Guaranteeing Reaction Times for Predictable Real-Time Applications on Multicore System with Shared Resources

1

Muhammad Nadeem, University of Auckland

zhenmin li, University of Auckland

Avinash malik, University of Auckland

Morteza biglari-abhari, University of Auckland

zoran salcic, University of Auckland

In this paper, we present a static analysisthat guarantees the execution times of concurrent programs that comply with Globally Asynchronous Locally Synchronous (GALS) formal model of computation (MoC) on chip multiprocessor with shared resources such as memory while using minimal resources. The interference to shared resources represent a bottleneck for timing predictability which is controlled by deploying Time Division Multiple Access (TDMA) and guarantees bounded delays. TDMA may lead to at the same time to increasing memory access latencies exacerbating the execution time of task dominated with memory access instructions on a particular processor with fair based slot allocation technique. The analytical tool statically analyses the program during compilation and finds out memory reference and non-memory reference instruction along the worst case execution path. It then computes the worst case execution time for each of the task running on a processor and calculates individual slot lengths for each processor so that all the tasks meet the deadlines which is inter arrival time between inputs. We demonstrate the applicability and computational efficiency of the presented methods by execution of benchmarks on multiprocessor based embedded system with a TDMA bus, where we will find the optimal parameter set for the TDMA bus.

Categories and Subject Descriptors: **C.2.2 [Computer-Communication Networks]**: Network Protocols

General Terms: GALS, Real-time, Multicore

Additional Key Words and Phrases: Wireless sensor networks, media access control, multi-channel, radio interference, time synchronization

**ACM Reference Format**:

Gang Zhou, Yafeng Wu, Ting Yan, Tian He, Chengdu Huang, John A. Stankovic, and Tarek F. Abdelzaher,2010. A multi-frequency MAC specially designed for wireless sensor network applications. *ACM Trans. Embedd. Comput. Syst.*9,4,Article 39 (March 2010), 6 pages*.*   
DOI:http://dx.doi.org/10.1145/0000000.0000000

# INTRODUCTION

The growth in semiconductor industry has allowed us to realize more transistors and as a result, more computational power on a single larger dies. This has helped us to realize even more complex applications which were not possible to realize due to limited computational capabilities. These complex applications required more and more processing speeds which have been achieved by lowering the feature size and increasing the frequency of a uniprocessor in the past. Due to increased power consumption area and lack of cooling mechanism does not allow to exploit this feature any further for the quest of more processing power. This lead to a system where multiple processors are used to exploit the parallelism of the application by executing the different tasks or processes on different processors. The performance gain also achieved even if there is no parallelism as the task being executed on individual processor need not to be interrupted thus avoiding the context switch time. The multiprocessor system integrates two or more processing units and a sophisticated communication network into a single integrated circuit. The homogeneous multiprocessor processor systems consist of identical processors connected to a single shared memory to save resources and keep the price real-state low.

Today, many embedded systems are deployed for safety critical applications where real-time behavior is more important than computation power. Failure to meet deadlines may result in the loss of life or in large damages. Therefore, such systems must undergo a timing analysis. In general, upper bounds on the execution times, also called worst case execution times, are needed to show the satisfaction of stringent timing constraints, which are derived from the hard real-time systems controlled.

Although the multiprocessor system results in increase in the performance but it is still desired that the number of cores still be kept to minimum to the save the real-state cost and e reduce the energy consumption to elongate the battery life.

Many tasks in safety-critical embedded systems have hard real-time characteristics. Utmost carefulness and state-of-the-art machinery have to be applied to make sure that all A lot research has been carried out to calculate the worst-case execution time analysis. However according to proposed techniques, each task is analyzed in isolation, as if it was running on a mono-processor system. Consequently, it is assumed that memory accesses over the bus take a constant amount of time to process, since no bus conflicts can occur. But, Shared memory introduces the possibility of timing interference between tasks [[Paolieri et al. 2013](#_ENREF_28); [Pitter and Schoeberl 2007](#_ENREF_29); [Thiele and Wilhelm 2004](#_ENREF_39)]. This occurs when one task affects the execution time of another. One way to eliminate shared bus interference is to adopt time-division multiple access (TDMA) arbitration, which grants each CPU access to the bus on a fixed, statically predictable schedule and consequently, a constant bandwidth to each processor [[Andrei et al. 2008](#_ENREF_1); [Poletti et al. 2003](#_ENREF_30); [Rosen et al. 2007](#_ENREF_32); [Schranzhofer et al. 2010](#_ENREF_37)]. The simplest form of TDMA schedule provides bus access to CPUs in a strict rotation, providing a fixed time-slice to each. This could be called a generic TDMA schedule.

A reliable guarantee based on the worst-case execution time of a task could easily be given if the worst-case input for the task were known. Unfortunately, in general the worst-case input is not known and hard to derive.

Say something about schedule

Furthermore, modern embedded systems consist of many computational units leading to GALS paradigm [[Chapiro 1985](#_ENREF_5)]which is based on a system composed of a number of synchronous modules designed in a traditional way which communicate asynchronously using handshake. Modeling such system is a tedious task and need formal verification before they can be deployed in safety-critical information systems.

-----------------------

The rapid growth in semiconductor technology have enabled the implementation of larger systems with more They also tend to interact with the environment making them reactive by repeatedly reading inputs, doing computations and generating outputs. These computational units have different response times; hence, they need to run concurrently at different speeds. These types of embedded systems can either be control-dominated or data-dominated or mix of both. Embedded systems based on GALS paradigm introduce a growing amount of synchronism and non-determinism and are the key issues in modeling such systems. Furthermore, many modern information systems are becoming safety-critical in a general sense because financial loss and even loss of life can result from their failure. Therefore, such systems need formal verification before they can be deployed. Different approaches have been used to model such systems such as RTOS based [], synchronous languages based approach [[Berry 1999](#_ENREF_3)], [[Milner 1982](#_ENREF_22)] [[Halbwachs 2005](#_ENREF_7); [Le Guernic et al. 2003](#_ENREF_14); [Mousavi et al. 2004](#_ENREF_23)] which suggest ways to describe GALS systems using synchronous languages but fail to recognized asynchronism and non-determinism as first-class concepts. Recently, there has been emergence of languages specifically designed to model asynchrony and non-determinism, and equipped with powerful verification tools, namely process calculi such as CSP [[Hoare 1985](#_ENREF_9)], Lotos[[Bowman 2006](#_ENREF_4)], Promela [18] and SystemJ[[Gruian et al. 2006](#_ENREF_6)]and GRL[[Jebali et al. 2014](#_ENREF_10)]. These languages tremendously reduce the design time as well as time to market and resulting in reduced productivity gap.

SystemJ [[[Malik et al. 2010](#_ENREF_21)]] is a system level programming language based on the globally Asynchronous Locally Synchronous (GALS) model of computation and allows the asynchronous coupling of synchronous reactive modules at the top level, which execute at different speeds. It extends Java with Esterel-like [[Berry 1999](#_ENREF_3)] constructs for the synchronous concurrency and reactivity, and CSP-like [[Hoare 1985](#_ENREF_9)] constructs for the asynchronous concurrency. A SystemJ program consists of multiple asynchronous processes, called clock-domains (CD), which are described at the top design level. The clock-domains are composed together with the asynchronous parallel operator (><). Each clock-domain consists of a number of synchronous concurrent processes, called reactions, which execute in lock-step, driven by a logical clock, called tick. A logical tick is used to represent a discrete time instant for a clock domain and all its synchronous reactions, where each clock domain executes at its own logical tick. SystemJ is based on the synchrony hypothesis which makes the abstractions that the computer is infinitely fast, each reaction is instantaneous and atomic, dividing time into a sequence of discrete instants, different reactions cannot interfere with one another, and a system’s reaction to an input appears at the same instant as the input. The time between two consecutive logical ticks is typically called reaction time. Reaction times can vary from tick to tick: reaction time is elastic, but the longest reaction time that ensures no loss of reaction on the next input event, if it can be determined, is called the Worst Case Reaction Time (WCRT) of a synchronous program.

A SystemJ program can be compiled and run on JOPplus[] processor where clock-domains are executed one after the other scheduled in a cyclic executive fashion. This results in increased reaction time as a clock-domain is able to respond to the environment only after all other clock-domains have finished the execution. A hard real-time system can only be described as safe if deadlines are always met. So there is a problem: how can speed be increased while preserving safety and without consuming much power? One way to boast performance is by introducing hardware platforms consisting of multiple cores to provide parallel and faster execution. To achieve efficient execution, programs can be divided into parallel operations at the clock-domain boundaries. But allocating each clock-domain to a separate processor is not a very efficient solution and might be infeasible as a typical embedded system described in SystemJ as it may comprise of tens of clock-domains resulting in too many logic resources usage impractical to realize. Therefore, the multicore system has to be tuned in order to maximize platform performances while minimizing non-functional costs such as the power consumption and resource utilization. This tuning process is called Design Space Exploration (DSE), and can be formalized as a multi-objective optimization problem where non-commensurable objectives have to be maximized (or minimized). Given the large amount of platform parameters and the large number of values that these parameters can assume, the design space of advanced computer architectures is huge[[Lukasiewycz et al.](#_ENREF_18)]. Theoretically, identifying the Pareto-optimal solutions in such space requires the evaluation of each platform configuration value sets. This is impracticable. Furthermore, real-time system require timing validation by schedulability analysis [[Joseph and Pandya 1986](#_ENREF_11); [Liu and Layland 1973](#_ENREF_17)] which can only be performed if WCET of the task is known.

Hard real-time systems need to satisfy stringent timing constraints, which are derived from the systems they control. In general, upper bounds on the execution times are needed to show the satisfaction of these constraints. Unfortunately, it is not possible in general to obtain upper bounds on execution times for programs. Otherwise, one could solve the halting problem. A reliable guarantee based on the worst-case execution time of a task could easily be given if the worst-case input for the task were known. Unfortunately, in general the worst-case input is not known and hard to derive. A task (clock-domain) typically shows a certain variation of execution times depending on the input data or different behavior of the environment.

To achieve predictability with respect to time, various techniques are applied, assuming that the worst-case execution time (WCET) of every task is known. A lot of research has been carried out within the area of worst-case execution time analysis[[Puschner and Burns 2000](#_ENREF_31); [Wilhelm et al. 2008](#_ENREF_40)]. However, according to the proposed techniques, each task is analyzed in isolation, as if it was running on a mono processor system. Consequently, it is assumed that memory accesses over the bus take a constant amount of time to process, since no bus conflicts can occur. But, Shared memory introduces the possibility of timing interference between tasks [[Paolieri, #246, Mische, Metzlaff, Gerdes, Qui, #241, ones, Uhrig, Ungerer and Cazorla 2013](#_ENREF_28); [Pitter and Schoeberl 2007](#_ENREF_29); [Thiele and Wilhelm 2004](#_ENREF_39)]. This occurs when one task affects the execution time of another. One way to eliminate shared bus interference is to adopt time-division multiple access (TDMA) arbitration, which grants each CPU access to the bus on a fixed, statically predictable schedule and consequently, a constant bandwidth to each processor [[Andrei, Eles, Peng and Rosen 2008](#_ENREF_1); [Poletti, Bertozzi, Benini and Bogliolo 2003](#_ENREF_30); [Rosen, Andrei, Eles and Peng 2007](#_ENREF_32); [Schranzhofer, Pellizzoni, Chen, Thiele and Caccamo 2010](#_ENREF_37)]. The simplest form of TDMA schedule provides bus access to CPUs in a strict rotation, providing a fixed time-slice to each. This could be called a generic TDMA schedule. A GALS program may consist of a number of clock-domains with different amount of data computation and memory access. If the clock-domain(s) scheduled on one core contains computations requiring more frequent access to memory than the other, the general TDMA policy may lengthen the WCET due to allocation of equal amount of time (slot) to a processor with no or minimal memory access requirements[[Pitter and Schoeberl 2007](#_ENREF_29)]. The solution to this problem is weighted TDMA schedule where each processor’s slot length is proportional to memory accesses contained by the clock-domains executing on it. The slots in TDMA schedule should be allocated in a way so that they reduce the worst case access time and, consequently, the worst case reaction time.

In this paper, we present a framework which takes a SystemJ program and analyze and calculate the worst case reaction time of each clock-domain assuming that the processor executing it has exclusive access to the memory. It than allocates the clock-domains to the processors such that the required execution times are met while using minimum number of cores. It means each core is allocated at least one clock-domian and may execute more if timing constraints are met. If required run time is too large, then all the cores can be run on a single processor. As mentioned earlier, shared resources (memory in this case) affect the execution of the clock-domains on a processor, resulting in lengthening of execution times, it recalculates the worst case reaction time of each clock-domain assuming a generic TDMA schedule. If timing guarantees are violated, the tool gives weighted TDMA schedule where each processor is allocated a number of slots in a TDMA cycle while guaranteeing the required reaction times.

**Contributions**

A contribution of this paper is

* The extension of the WCET tool for timing analysis of the SystemJ programs running on JOPPlus.
* The design space exploration of system which allocates the clock-domains to the individual processor in a clock-domain with an objective to minimize the resource usage while guaranteeing the required reaction times.
* Find out the TDMA weights for each processor so that all the clock-domains meet the GRT. It can be composed into: that makes at least one individual [better off](http://en.wikipedia.org/wiki/Utility) without making any other individual worse off is called a Pareto

# Related Work

WCET analysis has been an investigation topic for already a long time, the basic driving force of this research has been, and still is, to improve the tightness of the analysis and to incorporate more and more features of modern processor architectures. However, one of the basic assumptions of this research is that WCETs are determined for each task in isolation and then, in a separate step, task scheduling analysis takes the global view of the system [[Thiele and Wilhelm 2004](#_ENREF_39)]. This approach is valid as long as the applications are implemented either on single processor systems or on very particular multiprocessor architectures in which, for example, each processor has a dedicated, private access to an exclusively private memory.

The WCET analysis of multiprocessors [[Andrei, Eles, Peng and Rosen 2008](#_ENREF_1); [Rosen, Andrei, Eles and Peng 2007](#_ENREF_32)] is based on a multiprocessor system-on-chip with a shared communication bus, connecting several CPUs with two different types of memory. Each CPU has a private memory and all the processing units share a common memory for communication. A CPU is equipped with instruction and data caches, which are used to fetch data and instructions from the private memory. During execution, a task can only access private memory and no shared data objects. Hence, all input data must be placed into the private memory before the task can start executing. Consequently, in most cases the execution time of the task can never be influenced by other tasks (compare the simple-task model [[Kopetz 1997](#_ENREF_12)]. However, the communication bus serves as a communication interface between the CPUs and the private memories, and the CPUs and the shared memory. If a cache miss occurs during task execution, data has to be fetched from private memory using the communication bus. Therefore, some sort of bus arbitration is necessary because several CPUs may request a cache line from their private memories simultaneously.[[Gruian, Roop, Salcic and Radojevic 2006](#_ENREF_6)].

For analyses that include the shared bus, the choice of the bus arbitration method

is crucial. [Pitter and Schoeberl (2010)] compared the predominant arbitration methods, and TDMA arbitration resulted as the most predictable method, because it allows deriving an upper bound on the delay that any access to the shared resource may incur until it is granted. However, in order to avoid the problem of scheduling tasks, they assume that the numbers of cores are greater than the number of tasks. This is also the conclusion that is drawn in [Wilhelm et al (2009)], where future architectures of time-critical embedded systems are discussed.

The predictable multi-threading is developed within the PRET project [14], [[Schoeberl et al. 2009](#_ENREF_36)] achieved good performance and high temporal predictability through an offline-planning approach of single-path programming which avoids control flow dependent timing variations by taking as many control decisions as possible before the system is actually run. The single-path approach allows translating task code in a way that the resulting code has exactly one execution trace that all executions of the task have to follow. To this end, the single-path conversion eliminates all input-dependent control flow decisions but this approach is not applicable to reactive systems.

In Andrei et al. [2008] and Schoeberl and Puschner [2009], the worst-case cost of orthogonalization was significantly reduced by deriving optimal bus schedules given the memory access pattern of each task. However, the approaches do not support local task scheduling and are thus applicable only as long as the number of tasks in the system does not outgrow the number of processors. [[Schranzhofer, Pellizzoni, Chen, Thiele and Caccamo 2010](#_ENREF_37)]relaxed the assumption of fixed positions for the bus access using the same TDMA technique to arbitrate the shared bus. They divide tasks into sets of super blocks that are specified with a maximum execution time and a maximum number of memory accesses, and these superblocks are executed either sequentially or time-triggered. TDMA arbitration techniques eliminate the interference of other tasks due to accessing shared resources through isolation; however, they are limited in the usage of only a specified hardware.

The main problems that researchers have tried to solve are (1) the identification of the possible execution sequences inside a task and (2) the characterization of the time needed to execute each individual action[[Puschner and Burns 2000](#_ENREF_31)]. With advanced processor architectures, effects due to caches, pipelines, and branch prediction have to be considered in order to determine the execution time of individual actions. There have been attempts to model both problems as a single ILP formulation [[Li et al. 1996](#_ENREF_15)]. Other approaches combine abstract interpretation for cache and pipeline analysis with ILP formulations for path analysis[[Theiling et al. 2000](#_ENREF_38)], or even integrate simulation into the WCET analysis flow [[Wolf et al. 2002](#_ENREF_41)].

The general worst-case bound on access delays in a TDMA schedule is usually not

tight. It tries to determine the precise time at which every single memory access takes place. The bus access delay estimation is then performed separately for each access. The main problem is that accesses in loops with an iteration count of i can potentially have i different access times associated to the same memory access. Therefore, the analysis has to unroll all loops virtually to determine the access times for each access individually, which makes the analysis runtime dependent on the loop iteration counts. It also assume that application is running periodically with a period greater than or equal to the application deadline.

Implementations of buses or networks on chip which use TDMA arbitration include the Aethereal Network-On-Chip [Goossens and Hanson (2010)], the Time-Triggered Architecture [Paukovits and Kopetz (2008)] and the FlexRay bus from the automotive domain [FlexRay Consortium (2010)]. The basic property in favor of TDMA is, that TDMA allows to statically determining the delay of a request, once we know the point in time when the request is issued. Other protocols which build upon dynamic resolution of conflicts between multiple competing accesses do not have this property (e.g. priority-based arbitration, FIFO arbitration). The access delay in these “dynamic” protocols can only be bounded if all interference between possibly concurrent accesses is known. The drawbacks of this are that– the determination of the set of possibly concurrently executed bus accesses is much harder than the determination of the access time as required for TDMA and the delay of an access and thus the WCET of a task is no longer computable for agiven, single task on a given platform in isolation, but only if the full set of tasks

running on the platform is known. This means that timing composability is lost

on platforms with “dynamic” protocols. This means that even though these “dynamic” protocols provide better average-case performance than TDMA, the worst-case bounds that are computable from the microarchitectural view of a single core are usually not better or even worse than those for TDMA. Therefore, in this work we focus exclusively on TDMA.

# Preliminaries

SystemJ

The system-level programming language SystemJ [[Gruian, Roop, Salcic and Radojevic 2006](#_ENREF_6); [Malik, Salcic, Roop and Girault 2010](#_ENREF_21)]extends Java with synchronous andasynchronous concurrency and reactivity, making it suitable for designing complex embedded programs. The language allows use of full Java and discourages the use of Java concurrency (threading library). Instead, SystemJ provides its own concurrency model based on the formal Globally Asynchronous Locally Synchronous (GALS) model of computation (MoC). A SystemJ program consists of multiple asynchronous processes, called clock-domains (CD),which are described at the top design level. The clock-domains are composed together with the asynchronous parallel operator (><). Each clock-domain consists of a number of synchronous concurrent processes, called reactions, which execute in lock-step, driven by a logical clock ,called tick. A synchronous program reacts to its environment in a sequence of ticks, and computations within a tick are assumed to be instantaneous, i.e., as if the processor executing them was infinitely fast. The reactions communicate within a clock-domain, as well as with the external environment (input/output) through signals, which are broadcast and present within thecurrent tick and comply with synchronous reactive MoC [16, 41–43]. The reactions are represented as concurrent processes within a clock-domain using the synchronous parallel operator(||). Communication between reactions in different clock-domains, which are asynchronous each to the other, is carried out through the exchange of messages over channels, which are semantically the same as channels used in CSP MoC [1, 23]. Besides operations on signals and channels, SystemJ allows free use of Java data objects and statements in its reactions, and those statements are considered instantaneous in terms of logical time (i.e. they do not consume logical time or ticks). Control flow of a SystemJ program incorporates scheduling of all reactions and clock-domains, as well as communication between reactions, and communication with the external environment. The data-driven computations and transformations are performed in Java.

The system is the top level entity through which a SystemJ program sees its environment and vice versa. In the declarative part of the system entity, the interface with the environment (whichare input/output signals) and channels are declared. The clock-domains are declared inside thebody of the system. The channels serve as the communication medium between the clock-domains.Multiple clock-domains, which are asynchronous processes, are also declared in thebody of this entity. Within a system, each clock-domain executes at its own tick, and any two clock-domain ticks are unrelated.

Each clock-domain consists of a number of synchronous concurrent processes called reactions. The clock-domain can be given some name or can be unnamed. The clock-domains can communicate with each other and with the environment using their own set of channels and signals, respectively. Two clock-domains synchronizing with each other using CSP style communication are called partner clock-domains. The data structures cannot be shared among the reactions and among the clock-domains. This prohibition allows communication only through the signals and channels, which is safer and easier to validate. Reactions are combined and controlled within a clock-domain using the synchronous parallel operator (||). Reactions can be either named or unnamed and may have child reactions. Reactions use sequential and concurrent statements to specify their operation and those statements are from both SystemJ and Java repertoire. Reactions communicate within a clock-domain, as well as with the external environment (input/output) through signals, which are broadcast and present within the current tick if emitted, otherwise they are absent. Communication between reactions in different clock-domains is carried out through the exchange of messages over channels, which are semantically the same as channels used in the CSP MoC [1,23]. The signals are used for communication among reactions within a clock-domain as well as with the external environment (input/output). The signals communicating with the environment, whether input or output are called interface signals and are declared in the interface part of the body of the system. The signals which are used to communicate among the reactions in the clock-domain are termed as local signals and are declared within the body of a clock-domain. The signals are always broadcast and present within the current tick if emitted, otherwise they are absent. The signal can be either a pure signal or a valued signal. A pure signal has a status only, while a valued signal has both status as well as value of any Java data-type. As mentioned earlier; communication between reactions of different clock-domains is carried out through the exchange of messages over channels. They are used to synchronize the clock-domains. The channels can be of any Java type and are declared in the interface body of the system. The clock-domains are synchronized through channels using CSP style rendezvous mechanism. The idea is that two clock-domains rendezvous at a point of execution, and neither is allowed to proceed from that point until both have arrived, but they can proceed to execute other reactions of the same clock-domain if needed. The channels are point-to-point i.e., for every sending channel port there needs to be a corresponding receiving channel. The channels are unidirectional, therefore, they are always declared in pairs (input and output).Every channel has status and value buffers which are used to implement the rendezvous communication. Besides signals and channels objects, SystemJ allows free use of Java data objects in the program. The SystemJ objects declared are global but they cannot be used for the communication among the clock-domains as it is allowed through channels only.

There are two dummy statements, ’;’ and pause. The ’;’ does not perform any operation. The pause statement also does not perform any operation but differs from the ’;’statement as pause consumes one tick and the following instruction is executed only in the next tick. Sequential statements: The SystemJ allows the combining of synchronous reactive statements by using the ’;’ operator, making them execute sequentially one after the other. The SystemJ programs run forever and this feature is implemented by using *while* statement which provides an infinite loop mechanism. The present statement checks the status of a signal expression at any given instant of time. The signal can only be present if it is emitted or it is coming from the environment.

The synchronous concurrency causes reactions to run in parallel and in lockstep. This is done by using the synchronous concurrency primitive construct (||).Each reaction in the clock-domain finishes with a termination code having an integer value ranging from 0 to infinity (1). A reaction terminates with the code of 0 if it is completed and will never run again. The SystemJ requires a pair of input and output channels to have the same name. The output channel can only send the data as it is a simplex channel.

Table I. Simulation Configuration

|  |  |
| --- | --- |
| Syntactic constructs | Meaning |
| p1;p2 | p1 and p2 in sequence |
| pause | Dummy statement consuming a tick (i.e, completing a logical instant); |
| [input][output][type] signal Signal Name | Signal declaration statement |
| emit s [(exp)] | Emitting a signal with a possible value |
| while (true) p | Infinite temporal loop every iteration consumes at least a single tick |
| present SignalName p1 else p2 |  |
| [weak]abort([immediate]SignalNamefpg | Preempt if S was present in the previous tick |
| [weak]suspend([immediate]SignalName{p} | Suspend for one tick if S was present in the previous tick |
| trap (T) {p} | User controlled software exception based on signal |
| exit (T) | Throw a software exception |
| p1 || p2 | Run p1 and p2 in parallel and in lock step |
| jterm(p) | Java instantaneous term |
| #SignalName | Signal value |
| output [type] channel ChannelName | Declaring output channel |
| input [type] channel ChannelName | Declaring input channel |
| #ChannelName | Channel value |
| p1 >< p2 | Run clock-domains p1 and p2 asynchronously parallel |
| send C([exp]) | sending data on channel |
| receive C() | Receiving data on channel |

The SystemJ is amenable to verification as it is based on mathematical semantics and can be deployed in mission critical systems. Figure 1 gives an abstract representation of a SystemJ program that implements an extended version of asynchronous protocol stack[[Nadeem et al. 2013](#_ENREF_27)]. This program is readily runnable on any processor with JVM. It has three clock-domains; the first clock-domain models the packet generation from a network device, while the second and third clock-domains implement the main functionality of ***protocol stack*** [[Gruian, Roop, Salcic and Radojevic 2006](#_ENREF_6); [Lavagno and Sentovich 1999](#_ENREF_13)], which can be extended to model complete communication stack like TCP/IP. There are three synchronous parallel reactions in each protocol\_stack clock-domain: *Assemble*, *Checkhdr* and *Stack*. The *Assemble* reaction receives the bytes asynchronously on channel and assembles them into a packet that is sent synchronously through a valued signal for further processing. The packet is awaited for in reaction *Checkcrc* (line 34), a *Cyclic Redundancy Check* is carried out on it and the outcome of this test is emitted through a boolean signal further (line 36).Note how both the data and data operations are neatly encapsulated in the *Packet*, thanks to Java object orientation, making the code more legible. The *Prochdr* reaction carries out an address match computation (line 47) in parallel with checking the *crc ok* signal (lines 51–52). If the packet is faulty, the address calculation is aborted by emitting a signal (line 52), monitored by the address matching sub-reaction (line 46). Finally, the full protocol stack is another reaction, *TheStack*, which contains just a synchronous composition of the *Assemble*, *Checkcrc*, and *Prochdr* reactions. Note that the valued signals *packet* and *crc ok* are local signals, hidden from the environment outside *TheStack*.



Fig. 1. The asynchronous protocol stack

|  |  |  |  |
| --- | --- | --- | --- |
| **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**  **45**  **46**  **47**  **48**  **49**  **50**  **51**  **52**  **53**  **54**  **55**  **56**  **57**  **58**  **59**  **60** | import Asproto.\*;  Aspcd2(output int channel reset2; output byte channel data2; ) -> {  //details abstracted  }  Aspcd2(input int channel reset1; input byte channel data1;) -> {  signal packet1,kill\_check1;  signal res11,res12,res13;  Integer signal crc\_ok1;  Asproto buffer1 = null;  {  while (true) {  receive reset1;  int u1= 0;  u1 = #reset1;  if (u1==1) {  emit res11; emit res12; emit res13;  }  pause;  }  || //Assemble  {  int cnt1=0;  buffer1 = new Asproto();  while(true) {  abort(res11){  int e1 =0;  trap(T){  int len12 = Asproto.PKTSIZE;  if (e1 == len12){  exit(T);  }  else{  receive data1;  byte t1 = 0; t1 = #data1;  e1=e1+1;  buffer1.setRaw(e1,t1);  }  }  emit packet1;  }  pause;  }  }  || //Checkcrc(reset,packet,crcok)  {  int crc=0;  while(true) {  abort(res2){  await(packet);  crc = ((Asproto)#packet).computeCRC();  int val = 0;  val = (crc == ((Asproto)#packet).getCRC()) ? 1:0;  emit crcok(val);  }//abort  pause;  } // while\*/  } | **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16** | ||  {  int match\_ok=0;  while(true){  abort(res3){  await(packet);//now buffer is avail  {  abort(killcheck){  match\_ok = 1;  pause;  pause;  pause;  pause;  }  }  ||  {  await(crcok);  int re = 0;  re= (Integer)#crcok;  if(re==0){  emit killcheck;  }  }  int there = 0;  there = (Integer)#crcok;  if(there==1 /\*&& match\_ok==1\*/){  System.out.println("Address match!");  }  }//abort  pause;  }//while\*/  }  Aspcd3(input int channel reset2; input byte channel data2;) -> {  signal packet2,kill\_check2;  signal res21,res22,res23;  Integer signal crc\_ok2;  Asproto buffer1 = null;  // details abstracted  } |

Fig. 2. Asynchronous protocol stack in SystemJ

Task Graph Representation of Asynchronous Protocol stack

Time-Analyzable Chip Multicore Processor

SystemJ programs integrate both control-driven and data-driven operations which are separated at the compiler back-end. The control flow representing control-driven operations is named *Concurrency and Reactivity Control Flow* (CRCF). 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. 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. Then these control flows can be supported in the execution platform by using the extended JOP[[Schoeberl 2005](#_ENREF_33); [Schoeberl 2008](#_ENREF_35)] execution unit. This new execution unit has single instruction decoding unit and performs very efficient switching between CRCF and JCF when necessary. At the same time the compiler did not require any modifications. Thus, all SystemJ and JOP tools, without practically any modification, are made ready for the new processor.

The multicore platform used in this research is called GALS-CMP which implements the symmetric shared-memory multiprocessor model [[Hennessy and Patterson 2003](#_ENREF_8)]. The GALS-CMP follows the methodology adopted by the time predictable multiprocessor system JopCMP [[Pitter and Schoeberl 2007](#_ENREF_29)]. Several JOP-plus [[Nadeem et al. 2012](#_ENREF_25)] cores provide the basis of the homogeneous CMP as depicted in Figure ccc. JOP-plus provides a seamless integration of both control and data execution modes in one processor unlike TP-JOP [[Nadeem et al. 2011](#_ENREF_24)]which uses two different processors thus consuming too many logic resources. The JOP-plus supports two separate components of the control flow in SystemJ programs, Concurrency and Reactive Control Flow (CRCF) and Java Control Flow (JCF), by extending the instruction set of the original JOP while using single execution unit. JCF is dealt in conventional was as the JOP while CRCF is stored in a separate memory. The control execution is capable of invoking the data computations in Java, which then returns to the control directly. Shared memory is used to store Java data computations (JCF) made of a collection of Java methods that are loaded upon call from the control code into local method caches of individual cores before they are executed. Also, shared memory is used to store the JVM run time data areas.

The JOP is an open-source hardware (soft-core) implementation of the Java Virtual Machine (JVM)[[Schoeberl 2005](#_ENREF_33); [Schoeberl 2008](#_ENREF_35)] 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 [Schoeberl 2009]. It features a stack cache for the private data of each thread. Additionally, a kind of instruction cache (called method cache) limits the memory access frequency and increases the processing power. These real-time processing elements perform computations in parallel.JOP is proven to be much more efficient in running Java programs compared to software implementations of embedded JVM. JOP is also a time-predictable processor due to its architecture for time-predictable execution of Java bytecodes and Java methods. The time-analyzable 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 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.One of the highlighting features of JOP-Plus is that it is designed in a way so that it can use all SystemJ and JOP tools without practically any modification.

The depicted GALS-CMP architecture shows a synchronization unit which has the responsibility to coordinate access to the shared objects by a mutual exclusion mechanism. On-chip IO devices, such as a controller for real-time Ethernet or a real-time field bus, may be mapped to shared memory addresses and areconnected via the memory arbiter. All cores are connected to each other and to the memory via simple SoC interconnect(SimpCon) [[Schoeberl 2007](#_ENREF_34)] which provides point-to-point interconnections between components. All cores communicate with each other through the shared memory and an arbiter is used when multiple masters try to access the same slave.



Fig. ccc. GALS-CMP system.

To prevent inter-task interferences that disrupt the analyzability of HRT tasks

Algorithm 1. Frequency Number Computation

**Input:** Node *α*’s ID (*IDα*), and node *α*’s neighbors’ IDs within two communication hops.

**Output:** The frequency number (*FreNumα*) node *α*•gets assigned.

*index* = 0; *FreNumα*=−1;

**repeat**

*Rndα* = Random(*IDα*, *index*);

*Found* = *TRUE*;

**for***each nodeβ*• *inα’s two communication hops***do**

*Rndβ*= Random (*IDβ*, *index*);

**if** (*Rndα*<*Rndβ*) or (*Rndα* = = *Rndβ* and *IDα*<*IDβ*)

**then**

*Found* = *FALSE*; *break*;

**end**

**end**

**if***Found***then**

*FreNumα* = *index*;

**else**

*index* ++;

**end**

until*FreNumα*>−1;

# Methodology

Figure ccc gives an overview of our framework. SystemJ compiler front end compiles the GALS program described in SystemJ and produces AGRC in step 1. This together with abstract details of processor is fed to TACO framework in step 2 which calculates the WCRT of each clock-domain in isolation as if running on a dedicated processor. The framework analyses for worst case execution path and finds out all instructions falling in this path by separating them into memory referenced (M) and non-memory reference (I) instruction. This information together with the execution platform details is used to calculate the worst case reaction time. It then generates a Task Graph (TG) where each clock-domain is represented as a task annotated with its WCRT. The details of TACO framework are provided in Section 4.2. This TG together with AG and timing information is fed to design space exploration tool in step 3 with an objective to schedule all the clock-domains on minimum number of processor while meeting all the timing constraints. These objective functions and constraints are also formulated using ILP. Any ILP solver (e.g. Gurobi) can be used for deriving the whole program’s schedule whose details are provided in Section 4.3.The result of step 3 is multicore system connected to a shared memory via bus. For shared bus, we assume a TDMA-based round robin arbitration policy, where a fixed length bus slot is assigned to each core

The WCRT of all the clock-domains are recomputed for multicore architecture at step 4 accounting for allocation and delays due to TDMA. In case, time constraints are violated which are due to memory access, a weighted schedule for TDMA is computed at step 5 thus providing more bandwidth to processor excessive memory reference instructions so that it is able to meet the timing requirements. The details are provided in Section



Fig. ccc. GALS-CMP system.

## Memory Access Pattern

The SystemJ compiler compiles the program into CRCF instructions and JCF instructions which are executed on JOPPlus, an extension of JOP. JOP translates most of the byte-codes to its native microcode instructions. Each byte-code is composed of a microcode instruction or a series of microcode instructions. Some byte-codes are actually implemented in hardware. A couple of byte-codes are implemented in Java. According to the JVM specification [], the heap and the method area are shared data areas, whereas the stack is a private data area for each thread. In JOP, the heap and the method area are located in the main memory. Consequently, all byte-codes that work on these areas have to be carefully examined. Some byte-codes access the memory several times, some only once. Hence, it makes sense to have a closer look at the different instructions.

Table II summarizes the bytecodes that access the main memory. Most memory access patterns of the bytecodes can be statically analyzed; i.e. bytecodes that access the heap and those of type *const*. The pattern is only dependent on the memory access time. If the memory access time is known, the memory access pattern of the bytecodes can be analyzed regardless of the source code of the program. An example of such a bytecode is *ldc*, which pushes a single word constant onto the stack. Therefore, only one memory access to the method area is needed. JOP translates this bytecode into a series of microcodes. If the memory access time is known, the memory access pattern can be specified using JOP’s bytecode implementation. Another example is *iaload*, which is implemented in hardware. For the analysis of the memory access pattern, we determine the VHDL implementation in combination with ModelSim simulations.

The memory access patterns of the bytecodes of type call and return need a dynamic analysis and are more difficult to attain. Each JOP is equipped with an instruction cache that caches complete Java methods [14]. Consequently, the memory access patterns of these bytecodes vary, depending on the history of the execution. If the method is already in the cache, no additional memory accesses are needed to load the method into the cache. If a cache miss occurs, JOP will have to load the whole method into the cache.

The bytecodes data*call* and *returnresult* have different behaviour as compared to return and call. The methodcall bytecode directly invokes a method and always result in cache miss. Similarly, *returnresult* from a Java method to CRCF will not return to cache loading instead simply writes to CRCF memory and return to CRCF instruction execution. Depending on a cache hit or a cache miss and the length of the method that has to be loaded, the access pattern is created for each individual occurrence of the bytecode in the source code and integrated in the generation of the patterns into the WCET analysis tool where the cache information is available.

Table II. Bytecodes accessing the shared memory

|  |  |  |
| --- | --- | --- |
| **Type** | **Bytecode** | **Memory Area** |
| const | ldc, ldc\_w, ldc2\_w | Method Area |
| get | getfield, getstatic | Heap |
| put | putfield, putstatic | Heap |
| array | aaload, aastore, baload, bastore, caload, castore, daload, dastore, faload, fastore, iaload, iastore, laload, lastore, saload, sastore, arraylength | Heap |
| call | invokeinterface, invokespecial, invokestatic, invokevirtual, datacall | Method Area |
| return | areturn, dreturn, freturn, ireturn, lreturn, return, returncrcf | Method Area |
| new | anewarray, multianewarray, new, newarray | Heap |
| switch | lookupswitch, tableswitch | Method Area |
| cast | checkcast, instanceof | Heap |

## WCET Analysis

We use TACO [[Zhenmin et al. 2014](#_ENREF_42)], static analysis framework, for tight WCRT analysis and code optimization for each clock-domain of SystemJ programs. As shown in Fig [TODO] in Section [TODO], the TACO framework is invoked from Steps [TODO] to [TODO] in the complete flow of our approach. The source code of SystemJ program is firstly compiled into *Asynchronous GRaph Code* (AGRC) intermediate representation using the front-end of SystemJ compiler presented in [[Malik et al. 2011](#_ENREF_20)] (step 1). The AGRC extends the *GRaph Code* (GRC) with asynchrony by composing separated GRCs, one for every clock-domain in the program, using afork and ajoin nodes. The AGRC representation at this step does not contain any timing information and is then sent to the back-end of SystemJ compiler for target code generation. The target code generated by SystemJ compiler for JOP-Plus is based on the standard Java Virtual Machine (JVM) instruction set for JCF and a customized instruction set for CRCF (step 2) [[Nadeem et al. 2012](#_ENREF_26)]. The produced codes are in the format of Java byte codes. During target code generation in step 2, the tracking information is also produced, indicating the original AGRC node from which a segment of target code is generated.Both CRCF assembly code and JCF Java code are analyzed by our customized WCET analyzer to calculate WCET information for each AGRC node (step 3). We modify the WCET Analyzer used in TACO framework [[Li et al. 2014](#_ENREF_16)] to accommodate JOP-Plus execution platform. We first extend the JOP WCET analysis tool, called *JOP WCET Analyzer* (WCA), to be able to analyze the byte code generated based on the newly added instruction set to implement CRCF. We then analyze the Java byte codes for CRCF and JCF separately using extended JOP WCA to attain node-level WCET information. When analyzing the JCF code for each Java data computation, we separately calculate the worst-case execution time for only memory accesses (WCMT) and instruction execution time without memory access (WCIT). The original AGRC is then back-annotated with the node-level WCET information acquired from step 3 to obtain *Timed Asynchronous GRaph Code* (TAGRC). We annotate both WCMT and WCIT for each node in the AGRC. Note here, the WCMT for any control node representing CRCF is always 0 since no shared memory access occur during CRCF execution. In order to compartmentalize our WCRT analysis to each synchronous clock-domain, TAGRC is partitioned into a set of *Timed GRaph Codes* (TGRCs), one for every clock-domain. After obtaining TGRCs, model checking based tight WCRT analysis is carried out in steps 4-6. In step 4, each TGRC obtained from step 3 is translated to *Timed Automaton* (TA). Instead of using real-valued clocks, we employ a single bounded integer to capture the total time cost of a complete transition from the beginning to the end of a logic tick.The UPPAAL model checker [[Behrmann et al. 2004](#_ENREF_2)] is used in TACO framework as it offers efficient algorithms for both TA and automata with integer operations, while offering an excellent graphical user interface. All the timed automata produced in TACO framework are in the format of UPPAAL model. During the translation from TGRC to TA we preserve the following information in the UPPAAL model: (1) the state encoding and decoding that capture tick transition semantics and (2) internal and output signal emission and testing. The preserved information essentially models the execution flow of the SystemJ program. After the UPPAAL model is produced, tight WCRT result can be obtained by using the model checker which will automatically identify the scenarios where conventional WCRT estimation techniques like Max-Plus algebra would produce large over-estimates.



Fig. 5. Overview of TACO framework

 

(a) (b)

Fig. 7. Translation from TGRC to UPPAAL Model of TA

## Translation from Timed GRC to UPPAAL Model

This sub-section describes the translation from TGRC to UPPAAL model (UM) of TA, which is illustrated in Fig. 7. The graph code shown in Fig. 7(a) is the TGRC for the Asproto example introduced in Section [TODO]. The corresponding UPPAAL model translated from the TGRC is shown in Fig. 7(b). The first step in the translation is to perform a one-to-one mapping for each node in the TGRC to a location in the UM. For example, afork node AF0 in the TGRC is mapped to the initial location AF0 in UM. Similarly switch node S1 is mapped to a location S1, and so on. Next, we map each control flow edge in the TGRC to a transition between two locations in the UM. The conditional branching is modeled by annotating the transitions with additional guards. For instance, the transition from location S1 to E3 in Fig. 7(b) has the expression S633==636 as its guard to capture the tick transition semantics based on state encoding and decoding of switch node S1 in Fig. 7(a).

The signal emission and state encoding process are captured by the assignments on the transition. In Fig. 7(b), assignments on transitions from E3, E7 and E8 model three distinct state encoding. Three bounded integers, i.e. wcrt, wcmt and wcit, are used to count the execution time for only memory access, the execution time without any memory access, and the total execution time for both types of execution, respectively. The execution time is counted within a single global tick and reset to zero upon completion of one tick (during transition AJ5 to AF0). Therefore, every transition in the UM is annotated with an assignment that increases the value of integers wcrt, wcmt and wcit by the corresponding WCET values of the target location. Due to the semantics of AGRC nodes related with control flow, modifications should be done to the UM to correctly model the execution flow. Because the parallelism of the branches forked out by a fork node is compiled away so that each branch is executed one after another. Thus, the join node should have a transition back 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. The bounded integer wcrt should also be cleared to zero during the transition from ajoin node to afork node (from AJ5 to AF0), and at the same time all the signal statues and values are updated.

We model the tight WCRT analysis problem as verifying a CTL property in UPPAAL model checker. The tight WCRT, named WCRTtight, is the objective of WCRT analysis. WCRT-tight lies in the bounded range denoted by [WCRTlb, WCRTub]. WCRTub is a safe upper-bound of WCRT obtained by applying Max-Plus algebra to the TGRC, which is essentially summing up the maximum tick execution time of every reaction in the clock-domain. Similarly, we calculate a lower-bound of WCRT, termed WCRTlb, by summing up the minimum tick execution time of every reaction in the clock-domain. After translating TGRC to UM, we check the validity of a WCRT estimate of the clock-domain, termed WCRTest, by verifying a CTL property upon the UM using UPPAAL model checker. The CTL property is written as A[](wcrt ≤ WCRTest), meaning that the value of integer wcrt is less or equal to the WCRTest for every path starting from the initial location. In order to minimize the number of queries, we use standard binary search algorithm to find WCRTtight.

## Design Space Exploration

The allocation of clock-domains to processor in a multicore processor system is an issue which needs some attention. Two questions needed to be answered: 1) what is the minimum number of processor core to be used so that system meets all timing construct while using minimum logic resource and 2) What number and combination of clock-domains should be allocated to each core in the system so that the system meets all the timing constraints. The simplest answer can be to allocate a single clock-domain to each processor node in the system. But, systemJ programs can be very large and comprise of hundreds of clock-domains and allocating a single clock-domain allocation policy will result in hundreds of processors used and it may turn out impractical to implement such a huge number of cores. This need design space exploration to come out the system implementation optimized in terms of latency and resource requirements. The parameters which need to be evaluated are the number of cores objective for design space exploration will be to meet the reaction time.

The SystemCoDesigner ESL tool [Keinert et al. 2009] offers automated fast DSE and implementation flows from behavioural SystemC models. The performance information together with the executable specification, Task Graph (TG), and a so called architecture template serves as the inputmodel for design space exploration. The architecture template is represented by an Architecture Graph (AG) which contains all possible hardware accelerators, processors, memories, and the communication infrastructure from which the design space exploration has to select those ones which are necessary in order to fulfil the user requirements interms of overall worst case reaction time and number of cores. The fewer coresare allocated, the fewer hardware resources are required. This, however,in general comes along with a increase in wcrt in throughput, thus leading to trade-offs between execution speed and implementation costs. SystemCoDesigner uses multi-objective optimization together with symbolic optimization techniques [[Lukasiewycz et al. 2008](#_ENREF_19)]to find sets of optimized solutions. From this set of optimized solutions the designer selects a solution which best suited for his needs. If multiple clock-domains are bound to a single processor, a simple round-robin scheduling policy is deployed.

A SystemJ program is compiled by the compiler front-end into an AGRC representation all clock-domain. AGRC of each clock-domain, consisting of both CRCF and JCF nodes, i analyzed by Task WCET calculator where every clock-domain’s WCET is calculated (see Section 4.2). The time-analyzable multi-core execution platform employed in this work is introduced in Sectio. We then generate a task graph TG which consists of clock-domains nodes annotated with WCET. This TG is used by design space exploration tool to obtain task allocation for the given types of underlying processor cores, in our case JOPPlus. A generic Architecture Graph (see Section 4.4) along with the TG annotated with WCET of all nodes are input into the design space exploration tool named SystemCoDesigner [Keinert et al. 2009]. The mapping from TG to Architecture Graph (see Section 4.4) together with the objective and constraint specification (see Section 4.5) are also provided to the design space exploration tool. The SystemCoDesigner employs a meta-heuristic optimization approach, in particular an evolutionary algorithm, from the Opt4J {Lukasiewycz, 2011 #27} framework. Here, a front of high-quality solutions (if global optima are found by the heuristic, the solutions are Pareto-optimal) along with the task allocation and schedule are obtained (see Section 5). Then, the designer selects the solution that satisfies all the constraints (i.e. the GRT and the cost in terms of the number of cores). The selected solution which is a scheduled execution model is then automatically transformed into an architecture instance and the corresponding architecture-dependent object code for all cores is generated.

# TDMA

Let us consider four clock-domains C1, C2, C3 and C4 with worst case reaction time of WR1, WR2, WR3 and WR4 having values 20 (5I + 15M), 15 (5I+10M), 25(10I+15M) and 35 (20I + 15M) cycles, respectively. The worst case reaction time is equal to number of instructions falling in the worst case execution path assuming that each instruction takes single cycle to execute while I and M indicates non-memory referenced and memory referenced instructions, respectively. The required run time of the clock-domains should always be greater than the worst case reactions. Let RR1, RR2, RR3 and RR4 are the required run times of each individual clock-domain having values of 30, 40, 45 and 50 respectively. These clock-domains will be executed on a multiprocessor system. The framework is provided with worst case reaction time and required run times as constraints with an objective that these clock-domains are allocated to minimum number of processors. If all clock-domains are allocated on single processor with cyclic execution scheduling policy, then the reaction time of each clock-domain is 175 (=30+40+25+35) which is much higher than the required run time resulting timing violations. Therefore, they cannot be allocated to a single processor. In other extreme case, each clock-domain can be allocated to an individual processor in shared memory multiprocessor system with fair based TDMA interconnection mechanism. The introduction of fair TDMA access introduces delays as each processor has to wait for its slot before it can access the memory. Since there are four slots, each processor will wait for 3 cycles when it comes across a memory access instruction. This results in degraded reaction times of 25, 45, 70 and 80, respectively. This highlights two issues: firstly, too many logic resources (processors) are needed and secondly, the clock-domains are unable to meet the required execution times due to excessive delays introduced by the deployment of TDMA.

As it can be seen that CD 1 and CD2 can be assigned to a single processor as the WR1 + WR2 > MIN (RR1, RR2), the worst case reaction time of all clock-domains is sum of their individual worst case reaction times when executed in a cyclic executive fashion. The remaining clock-domains will be allocated to individual processor as they will be unable to satisfy conditions mentioned above. The frame tries all possible combinations of clock-domain which could allocate maximum number of clock-domains to a single processor as while meeting the required run time at the same time. As a result, the multiprocessor system consists of three processors P1, P2 and P3 with CD1 & CD2 being allocated to P1 and whereas CD2 and CD3 are allocated to P2 and P3, respectively. The reaction times for clock-domains without considering the shared resource are 35, 35, 25 and 35 in the same order for all clock-domains. Assuming shared resources connected through fair based TDMA, the worst reaction times are inflated and become 60, 60, 30 and 50. An analysis of the instructions reveals that the processor P1 needs to access the shared the resources more often than others, therefore, it should be given more time (slots here) to access the shared memory. In other words slots should be according to the memory access requirement of each processor.

The framework provides the weight of each slot so that all the time requirements are met. It comes out with a solution where base cycle has four slots with one slot allocated to P1 and one slot is allocated to each of the other clock-domains.

P1, P1, P2, P3

Since the memory access pattern is not known, having two consecutive slots in a base cycle does not improve the worst case execution time of any of the clock-domains, instead, it worsen the execution of time of CD3 and CD4 which now becomes 45 and 70. If we allocate the slots in such a way such that the distance between two consecutive slots of P1 becomes minimum, this will improve the execution time of the clock-domains running on the P1. The following slot allocation will result in a single wait cycle for P1:

P1, P2, P1, P3

Now the memory access times for all the four clock-domains become 60, 60, 70 and 80, in the order.

# milp formulations for weighted tdma

In this section, we first state the problem formally and then present the MILP constraint formulation for finding the optimal weighted TDMA solution in terms of resource usage while satisfying timing constraints for every clock-domain considering resource binding. In Section 4.1 we…, Section 4.2 presents the optimal constraint formulation (MILP). In Section 4.3 we ….

## Problem definition

The problem that we address consists of binding clock-domains to allocable JOP-Plus cores and obtaining a weighted TDMA scheme such that the minimum hardware resource usage is achieved and the timeliness of the system is guaranteed. Formally, the problem can be stated as follows.

Given:

* A GALS system described by a SystemJ program consisting of a set of asynchronous clock-domains , where denotes a clock-domain that is comprised of parallel reactions running synchronously in its own lockstep.
* An execution platform named GALS-CMP with TDMA based shared memory architecture. The GALS-CMP multi-core architecture is abstracted as a set of JOP-Plus cores , where represents a JOP-Plus core on which single or multiple asynchronous clock-domains are running.
* A TDMA bus access schedule specified as a set of time slot allocation schemes , where each time slot allocation scheme is applied to a unique JOP-Plus core. The representation of time slot allocation scheme is further discussed in Section TODO.
* A resource binding function associating clock-domains of the SystemJ program with JOP-Plus cores of GALS-CMP platform. Each JOP-Plus core is capable of executing any number of clock-domains.
* A set *Worst-Case Execution Time* (WCET) profiles, , for every clock-domain of the SystemJ program, where denotes the WCET profile for a single clock-domain. Each is characterized by a tuple , where represent the Worst-Case Memory Access Time (WCMAT) and denotes the Worst-Case Instruction Execution Time (WCIET) that does not involve memory accessing.
* A set of timing requirements, , for every clock-domain of the SystemJ program, where refers to the timing constraint for . Since every clock-domain is a unique synchronous hard real-time reactive system and the execution of a clock-domain is based on a logical and discrete clock event named *tick*, the timing requirements for is essentially the upper bound of the execution time between any two consecutive ticks, namely *Worst-Case Reaction Time* (WCRT) [TODO]. In other words, the WCRT for should be less or equal to its timing requirement, i.e. .

The objective is to find an optimal resource binding allocating clock-domains to the available JOP-Plus cores as well as an optimal TDMA bus access schedule for every JOP-Plus core used, such that all the timing requirements for every clock-domain are satisfied and the hardware resource usage in terms of the number of the JOP-Plus cores is minimized. Each feasible solution in the design space formed by varied resource binding and TDMA bus access schedule corresponds to a unique real-time characteristic of the system described by SystemJ program, in terms of WCRT estimates for every clock-domain. In this article we employ MILP method to explore the design space efficiently for finding the optimal solution in the matter of resource usage while meeting all the timing requirements. The MILP formulation the discussion about TDMA bus access schedule is given in Section TODO.

## TDMA bus time slot allocation scheme and access scheduling

In this section we discuss the time slot allocation scheme for TDMA based shared memory architecture used in GALS-CMP execution platform. We first give the definition for TDMA time slot allocation scheme a schedulability analysis for TDMA bus access.

As specified in Section TODO, the TDMA bus access schedule is represented as a set of time slot allocation schemes , where is the time slot allocation scheme applied to a unique JOP-Plus core . The execution time spent for to complete a single memory access (either read or write) is determine by TDMA bus access schedule.

**Property 3.1.** For any JOP-Plus core of GALS-CMP execution platform, the worst-case execution time for any single memory access (either read or write), denoted as , is bounded by the longest time interval between two consecutive memory access time slots plus the actual access time.

Due to the fact that is irrelevant to the length of TDMA time slots that are assigned to contiguous memory accesses, in our approach memory access time slots assigned to any JOP-Plus core are not contiguous, in order to simplify TDMA bus scheduling and facilitate schedulability analysis.

Let be a TDMA bus schedule for all the JOP-Plus cores sharing the bus. can be described by the complete set of time slot allocation schemes , where an allocation scheme for JOP-Plus core can be represented as its worst-case execution time for any single memory access, therefore TDMA bus schedule can be formulated as .

We name to be *unit memory access time* (UMAT) for core

Dynamic TDMA schedule, Static schedule

**Theorem 3.2.**

**Property 3.2.** The timing requirement for any JOP-Plus core of GALS-CMP execution platform, denoted as , is the minimum timing requirement of all the clock-domains mapped onto core .

## The MILP formulation for optimal solution

In this section, we present the MILP formulation that gives an optimum solution for the problem defined in Section TODO. We first discuss the TDMA bus schedule scheme for the

In the following description

Table 1 gives the nomenclature of various variables utilized in the formulation and acts as a quick reference for the reader.

Referring to Figure 1,

Integer linear programming (ILP) provides a method to achieve the optimal solution of a problem in which all of its constraints can be formulated as linear constraints of integer variables. Mixed-integer linear programming (MILP) is an extension of ILP for capturing problems with both discrete decisions and continuous variables. MILP provides a systematic approach to describe the complex interaction among resource binding, TMDA bus access scheduling, timing analysis and real-time property verification. Note here the resource binding and the TMDA bus access schedule jointly affects the worst-case execution time for each clock-domain.

*1.3.1 Constants*

Firstly we define the following constants that are derived from the problem definition:

* *Number of the clock-domains* : Let be the number of clock-domains comprising the GALS system described by a SystemJ program.
* *Number of the JOP-Plus cores* : Let be the number of JOP-Plus cores within the GALS-CMP multi-core execution platform.
* *Number of the unit memory access time (UMAT) candidate groups* : Let be the number of UMAT Candidate Groups comprised of sets of UMAT candidates.
* *Number of the unit memory access time (UMAT) candidates*: Let be the number of UMAT candidates forming a UMAT Candidate Group. For the sake of simplicity we assume that all UMAT candidate groups have the same number of UMAT candidates.
* *WCET profiles for every clock-domain* : Let be the set of WCET profiles, where is characterized by a tuple and denotes the WCET profile for clock-domain .
* *Timing requirements* : Let be the set of timing requirements for the SystemJ program, where refers to the WCRT constraint for .
* *Group of UMAT candidate*: TODO TODO Let

*1.3.2 Variables*

In this section, we define the independent and the dependent variables that are used in the formulation.

We define the following base (independent) variables:

* *Binding decision variable matrix* : We define a binary variable such that and ,
* *Unit memory access time (UMAT) Candidate Group decision variable* : We define a binary variable such that ,
* *Unit memory access time (UMAT) decision variable matrix* : We define a binary variable such that , and ,

We define the following derived variables:

* *Core utilization decision variable* : We define a binary variable such that ,
* *Worst-case memory access time variable*: We define an integer variable indicating the total worst-case memory access time for core after resource binding such that , is the sum of the worst-case memory access time (WCMAT) for all the clock-domains mapped onto core . Note here is calculated without considering the TDMA bus sharing.
* *Worst-case instruction execution time variable*: We define an integer variable indicating the total worst-case instruction execution time for core after resource binding such that , is the sum of the worst-case instruction execution time (WCIET) for all the clock-domains mapped onto core
* *Timing Requirement variable*: We define an integer variable indicating the timing requirement for core after resource binding such that , is the minimum timing requirement of all the clock-domains mapped onto core , as specified in Property TODO.

*1.3.3 Objective function*

The objective of the proposed MILP formulation is to minimize the hardware resource usage in terms of the number of the JOP-Plus cores, while satisfying the timing requirements for all the clock-domains mapped onto a set of JOP-Plus cores. The objective function can be stated as follows:

*1.3.4 Constraints*

In this section we elaborate the constraints to which the objective function is subject.

* *Resource binding constraint* : A clock-domain should be mapped onto only one JOP-Plus core. Therefore, for each , i.e.

(1)

* *Core utilization constraint* : If no clock-domain is mapped to core the core utilization decision variable for core should be zero, otherwise should be one. Therefore, for each , i.e.

(1)

For each and

(1)

* *Unit memory access time (UMAT) candidate group selection constraint*: In order to enable schedulable TDMA bus access, only one UMAT candidate group can be selected, as elaborated in Section TODO. Thus,

(1)

* *Unit memory access time (UMAT) candidate constraint*: If the unit memory access time (UMAT) candidate group is not selected, the corresponding UMAT decision variable should be zero. For each and

(1)

If core is utilized, i.e. at least one clock-domain is mapped onto core , only one UMAT value should be chosen from a UMAT candidate group. Therefore, For each ,

(1)

(1)

If core is not utilized, i.e. no clock-domain is mapped onto core , all *UMAT* candidate decision variable should be zero. Therefore, For each and

* *TDMA bus access schedulability constraint*: According to the schedulability analysis in Section TODO, the sum of reciprocal of UMAT should be less than one.

(1)

* *Worst-case memory access time variable constraint*: According to the definition of worst-case memory access time variable*,* For each ,

(1)

* *Worst-case instruction execution time variable constraint*: According to the definition of worst-case instruction executiontime variable*,* For each ,

(1)

* *Timing Requirement variable constraint*: According to the definition of timing requirement variable*,* For each ,

(1)

* *Real-time property constraint*: The timing requirement for each clock-domain should be obeyed after resource binding. Considering TDMA bus sharing for memory access, the worst-case memory access time for core should be multiplied by its unit memory access time (UMAT).Therefore, for each ,

(1)

Efficient formulation of the problem with MILP is not a trivial task.

The objective function of the proposed MILP formulation is to maximize the response time of a certain task while obeying all the constraints. The key variables in the formulation are the schedule times and execution times of all subtasks.

Consequently, the separate analysis of three factors does not guarantee the worst-case coverage. We need to consider all the factors in a single analysis framework. In this regard, we propose to use the MILP method because the complex interaction of the parameters, which are related to scheduling and contention modeling, can be expressed systematically in well-structured linear formulations. A state-of-the-art MILP solver facilitates the efficient computation of those formulations on the basis of sophisticated algorithms such as branch-and-bound, cutting-plane method, and so on [25].

Since the MILP solver basically explores the whole design space to find out an optimal solution, the time complexity is the critical problem for practical use. For the faster evaluation, we have to prune out nonsense solutions by adding problem specific constraints to MILP.

Although the time complexity of an MILP solver is inherently exponential, we observe that how to formulate the design constraints greatly affects the execution time of an ILP solver by orders of magnitude in time-complexity.

## Problem Formulation

# PERFORMANCE EVALUATION

First, we derive operating points for each of the benchmark’s five applications (consumer, auto, office, network, telecom) by performingdesign space exploration using an optimization frameworkwith the objectives of minimizing resource usage andpower consumption, and maximizing speedup (compared tothe operating point with highest latency).

most cases.

# CONCLUSIONS

In this article, ss design.

# TYPICAL REFERENCES IN NEW ACM REFERENCE FORMAT

thesis: [Anisi 2003], an online document / world wide web resource [Thornburg 2001], [Ablamowicz and Fauser 2007], [Poker-Edge.Com 2006], a video game (Case 1) [Obama 2008] and (Case 2) [Novak 2003] and [Lee 2005] and (Case 3) a patent Scientist 2009], work accepted for publication [Rous 2008], ‘YYYYb’-test for prolific author [Saeedi et al. 2010a] and [Saeedi et al. 2010b]. Other cites might contain ‘duplicate’ DOI and URLs (some SIAM articles) [Kirschmer and Voight 2010]. Boris / Barbara Beeton: multi-volume works as books [Hörmander 1985b] and [Hörmander 1985a].

APPENDIX

In this appendix, we measure the channel switching time of Micaz [CROSSBOW 2008] sensor devices. In our experiments, one mote alternatingly switches between Channels 11 and 12. Every time after the node switches to a channel, it sends out a packet immediately and then changes to a new channel as soon as the transmission is finished. e measure the number of packets the test mote can send in 10 seconds, denoted as 1. In contrast, we also measure the same value of the test mote without switching channels, denoted as N2. We calculate the channel-switching time s as



By repeating the experiments 100 times, we get the average channel-switching time of icaz motes: 24.3 *μ*s.

ELECTRONIC APPENDIX

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

ACKNOWLEDGMENTS

The authors would like to thank Dr. Maura Turolla of Telecom Italia for providing specifications about the application scenario.

REFERENCES

ANDREI, A., ELES, P., PENG, Z. AND ROSEN, J. 2008. Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *Proceedings of the Proceedings of the 21st International Conference on VLSI Design*2008 IEEE Computer Society, 1339374, 103-110.

BEHRMANN, G., DAVID, A. AND LARSEN, K.G. 2004. A tutorial on uppaal. In *Formal methods for the design of real-time systems* Springer, 200-236.

BERRY, G. 1999. *The Esterel v5 language primer*.

BOWMAN, H. 2006. Process Calculi: LOTOS. In *Concurrency Theory* Springer London, 19-54.

CHAPIRO, D.M. 1985. Globally-asynchronous locally-synchronous systems (performance, reliability, digital) Stanford University, 136.

GRUIAN, F., ROOP, P., SALCIC, Z. AND RADOJEVIC, I. 2006. The SystemJ approach to system-level design. In *Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings. Fourth ACM and IEEE International Conference on*, 149-158.

HALBWACHS, N. 2005. A synchronous language at work: the story of Lustre. In *Proceedings of the Proceedings of the 2nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design*2005 IEEE Computer Society, 1322140, 3-11.

HENNESSY, J.L. AND PATTERSON, D.A. 2003. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc.

HOARE, C.A.R. 1985. *Communicating sequential processes*. Prentice-Hall, Inc.

JEBALI, F., LANG, F. AND MATEESCU, R. 2014. GRL: A Specification Language for Globally Asynchronous Locally Synchronous Systems. In *Formal Methods and Software Engineering*, S. MERZ AND J. PANG Eds. Springer International Publishing, 219-234.

JOSEPH, M. AND PANDYA, P. 1986. Finding Response Times in a Real-Time System. *The Computer Journal 29*, 390-395.

KOPETZ, H. 1997. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers.

LAVAGNO, L. AND SENTOVICH, E. 1999. ECL: a specification environment for system-level design. In *Proceedings of the Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, New Orleans, Louisiana, USA1999 ACM, 309989, 511-516.

LE GUERNIC, P., TALPIN, J.-P. AND LE LANN, J.-C. 2003. POLYCHRONY for System Design. *Journal of Circuits, Systems and Computers 12*, 261-303.

LI, Y.T.S., MALIK, S. AND WOLFE, A. 1996. Cache modeling for real-time software: beyond direct mapped instruction caches. In *Real-Time Systems Symposium, 1996., 17th IEEE*, 254-263.

LI, Z., MALIK, A. AND SALCIC, Z. 2014. TACO: A scalable framework for timing analysis and code optimization of synchronous programs. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on* IEEE, 1-8.

LIU, C.L. AND LAYLAND, J.W. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM 20*, 46-61.

LUKASIEWYCZ, M., GLA, M., #223, REIMANN, F., #252 AND TEICH, R. 2011. Opt4J: a modular framework for meta-heuristic optimization. In *Proceedings of the Proceedings of the 13th annual conference on Genetic and evolutionary computation*, Dublin, Ireland2011 ACM, 2001808, 1723-1730.

LUKASIEWYCZ, M., GLASS, M., HAUBELT, C. AND TEICH, J. 2008. Efficient symbolic multi-objective design space exploration. In *Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific*, 691-696.

MALIK, A., GIRAULT, A. AND SALCIC, Z. 2011. A GALS Language for Dynamic Distributed and Reactive Programs. In *Application of Concurrency to System Design (ACSD), 2011 11th International Conference on*, 173-182.

MALIK, A., SALCIC, Z., ROOP, P.S. AND GIRAULT, A. 2010. SystemJ: A GALS language for system level design. *Comput. Lang. Syst. Struct. 36*, 317-344.

MILNER, R. 1982. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc.

MOUSAVI, M.R., GUERNIC, P.L., TALPIN, J.-P., SHUKLA, S.K. AND BASTEN, T. 2004. Modeling and Validating Globally Asynchronous Design in Synchronous Frameworks. In *Proceedings of the Proceedings of the conference on Design, automation and test in Europe - Volume 1*2004 IEEE Computer Society, 969070, 10384.

NADEEM, M., BIGLARI-ABHARI, M. AND SALCIC, Z. 2011. GALS-JOP – A Java Embedded Processor for GALS Reactive Programs. In *EmbeddedCom 2011*, Sydney, Australia.

NADEEM, M., BIGLARI-ABHARI, M. AND SALCIC, Z. 2012. JOP-Plus - A Processor for Efficient Execution of Java Programs Extended with GALS Concurrency. In *ASPDAC*.

NADEEM, M., BIGLARI-ABHARI, M. AND SALCIC, Z. 2012. JOP-plus - A processor for efficient execution of java programs extended with GALS concurrency. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, 17-22.

NADEEM, M., PARK, H., LI, Z., BIGLARI-ABHARI, M. AND SALCIC, Z. 2013. GALS-CMP: chip-multiprocessor for GALS embedded systems. In *Proceedings of the Proceedings of the 26th international conference on Architecture of Computing Systems*, Prague, Czech Republic2013 Springer-Verlag, 2450040, 147-158.

PAOLIERI, M., #246, MISCHE, R., METZLAFF, S., GERDES, M., QUI, E., #241, ONES, UHRIG, S., UNGERER, T. AND CAZORLA, F.J. 2013. A hard real-time capable multi-core SMT processor. *ACM Trans. Embed. Comput. Syst. 12*, 1-26.

PITTER, C. AND SCHOEBERL, M. 2007. Towards a Java Multiprocessor. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)* ACM Press, 144-151.

POLETTI, F., BERTOZZI, D., BENINI, L. AND BOGLIOLO, A. 2003. Performance Analysis of Arbitration Policies for SoC Communication Architectures. *Design Automation for Embedded Systems 8*, 189-210.

PUSCHNER, P. AND BURNS, A. 2000. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems 18*, 115-128.

ROSEN, J., ANDREI, A., ELES, P. AND PENG, Z. 2007. Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *Proceedings of the Proceedings of the 28th IEEE International Real-Time Systems Symposium*2007 IEEE Computer Society, 1338641, 49-60.

SCHOEBERL, M. 2005. JOP: A Java Optimized Processor for Embedded Real-Time Systems Vienna University of Technology.

SCHOEBERL, M. 2007. SimpCon - a Simple and Efficient SoC Interconnect. In *Proceedings of the 15th Austrian Workhop on Microelectronics, Austrochip 2007*.

SCHOEBERL, M. 2008. A Java processor architecture for embedded real-time systems. *J. Syst. Archit. 54*, 265-286.

SCHOEBERL, M., PUSCHNER, P. AND KIRNER, R. 2009. A Single-Path Chip-Multiprocessor System. In *Software Technologies for Embedded and Ubiquitous Systems*, S. LEE AND P. NARASIMHAN Eds. Springer Berlin Heidelberg, 47-57.

SCHRANZHOFER, A., PELLIZZONI, R., CHEN, J.-J., THIELE, L. AND CACCAMO, M. 2010. Worst-case response time analysis of resource access models in multi-core systems. In *Proceedings of the Proceedings of the 47th Design Automation Conference*, Anaheim, California2010 ACM, 1837359, 332-337.

THEILING, H., FERDINAND, C. AND WILHELM, R. 2000. Fast and Precise WCET Prediction by Separated Cache andPath Analyses. *Real-Time Syst. 18*, 157-179.

THIELE, L. AND WILHELM, R. 2004. Design for Timing Predictability. *Real-Time Syst. 28*, 157-177.

WILHELM, R., ENGBLOM, J., ERMEDAHL, A., HOLSTI, N., THESING, S., WHALLEY, D., BERNAT, G., FERDINAND, C., HECKMANN, R., MITRA, T., MUELLER, F., PUAUT, I., PUSCHNER, P., STASCHULAT, J., STENSTR, P. AND #246 2008. The worst-case execution-time problem&mdash;overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst. 7*, 1-53.

WOLF, F., STASCHULAT, J. AND ERNST, R. 2002. Associative caches in formal software timing analysis. In *Proceedings of the Proceedings of the 39th annual Design Automation Conference*, New Orleans, Louisiana, USA2002 ACM, 514076, 622-627.

ZHENMIN, L., MALIK, A. AND SALCIC, Z. 2014. TACO: A scalable framework for timing analysis and code optimization of synchronous programs. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, 1-8.