An Interval Algebra for Multiprocessor Resource Allocation

Leandro Soares Indrusiak and Piotr Dziurzanski
Department of Computer Science, University of York, Deramore Lane, Heslington, York, YO10 5GH, UK.
{Leandro.Indrusiak, Piotr.Dziurzanski}@york.ac.uk

***Abstract*—This paper presents an interval algebra created specifically to evaluate timing properties of multiprocessor systems. It models the application load as intervals, and considers allocation and scheduling as algebraic operations over those intervals, aiming to analyse the impact of resource allocation decisions on application response times or schedulability. The theoretical background is introduced informally, followed by the description of a reference implementation of the interval algebra in C++, aiming to appeal to the design practitioner rather than the formalist. Examples of the usage of the proposed algebra are also provided, showing its applicability to the performance evaluation of industrial systems implemented over bus-based and Network-on-Chip multiprocessor platforms. A particular design flow is highlighted, where the interval algebra is used as a fitness function in a genetic algorithm tailored to optimise resource allocation in hard real-time multiprocessors.**

I. Introduction

A multiprocessor system is a composite of computation, communication and storage resources, and each of them contributes to the overall timing behaviour of the system as it processes application load. The way the application load is allocated to those resources has crucial impact to its performance and timeliness. Thus, resource allocation is an increasingly important part of the design flow of such systems, specially as the number of processors keeps increasing in both embedded and high-performance domains.

Resource allocation is a well known problem and most of its formulations belong to the NP-hard class [3]. For current multiprocessor systems with hundreds of communicating tasks, dozens to hundreds of processors and sophisticated interconnects, it is not practical to optimally solve resource allocation problems. Heuristic solutions are currently the state-of-the art, trying to achieve acceptable allocations by sampling only a small subset of the very large solution space of such problems [11]. To find an acceptable solution, models of the application load and the multiprocessor platforms can be used to forecast performance metrics under different allocation alternatives. Such models must be expressive enough to describe diverse system architectures, load patterns, resource constraints and timing requirements.

In this paper, we propose an interval algebra (IA) that models all those aspects and uses them to evaluate the timeliness of multiprocessor systems under different resource allocations. It models the application load using the mathematical notion of intervals, which are used to denote the amount of time an application component (e.g. computational task, communication message) uses a specific multiprocessor platform resource (e.g. CPU, communication bus). It then models the allocation and scheduling of those application components as algebraic operations over those intervals. By applying those algebraic operations, it is possible to obtain performance figures such as the response times of the application components, which can be in turn aggregated to obtain e.g. average and worst-case performance.

To establish the need for the proposed algebra, we review a number of performance analysis formalisms, aiming to show that while very useful in their specific domains they cannot be easily integrated into a common framework to analyse different kinds of application load (e.g. periodic or aperiodic, with or without dependencies between tasks) or multiprocessor platform (e.g. homogeneous or heterogeneous processors, shared or distributed memory). We then informally present the foundations of the proposed algebra through examples, using a simple yet consistent ASCII-based notation, as our aim is to be appealling to design practitioners rather than to formalists. This is followed by the description of our first reference implementation of the algebra, which follows object-oriented principles and provides an API of classes and methods to solve interval-algebraic representations of multiprocessor systems under load. By solving valid expressions of the proposed algebra, designers can investigate the timeliness of applications running over multiprocessors under different resource allocation policies, enabling them to design systems that can meet the application's timing constraints. Furthermore, the compact and simple implementation of the algebra can also be used during runtime to guide resource allocation in dynamic systems, where the application load is not known at design time and must be allocated on demand. The paper is then closed with a number of examples of the applicability of the proposed algebra and a discussion of its current and potential use.

II. Related Work

The inherent complexity of contemporary multiprocessor systems limits the possibility of a complete analysis using simulations, thus analytical models are of increasing importance as part of modern system design [12]. In this section, a

The research leading to this work has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 611411.

number of widely used analytical models are briefly reviewed.

Queueing theory can be applied to systems in which customers come to be processed by a service facility. When a customer arrives but there is no idle server, it waits in a queue. The interarrival time and service time are specified probabilistically. Using a queueing model, a number of vital performance measures can be estimated, such as a number of customers waiting in the system for service, a mean queue length, waiting time in a single queue or the whole system, duration of a busy period, etc. Queueing theory is often used in communication networks [1], including Networks on Chip [11], where customers represent packets transmitted between nodes. An example of queueing-theory-based model for evaluating energy dissipation in a network node can be found in [13]. The main feature preventing direct queueing theory application in the real-time domain is related with its strictly probabilistic nature. It allows computing average quantities in an equilibrium state and is not intended to infer about the worst-case behaviour.

Network Calculus is a theory of deterministic queuing systems that has been proposed by Cruz in [7]. It is an alternative approach to queueing theory, using upper bounds to characterize arrivals and lower bounds to describe services. Using this approach it is easy to compute the bounds of network performance metrics, such as a delay or backlog [4]. Network Calculus describes data flows by the cumulative function, $R(t)$, being a number of the bits transmitted in a particular data flow during time interval $[0, t]$. Both continuous and discrete time models can be applied. Since bounds obtained with Network Calculus hold with probability 1, they can be treated as the worst-case values, essential to evaluate schedulability in hard real-time systems. To compute a delay or backlog for an average case, a stochastic extension of network calculus has been introduced [10]. It can be applied to provide particular stochastic (soft) guarantees [6].

Network Calculus has been extended to Real-Time Calculus in [5]. It uses upper and lower arrival curves as functions bounding the amount of the events arriving in a time interval. These events may be treated as task arrivals. Using this notation, it is possible to represent periodic or sporadic tasks. A number of schedulability tests based on Real-Time Calculus have been proposed in [14]. Furthermore, Real-Time Calculus has been extended to globally-scheduled multiprocessor systems in [14]. In [9] the modelling capability of Real-Time Calculus has been extended with the execution of a task triggered by events on multiple input event streams using OR-activation, AND-activation, or their combination. Despite relatively large research on Real-Time Calculus and its vast applicability, there is still lack of efficient methods for determining response-time bounds in case they are unspecified. The multiprocessor case is not fully compatible with uniprocessor Real-Time Calculus. The pessimism introduced by applying real-time calculus methods has not been assessed in [14].

Schedulability analysis is a formalism to evaluate the timing properties in real-time systems introduced in [15]. In this technique, a workflow is usually given as a set of independent periodic or sporadic tasks, where each task is defined as a tuple with its worst-case execution time, relative deadline, and priority. With this tuple, it can be verified if all tasks mapped to a particular processor, do not exceed the processors capacity. An example formula of direct schedulability tests, which checks task response time, is proposed in [2]. The majority of schedulability analysis research assumes task independence. A solution considering control and data dependencies, presented in [17], requires relatively complicated system modelling by means of conditional process graphs. Earlier proposals of dealing with dependencies included using even more sophisticated techniques like appropriate release jitters, or time offsets of phases [16]. Lacking of any simple extension to application models with dependent or single appearance tasks can be viewed as a strong disadvantage.

An exhaustive survey of hard real-time scheduling analysis for multiprocessor systems has been presented in [8]. The number of already proposed schedulability tests can be considered as rather high. Nearly each of these tests has a different capability or applicability. Since schedulability analysis is dedicated to hard real-time systems, it is not easily applicable when soft timing constraints are assumed. In the case of periodic and sporadic intervals, schedulability analysis can be easily integrated to the proposed algebra as one restricted type of algebraic transformation.

From this survey it follows that there is not a single analytical model that is expressive enough to describe various application dependency patterns, with different task temporal behaviour, capable of representing resource affinities and applicable to both soft and hard-real time systems. We foresee, however, the increasing need for such a formalism, as high-performance and cloud computing become more time-critical and start to be seen as soft real-time systems, and hard real-time systems become more dynamic and can only provide schedulability guarantees during runtime for a finite time horizon.

III. Interval Algebra Principles

An algebra is a definition of symbols and the rules for manipulating those symbols. An interval algebra (IA) therefore establishes rules for the manipulation of intervals. The proposed IA defines different types of intervals, which represent the amount of time a particular application component requires from a notional platform component. It also defines rules for the manipulation of such intervals: what happens when an interval is allocated to a specific type of resource, what if two intervals are allocated to the same resource, etc. Two basic algebraic operations are needed: time displacement and partition. Time displacement changes the endpoints of an interval by an arbitrary value X, and denotes that the application component had to wait for its allocated resource (i.e. its starting and ending times were moved X time units to the future). Partition simply breaks one interval in two, and denotes that an application component was preempted from a resource (and the second interval produced by the partition likely to be time-displaced). All other interval-algebraic operations of

the proposed IA, which can represent an arbitrarily large set of allocation and scheduling mechanisms, can be expressed as compositions of those two. By applying those operations, it is possible to investigate the impact of different resource allocation and scheduling mechanisms on the endpoints of the intervals, which in turn denote the completion times of each application component.

Throughout this paper an application is viewed as a set of *tasks* (a taskset). The tasks appearing exactly once during the application execution are often referred to as *singletons* and are composed of a single *job*. A periodic or sporadic task can be treated as an infinite series of jobs that are released periodically or less often than the provided inter-release time, respectively.

Let us consider a simple example. A given application is composed of three singleton tasks: A, B and C, and a given homogeneous platform is composed of two processors with the first-in-first-out (FIFO) scheduling. Each of the tasks can be represented by an interval that denotes the time it needs to run using one of the platform processors: $A = [0, 30)$, $B = [0, 45)$, $C = [0, 20)$ (assuming in this example that A, B, C are all independent and ready to run at $time = 0$). By using simple interval algebra operations, a resource allocation heuristic can estimate the response time R of the three tasks under different allocation schemes (e.g. $R_A = 30$, $R_B = 45$ and $R_C = 50$ if A and C are allocated, in that order, to one of the processors and B is allocated to another), and thus can dynamically decide whether it is likely to meet the application's constraints when using a given allocation. While trivial, such example can be made arbitrarily complex by allowing different resource scheduling disciplines, a larger number of tasks and processors. For the proposed algebra, however, the analysis of the response times under a specific allocation would still involve the application of the same interval manipulation rules.

The advantages of such an approach are numerous, including the following.

- It enables dynamic allocation heuristics to have an appropriate level of confidence on whether the chosen allocation meets the applications constraints.
- The approach can be used as a fitness function of search-based allocation heuristics, if the algebraic operations are sufficiently lightweight as they have to be applied over a potentially large search space (some examples of applying IA to genetic algorithms are provided in Section V).
- The solution of algebraic operations can be found in multiple ways, with different levels of performance. Therefore, resource allocation heuristics can be improved simply by optimising the solution of the employed algebraic operations.
- If absolute predictability is not required (i.e. in soft real-time and best-effort applications), algebraic operations can be solved faster by applying approximations that sacrifice the accuracy of the final result. This enables applying of heuristics that can be applied to systems with different levels of strictness of their timing requirements.

In the following subsections, we briefly introduce the main features of the application modelling approach based on the proposed interval algebra from different aspects, such as: modelling application architecture with respect to various dependency patterns, modelling diverse temporal behaviour, including periodicity, modelling task affinities to certain resources, and also describing assorted loads.

A. Modelling Application Architecture

Using IA, application jobs are represented as intervals. For example, a singleton task can be represented by the time interval it requires from a notional resource. It can be denoted with the notation[2] exemplified below:

$$\#A\#0\#40 \tag{1}$$

where the first element of the tuple is a unique job identifier, the second is a non-negative real number representing the release time of the job and the third is a positive real number representing the job's load, i.e. the actual length of the time interval. In the example above, the job A is released at time 0 and requires 40 time units of a resource. The same concept can also be represented using the mathematical notation for a left-closed right-open bounded interval $[0, 40)$.

Such interval-based representation of a job is sufficient to express a singleton, and by using a set of intervals, independent jobs can be also represented. To denote a dependency between two tasks A and B, the notation can be extended to include a job identifier instead of the release time of a job:

$$\#B\#A\#50 \tag{2}$$

This notation is capable of denoting single dependency jobs, and conveys that interval B's first endpoint depends on interval A. Multiple dependencies can also be specified as a dependency set, and thus multi-dependency jobs can be covered:

$$\#C\#\{A, B\}\#260 \tag{3}$$

This notation assumes that whenever an interval has dependencies, its first endpoint lies exactly at the highest second endpoint among all the intervals it depends on. In this example, assuming that tasks A and B are defined as in formulas (1) and (2), this leads to: $A = [0, 40), B = [40, 90), C = [90, 350)$.

B. Modelling Application Temporal Behaviour

The intervals described in the previous subsection are single-appearance and have a fixed release time, therefore express singleton tasks. A strictly periodic series of jobs can be characterised by its release time, the period after which a new job is released, and the time interval each job requires from a notional resource. We denote such job series with the notation exemplified below, which is exactly the same as the notation of a singleton task followed by the period:

$$\#P\#0\#40\#100 \tag{4}$$

[2]The formal description of interval algebra grammar, specified with Extended Backus-Naur Form (EBNF), can be found at https://www.cs.york.ac.uk/rts/rtslab/wiki.

Mathematically, it represents an infinite series of intervals, such as: $P = [0, 40), [100, 140), [200, 240), \ldots$. This extension is expressive enough to represent strictly periodic tasks.

The release time of sporadic tasks is not deterministic but has well defined bounds. In case of aperiodic tasks, those bounds do not exist. To model those cases, we can represent release times with so-called *aleatory variables*. Those variables are associated with probability distributions that can constrain assumed values. The interval algebra notion does not impose any limitation on the choice of probability distributions. Their parameters should be provided following the usual notation. For example, a normal distribution $\mathcal{N}(\mu, \sigma^2)$ with parameters mean $\mu = 2$ and variance $\sigma^2 = 1$, $\mathcal{N}(2, 1)$ can be used to denote the release time of task R, and similarly $\mathcal{N}(40, 1)$ can denote its execution time:

$$\#R\#normal(2,1)\#normal(40,1) \tag{5}$$

The time when R finishes its execution is described by the convolution of two Gaussians: $\mathcal{N}(2, 1) * \mathcal{N}(40, 1)$.

C. Modelling Application Resourcing Constraints

A resource can be represented by an algebraic operation over all the jobs mapped onto it, each represented by its respective interval. The algebraic operation determines how the resource is shared between the jobs mapped to it, and how the sharing affects their timings. We denote a resource with the notation exemplified below:

$$+\Pi_1(\#A\#0\#40) \tag{6}$$

where the algebraic operation Π_1 is applied to the set of intervals surrounded by brackets (only A in the example above). The example below shows the same resource, but this time with two distinct jobs mapped to it:

$$\begin{aligned}&+\Pi_1(\#A\#0\#40, \#B\#0\#50) = \\ &+\Pi_1(\#A\&40, \#B\&90) = \\ &+\Pi_1([0, 90))\end{aligned} \tag{7}$$

In this example, we introduce two different ways to evaluate the operator Π_1 (which we can intuitively understand as a resource serving jobs under a FIFO schedule). The first evaluation of the operator preserves the identities of the mapped jobs, and it indicates the completion times of each one of them after the symbol "&". We will refer to this type of evaluation as *information-preserving* (or simply *preserving*). The second way to evaluate the operator is equivalent to the first, but it does not preserve any information about the individual operands. It simply determines the busy period(s) of the resource with one or more intervals. We refer to this type of evaluation as *information-collapsing* (or simply *collapsing*).

A slightly different example is shown below, using the same jobs but this time mapped onto resource Π_2 that uses a time-division multiplexing (TDM) scheduler with a quantum of 8 time units:

$$\begin{aligned}&+\Pi_2(\#A\#0\#40, \#B\#0\#50) = \\ &+\Pi_2(\#A\&72, \#B\&90) = \\ &+\Pi_2([0, 90))\end{aligned} \tag{8}$$

It is worth noticing that only the intermediate expression (i.e. after the preserving evaluation) differs, and the final result after the collapsing evaluation is the same. This is always the case if the operand denote a work-preserving scheduler, when no processor is idle as long as there are tasks ready to be executed.

The two following examples show jobs mapped onto a resource that is shared under a priority-preemptive scheduler, assigning priorities in the same order the jobs are passed to the operator (higher to lower):

$$\begin{aligned}&+\Pi_3(\#C\#15\#40, \#D\#10\#50, \#E\#0\#50) = \\ &+\Pi_3(\#C\&55, \#D\&100, \#E\&140) = \\ &+\Pi_3([0, 140))\end{aligned} \tag{9}$$

$$\begin{aligned}&+\Pi_4(\#F\#10\#4, \#G\#0\#18, \#H\#26\#5, \\ &\#I\#24\#8) = \\ &+\Pi_4(\#F\&14, \#G\&22, \#H\&31, \#I\&37) = \\ &+\Pi_4([0, 22), [24, 37))\end{aligned} \tag{10}$$

In both cases, the algebraic operations abstracts away the specific interleaving patterns of the execution of each job. Each of the evaluation types focusses solely on, respectively, the finish times of each job or the idleness of the resource. For example, (10) represents the following: task G starts to be executed at time zero, but after 10 time units it is preempted by task F which runs t completion for 10 time units; then G resumes and runs for its remaining execution time until time equals 22 units; resource Π_4 becomes idle until task I is released at 24 time units, which in turn executes until time equals 37 units.

Just like single appearance jobs, periodic jobs can be mapped to resources:

$$\begin{aligned}&+\Pi_1(\#A\#0\#40\#100, \#B\#0\#50) = \\ &+\Pi_1(\#A\&40, \#B\&90, \#A\#100\#40\#100) = \\ &+\Pi_1([0, 90), \#A\#100\#40\#100)\end{aligned} \tag{11}$$

It is important to notice that a periodic job series always remains as a distinct interval in the result of both preserving and collapsing evaluations of an operator. This reflects the infinite nature of the series.

One of crucial properties of each task is a list of resources that can execute this task. The task that can be executed on any resource available in a system is referred to as *untyped task*. If a task can be executed on a single type of resources only, it is a *single-typed* task. A *multi-typed* task can be executed on a few (enumerated) resource types, possibly with different execution time. In all the earlier examples, untyped tasks have

been presented. To describe a single-typed or multi-typed task, the notation should support the definition of different types of resources and different types of resource affinity. This can be expressed as follows, where each scalar in pointy brackets denotes a different type and the absence of type constraints implies untyped jobs or resources (as earlier):

$$+X < 2 > (\#J < 2 > \#0\#15, \#K < 2, 3, 8 > \#0\#20, \#L\#0\#14) \quad (12)$$

By allowing the definition of resources types and resource requirements, it is also possible to present communicating jobs by modelling the job as two fully dependent intervals with distinct resource requirements, one for computation and one for communication (i.e. the job can only communicate over resource 2 once it has finished being computed over resource 1):

$$\begin{aligned} &\#L < 1 > \#0\#14 \\ &\#M < 2 > \#L\#340 \end{aligned} \quad (13)$$

D. Modelling Application Load Characterisation

The representation of load as the interval length, denoted by a positive real number (as defined in subsection III-A), is already capable of representing a fixed load.

To represent a typed fixed load, we allow the specification of different interval lengths for different resource types using a similar notation as the one introduced at the end of subsection III-C:

$$\#M < 2, 4, 6 > \#0\# < 10, 20, 20 > \quad (14)$$

To represent a probabilistic load or typed probabilistic load, we have to rely again on aleatory variables to represent the load. This can be done for both typed and untyped jobs.

IV. Application Modelling Using Interval Algebra Reference Implementation

In this section, a brief description of the reference implementation of the proposed interval algebra is provided. Its software architecture follows the principles of object-orientation and object-oriented frameworks, allowing for further extensions through inheritance. It has been implemented in C++ language.

The most important classes of the interval algebra reference implementation are presented in Figure 1. Among these classes four clusters can be identified:

- related with various notion of time (*Time*[1], `TimeDeterministic`, `TimeStochastic`),
- related with jobs to be allocated (`Job`, `JobTree`, `JobTreeNode`, `TreeNode`),
- related with various policies of scheduling (*Scheduler*, `SchedulerTDM`, `SchedulerFIFO`, `SchedulerPriorityTDM`, `SchedulerPriorityNonPreemptive`, `SchedulerPriorityPreemptive`),

[1]Classes written in italic are abstract.

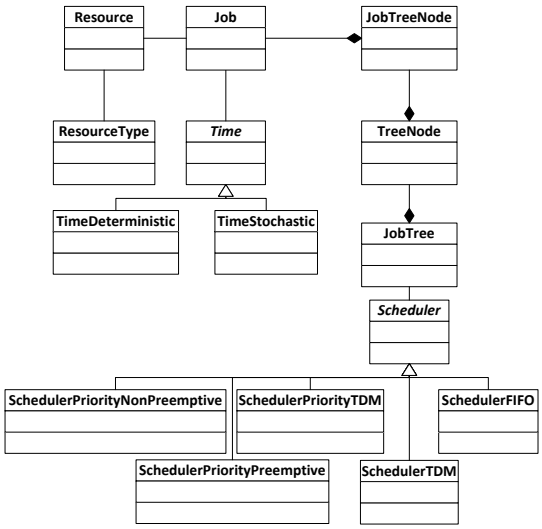

Fig. 1. Main classes of the reference implementation of the interval algebra and their dependencies

- related with hardware platforms (`Resource`, `ResourceType`).

In this section, we demonstrate simple examples (written in C++) of the interval algebra reference implementation usage.

An application is modelled by defining parameters of its jobs. After this stage, each job shall be added to a job list held by a scheduler. In the examples below we assume that a scheduler object has been already created, for example a scheduler with the FIFO policy (`SchedulerPriorityFIFO` class), but any other schedulers derived from abstract class `Scheduler` can be used instead. The scheduler can be instantiated in the same manner as any other C++ object: `SchedulerFIFO *MySchedulerFIFO=new SchedulerFIFO;`.

Applications are modelled with instances of class `Job`. The constructor of this class requires the job name as a parameter (string). Then the required parameters of the job are set by means of executing the member functions of the `Job` class. These parameters include: release time, period, deadline, parent jobs, resources the job can be executed on, execution time for each possible resource and the job priority.

Applications that are composed of a single job can be modelled by creating an instance of class `Job`. In the example below, the task described with IA by formula (1), i.e. a singleton task named *A* is defined, released at 0ms (the default time unit), of the execution time equal to 40ms on the default resource.

```
Job *A = new Job("A");
Time *TimeReleaseA = new TimeDeterministic(0);
A->SetReleaseTime(TimeReleaseA);
Time *TimeA=new TimeDeterministic(40);
A->SetExecutionTimeForExecutingResource(TimeA);
MySchedulerFIFO->AddJob(A);
```

There is a possibility of modelling applications that are

composed of an arbitrary number of single-dependency jobs, i.e. the jobs that can depend on one and only one other job. The preceding job, whose execution is required before the execution of the given job, is set with member function `Job::AddDependency(Job*)`. For example, the task given by formula (2) can be described with the following code (assuming job A has been already created).

```
Job *B = new Job("B");
Time *TimeB=new TimeDeterministic(50);
B->SetExecutionTimeForExecutingResource(TimeB);
B->AddDependency(A);
MySchedulerFIFO->AddJob(B);
```

A strictly periodic task is comprised of a series of jobs with release times separated by a constant time interval. Internally, a periodic task is split into a series of single appearance jobs. The number of instances is set so that the release time of no instance is higher than the provided time. The pointer to this time is given as the second parameter of member function `Scheduler::AddPeriodicJobs`, whereas the first parameter is, similarly to the single appearance job, the pointer to the Job object itself. In the example below, representing the task described with formula (4), this time is named `MaxTime`. We assume the task has an implicit deadline, i.e. the relative deadline of each job is equal to its period. However, any other positive value can be used instead.

```
Time *MaxTime = new TimeDeterministic(300);
Job *P =new Job("P");
Time *TimeReleaseP = new TimeDeterministic(0);
P->SetReleaseTime(TimeReleaseP);
Time *TimeP=new TimeDeterministic(40);
P->SetExecutionTimeForExecutingResource(TimeP);
Time *PeriodP = new TimeDeterministic(100);
P->SetPeriod(PeriodP);
Time *DeadlineP = new TimeDeterministic(40);
P->SetDeadline(DeadlineP);
MySchedulerFIFO->AddPeriodicJobs(P,MaxTime);
```

The affinity of applications defines which kind of resources a given job requires for its execution. In order to be used with the interval algebra, each resource has to be defined and instantiated. For example, to create a resource named *Processor1*, the following line of code should be written: `Resource *Processor1=new Resource("Processor1");`. Then, to allow a job to be executed on this resource, one should use the `Job::AddExecutingResource(Resource*)` with a pointer to the resource as the parameter, for example: `A->AddExecutingResource(Processor1);`.

To set the job execution time for a particular resource, one should use member function `Job::SetExecutionTimeForResource(Time*, Resource*)`.

V. Examples and Experiments

In this section, some capabilities of the proposed algebra are presented using Bosch's DemoCar benchmark, a lightweight engine control system composed of 43 tasks (basic execution units) and 71 labels (memory locations of given lengths) for inter-task communication.

The actual computation time of tasks is not known a priori, only its lower and upper bounds are provided together with a probability distribution function representing the likelihood of the values inbetween them. Since DemoCar contains hard real-time constraints, we use the worst-case execution time (WCET) to determine the length of the intervals representing tasks. To illustrate this issue, let us present an IA formula of one arbitrary task of this benchmark, for example *CylNumObserver*. This periodic task is released every 10000μs and its execution time is described with a Weibull distribution with parameters $\lambda = 7534.51$ and $k = 1.51$. However, the WCET is also specified to be equal to 440μs and this value is employed in formula $CylNumObserver\#0\#440\#10000$ that is used for response time evaluation. For soft real-time systems, an aleatory variable could be created for execution time and it would be used to determine probability distribution function of the system response time.

A. Number of Processors and Scheduler Selection

In the first experiment we present how interval algebra can be used to choose an appropriate number of processors and a scheduling discipline for a particular system so that no deadline is violated. Firstly, we model a simple bus-based architecture with the number of processors ranging from 1 to 5, where data transfer overheads from and to a shared memory has been assumed to be negligible (i.e. a contention on the bus is not modelled).

Let us compare the influence of various schedulers in the DemoCar example. For this relatively simple case, the obtained makespan (aka response time) of the whole taskset does not depend on the chosen scheduler type, and is depicted in Figure 2 for processor number ranging from 1 to 5. However, the number of missed deadlines varies significantly for different scheduler types, as presented in Figure 3. For TDM in the single processor system and quantum 100μs, 42 out of 43 deadlines are missed, whereas with the remaining scheduler types only about 20 tasks have been executed on time assuming WCET. For priority schedulers (where priorities of tasks have been assigned statically depending on the task deadline - the lower deadline, the higher priority), three processors are sufficient to meet all the deadlines, whereas for the FIFO scheduler one deadline remains violated even in the 5-processor system. This simple experiment shows both the significant influence of the chosen scheduler as well as capabilities of the interval-algebra-based evaluation.

The proposed technique can be also applied to more sophisticated platform architectures, such as a mesh Networks on Chip (NoC). In contrast with the previous case, data transfer overhead has been taken into consideration, assuming constant time for transferring a single flit (flow control digit, a small piece of a packet to be transferred) between two neighbouring nodes if no contention is present. Each link is used as a single resource, so for example to transfer one data from $Processor_{0,1}$ to appropriate sink $Processor_{2,0}$ we need such resources allocated simultaneously: $Processor_{0,1} - Router_{0,1}$, $Router_{0,1} - Router_{1,1}$, $Router_{1,1} - Router_{2,1}$,

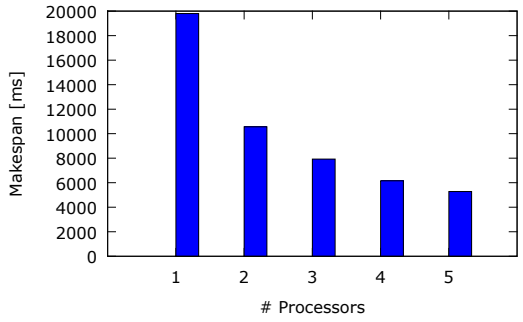

Fig. 2. Makespan for DemoCar use case with different number of processors

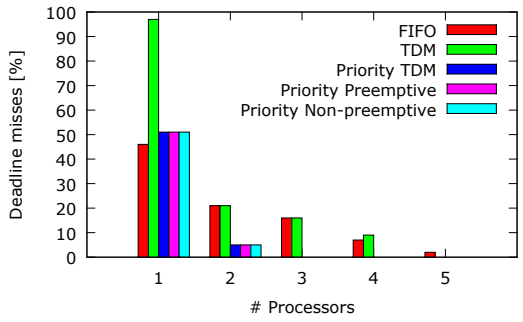

Fig. 3. Deadline misses for DemoCar use case with different schedulers and number of processors

$Router_{2,1} - Router_{2,0}$, $Router_{2,0} - Processor_{2,0}$, as shown in Figure 4. Communicating tasks can be allocated in different processors, resulting in potentially large transmission overheads.

Both processor functionalities and labels have been assigned with the round-robin order, presumably far from being optimal (finding an optimal mapping belongs to the NP-hard problems and thus is intractable [3]). The makespans for a few different NoC sizes executing the DemoCar example with the FIFO scheduler are presented in Figure 5. In line with our expectations, the makespan decreases with the NoC size growth due to the lower contention and processor utilization up to a certain mesh size (here: 3x3), after which the potentially increased distance between message senders and receivers changes this

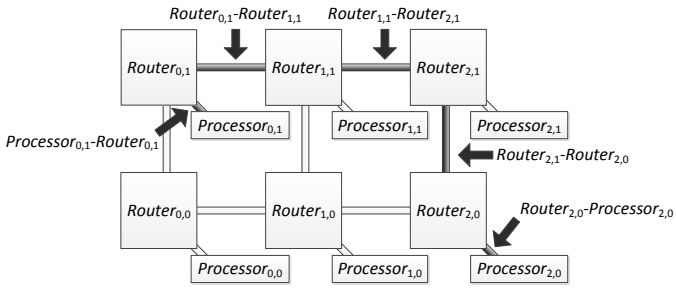

Fig. 4. Example of a path in a mesh Network on Chip

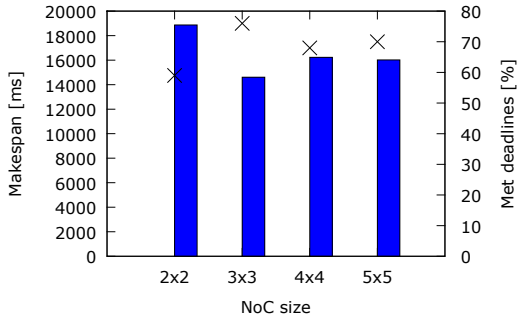

Fig. 5. Makespan values for mesh NoC with FIFO scheduler (bars) and percentage of met deadlines (crosses) - DemoCar use case

trend. The obtained worse results than those presented in Figure 3 show the price of using NoC communication without any task allocation optimization and motivate developing methods aiming at makespan shortening and improving meeting timing constraints.

B. Task and Memory Allocation

Interval algebra can be also used to evaluate the quality of the allocation of tasks and labels into processors to decrease the makespan and meet all timing constraints. In this experiment, we use it as a fitness function in a genetic algorithm that aims to explore the allocation space towards solutions with optimised timing behaviour statically, during the system design stage [18]. To demonstrate this possibility, a NoC mesh platform with XY routing algorithm has been chosen. For the DemoCar application, the size of the mesh has been initially configured as 4x4. The application model has been extended with communication messages between tasks and labels. The genetic algorithm is then executed to perform both task and label allocations to processors during 100 generations of 20 individuals each. The first fully schedulable allocation has been found in the 17-th generation, but the fully schedulable allocation found in the 95-th generation has had 20% lower makespan value. This workload is also fully schedulable in a 3x3 mesh, as a allocation with no violations has been found in the 12-th generation (Figure 6 top). For a 3x2 NoC, the fully schedulable allocation has been found in the 18-th generation, whereas the minimum makespan in the 32-nd (Figure 6 bottom). A fully schedulable allocation has not been found for 2x2 mesh NoC, despite analysing much wider search space than previously - spanning over four islands with 100 individuals each. The best found allocation leads to violation 14 out of 186 deadlines.

C. Performance and scalability

The average execution time of performing the interval algebra preserve operation during the experiment with the bus-based system has been lower than 0.002s regardless the scheduler applied. To determine the approach scalability, a real-life engine control system, composed of 1297 tasks and 46929 labels, has been chosen as a taskset. It has been

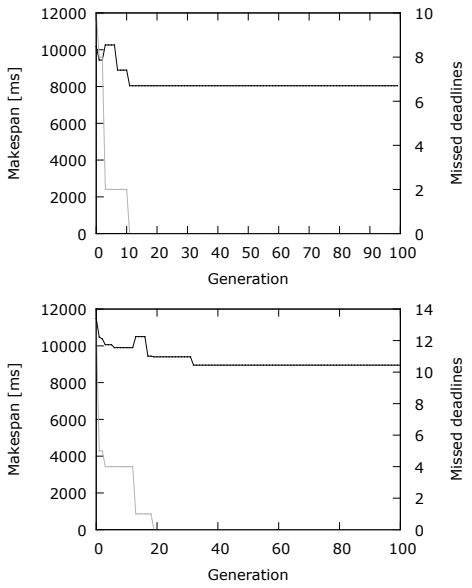

Fig. 6. Missed deadlines (grey) and makespan (black) value optimization for DemoCar implemented on 3x3 (above) and 3x2 (below) mesh-based NoC

evaluated by the IA reference implementation in 0.25s. All these computations have been performed by a single core in a typical desktop computer. These results confirm that the proposed approach is applicable to industrial-size cases. For comparison with another analytical method, we have rewritten the DemoCar use case in MAST-1 model and performed its schedulability analysis with MAST 1.5.0.1 tool from University of Cantabria[3]. To make the comparison fair, the software has been configured for multiprocessor and distributed systems and to reflect task dependencies with offsets. With the fastest technique available for these settings, *Offset Based Approximate Analysis*, the analysis of a single mapping takes 31s, which is too long to be used as a GA fitness function. Comparisons with other formalisms and with real system performance are planned as future work.

VI. Conclusions

An interval algebra has been proposed for evaluating performance parameters of multiprocessor systems. This algebra is expressive enough to model a wide class of platforms, including homogeneous or heterogeneous processors connected with buses or NoCs. The application model covers broad range of tasksets, with deterministic or stochastic execution time and deadlines, with any dependency patterns or processor affinities. The modelled tasks can be singletons, strictly periodic, sporadic or aperiodic. The efficiency and scalability of the reference implementation facilitates using the interval-algebra-based evaluation as a fitness function in various search-space heuristics even with industrial-size cases.

The conducted experiments demonstrated a selection of an appropriate number of processors to satisfy all the application's timing constraints and a choice of a suitable scheduling policy. The resource contention was assessed and resolved by using different platform architectures and resource allocations. The taskset schedulability and makespan were optimized by using genetic algorithms in mesh-based NoCs with various processor numbers.

The library source is planned to be publicly released under the GNU licence in the second half of 2015.

[3]http://mast.unican.es

Acknowledgement

The authors would like to thank Björn Saballus from Robert Bosch GmbH for providing DemoCar and the real-life engine control.

References

[1] A.S. Alfa, Queueing theory for telecommunications. Discrete time modelling of a single node system, Springer, 2010.
[2] N. Audsley, A. Burns, M. Richardson, K. Tindell, A.J. Wellings, Applying new scheduling theory to static priority preemptive scheduling, Softw. Eng. J., Volume 5, Issue 8, 1993, Pages 284–292.
[3] S. H. Bokhari, On the Mapping Problem. IEEE Trans. Comput., Volume 30, Issue 3, 1981, Pages 207–214.
[4] J.-Y. Le Boudec, P. Thiran, Network Calculus: a theory of deterministic queuing systems for the Internet, Springer-Verlag, Berlin, Heidelberg, 2001.
[5] S. Chakraborty, S. Knzli, L. Thiele, A General framework for analysing system properties in platform-based embedded system designs, Design, Automation and Test in Europe Conference and Exhibition (DATE'03), 2003, Pages 190–195.
[6] C.-S. Chang, Performance guarantees in communication networks, Springer-Verlag, Berlin, Heidelberg, 2000.
[7] R. Cruz, A calculus for network delay, Part I: Network elements in isolation, IEEE Trans. on Information Theory, Volume 37, Issue 1, 1991.
[8] R.I. Davis, A. Burns, A survey of hard real-time scheduling for multiprocessor systems, ACM Comput. Surv., Volume 43, Issue 4, Article 35, October 2011, 44 pages.
[9] W. Haid, L. Thiele, Complex task activation schemes in system level performance analysis, 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'07), 2007, Pages 173–178.
[10] Y. Jiang, Y. Liu, Stochastic Network Calculus, Springer-Verlag, Berlin, 2008.
[11] A E. Kiasari, S. Hessabi, H. Sarbazi-Azad, PERMAP: A performance-aware mapping for application-specific SoCs, International Conference on Application-Specific Systems, Architectures and Processors (ASAP'08), 2008, Pages 73–78.
[12] A.E. Kiasari, A. Jantsch, Z. Lu, Mathematical formalisms for performance evaluation of networks-on-chip. ACM Comput. Surv., Volume 45, Issue 3, Article 38, July 2013, 41 pages.
[13] J. Kim, D. Park, C. Nicopoulos, N. Vijaykrishnan, C.R. Das, Design and analysis of an NoC architecture from performance, reliability and energy perspective, ACM Symposium on Architecture for Networking and Communications Systems (ANCS'05), 2005, Pages 173–182.
[14] H. Leontyev, S. Chakraborty, J. Anderson, Multiprocessor extensions to real-time calculus, 30th IEEE Real-Time Systems Symposium (RTSS'09), 2009, Pages 410–421.
[15] C.L. Liu, J. Layland, Scheduling algorithms for multiprogramming in a hard real-time environment, Journal of the ACM, Volume 20, Issue 1, 1973, Pages 46–61.
[16] J.C. Palencia, M.G. Harbour, Exploiting precedence relations in the schedulability analysis of distributed real-time systems, 20th IEEE Real-Time Systems Symposium (RTSS'99), 1999, Pages 328–339.
[17] P. Pop, P. Eles, Z. Peng, Schedulability analysis for systems with data and control dependencies, 12th Euromicro Conference on Real-Time Systems (Euromicro-RTS'00), 2000, Pages 201–208.
[18] M.N.S.M. Sayuti, L.S. Indrusiak, Real-time low-power task mapping in Networks-on-Chip, IEEE Annual Symposium on VLSI (ISVLSI'13), 2013, Pages 14–19.