TODO. As described in Section V-B, to evaluate the performance of proposed model checking based WCTT analysis approach, each GALS system benchmark is divided into individual clock domains. The performance WCTT analysis is evaluated as the number of reactions grows. There are two synthetic benchmarks with single clock domain containing large number of reactions to serve as extreme cases. The results of these experiments are presented in TABLE III. We implement the Max-Plus algebra specified in [23] to calculate WCTTmax and WCTTmin. WCTTtight is obtained by running UPPAAL model checker.

The complexity increased by tracking different numbers of signals when performing model checking based WCTT analysis is shown in TABLE IV.

The performance of code optimization based on WCTT analysis is shown in TABLE V.

//TODO results are testbench dependent.

//Delete all ConveyorController

# Conclutions

##### References

1. J. Stankovic and K. Ramamritham, editors. Tutorial on Hard Real-Time Systems. IEEE Computer Society Press, 1988.
2. Knight, J.C., 2002. Safety critical systems: Challenges and directions. Proceeding of the 24th International Conference on Software Engineering, May 19-25, IEEE Xplore Press, Orlando, Florida, pp: 547-550.
3. Nicolas Halbwachs. Synchronous Programming of Reactivce Systems.
4. Compiling ESTEREL
5. A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. Proceedings of the IEEE, 79(9):1270–1282, 1991.
6. The synchronous languages 12 years later
7. G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming, 19(2):87–152, 1992.
8. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. Proceedings of the IEEE, 79(9):1305–1319, 1991.
9. P. L. Guernic, T. Gauthier, M. L. Borgne, and C. L. Maire. Programming real-time applications with signal. Proceedings of the IEEE, 79(9), 1991.
10. Leveraging Synchronous Language Principles for Heterogeneous Modeling and Design of Embedded Systems
11. Keating M, Bricaud P. Reuse methodology manual: for system-on-a-chip designs. Norwell, MA, USA: KluwerAcademicPublishers; 1998.
12. Tranoris C, Thramboulidis K. A tool supported engineering process for developing control applications. ComputInd 2006;57(5):462–72
13. The SystemJ Approach to System-Level Design
14. A. Malik, Z. Salcic, P. Roop, A. Girault: SystemJ: A GALS language for system level design, Elsevier Comput. Lang. Syst. Struct., vol. 36, 2010, pp. 317-344
15. Gérard Berry, The Esterel v5 Language Primer, Ecole des Mines and INRIA, 2004.
16. C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.
17. Luciano Lavagno , Alberto Sangiovanni-Vincentelli , Ellen Sentovich, Models of Computation for Embedded System Design, System-Level Synthesis, 45-102, Kluwer Academic Publishers, 1998.
18. G. Berry, S. Ramesh, and R. K. Shyamasundar, Communicating reactive processes, Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1993, pp. 85–98
19. S. Ramesh, Communicating reactive state machines: design, model and implementation, IFAC Workshop on Distributed Computer Control Systems, 1998.
20. B. Rajan and R. K. Shyamasundar, Multiclock ESTEREL: A reactive framework for asynchronous design, Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International, 2000, pp. 201–209.
21. Reinhard Wilhelm, et al., The worst-case execution-time problem—overview of methods and survey of tools, ACM Transactions on Embedded Computing Systems (TECS) 7(3), 1–53 2008.
22. Avinash Malik, Principia lingua SystemJ, PhD thesis, The University of Auckland, 2009.
23. Marian Boldt, Claus Traulsen, and Reinhard von Hanxleden, Worst Case Reaction Time Analysis of Concurrent Reactive Programs, Electronic Notes in Theoretical Computer Science (ENTCS), Volume 203 Issue 4, Pages 65-79, Jun 2008.
24. V. Bertin and et al. Taxys = Esterel + Kronos - a tool for verifying real-time properties of embedded systems. In IEEE Conference on Decision and Control, 2001.
25. P. S. Roop, S. Andalam, R. von Hanxleden, S. Yuan, and C. Traulsen. Tight WCRT analysis for synchronous C programs. In Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES’09), Grenoble, France, October 2009. IEEE.
26. Pruning Infeasible Paths for Tight WCRT Analysis of Synchronous Programs
27. Efficient WCRT Analysis of Synchronous Programs using Reachability
28. L. Ju, B. K. Huynh, A. Roychoudhury, and S. Chakraborty. Performance debugging of Esterel specifications. In CODES+ISSS, pages 173–178, 2008.
29. Context-sensitive timing analysis of Esterel programs
30. L. Ju, B. K. Huynh, A. Roychoudhury, and S. Chakraborty. Timing analysis of esterel programs on general-purpose multiprocessors. In Proceedings of the 47th Design Automation Conference, DAC ’10, pages 48–51, New York, NY, USA, 2010. ACM.
31. L. Yoong, P. Roop, Z. Salcic, “Compiling Esterel for Distributed Execution,” in Proceedings of Synchronous Languages, Applications and Programming, 2006.
32. A. Malik, Z. Salcic, and P. S. Roop. SystemJ compilation using the tandem virtual machine approach. ACM Trans. Des. Autom. Electron. Syst., 14(3):1{37, 2009.
33. A. Malik, Z. Salcic, A. Girault, A. Walker, S.C. Lee, A Customizable Multiprocessor for Globally Asynchronous Locally Synchronous Execution, Proceedings of Java Technologies for Real-time and Embedded Systems, Madrid, Spain, 2009, ACM.
34. M. Nadeem, M. Biglari-Abhari, Z. Salcic, "RJOP - A customized Java processor for reactive embedded systems," Proceedings of the 48th Design Automation Conference (DAC), pp.1038-1043, 5-9 June 2011.
35. M. Nadeem, M. Biglari-Abhari, Z. Salcic, GALS-JOP: A Java Embedded Processor for GALS Reactive Programs, IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing (DASC), pp.292-299, 12-14 Dec. 2011.
36. M. Nadeem, M. Biglari-Abhari, Z. Salcic, JOP-Plus - A Processor for Efficient Execution of Java Programs Extended with GALS Concurrency, Proceedings of the 17th Asia and South Pacific Design Automation Conference Technical Program, 2012, pp. 17-22.
37. GALS-HMP: A heterogeneous multiprocessor for embedded applications
38. M. Schoeberl. JOP Reference Handbook: Building Embedded Systems with a Java Processor. 2009.
39. Li, X., J. Lukoschus, M. Boldt, M. Harder and R. von Hanxleden, An Esterel Processor with Full Preemption Support and its Worst Case Reaction Time Analysis, in: Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) (2005), pp. 225–236.
40. Simon Yuan, Li Hsien Yoong, Sidharta Andalam, Partha S. Roop, and Zoran Salcic. 2009. A new multithreaded architecture supporting direct execution of Esterel. *EURASIP J. Embedded Syst.* 2009, Article 4 (January 2009), 19 pages.
41. P. S. Roop, S. Andalam, R. von Hanxleden, S. Yuan, and C. Traulsen. Tight WCRT analysis for synchronous C programs. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES’09)*, Grenoble, France, October 2009. IEEE.
42. S. Andalam, P.S. Roop, A. Girault, "Pruning infeasible paths for tight WCRT analysis of synchronous programs," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011* , vol., no., pp.1,6, 14-18 March 2011
43. Uppaal tool. www.uppaal.com. last accessed on April 2014
44. A Tutorial on UPPAAL 4.0

# SystemJ Execution platform