Contract no. 248972
FP7 STREP Project

NaNoC

Nanoscale Silicon-Aware Network-on-Chip Design Platform

D2.3: Report on Service Differentiation Technique with Packet Formats to Support Quality of Service

Due Date of Deliverable	31st December, 2012
Completion Date of Deliverable	31st December, 2012
Start Date of Project	1st January, 2010 - Duration 36 Months
Lead partner for Deliverable	Lantiq Deutschland GmbH (Lorenzo Di Gregorio)
Approval Status	Approved by all partners on February 13, 2013

Revision: v1.0

Project co-funded by the European Commission within the 7th Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Contents

List of Figures

1. Introduction

In this document we report on investigation and implemention activities within the NaNoC project regarding the *definition and experimentation of novel techniques for supporting quality-of-service (QoS) within generic networks-on-chip (NoCs) from the microarchitecture level up to the application level*.

As explained in the subsequent section 1.1, our goal has been to identify and develop lightweight techniques for providing QoS guarantees at microarchitectural level, because single NoC transactions take place at a very small time scale and do not impact directly application-level QoS. Instead, QoS at application-level is guaranteed by software applications whose performance is in turn affected at the microarchitectural level by overall NoC transport characteristics. Because these characteristics are only indirectly and loosely related to the hardware structures on chip, we are not willing to spend significant silicon and workload on QoS and an outcome of this work is that this cost is also not necessary to obtain very good QoS guarantees, because they can be achieved already by defining static routes, computing schedulability conditions and performance bounds already at design time rather than at execution time.

The novel problem setting of QoS over NoC has driven us to obtain significant theoretical advances in two areas which have been rather dry for almost a decade:

- utilization-based scheduling
- service curve scheduling

In order to validate and evaluate these advances, we have developed simulation infrastructures based on SystemC:

- NoC models employing the SystemQ library [17]
- MPSoC (multiprocessor systems-on-chip) models employing the Open Virtual Platform of Imperas with SystemC wrappers

1.1 Motivation for a lightweight approach

In this section we state the need we see in QoS over NoC at both microarchitectural and application-level. We argue that the purpose of QoS at microarchitectural level is chiefly to ease the design process, whereas guarantees obtained at the application level demand a different solution based on holistic techniques.

1.1.1 Microarchitectural level

From a practical and systemic point of view, the extent to which QoS is a necessity within a NoC is questionable: in real-time applications at a macroscopic level, data transactions over a NoC are hardly perceived and their lack of determinism is usually only a small component of the overall variability in the system. Existing real-time operating systems are able to achieve sub-millisecond accuracy without any special hardware support.

Nevertheless, hardware must be designed for absolute worst cases and QoS in NoCs significantly impacts on design in several microarchitectural aspects of multiprocessor systems-on-chip (MPSoC): from buffer sizing to composition of hardware and software modules, reliable timing characteristics are known to be largely beneficial to the overall design process. For example, due to long worst case response times, costly measures are required to prevent devices' starvation or race conditions: with QoS and deterministic response times in place, this prevention can be achieved by scheduling rather than, for example, by *bloating hardware buffers* of every individual module to ensure that enough data is available under worst case circumstances. Furthermore, QoS enables the allocation of cycle budgets for multiple components sharing common resources over the network: this budgeting avoids the typical *simulate-and-tweak loop*, offering a widely acknowledged benefit in any platform-based design process.

For all these reasons, we have sought an approach which does enable QoS in NoCs, although with little or possibly no hardware impact. Although we have tried to cover all directions, we have focused on techniques in which the nodes are not burdened by additional dedicated features: static routes, which can be set and canceled according to usual *connection* protocols, are provisioned with QoS by non-exclusive bandwidth reservation managed in software. Consequently, we have focused on techniques by which the software overhead for defining many routes through a NoC remains low, in fact we do not see a need to squeeze out the last bit of performance on QoS routes and expect that at the least some best-effort traffic will be transported under all practical circumstances.

1.1.2 Application-level

We know from experience in *soft real-time* scheduling that remarkable improvements of the processor's utilization can be achieved allowing small overload phases, which show no practical impact on the application's quality. This happens because the models employed by the theory must make largely conservative assumptions and absolute worst cases are anyway extremely unlikely to be hit in practice.

While the timing of individual transactions over a NoC does not practically affect application level guarantees, the plurality of transactions taking place in a NoC does affect the performance of the software which provides application level guarantees. A QoS guarantee provided *to an application by a processor* consists of sustainability for workload characteristics seen at the application's programming interfaces. A QoS guarantee provided *to a processor by a NoC* consists of sustainability for traffic characteristics seen at the processor's interfaces like latency, data rate, burst size etc. The application-level characteristics in a network device are conceptually pretty much the same latency, data rate, burst size etc, seen at the microarchitectural level, just on another order of magnitude.

Since *service to an application by a processor* depends largely, although not only, on the *service to that the processor by a NoC*, we could term the problem setting as *"nested QoS"* problem: how does the QoS guarantee to an application change, if the QoS to the underlaying processor changes?

In section 4.4 we give a precise formulation of this *nested QoS* problem, which is illustrated by figure 1.1. In order to master such complex of problems it is most useful to gain confidence with the concept of *service curve*, which is presented in textbooks like [8]. We can say in informal wording that service curve is the *worst case* service seen by a serviced *item*, such as a data packet, arriving at any random time at a server, such as a processor.

One important feature of the concept behind service curves is that it is *holistic*: it does not require a decomposition of the system under analysis and can hence be employed for complex systems. The service curve can be devised from design parameters or actually measured from workload realizations. In the latter case it models only corner cases which are actually hit during the system's evolution and it might

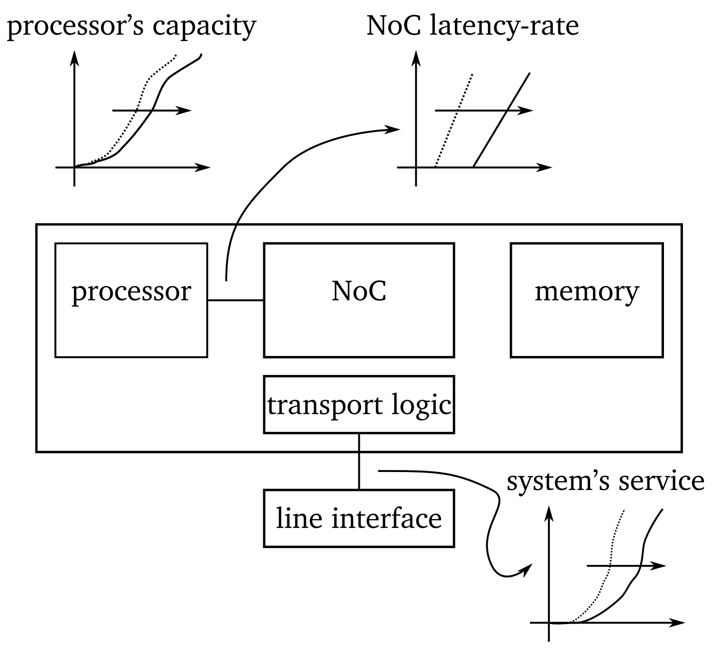

Figure 1.1: Illustration of the "nested QoS" problem: the QoS provided by the NoC to the processor affects the QoS provided by the processor to the line interface.

turn out in unlikely situations to be over-pessimistic: in such cases the analysis must be done for multiple service curves, since a single one cannot represent the characteristics of interest. In principle it is also possible and actually straightforward to disregard corner cases which are extremely unlikely, though this is seldom a practice because one must discover that such cases are unlikely to take place and this requires retaining statistical information about the whole evolution of the system.

A solution for the nested QoS problem, is reported in section 4.5 and the presentation relies heavily on network calculus formalisms. The proposed sólution performs the conversion of the service curve presented by a processor to an application into a service curve presented by the NoC to the processor.

Once the service curves demanded on the interfaces of a processor have been determined according to the algorithm presented in section 4.5, the required latencies and throughputs can be obtained from the NoC employing the techniques we have devised at microarchitectural level for defining static routes under QoS provisioning.

1.1.3 Experimental Results

Section 6.2 describes in practical details the simulation infrastructure and application scenarios required to carry out this work.

1.2 Related work

The groundwork for supporting QoS by reserving bandwidth in the nodes of a NoC consists of the two classical schemes for resource reservation: *rate monotonic* and *earliest deadline first* scheduling (for example see [15] for an overview).

Qian et al. have employed network calculus concatenations in [11] to calculate the service curves for a priority scheduler and a weighted round-robin scheduled and concluded that weighted round-robin schedulers are more flexible than priority schedulers because of the difficulties involved in obtaining guarantees on delays employing priorities. Quite surprisingly, though, the authors appear to have missed the whole literature about periodic scheduling.

In contrast, Shi and Burns presented in [16] a framework based on to predict the latency of traffic flows under periodic scheduling, regarding the NoC nodes as resources which can be reserved in a preemptive manner through a monotonic priority assignment of concurring transactions. They develop a method for analyzing the *interference jitters* within a NoC, i.e. the deviations in the release times of flits indirectly caused by higher priority traffic flows. An obstacle to the practical application of their work, though, is that they employ the classical latency formula by Audsley in [2], which is computationally quite demanding (pseudo-polynomial complexity) for being employed within a traffic management software for a NoC. One can conjecture that a redesign of the algorithm proposed by Bini e Buttazzo in [5] would reduce this complexity.

We address both the priority scheduling area, employing the *hyperbolic bound* presented by Bini et al. in [4], and the network calculus area, pointing out that the *service-curve earliest deadline* (SCED) algorithm by Sariowan et al. in [14] can be employed to derive further policies. A work in this context which is losely related to ours is the one initiated by Thiele in [18], where the *capacity curve* has been introduced and the calculation of the *service curve* obtained from a variable capacity has been presented along with other results. Our work can be viewed as a conceptual inversion, in which we devise how far a capacity curve can be distorted in order to remain compliant to a given service curve.

2. Reservation Frameworks

In the reservation frameworks we have investigated, a NoC node is regarded as a resource to be reserved and for which multiple data transactions are competing. Every transaction consists of a set of data *transfers* to be transported and a set of deadlines, each associated to a transfer, to be met. The goal of the schedule is to ensure, whenever possible, that all transactions meet their deadlines while the data is being transported over the NoC node.

2.1 Reservations on individual nodes

Two major models are available for resource reservation among concurring transactions:

- *task model:* in this model a node can be either busy on a transfer or not. Meeting a deadline means that a total amount of busy time must be served to a transaction before the deadline expires.
- *flow model:* in this model, a cumulative function represents how much data has been transfered. The cumulative function must lay above a boundary which represents the deadlines.

These models are different and equivalent interpretations of the transactions which take place in a NoC: the task model focuses on busy time of the node while the flow model focuses on the amount of data transported by the node. Two graphical representations of these models, showing their equivalence, are provided in figure 2.1 for three transactions which transport *periodic* data transfers. The upper three diagrams represent these transactions according to a *task model* while the lower three diagrams represent the same transactions according to a *flow model*. These data transfers can be *sporadic*: a common misconception about the scheduling of periodic tasks is that the tasks must actually show periodicity over the whole time axis, indeed the theory behind the task model is devised for scheduling tasks which may created and terminate at any time, as long as they show periodicity during their life time.

The task model can be employed for simple traffic models, introduced in section 3.1.1, while the flow model is employed for more complex traffic models introduced in section 3.1.2.

2.2 The NoC as a resource

Having implemented reservations on individual nodes, we can regard a route as a *virtual path* with given transport characteristics. These characteristics can become, in general, rather complex and difficult to analyze at larget time scales than the one of single transactions.

In order to solve these problems, we have elaborated an holistic solution based on network calculus and in particular on the concept of the service curve, which opens the possibility to effectively use further readily available results from the existing literature like the SCED scheduling [13] discipline.

In the following two sections we provide a background on the concepts of *service curve* and *traffic envelope*: these are fundamental tools for the analysis of complex systems, because complex service curves are hardly manageable and they must be approximated by simpler service curves.

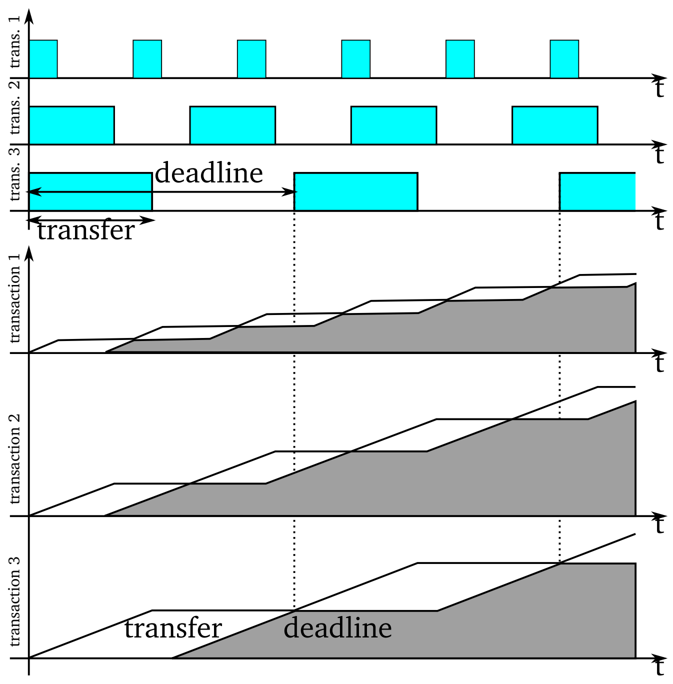

Figure 2.1: Relation of the task models to the traffic models

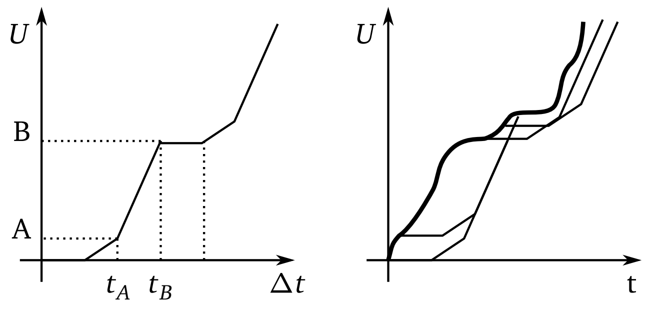

Figure 2.2: On the left a service curve and on the right a workload realization with the same service curve in evidence.

2.2.1 Service Curves

Figure 2.2 provides a graphical representation of the concept behind a service curve. U is a generic *utility*, that is a figure of merit like amount of data packets or amount of bytes. The plot on the right shows as a thick line a *workload realization*, that is the amount of utility which is serviced over time by an application. It is important to note that the application must be *under backlog*, that is it must have data to process. In lack of inputs an application would be clearly delivering no service merely because there is nothing to do. At any random point in time it can be guaranteed that the amount of U which shall be serviced lays above the service curve presented by the plot on the left. We see in the plot on the right that the service curve can be swept along the whole workload realization and the workload realization always lays above the service curve. This is the reason why the plot on the left has Δt rather than t on the horizontal axis: the service curve at time t_B represents the worst case service which can be seen over a timespan of t_B time units. In the plot on the left we can also recognize a packet processing characteristic: in the period between t_A and t_B the body of the packet is transported at high data rate, before t_A the

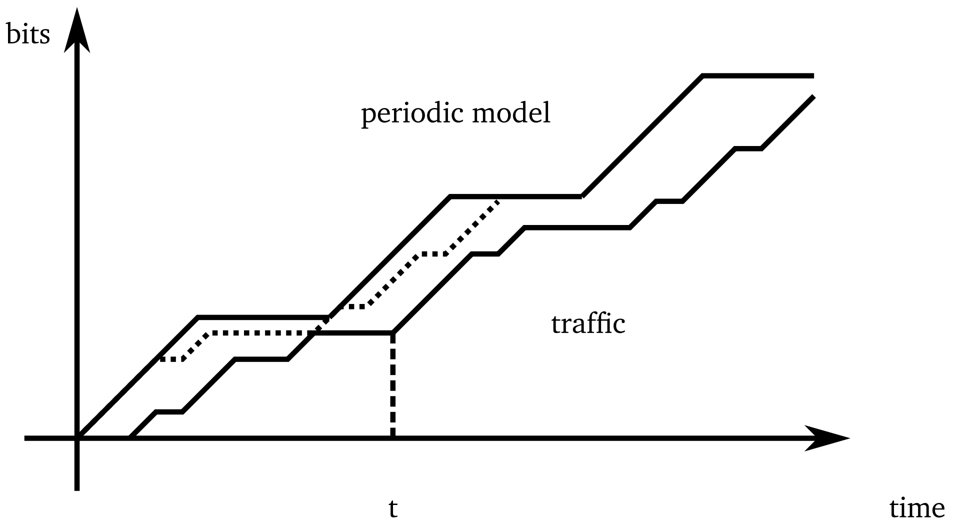

Figure 2.3: Periodic model which envelopes a traffic trace.

header is processed at a lower data rate, because the software must carry out more operations per byte, and after t_B the packet has been transported and there is a small inter-packet gap, likely to be caused by framing.

2.2.2 Traffic Envelope

In this section we explain in simple terms what is a *traffic envelope*.

A traffic *model* is a defined function which represents a traffic level over time. For example we can term a periodic model with period T and burst C the function:

$$f(x) = \int_0^x S(kT - C, kT)\,\mathrm{d}x$$

where $S(a,b)$ is C between a and b and 0 elsewhere.

The traffic obtained from such a periodic model is presented in figure 2.3. This model is an *envelope* for a traffic *trace*, i.e. a realization of traffic over time, if it lays always above the trace regardless of the point at which the trace is observed.

In figure 2.3 the worst case traffic begins at time point t: this traffic is represented at time point 0 by the dotted line. The periodic model represented in figure 2.3 is the lowest model which lays above the dotted line.

The dotted line is called in network calculus the *arrival curve* of the traffic and it represents at a generic time t the maximum increment presented by the traffic between two generic time points q and $q+t$.

3. Serving Traffic

For implementing QoS, incoming traffic must be served by NoC nodes. In the context of this document, *classes of service* presented in chapter 5 represent a classification of traffic within the NoC according to characteristics which are suitable for enforcing QoS. For determining which characteristics are relevant, in this chapter we study *traffic models* in section 3.1 and technique to serve traffic under these models in section 3.2. In order to increase the efficiency in QoS over NoCs, we propose in chapter 4 some advances on *utilization bounds* presented in section 3.2.1.

3.1 Traffic Models

Traffic models can be regarded as processes whose realizations are subject to given traffic characteristics. In sections 3.1.1 and 3.1.2 we present models for which schedulability analysis has been successfully employed and are hence supported by QoS techniques.

3.1.1 Simple traffic models

Rate-monotonic scheduling is the most common technique for resource reservation. It is commonly applied to a set of software tasks, but it can be equally applied to a set of transactions if the busy time of a task represents the busy time of a data transfer, as shown in section 2. With the terminology introduced to this purpose within this document, rate-monotonic scheduling consists of associating to each transaction a priority which is inversely proportional to the period of the deadlines. In their seminal work in [10], Liu and Leyland demonstrated that if the highest priority is always scheduled first and all transactions are preempt-able, all deadlines are met as long as the n *data rates* $U_i, \forall i \in 1, \ldots, n$ of all ongoing transactions satisfy:

$$\sum_{i=1}^{n} U_i \leq n(\sqrt[n]{2} - 1) \tag{3.1}$$

where:

$U_i \doteq C_i / T_i$	is the definition of *data rate*.
C_i	busy time for one periodic transfer within the transaction i.
T_i	period of the deadline within the transaction i.
n	number of simultaneous transactions.

The inequality (3.1) states a simple condition under which it is ensured that a set of transactions bearing periodic transfers *whose sole data rate is known*, is guaranteed specific bandwidth and latency.

This condition is sufficient but not necessary, and it is not tight in the sense that there exist sets $U_1, \ldots, U_n$ which violate (3.1) although there exist no combination of $C_1, \ldots, C_n$ and $T_1, \ldots, T_n$ which cannot meet all deadlines under rate-monotonic scheduling.

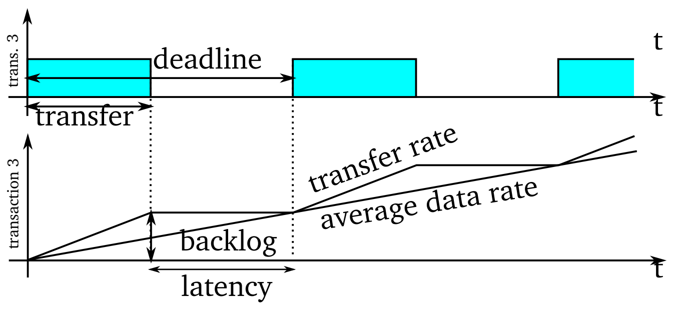

Figure 3.1: Relation between data rate, latency and backlog in a guaranteed traffic flow, derived from the real-time scheduling model of periodic tasks.

Bini et al. have identified in [4] the tightest sufficient condition based on sole *data rates* U_i, which is:

$$\prod_{i=1}^{n}(U_i+1)\leq 2 \tag{3.2}$$

This condition is still sufficient and not necessary, but it can be proved that this is the tightest condition if only data rates U_i are known, because for every set $U_1,\dots,U_n$ which violates (3.2) there is at the least one combination of $C_1,\dots,C_n$ and $T_1,\dots,T_n$ which cannot meet all deadlines under rate-monotonic scheduling.

Condition (3.2) imposes an upper limit to the *utilization* $U \doteq \prod_{i=1}^{n}(U_i+1)$ of a NoC node under periodic guaranteed traffic if only data rates are known.

The periodic or sporadic traffic flows, sustainable through bound (3.2), must bear sufficient backlog at the beginning of the period: to make this point clear consider that if all flows would start transporting their backlog just in time to comply with their deadline, without leaving any slack, it would not be possible to meet all deadlines if any two intervals between start and deadline would overlap. Figure 3.1 provides a graphical representation of the relations among the traffic characteristics of one individual flow, in particular:

$$\text{flow latency} = \text{deadline}\left(1-\frac{\text{average data rate}}{\text{transfer rate}}\right) \tag{3.3}$$

because it is clear from figure 3.1 that

$$\text{average data rate} = \frac{\text{backlog}}{\text{deadline}}$$

$$\text{transfer rate} = \frac{\text{backlog}}{\text{deadline}-\text{latency}}$$

In symbols, the *flow latency* expressed by (3.3) is merely $T_i - C_i$ and it states the necessity of having the backlog ready at the beginning of the period in order to schedule it under the test of (3.2), in fact the latency $T_i - C_i$ is exactly the one we have declared by setting the deadline T_i on the flow.

Transporting data over one node delays it because of other *interfering* transfers, i.e. transfers which bear a higher priority and increase the latency of the current transfer by preempting it. Consequently, the backlog after transversing the first node is not ready anymore at the beginning of the period and the preemption time due to higher priority transfers must be accounted in the schedulability condition. The delay to which a transfer is subject due to preemptions, is called *network latency*, not to be confused with the flow latency expressed by (3.3).

One straightforward approach to account network latency consists of enlarging C_i to cover the preemption time, but this grossly reduces the achievable *real* utilization of the node, i.e. its the busy time, because it disregards the fact that during the preemption time of one backlog the node could actually transport data of another backlog.

In order to increase the utilization of the NoC nodes under further types of guaranteed traffic, further insight into the traffic characteristics of the transactions must be won. A first step can be carried out by knowing the deadlines T_i of the transactions and the elapsed fraction of busy time C_i per transaction. With this data, the *earliest deadline first* (EDF) scheduling, consisting of scheduling always the transaction whose earliest deadline is next, can achieve up to 100% utilization of NoC nodes under guaranteed traffic. While EDF scheduling appears attractive of a single node, it comes on system level at the price of having to sort out at any generic time all transactions in order to find out which one bears the next earliest deadline on a node along the route through interfering transaction. This is clearly not feasible.

A feasible solution consists in applying a technique known as *deadline monotonic* scheduling, proposed by Audsley in [2], and consisting of assigning the higher priorities to transfers which get closer to their deadlines. In this solution, the network latency can be simply subtracted from the deadline to produce a new deadline on which the deadline monotonic scheme can be applied. Yet, before a route can be defined a *schedulability test* must be carried out to determine whether the QoS in presence of network latency can be met and this test is more demanding than the ones based on utilization, which we have seen so far. A test developed by Bini and Buttazzo in [5] offers a parametrized trade-off between tightness and complexity, hence it can be employed according to the demands of its users.

While EDF scheduling might definitely be an overkill for simple traffic models, traffic characteristics can definitely be too complex to be efficiently embedded in periodic sequences of data transfers at fix rates. If traffic significantly differs from the periodic model into which it has to be embedded, fitting it into a periodic model would lead to gross loss of utilization. In such cases, loading the node with a higher utilization might justify more costly solutions and requires exploiting further traffic characteristics for which classic EDF scheduling does not apply. Service curve scheduling techniques, introduced in the section 3.1.2, provide solutions to schedule more complex traffic.

3.1.2 Complex traffic models

The traffic characteristics which can be sustained by heterogeneous components can be rather complex: for example it is common to accept a *maximum burst size* and a lower *data rate*, which needs to be paused by short *recovery times*, sometimes known in packet-based networks as “inter-frame gap”.

Such components can be characterized by *service curves*, introduced by Cruz in [6, 7]: figure 3.2 provides a graphical representation of a service curve along with the traffic which has generated it. The service curve at time t represents the minimal amount of traffic that a component can serve within a time window of size t if backlog is available. The “translated service curve” in figure 3.2 demonstrates that if the origin of the service curve is translated to any generic point of a traffic realization, the service curve always constitutes a lower bound of the traffic. Obviously, the service curve must be the highest of all lower bounds in order to be a *tight* bound on the traffic.

An application of service curves in resource reservation for QoS is featured by the SCED algorithm, proposed in [12]. Behind the formalism employed in [12], the generic SCED algorithm consists of assigning deadlines to data packets outgoing from one node according to the service curves to be guaranteed to every traffic flow: if packets under such deadlines are scheduled by an EDF policy and the schedule is feasible (i.e. no overloading situation takes place), then the EDF policy guarantees to every flow its

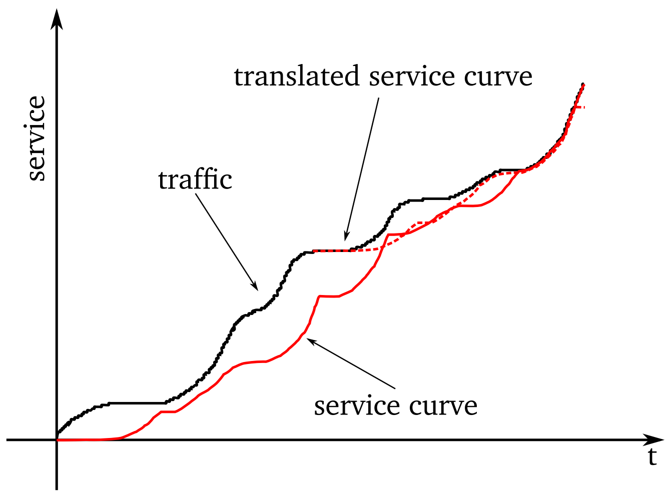

Figure 3.2: Relation of the service curve to a traffic realization

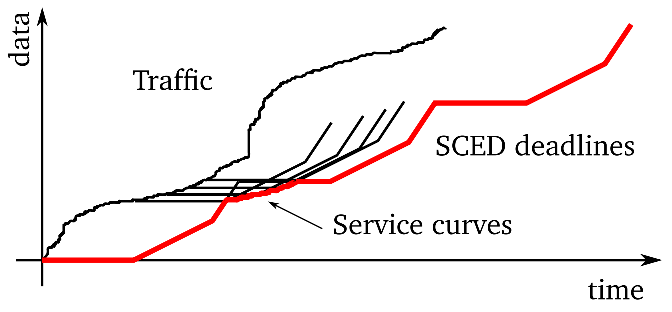

Figure 3.3: Relation of the SCED deadlines to the actual traffic

assigned service curve even if traffic is random. A representation of a traffic flow with the associated service curve and deadlines is presented in figure 3.3: the SCED deadlines are constructed sweeping the service curve along the traffic flow and obtaining the *infimum*[1] of all swept curves.

An implementation of the generic SCED algorithm is not feasible, because it requires iterations over a long record of past traffic and future deadlines: one should constantly calculate the infimum between the service curve applied to the current traffic and the already calculated future deadlines. In order to avoid this computational effort, the work in [12] proposes a class of simple service curves for which this computation reduces to few simple operations.

This class consists of curves shown in figure 3.4, which present a latency and a constant service rate:

In this case, the deadlines are a straightforward function of the accumulated traffic, with a *reset* as soon as the deadlines fall at the value of the latency. Figure 3.4 shows this computation in a graphical way: one can ideally picture to attach the service curve to the traffic as it had been done in figure 3.3 and as soon as one translated service curve falls below the currently effective one, it dominates it completely and becomes the new effective service curve.

[1]For practical purposes it is obvious that the *infimum*, also known as *greatest lower bound*, always corresponds to the *minimum*, but in network calculus theory it is common to refer to the infimum in order to keep results valid for *fluid* traffic models.

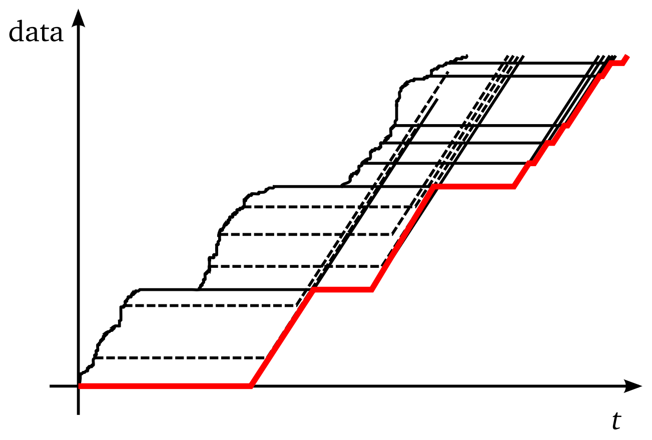

Figure 3.4: Deadlines for a server with fix latency and constant rate.

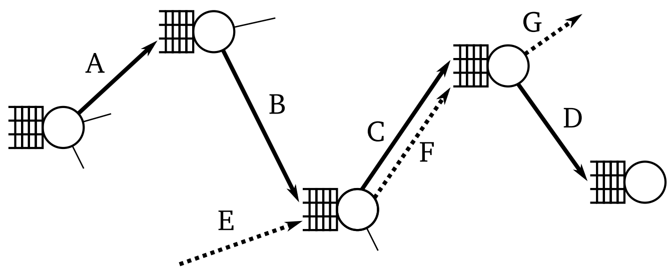

Figure 3.5: Route A-B-C-D interferes with route E-F-G.

3.2 Service models

Service models can be regarded as time partitioning rules for access to a NoC node or a communication sink. Simple traffic models are served by techniques mentioned in section 3.2.1. Some advances on these techniques have been reported in chapter 4. Section 3.2.2 reports management techniques for more complex traffic, discussed in section 3.1.2.

3.2.1 Serving simple traffic

In this section we employ the bound (3.2) for sustaining traffic guarantees for simple periodic traffic models to which we assign static priorities. Although the same reasoning applies also to the bound (3.1), we do not employ this bound because it is looser and merely bears the advantage of a slightly lower computational effort, which is not significant in our context. In chapter 4 we will present some novel tighter bounds.

For serving simple traffic, every traffic flow (C_i, T_i) must be transported over a static route through the NoC, ensuring that the utilization of every node is sufficient for sustaining required throughputs and achieving achieving given latencies.

Figure 3.5 presents a route across multiple nodes: the route E-F-G interferes with all nodes on the route A-B-C-D, because they all stop transporting data due to backpressure from the single link sustaining trunks C and F, if the transfer E-G has higher priority.

Although this is an idealization, because nodes have some buffer and backpressure has a propagation delay, schedulability analysis assumes worst case phase shifts and stationary behavior: under these assumptions the fill levels of the buffers and the backpressure propagation delay are negligible.

Consequently, the *schedulability test* is just (3.2), which can applied to all transfers which have been transported though *one* node with interference along their route.

When a transfer travels *over multiple nodes with interference*, instead, the schedulability test (3.2) is not directly applicable anymore because the backlog gets distributed over time due to preemption of interfering transfers. In this case *overall network latency* experienced by a single flow under preemption of higher priority transfers *over a single node* can be computed using a method proposed by Audsley in [2, 3]:

$$\begin{aligned}\text{overall latency} = &\min_{k=1,\ldots,+\infty} \{R_i^{(k)} : R_i^{(k)} = R_i^{(k-1)}\} \\ &+ T_i - C_i\end{aligned} \tag{3.4}$$

with $R_i^{(k)}$ defined by recurrence as

$$\begin{aligned} R_i^{(0)} &= C_i \\ R_i^{(k)} &= C_i + \sum_{j:T_j<T_i} \left\lceil \frac{R_i^{(k-1)}}{T_j} \right\rceil C_j \end{aligned}$$

In practice, the iteration over k must be carried out only until $R_i^{(k)} = R_i^{(k-1)}$. Still, this formula cannot be directly employed for an online schedulability test because of its pseudo-polynomial complexity. An algorithm by Bini and Buttazzo in [5] can be employed instead to test schedulability under several trades-off between computational demand and accuracy of the test.

From (3.4) we can note that we must discover all transfers which are interfering with the one for which we need to compute the network latency. These are not only transfers which directly preempt the one in question, but rather also transfer which preempts these preempting transfers. Shi and Burns have developed a technique for applying (3.4) to a NoC and published it in [16].

3.2.2 Serving complex traffic

Commonly, *dynamic priorities* refer to the *earliest deadline first* (EDF) scheduling policy, which states that at any point in time the transfer with the earliest deadline must get the highest scheduling priority, by dynamic assignment over time watching all deadlines of outstanding transfers. Obviously, the EDF policy can be implemented for the traffic models presented in section 3.2.1 and demands a very straightforward and simple schedulability test:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

Nevertheless, because of several known engineering difficulties, whose most relevant to NoCs have been mentioned in section 3.1.1, EDF schedulers are not commonly found in real-time computing systems, with some notable exceptions like the Xen scheduler.

Conversely, the fix transfer raster which is demanded by the static priority assignment presented in section 3.2.1 is highly inefficient if the traffic cannot be *served* with this fix raster. As discussed in section 3.1.2, such more complex traffic characteristics can be framed within the SCED scheduling policy, presented in [12], which reduces to the classic EDF policy if traffic is periodic.

The key result behind SCED policies is that the EDF assignment is still optimal if deadlines are assigned according to service curves and these service curves can be calculated from sinks to sources along the

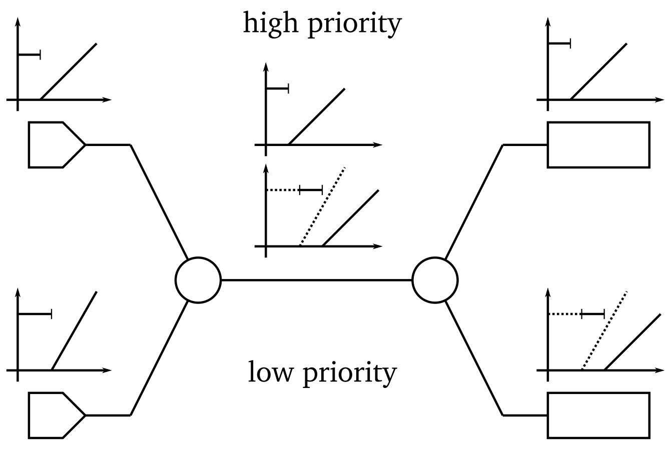

Figure 3.6: Service curves for a high and low priority transfer.

routes in a NoC. In general, these service curves need only to be convex, but the SCED policy can only be enforced at acceptable costs if the service curves consist of *latency-rate* characteristics, as shown in figure 3.4.

Figure 3.6 presents a graphical example of this calculation based on a *concatenation theorem* reported in [9]. In practice, the service curves are calculated along paths from sources to sinks, adding latencies and taking the lower slopes. When transfers are transported over common links, either one transfer is prioritized, as in the example in figure 3.6 or both are interleaved in some round-robin fashion. The increased latency must be accounted according to the chosen arbitration scheme: in the example shown in figure 3.6, the high priority transfer retains its latency while the low priority transfer must account for the preemption time due to the high priority transfer.

Once the service curves are calculated, SCED deadlines can be assigned on every node and EDF scheduling on every node ensures locally that these deadlines are not violated. The QoS experienced at the sink is clearly looser than the one imposed at the source: in order to meet given constraints, though, the computation of the service curves can also be carried out also backward, from sinks to sources. In this case the traffic is schedulable if the QoS demanded at the sinks is loose enough to accommodate rates and latencies met along the path and present feasible service curves at the sources.

4. Advances on QoS bounds

In this chapter we report some advances we developed on utilization-based scheduling, which is particularly attractive if the data rates are known but the burst characteristics of the traffic are not entirely known, and service curve scheduling, which is attractive to hierarchically propagate the effect of traffic characteristics of the NoC to the system by modeling the slowdown caused on the software. In section 4.1 a new theorem on utilitation-based scheduling will be introduced and we will show a graphical interpretation of the results which leads to two further theorems in sections 4.2 and 4.3. In section 4.5 a theory is given leading to an algorithm which determines a bound on the service provided by a NoC in order to let the software being executed be compliant with service constraints imposed at system level.

4.1 Remaining Utilization

In this section we introduce a theorem for calculating the utilization of the nodes in a NoC under periodic traffic models presented in section 3.1.1. We provide for this theorem an algebraic proof which demonstrates that its main formula (4.1) is equivalent to the hyperbolic bound (3.2) presented in [4]. More interestingly, we provide a graphical interpretation of this result from which tighter bounds are derived in sections 4.2 and 4.3: these results are no contradiction of the proof in [4] that the hyperbolic bound (4.1) is the tightest bound on utilization, because they employ the backlog C_k and period T_k individually and not just $U_k \doteq C_k/T_k$.

To begin with, we state the following theorem.

Theorem 1 (remaining utilization). *If a node under a utilization U_{i-1} becomes subject to a periodic traffic with backlog C_i and period T_i, its utilization becomes:*

$$U_i = \frac{T_i U_{i-1} - C_i}{T_i + C_i} \tag{4.1}$$

□

In order to prove this theorem, we employ just the hyperbolic bound (3.2) and show that formula (4.1 can be obtained by algebraic manipulation.

Proof. By employing the hyperbolic bound (3.2) for a set of transfers $(C_1, T_1), \ldots, (C_n, T_n)$, we can find out what is the remaining utilization U_n as:

$$\prod_{i=1}^{n} \left(\frac{C_i}{T_i} + 1 \right) (U_n + 1) = 2$$

from which one can rather simply obtain:

$$U_n = \frac{2 \prod_{i=1}^{n} T_i - \prod_{i=1}^{n} (C_i + T_i)}{\prod_{i=1}^{n} (C_i + T_i)} \tag{4.2}$$

Now we must express the product $\prod_{i=1}^{n}(C_i+T_i)$ in an explicit form. By executing the first iterations of the product:

$$\begin{aligned}\prod_{i=1}^{n}(C_i+T_i) &= C_1\prod_{i=2}^{n}(C_i+T_i)+T_1\prod_{i=2}^{n}(C_i+T_i)\\ &= C_1C_2\prod_{i=3}^{n}(C_i+T_i)+C_1T_2\prod_{i=3}^{n}(C_i+T_i)+T_1C_2\prod_{i=3}^{n}(C_i+T_i)+T_1T_2\prod_{i=3}^{n}(C_i+T_i)\end{aligned}$$

it can be seen that the product consists of a sum of all possible product sequences $M_1\cdots M_n$ with $M_i\in\{C_i,T_i\}$.

The product $\prod_{i=1}^{n}(C_i+T_i)$ can be expressed in the following form:

$$\begin{aligned}\prod_{i=1}^{n}(C_i+T_i) = C_n&\prod_{i=1}^{n-1}(C_i+T_i)+\\ &T_nC_{n-1}\prod_{i=1}^{n-2}(C_i+T_i)+\\ &T_nT_{n-1}C_{n-2}\prod_{i=1}^{n-3}(C_i+T_i)+\\ &\vdots\\ &T_n\cdots T_4C_3\prod_{i=1}^{2}(C_i+T_i)+\\ &T_n\cdots T_4T_3C_2(C_1+T_1)+\\ &T_n\cdots T_4T_3T_2C_1+\\ &T_n\cdots T_4T_3T_2T_1\end{aligned}\tag{4.3}$$

Observing that the last addend of (4.3) is $\prod_{i=1}^{n}T_i$, (4.3) can be substituted in (4.2), obtaining:

$$\begin{aligned}U_n = &-\frac{C_n\prod_{i=1}^{n-1}(C_i+T_i)}{\prod_{i=1}^{n}(C_i+T_i)}+\\ &-\frac{T_nC_{n-1}\prod_{i=1}^{n-2}(C_i+T_i)}{\prod_{i=1}^{n}(C_i+T_i)}+\\ &-\frac{T_nT_{n-1}C_{n-2}\prod_{i=1}^{n-3}(C_i+T_i)}{\prod_{i=1}^{n}(C_i+T_i)}+\\ &\vdots\\ &-\frac{T_n\cdots T_4C_3\prod_{i=1}^{2}(C_i+T_i)}{\prod_{i=1}^{n}(C_i+T_i)}+\\ &-\frac{T_n\cdots T_4T_3C_2(C_1+T_1)}{\prod_{i=1}^{n}(C_i+T_i)}+\\ &-\frac{T_n\cdots T_4T_3T_2C_1}{\prod_{i=1}^{n}(C_i+T_i)}+\\ &+\frac{T_n\cdots T_4T_3T_2T_1}{\prod_{i=1}^{n}(C_i+T_i)}\end{aligned}$$

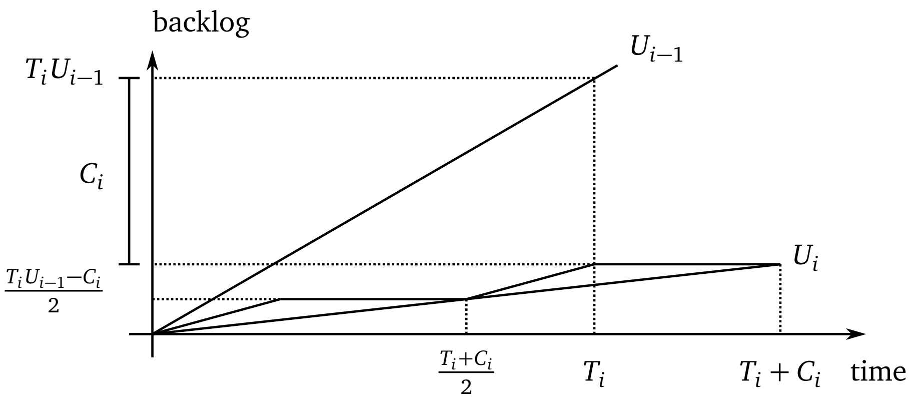

Figure 4.1: Graphical interpretation of the remaining utilization.

and simplifying the ratios we obtain:

$$\begin{aligned}
U_n = & -\frac{C_n}{C_n+T_n}+ \\
& -\frac{T_nC_{n-1}}{\prod_{i=n-1}^{n}(C_i+T_i)}+ \\
& -\frac{T_nT_{n-1}C_{n-2}}{\prod_{i=n-2}^{n}(C_i+T_i)}+ \\
& \vdots \\
& -\frac{T_n\cdots T_4C_3}{\prod_{i=3}^{n}(C_i+T_i)}+ \\
& -\frac{T_n\cdots T_4T_3C_2}{\prod_{i=2}^{n}(C_i+T_i)}+ \\
& +\frac{T_n\cdots T_4T_3T_2}{\prod_{i=2}^{n}(C_i+T_i)}\frac{T_1-C_1}{C_1+T_1}
\end{aligned}$$

These terms can be reorganized as:

$$\begin{aligned}
U_n = & \frac{T_n}{C_n+T_n}\frac{T_{n-1}}{C_{n-1}+T_{n-1}}\cdots\frac{T_2}{C_2+T_2}\frac{T_1-C_1}{C_1+T_1} \\
& -\frac{T_n}{C_n+T_n}\frac{T_{n-1}}{C_{n-1}+T_{n-1}}\cdots\frac{T_3}{C_3+T_3}\frac{C_2}{C_2+T_2} \\
& \vdots \\
& -\frac{T_n}{C_n+T_n}\frac{T_{n-1}}{C_{n-1}+T_{n-1}}\frac{C_{n-2}}{C_{n-2}+T_{n-2}} \\
& -\frac{T_n}{C_n+T_n}\frac{C_{n-1}}{C_{n-1}+T_{n-1}} \\
& -\frac{C_n}{C_n+T_n}
\end{aligned}$$

from which the recursive structure of (4.1) can be recognized assuming that the node is initially empty, hence $U_0 = 1$. □

A graphical interpretation of formula (4.1) is presented in figure 4.1: a backlog C_i must be deployed within a period T_i on a node under utilization U_{i-1}. The utilization U_{i-1} guarantees that a total backlog of

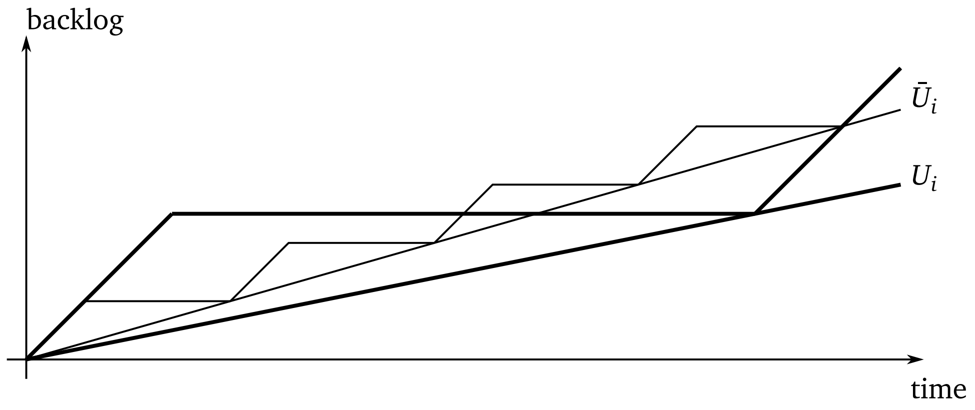

Figure 4.2: Higher utilization achieved by shorter periods.

$T_i U_{i-1}$ can be transported, hence a total backlog of $T_i U_{i-1} - C_i$ remains available for transport by a further transaction and a total time of $T_i - C_i$ remains available for transporting it. The remaining transactions must be periodic, transport the whole backlog $T_i U_{i-1} - C_i$, and the worst utilization is achieved if the whole remaining backlog is transported as shown in the figure, because it achieves the maximum distance from the utilization U_i: every shorter period would increase the utilization. The period for this transaction is exactly $\frac{T_i + C_i}{2}$ and the backlog is $\frac{T_i U_{i-1} - C_i}{2}$, the utilization U_i is defined as the fraction of backlog by utilization and corresponds to formula (4.1).

4.2 Bounded Periods

The graphical interpretation of figure 4.1 has shown that the utilization employed for the hyperbolic bound (3.2) involves no assumption on the individual backlog and period of the transfers. The utilization U_i can only be achieved if the period is $\frac{T_i + C_i}{2}$ and the corresponding backlog is $\frac{T_i U_{i-1} - C_i}{2}$ and gets transported at the beginning of the two periods: shorter periods would be able to transport the same backlog in more periods, achieving a higher utilization. This is demonstrated by figure 4.2, which displays a utilization $\bar{U}_i > U_i$ for transporting the same total backlog.

This reasoning can be applied to increase the available utilization if we know that the set of subsequent transfer, from $i + 1$ onward, will have an overall period bounded by a known upper limit $\bar{T}_{i+1}$. This is the case if merely one subsequent transfer is scheduled or if subsequent transfers $i + 1, i + 2, \ldots$ are *harmonic*: for example, if $T_{i+1} = 4$ and $T_{i+2} = 2$, they are *harmonic* because it holds with $T_{i+1} \geq T_{i+2}$ that $T_{i+1}/T_{i+2} \in \mathbb{N}$, then $\bar{T}_{i+1} = 4$. This is also the case if the subsequent transfers $i + 1, i + 2, \ldots$ are not harmonic but they bear relatively small periods with respect to the current transfer i: for example, if $T_i = 10$, $T_{i+1} = 3$ and $T_{i+2} = 2$, then $\bar{T}_{i+1} = 6$ is considerably lower than T_i.

Figure 4.1 has shown that the remaining utilization is the worst case one reached by a periodic transfer which transports the backlog $T_i U_{i-1} - C_i$ within the time T_i. The problem in this case consists of determining the $\bar{C}_{i+1}$ which transports the same backlog if the period T_{i+1} is bounded by the upper limit $\bar{T}_{i+1}$.

This problem is solved by the following theorem.

Theorem 2 (tighter utilization). *If a node under a utilization U_{i-1} becomes subject to a periodic traffic with backlog C_i and period T_i, and the schedule of further transfers bears a period equal to or lower than $\bar{T}_{i+1}$,*

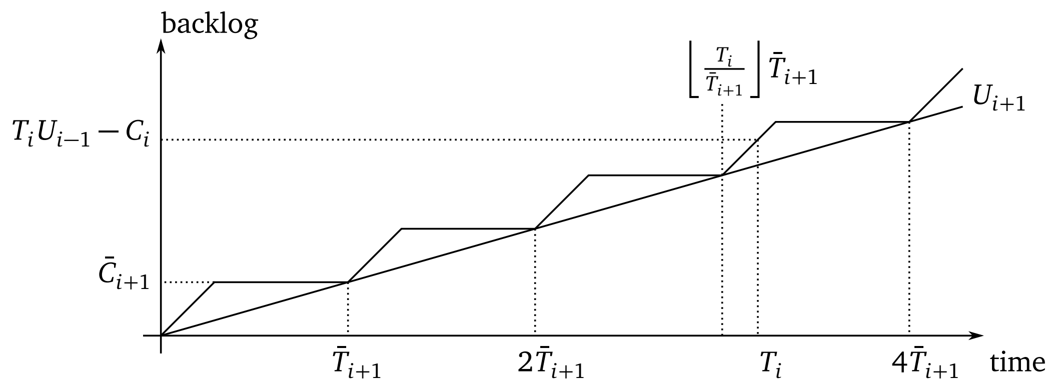

Figure 4.3: Calculation of utilization achieved by a bound on period.

its utilization becomes:

$$U_i = \frac{\bar{C}_{i+1}}{\bar{T}_{i+1}} \tag{4.4}$$

where

$$\bar{C}_{i+1} = \frac{T_iU_{i-1} - C_i}{\left\lfloor \frac{T_i}{\bar{T}_{i+1}} \right\rfloor + 1} \qquad \text{if } \bar{C}_{i+1} \le T_i - \left\lfloor \frac{T_i}{\bar{T}_{i+1}} \right\rfloor \bar{T}_{i+1} \tag{4.5}$$

$$\bar{C}_{i+1} = \frac{T_iU_{i-1} - C_i - T_i + \left\lfloor \frac{T_i}{\bar{T}_{i+1}} \right\rfloor \bar{T}_{i+1}}{\left\lfloor \frac{T_i}{\bar{T}_{i+1}} \right\rfloor} \qquad \text{if } \bar{C}_{i+1} \ge T_i - \left\lfloor \frac{T_i}{\bar{T}_{i+1}} \right\rfloor \bar{T}_{i+1} \tag{4.6}$$

□

Proof. Inspecting figure 4.3, it can be noted that the backlog transported within T_i depends on whether the last transfer C_i can complete within T_i or not. If it can be complete within T_i, then the transported backlog is merely $\bar{C}_{i+1}$ for the number of periods which fit into T_i, hence:

$$\bar{C}_{i+1} \left\lceil \frac{T_i}{\bar{T}_{i+1}} \right\rceil$$

if it cannot be completed within T_i, the part of backlog which can be transported within the last period is

$$T_i - \left\lfloor \frac{T_i}{\bar{T}_{i+1}} \right\rfloor \bar{T}_{i+1}$$

These formulae can be organized into the following equation which states that the transfer with period $\bar{T}_{i+1}$ must be able to transport the whole remaining backlog $T_iU_{i-1} - C_i$:

$$T_iU_{i-1} - C_i = \bar{C}_{i+1} \left\lfloor \frac{T_i}{\bar{T}_{i+1}} \right\rfloor + \min\left(\bar{C}_{i+1}, T_i - \left\lfloor \frac{T_i}{\bar{T}_{i+1}} \right\rfloor \bar{T}_{i+1} \right)$$

from this formula $\bar{C}_{i+1}$ can be determined according to (4.5) and (4.6). □

As shown in figure 4.4, one can simply try the computation of $\bar{C}_{i+1}$ according to the more likely between (4.5) and (4.6) and verify if the condition on $\bar{C}_{i+1}$ is met. If it is not met, then it must be met in the other case if the i-th transfer (C_i, T_i) is schedulable.

A careful reader might observe that for $\bar{T}_{i+1} \to 0$ the utilization $U_i \to \frac{T_iU_{i-1} - C_i}{T_i}$ (hint: $\left\lfloor \frac{T_i}{\bar{T}_{i+1}} \right\rfloor + 1 \approx \frac{T_i}{\bar{T}_{i+1}}$ when $\bar{T}_{i+1} \to 0$).

```
function [Ui] = utilization_bp(Ci,Ti,Uim,Tbip)

Cbip=(Ti*Uim-Ci)/(floor(Ti/Tbip)+1);

if Cbip >= Ti-floor(Ti/Tbip)*Tbip
    Cbip=(Ti*Uim-Ci-Ti+floor(Ti/Tbip)*Tbip)/floor(Ti/Tbip);
end

Ui=Cbip/Tbip;
```

Figure 4.4: Implementation of (4.6) in Octave: Ci is C_i, Ti is T_i, Uim is U_{i-1}, Tbip is $\bar{T}_{i+1}$, Cbip is $\bar{C}_{i+1}$ and Ui is U_i.

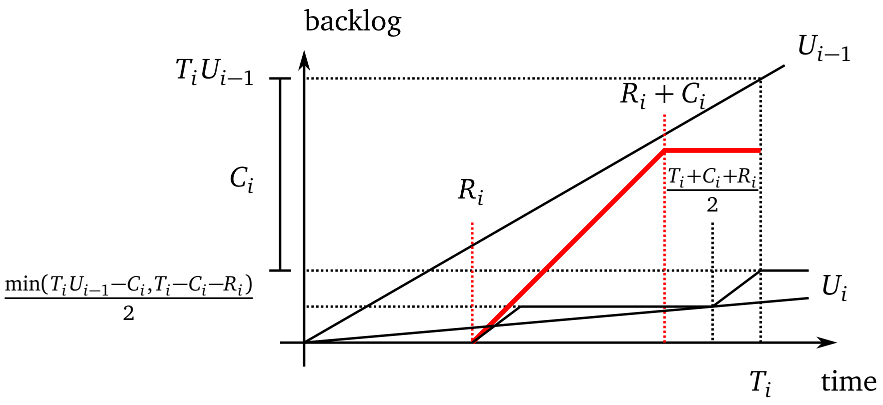

Figure 4.5: Utilization under rate monotonic with release times.

4.3 Network Latency

In a NoC, after a transfer has been subject to interference, the backlog has accumulated a *network latency* as high as the preemption time due to higher priority transfers. As mentioned in section 3.1.1, if the backlog to be transfered within a period is not available at the beginning of the period, the schedulability tests do not guarantee the deadlines anymore.

In this section we employ our graphical interpretation of the schedulability tests to give tighter bounds on schedulability of transfers subject to propagation delay. As Shi and Burns noted in [16], this problem is in general NP-hard.

While there are several techniques to deal with schedulability under release times, it is attractive to retain the simplicity of the rate monotonic scheduling. A straightforward way to do so consists of replacing the transfer length C_i with the sum $R_i + C_i$, hence considering the network latency a part of the transfer.

Figure 4.5 shows an improvement over this basic method: the red curve indicates a transfer of backlog C_i subject to network latency R_i. Taking the same approach employed in section 4.1, we allocate C_i idle time between R_i and T_i in the worst case condition, considering that if $C_i + R_i$ gets too close to T_i, there might be no time to transfer $T_iU_{i-1} - C_i$ backlog and only $T_i - C_i - R_i$ can be transported. The reason behind this observation is that it might be entirely possible to construct models with one instant in which a node remains idle for the whole time up to R_i and then must transport C_i plus the backlog resulting from the utilization U_i. These considerations can be casted as the following theorem.

Theorem 3 (utilization under release times). *If a node under a utilization U_{i-1} becomes subject to a periodic* *traffic with backlog C_i, period T_i and latency R_i, its utilization becomes:*

$$U_i = \frac{\min(T_iU_i - C_i, T_i - C_i - R_i)}{T_i + C_i + R_i} \tag{4.7}$$

□

Although this result is an improvement over the basic method, it is clear that the bound is still rather lose and especially if R_i grows high, the scheduling can become rather inefficient. For example note that if $C_i + R_i = T_i$, the remaining utilization drops to zero, so this single transfer can occupy an entire node regardless of the size of C_i.

The reason for this inefficiency is the fact that the utilization parameter does not contain enough information: in the case $C_i + R_i = T_i$ this is evident because when the transfer i starts at time R_i it may not be preempted anymore and any utilization greater than zero would allow another transfer with shorter period to preempt it.

Audsley has proposed in [2] a different policy called *deadline monotonic* scheduling. In this approach the priorities are not assigned according to the periods of the transfers, but instead they are assigned according to the slack available before each deadline. The network latency in this case can be simply considered a part of the size ($C_i \rightarrow C_i + R_i$) and results in a tighter deadline, which in turn results in a higher priority.

The network latency affecting a flow can be calculated according to [16, eq. (15)], which is a latency analysis equation to be solved iteratively. An important property is that if a node is executing a deadline monotonic policy, the utilization-based tests can still be applied: in fact it can be proved that if a set of transfers is schedulable, the deadline monotonic policy implements a feasible schedule.

We can improve theorem 3 for the deadline monotonic case observing that the problem discussed for the case $R_i + C_i = T_i$ does not exist under the deadline monotonic policy because the zero slack corresponds to the top priority. Consequently the minimum operator can be removed, leading to the following theorem.

Theorem 4 (utilization under release times in deadline monotonic scheduling). *If a node under a utilization U_{i-1} and a deadline monotonic policy becomes subject to a periodic traffic with backlog C_i, period T_i and latency R_i, its utilization becomes:*

$$U_i = \frac{T_iU_i - C_i}{T_i + C_i + R_i} \tag{4.8}$$

□

A better test, with the property of being scalable in workload versus accuracy, has been developed by Bini and Buttazzo in [5] but it could not be directly employed to our problems and a redesign to apply its concepts to our context has not been carried out because of a prospective little potential for practical improvements.

4.4 The Nested QoS problem

Rather than carrying out a static timing analysis of the worst-case execution times (WCET analysis), the framework of network calculus extracts performance characteristics from a statistically representative population of workload realizations (soft real-time) or from an analytical model of the workload (hard real-time). In network calculus, a *service curve* provides a *guaranteed lowest bound* on the amount of events which can be processed over time by the workload. This is a deterministic bound and, if the

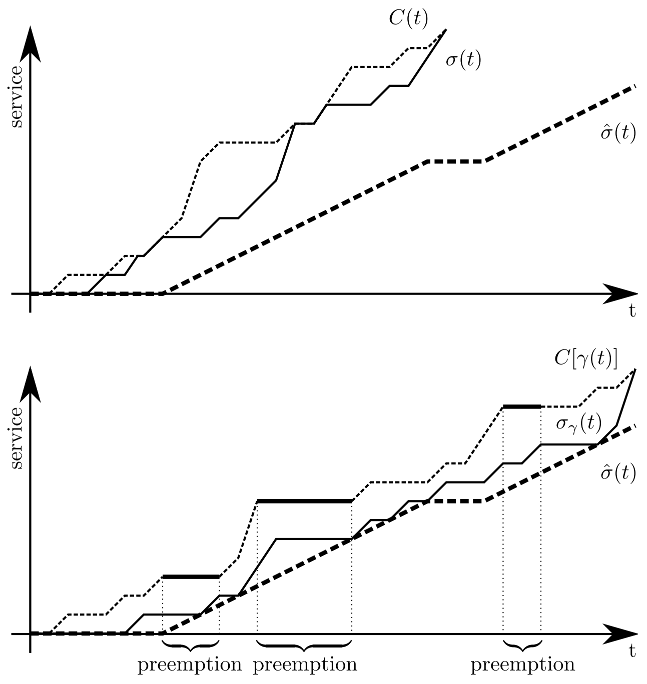

Figure 4.6: Top: a service curve $\sigma(t)$ is obtained from $C(t)$ by (4.9). Bottom: the preemption γ modifies the service curve from $\sigma(t)$ to $\sigma_\gamma(t)$.

worst-case is present among the realizations, it corresponds to the outcome of a WCET analysis. In practice the service curve presents often a more realistic and usually less pessimistic bound than the one derived by a WCET analysis, because it is extremely unlikely that any reasonable single realization can stimulate all longest code paths. Furthermore, WCET analysis can hardly account for *variability* effects such as the ones described in [1], which are more correctly represented in a proper population of workload realizations from which the service curve gets computed.

The lowest service curve which a software application must guarantee is a system demand and can be considered as given: we term it the "target" service curve. Conversely, it can be useful to compute the service curve presented by a software application with the purpose of testing whether the target curve lays below the offered one and hence it is satisfied.

For simplification we consider one single software thread in charge of executing an application, although our results can be easily extended to a whole system if the time scale being regarded is large enough. This thread may be preempted by stalling on NoC accesses only as long as the service curve resulting from the preempted realization of the thread remains above the target one. In order to highlight that our results are holistic and can be employed for any software which can be regarded as sequential at a time scale high enough, rather than referring to a thread we refer to a *virtual processor*.

An example showing the effect of preemption on a service curve is presented in figure 4.6. $C(t)$ is the *capacity* of a software to provide a service over time: we will show later how it can be extracted from several workload realizations. $\sigma(t)$ is the service curve obtained from $C(t)$ according to [18] as

$$\sigma(\Delta) = \min_{t\geq 0}\{C(\Delta + t) - C(t)\} \tag{4.9}$$

and it lays above $\hat{\sigma}(t)$, which is the target service curve. The effect of thread preemption on $C(t)$ is represented by $C[\gamma(t)]$, from which the service curve $\sigma_\gamma(t)$ is obtained as $\sigma_\gamma(\Delta) = \min_{t\geq 0}\{C[\gamma(\Delta+t)] - C[\gamma(t)]\}$ according to (4.9). $\sigma_\gamma(t)$ lays just above $\hat{\sigma}(t)$ and shows that $\gamma(t)$ is one limit of the preemption which $C(t)$ can tolerate if a minimal service $\hat{\sigma}(t)$ has to be guaranteed. In order to determine a valid thread schedule, we must determine for every thread a $\gamma(t)$ such that $C[\gamma(t)]$ delivers $\sigma_\gamma(t) \geq \hat{\sigma}(t)$.

Since we could find no solution for this problem in the existing literature, we have developed a novel algorithm for determining an upper bound on every feasible $\gamma(t)$ and hence providing necessary and sufficient conditions for the delay which may be introduced into the thread without violating the target curve. These upper bounds are called *deadline curves* $\delta_i(t)$ and their calculation can be carried out through algorithm 1 presented in section 4.5.

4.5 Calculation of Deadline Curves

Because the terminology in [8] has become quite widespread, we borrow and slightly reformulate it here to adapt it to our case. In agreement with [18] we mean by "traffic flow" a flow of demand for computation and by the related "backlog" the amount of demand can be placed before results are needed to generate further demand: this quantity models the latency tolerance capability of a processor, e.g. due to outstanding load mechanisms, as well as the latency tolerance capability of an application due to non-blocking accesses to components connected to a NoC.

Definition 1 (arrival curve). *Say $\alpha_i(\cdot)$ the* arrival curve *associated to the i-th virtual processor, such that:*

$$\alpha_i(t) = \sup_{R_i(\cdot)} \sup_{s} \{R_i(t+s) - R_i(s)\}$$

according to [18, Prop. 2], where $R_i(\cdot)$ varies over the family of input flows. □

This means that $\alpha_i(\cdot)$ is the arrival curve for the worst-case input flow.

Definition 2 (variable capacity). *Say $C_i(\cdot)$ the* variable capacity *associated to the trace of the i-th virtual processor, such that $C(t)$ is* maximum guaranteed service *provided by the i-th virtual processor up to t time after it starts.* □

Definition 3 (service curve). *Say $\sigma_i(\cdot)$ the* service curve *associated to the i-th virtual processor, such that:*

$$\sigma_i(t) \doteq \sup_{s} \{C_i(t+s) - C_i(s)\}$$

according to [18, Prop. 3]. □

Let the *maximal virtual delay* to which the i-th traffic flow is subject be D_i and let the *maximal backlog size* reserved to the said traffic flow be M_i: this bounds the service presented to the i-th traffic flow by a *minimal service curve*:

Lemma 1 (minimal service). *Given a maximal virtual delay D_i and a maximal backlog M_i associated to the i-th traffic flow, the* minimal service curve *$\hat{\sigma}_i(t)$ that the i-th thread may present to the flow is*

$$\hat{\sigma}_i(t) \doteq \sup_{t \geq 0} (\alpha_i(t - D_i), \alpha_i(t) - M_i, 0)$$

□

Proof. this is a straightforward consequence of [8, Th. 1.4.1] and [8, Th. 1.4.2] □

A deadline curve $\delta_i(\cdot)$ is a generalization of the usual concept of deadline. We define it as follows:

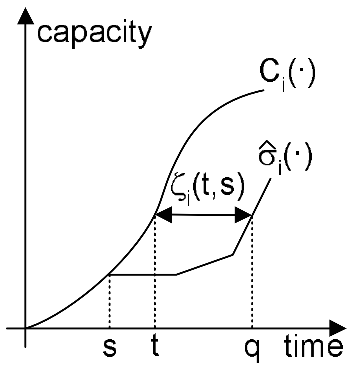

Figure 4.7: Tolerable capacity delay: assuming a flow can start only at time s, the variable capacity $C_i(\cdot)$ can be delayed at time t by an amount $q-t$ without violating the minimal service curve $\hat{\sigma}_i(\cdot)$.

Definition 4 (deadline curve)**.** *The* deadline curve $\delta_i(\cdot)$ *represents the amount of time $\delta_i(t)$ which has to be spent in the i-th virtual processor trace before time t, for avoiding service violations of the lower bound $\hat{\sigma}_i(t)$.* □

Obviously $\delta_i(t)$ is a positive wide-sense increasing function and, since in a scalar architecture no more than one instruction per unit time can be executed, it must hold that:

$$0 \leq \frac{\partial \sum_i \delta_i(t)}{\partial t} \leq 1 \tag{4.10}$$

Based on this definition, the capacity of the i-th virtual processor, once it has been *scheduled*, is $C_i[\delta_i(\cdot)]$. Now conditions must be determined on $\delta_i(\cdot)$, such that the service curve of the scheduled i-th virtual processor is above the the minimal service curve: $\sigma_i(t) \geq \hat{\sigma}_i(t), \forall t$. To this purpose the tolerable capacity delay is defined as follows:

Definition 5 (tolerable delay)**.** *Given a variable capacity curve $C_i(\cdot)$ and a minimal service curve $\hat{\sigma}_i(\cdot)$, let $\zeta_i(t,s)$ be the* tolerable capacity delay *at time t for a service begun at time s:*

$$\begin{aligned}\zeta_i(t,s) \doteq &\inf_{q\geq s}\{q : C_i(t) = \hat{\sigma}_i(q-s) + C_i(s)\}\\ &- sup_{\bar{t}}\{\bar{t} : C_i(\bar{t}) = C_i(t)\}\}\end{aligned}$$

□

Referring to the figure 4.7, a flow starting at time s must obtain at time q at least a service $\hat{\sigma}_i(q-s)+C_i(s)$. $C_i(\cdot)$ reaches this level at t and may not increase it until a time $\bar{t}$, motivating the definition. In order to effectively compute the values, it is useful to introduce the *pseudo-inverse* [8, Def. 3.1.7].

Definition 6 (pseudo-inverse)**.**

$$\begin{aligned}f^U(x) &\doteq \sup_t\{t : f(t) \leq x\} \quad \textit{upper pseudo-inverse}\\ f^L(x) &\doteq \inf_t\{t : f(t) \geq x\} \quad \textit{lower pseudo-inverse}\end{aligned}$$

□

Once $\hat{\sigma}_i^L(\cdot)$ and $C_i^U(\cdot)$ have been computed, the tolerable capacity delay can be obtained as follows.

Lemma 2 (tolerable delay). *The tolerable delay $\zeta_i(t,s)$ of a variable capacity $C_i(\cdot)$ serving within a service bound $\hat{\sigma}_i(\cdot)$ a flow started at time s is:*

$$\zeta_i(t,s) = \hat{\sigma}_i^L[C_i(t) - C_i(s)] + s - C_i^U(t)$$

□

Proof. per definition of $\zeta_i(t,s)$, $C_i(t) = \hat{\sigma}_i(q-s) + C_i(s)$. This leads to $q = \hat{\sigma}_i^L(C_i(t) - C_i(s)) + s$, where the lower pseudo-inverse is required by the inf in the definition of $\zeta_i(t,s)$. The proof is concluded observing that $\zeta_i(t,s) = q - \bar{t}$ and $\bar{t} = C_i^U(t)$ □

For each instant t, there is a minimal tolerable capacity delay $\zeta_i(t)$ which can be achieved for flows starting at a unique instant time $s(t)$:

Definition 7 (minimal tolerable delay). *Given a tolerable delay $\zeta_i(t,s)$, the* minimal *tolerable delay in t is:*

$$\zeta_i(t) \doteq \inf_s\{\zeta_i(t,s)\}$$

and this is achieved for a single bounding instant *$s(t)$ such that:*

$$s(t) \doteq \inf_s\{s : \zeta_i(t,s) \geq \zeta_i(t)\}$$

□

We just write $s(t)$ rather than $s_i(t)$ because we always refer to the same trace i. This capacity delay bound can be employed to bound the deadline curve:

Theorem 5 (deadline bound). *A virtual processor i, sustaining its arrival curve $\alpha_i(\cdot)$, services its flow with a maximal D_i virtual delay and a maximal M_i backlog if its deadline curve is such that:*

$$\delta_i^U(t) - \delta_i^L[s(t)] \leq \zeta_i(t) + t - s(t)$$

□

Proof. the condition on $\delta_i(\cdot)$ which makes $\sigma_i(t) \geq \hat{\sigma}_i(t)$, with $\sigma_i(t)$ being the arrival curve of the scheduled capacity $C_i[\delta_i(\cdot)]$, is:

$$\inf_s\{C_i[\delta_i(t+s)] - C_i[\delta_i(s)]\} \geq \hat{\sigma}_i(t)$$

this condition is equivalent to:

$$C_i[\delta_i(t)] - C_i[\delta_i(s)] \geq \hat{\sigma}_i(t-s), \forall s$$

Referring to figure 4.8, this condition is verified if the delay introduced between t and $s(t)$ by scheduling $C_i(t)$ is less than $\zeta_i(t)$, because $s(t)$ is the earliest point at which the tolerable delay reaches its minimum and $\partial\delta_i(t)/\partial t \leq 1$, i.e. the scheduling cannot increase the tolerable delay.

Say u the "delayed" $s(t)$, i.e. $u = \inf_{\bar{u}}\{\bar{u} : C_i[s(t)] = C_i[\delta_i(\bar{u})]\}$, and q the "delayed" t, i.e. $q = \inf_{\bar{q}}\{\bar{q} : C_i(t) = C_i[\delta_i(\bar{q})]\}$. The scheduling does not violate the minimal tolerable delay if:

$$\zeta_i(t) \geq (q-u) - [t - s(t)]$$

the proof is concluded observing that $s(t) = \delta_i(u)$ and $t = \delta_i(q)$, hence $u = \delta_i^L[s(t)]$ and $q = \delta_i^U(t)$ □

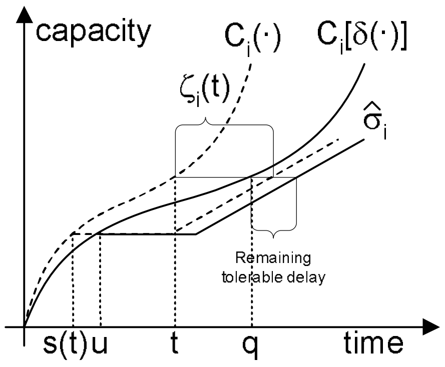

Figure 4.8: Tolerable delay during scheduling: the tolerable delay $\zeta_i(t)$ is reduced after the variable capacity $C_i(\cdot)$ has been scheduled by $\delta_i(\cdot)$.

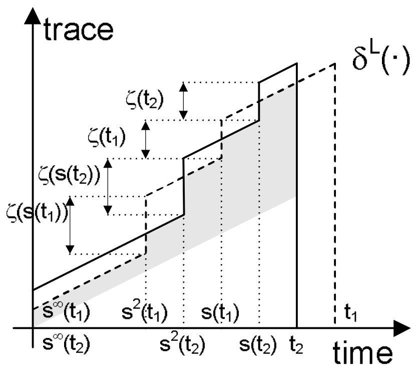

Figure 4.9: Feasible Region for deadlines: the upper bound of the gray area is the pseudo-inverse of the "as-late-as-possible" deadline curve.

This theorem shows that for every instant t, there is a limit to the delay that the deadline curve may introduce during the last $t - s(t)$ execution time. If this limit is not exceeded, the service provided by the scheduled thread's capacity satisfies the requirements.

We can employ this observation for defining a *feasible region* in which the deadline curve must lay.

Algorithm 1 (Feasibility Region)**.** *Say*

$$s^n(t) \doteq \underbrace{s(\cdots s}_{n\ times}(t)\cdots), \Phi_{s,h,e}(t) \doteq \begin{cases} h + t - s, \forall t \in [s, e] \\ 0\ otherwise \end{cases}$$

and build the sequence

$$\begin{aligned} \delta_i^L(t, \bar{t}) &\doteq \Phi_{s(\bar{t}),\zeta_i(\bar{t}),\bar{t}}(t) \\ &\dots \\ \delta_i^L(t, s^n(t_1)) &\doteq \Phi_{s^{n+1}(\bar{t}),\zeta_i(s^n(\bar{t})),s^n(\bar{t})}(t) \\ &\quad + \delta_i^L[t, s^{n-1}(\bar{t})] \end{aligned}$$

then take

$$\delta_i^L(t) = \inf_{\bar{t}}\{\lim_{n\to\infty} \delta_i^L[t, s^n(\bar{t})]\}$$

□

Referring to figure 4.9, it is easy to see that $\lim_{n\to\infty} \delta_i^L(t, s^n(t_1))$ converges to the dashed line from below. For a different time instant t_2, the continuous line in figure 4.9 is generated. Repeating this process for a sufficiently large number of instants and taking the infimum of all the generated lines, an upper bound to $\delta_i^L(t)$ can be determined and this directly corresponds by pseudo-inversion to a lower bound to $\delta_i(t)$.

5. Classes of Service

Based on the traffic and service models studied in sections 3.1 and 3.2, we can identify two main classes of service which can be offered for QoS over NoC: the *periodic / sporadic* classes presented in section 5.1 and the *latency-rate* classes presented in section 5.2. In these sections we describe the information which has to be included in the packet formats for supporting these classes.

5.1 Periodic / sporadic

While scheduling transactions has been traditionally linked to a *priority* field, we have seen that for periodic models the knowledge of the *backlog* per period C_i and the *duration* of the period T_i allow the computation of tighter bounds on real-time traffic. This information can be employed by the software for schedulability testing and for assigning a priority to a transaction and does not need to be included in an header flit: for canonical NoC models which feature virtual channels for wormhole routing and a scheme for prevention of head-of-line-blocking like the one presented later in figure 6.4, the priority field is everything we need in the header.

5.1.1 Indirection of priorities

The priority field must be large enough to accommodate a priority for every possible parallel transfer present in the NoC. The software must hash the values $1/T_i, \forall i$ to an ordered set of priorities such that $T_i > T_j \rightarrow \text{priority}(T_i) <\rightarrow \text{priority}(T_j)$. If the width of this field is limited, this hashing is impossible to achieve in the general case, hence the priority assignment must always be carried out exploiting specific knowledge of the traffic. One possibility to provide a general scheme is to add one level of indirection to the values, hence rather than the priorities themselves the header flit includes merely an index to a table of priorities and the software has the possibility of dynamically updating this table. For doing that, though, it must address all nodes containing a replica of this table and update it, consequently a protocol must be supported to make all transactions update the tables along their static routes. This can be implemented as additional set of parameters on the same model employed for programming tables to define static routes.

5.1.2 Route discovery

In some implementations the addressing scheme of the nodes might enable some *route discovery* techniques: every node can address a multicast set of nodes, so that from source to destination a directed multi-level network is available in which header flits must select one static route, if available, or cause backpressure. This scheme is more demanding on the nodes but eases the traffic management in software. The computation of the remaining utilization, shown throughout chapter 4, can be employed for discovering routes which provide QoS. In this case the priority information is not enough anymore and both C_i and T_i must be communicated in order to allow the nodes to compute themselves their utilization and accept or reject the route.

5.2 Latency-Rate

The studies presented in sections 3.1.2 and 3.2.2 have shown that the only parameters which can be employed by scheduling algorithms are *latency* and *rate* constraints.

Consequently, a node must present a buffer per flow and every flow must be served according to a *leaky bucket* scheme with a specified rate. This information must be local to every node and the only information which must be carried by the traffic flow is an *identifier* which addresses the flits into the corresponding buffers of the nodes.

6. Experimental Results

6.1 Models for networks-on-chip

In order to study QoS in NoCs and carrying out experiments, we have built simulation consisting of the components represented in figure 6.1.

These models can be assembled in arbitrary topologies and include processing of header flits, they are developed in SystemC, employing the SystemQ [17] library for traffic generation, network modeling and probing with data export, for example toward the R package (`http://www.r-project.org`).

The whole work enviroment includes a number of monitors as well as data reporting scripts and setups for Eclipse, GDB and SystemQ.

The SystemC models of the nodes are schematically represented by the block diagram shown in figure 6.2. The models implement output queuing and support *static routing* with n virtual channels. The routing tables in the node models support $m \geq n$ entries and if m transactions are pending in a traffic source while only n virtual channels are available along a static path, this traffic source is subject to backpressure until the individual *flits* have been transported along the route, freeing the virtual channels.

The MAC interfaces support a request/response scheme for generating backpressure under congestion and methods for scheduling can be redefined by inheritance. Inheritance is also the extension mechanism for decoding of additional header flit types, but all parameters can be set as well through configuration files.

Figure 6.3 shows a full mesh configuration which has been used for the experiments reported in this document. All NoC nodes communicate through their network interface with 4-ports nodes. These nodes feature wormhole routing, so interference is only generated if traffic must share a port and a channel.

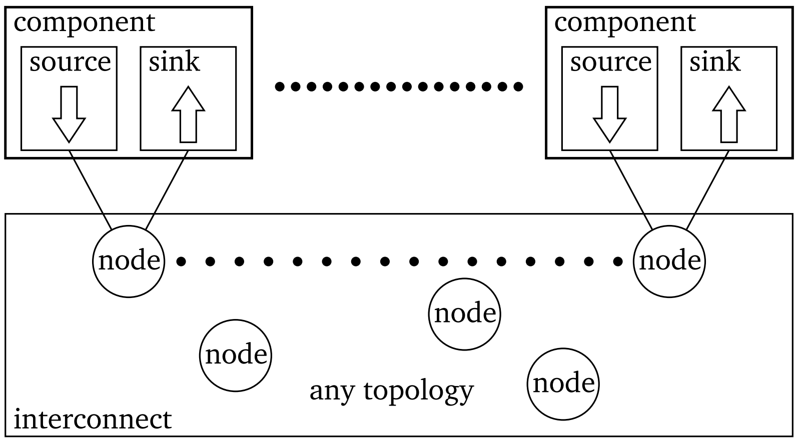

Figure 6.1: Components of the simulation infrastructure

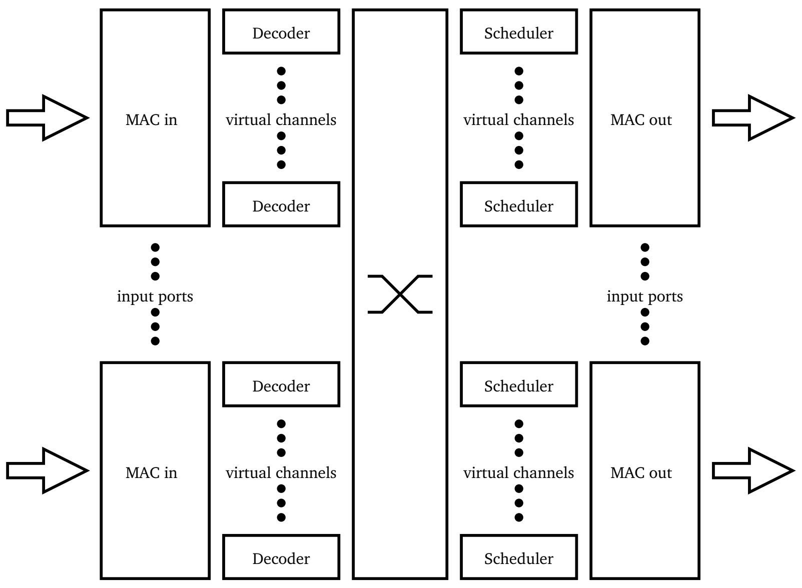

Figure 6.2: Block diagram of a SystemC model of a NoC node

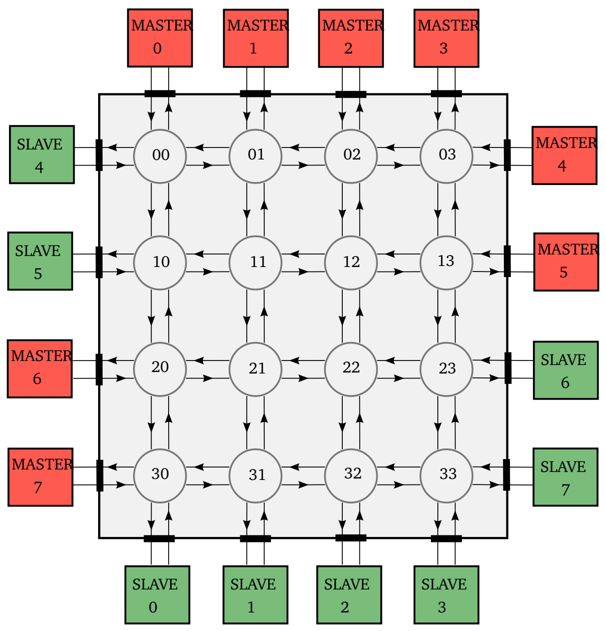

Figure 6.3: One full mash configuration used for experiments.

```
Require: virtual_channels ← [(channel,priority),...]
  random_shuffle(virtual_channels) // shuffle vector of pairs in random order
  for all pair ∈ virtual_channels do
    if priority = pair[1] then
      if last flit of a transfer then
        pair[1] = ∅
      end if
      return pair[0]
    end if
  end for
  for all pair ∈ virtual_channels do
    if pair[1] = ∅ then
      pair[1] = priority
      return pair[0]
    end if
  end for
  return ∅
```

Figure 6.4: Dynamic assignment of virtual channels to prioritized traffic

6.1.1 Note on topologies for NoC

While we present a mesh topology, it is obvious that several different NoC topologies can be designed. In our experiments, we analyze the interference across individual paths which are statically defined hence the overall topology is not really relevant to our results. Indeed, since no dynamic routing is carried out, one could consider the nodes not involved in paths are non-existing. The only reason for having worked on a mesh NoC has been the ease of configuring and monitoring it.

6.1.2 Head-of-Line-Blocking Prevention

The nodes support header flits with priority information and a hashing method for dynamic virtual channel assignment which prevents head-of-line-blocking situations. This method is represented in pseudo-code in figure 6.4.

The vector "virtual_channels" contains the assignments of channel numbers and priorities. Whenever a flit with a priority arrives to a node, a scheduler searches for a virtual channel which is already hashed to this priority. If none is found, it searches for a free virtual channel and on failing that as well, it causes backpressure. If a flit terminates a transfer, then it frees the hashed channel. This simple method, which ensures no head-of-line-blocking if a node sustains as many transfers as virtual channels are available, is sufficient for experimenting and validating QoS algorithms in simulation.

6.2 Simulation platform

Figure 6.5 shows a block diagram for a rather generic multiprocessor system-on-chip which is common in our application domain. An ingress queuing system dispatches packets to an array of processors by storing them in a shared memory system and issuing interrupts to the processors. An application being

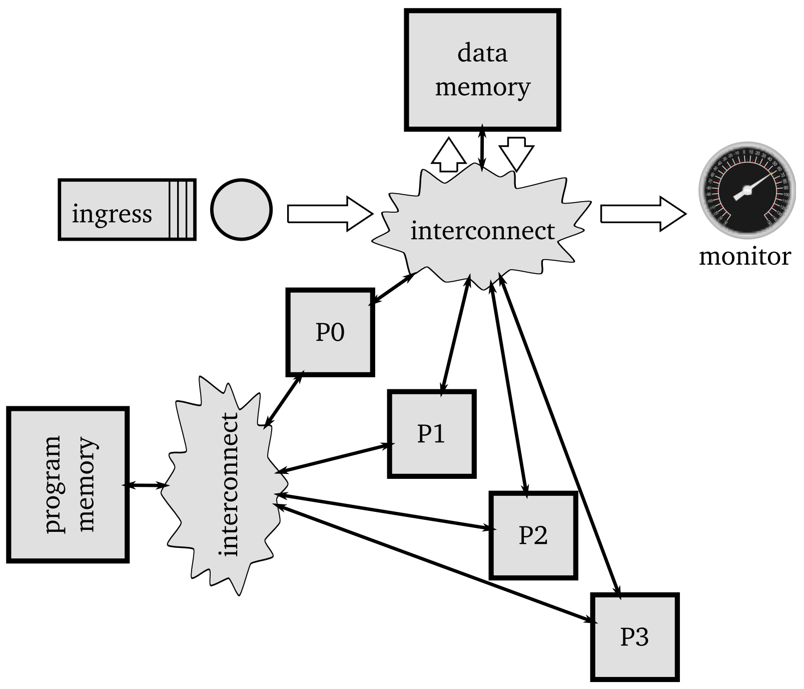

Figure 6.5: Block diagram of a generic multiprocessor system-on-chip.

executed on these processors acknowledges the interrupts and processes the packets and obtain some figures of interest.

In our case study the application has been deep packet inspection and the figure of interest has been the amount of classified data in the packets.

We have employed the Imperas virtual prototyping platform (`http://www.ovpworld.org`) which is freely available as collection of SystemC models and we herewith thankfully acknowledge the kind support of Imperas in helping us to get started.

The software infrastructure on which the simulation environment has been built is:

Linux Red Hat Enterprise Edition
: This is the standard operating system employed for research and development on the Lantiq computer farm.

SystemC 2.2
: The Imperas models have wrappers for SystemC 2.2.

TLM 2.0.1
: The Imperas software employs the SystemC/TLM2 semantics.

GCC 4.1.2
: We have tested that this version links all precompiled object codes.

Imperas OVP for MIPS
: We have employed Imperas MIPS models embedded in SystemC 2.2 / TLM 2.0.1 wrappers.

Wireshark
: Traffic dumps need to be obtained and Wireshark is the premier tool to obtain them.

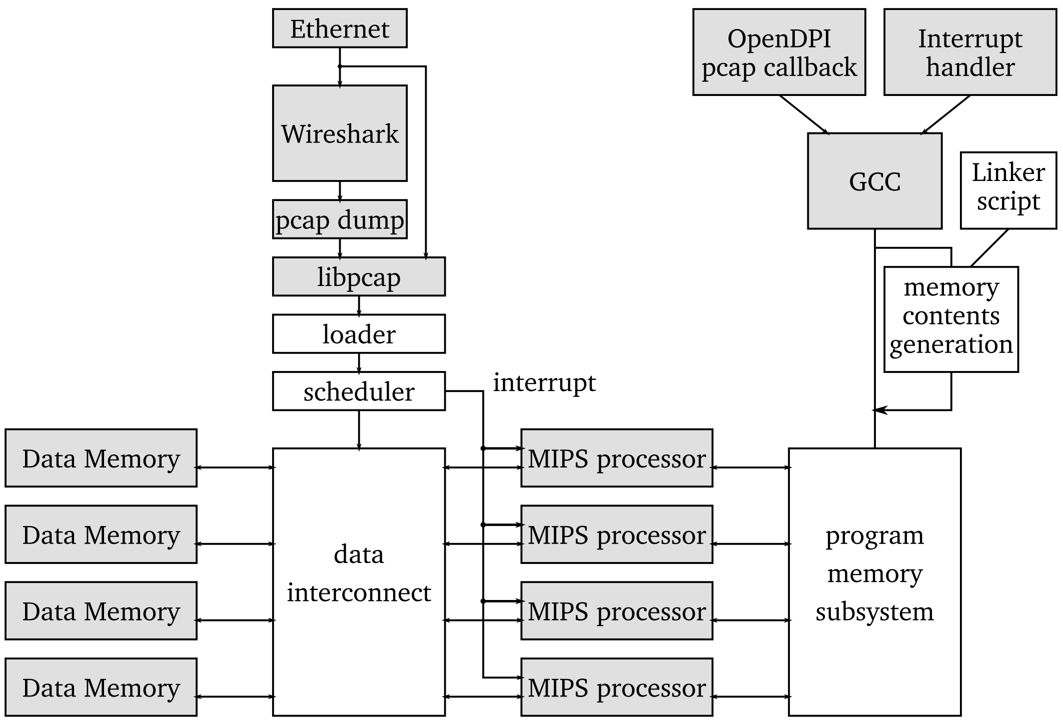

Figure 6.6: Top level specification of the simulation environment, blocks in gray are available from third parties.

The application code has been simulated on bare metal, i.e. without operating system, but employing the standard C library and porting a few additional libraries. The Imperas OVP platform provided code interception (a.k.a. syscall emulation or semihosting) for user output calls to display and store the results. In order to execure code on bare metal, linker scripts as well as boot routines to allocate the code have been employed and interrupt handler has been developed.

6.2.1 Processor-based System

Figure 6.6 presents an top-level specification of the simulation environment, with blocks in gray made available from third parties.

The hardware platform consists of a cluster of parallel processors sharing a program memory and an interleaved data memory. Packets get uploaded to the data memory which is accessed through an interconnect. The interconnect on the program memory is abstracted away but this is a legitimate abstraction because the application code is rather small and would mostly fit within the caches of high-end MIPS processors.

The software platform is based on the classical GCC compilation suite and on the PCAP (packet capture) library for extracting packets sampled by Wireshark.

The SystemC environment includes a loader which performs acquisition from a PCAP interface and a scheduler which looks for the next available processor and triggers it to load a packet from the data memory. The processor acknowledges the interrupt and releases the packet entry on terminating the interrupt service routine.

6.2.2 Software Application

The application of choice is OpenDPI (`http://www.opendpi.org`). This is an open source package for deep packet inspection. It has been created and released from a startup company called Ipoque, which has been acquired by Rodhe & Schwarz. Recently the distribution has been discontinued and the package has been retired from the repositories, but the LGPLv3 (GNU Lesser General Public License Version 3) under which the package has been released gives rights to further utilize, modify and distribute the release.

OpenDPI provides a library of C functions to determine the application layer protocol to which a data packet belongs. For example, OpenDPI can determine with a very low error rate whether an Ethernet frame is transporting a part of a Flash video or a part of a large E-Mail.

A C source file named `OpenDPI_demo.c` contains code which employs the packet capture library *libpcap* (`http://www.tcpdump.org`) to analyze traffic dumps. libpcap library allows registering a C function as *callback* and executes an endless loop which calls this registered callback whenever a packet is received either from an Ethernet card or read from a dump file. Not surprisingly, the C function registered in *OpenDPI_demo.c* is called `pcap_packet_callback()`.

`pcap_packet_callback()` accepts as input a pcap header consisting of packet length, timestamp and pointer to the first byte of the packet, which must be an Ethernet frame, and returns an identification number associated to the application layer protocol which is being transported. This packet header has been assembled in an interrupt service routine which has obtained base address and packet length from the loader unit in the SystemC environment and has been employed to call `pcap_packet_callback()`.

6.3 Results at NoC-level

This chapter presents outcomes from some simulations which validate the QoS formulas for reserving individual nodes. In order to test the utilization bound (3.2), a case study illustrated by figure 6.3 has been set up.

In this case study, a network-on-chip is built out of

- nodes which support virtual channels and
- packet formats which support a priority field.

Every network node ensures that up to n transfers are not affected by *head-of-line-blocking* by hashing the incoming priority to the n available virtual channels: this hashing implies that all transfers with the same priority are served in first-come-first-served policy within the same virtual channel. Routes are static, but can be dynamically defined employing (3.2) to test the availability of node bandwidth for real-time traffic.

Furthermore, network latency has been generated and it has been shown that for met schedulability conditions the deadline is not violated.

Figure 6.8 shows the channel under a low utilization condition: all transfers have some slack (horizontal difference between normal and dashed line). When the traffic level are increased to reach the critical utilization, the plot in figure 6.9 shows that the transfers still remain within the deadlines, leaving zero slack in some critical instants. If the traffic levels exceed this critical utilization limits, the deadline violations shown in figure 6.10 take place.

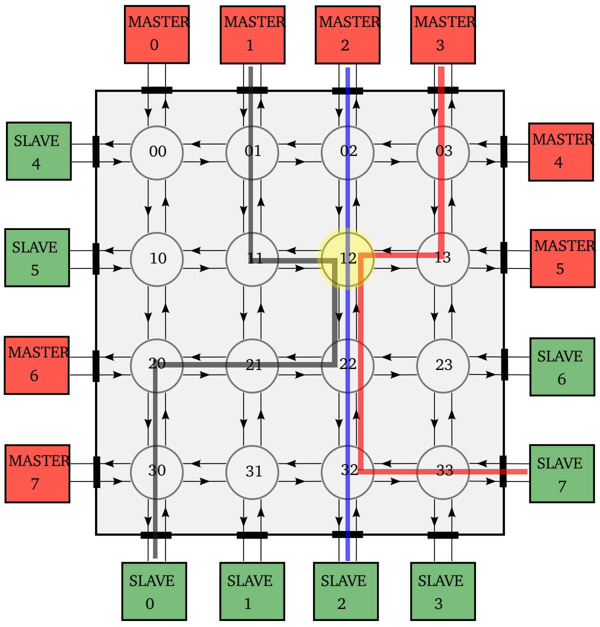

Figure 6.7: Case study for hyperbolic bound on node reservations.

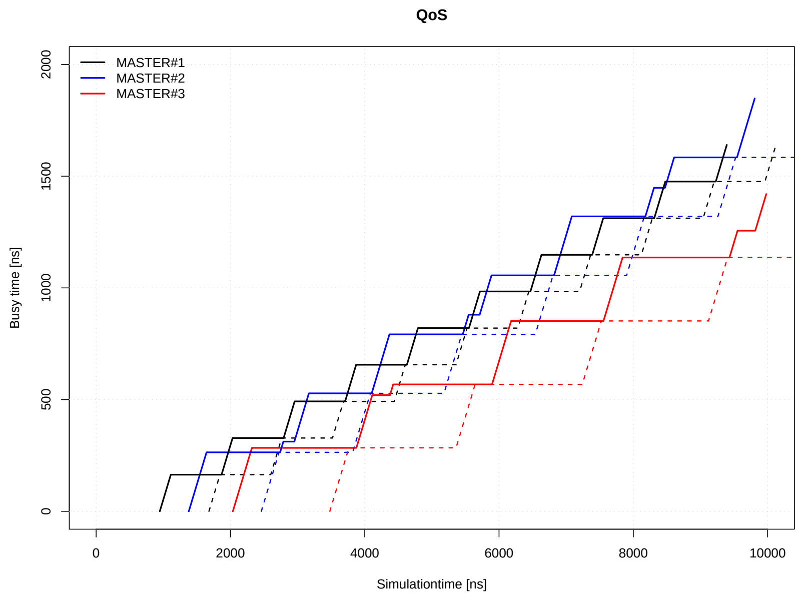

Figure 6.8: Low utilization for figure 6.7: slack available.

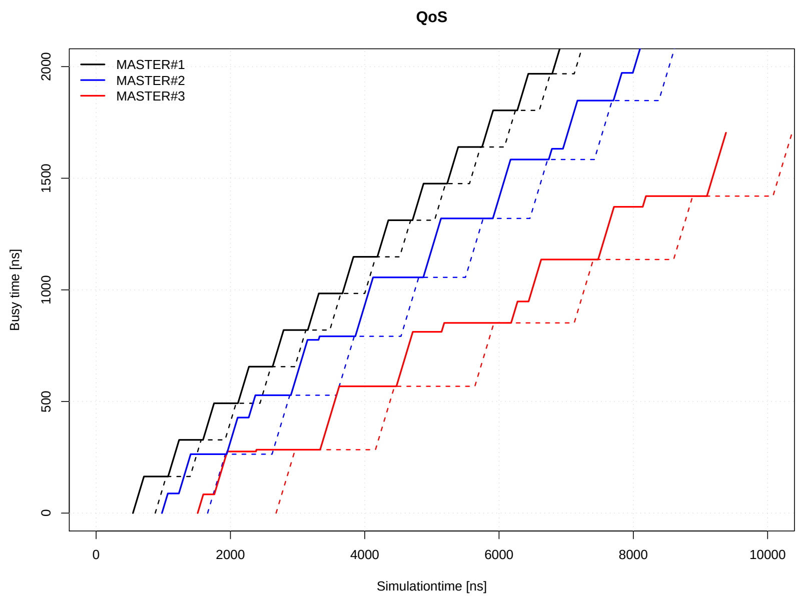

Figure 6.9: Critical utilization for figure 6.7: within deadlines.

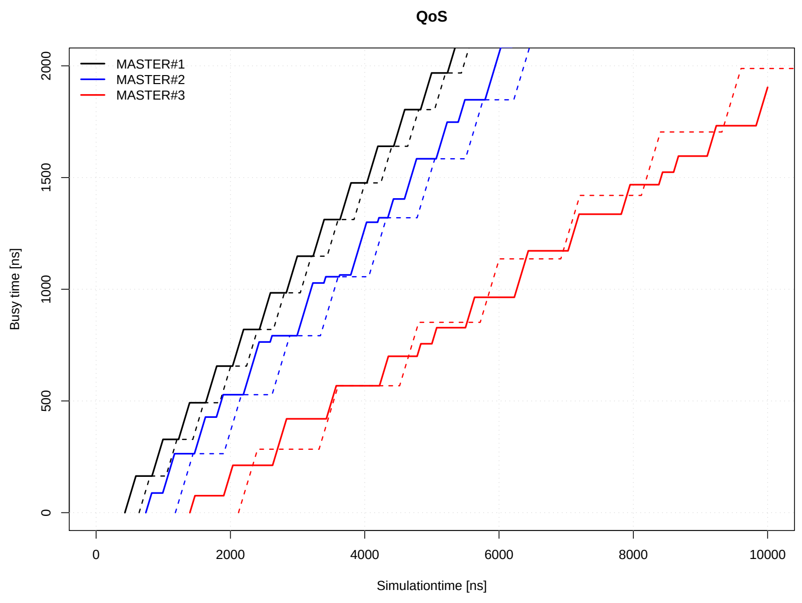

Figure 6.10: Overload condition for figure 6.7: deadlines missed.

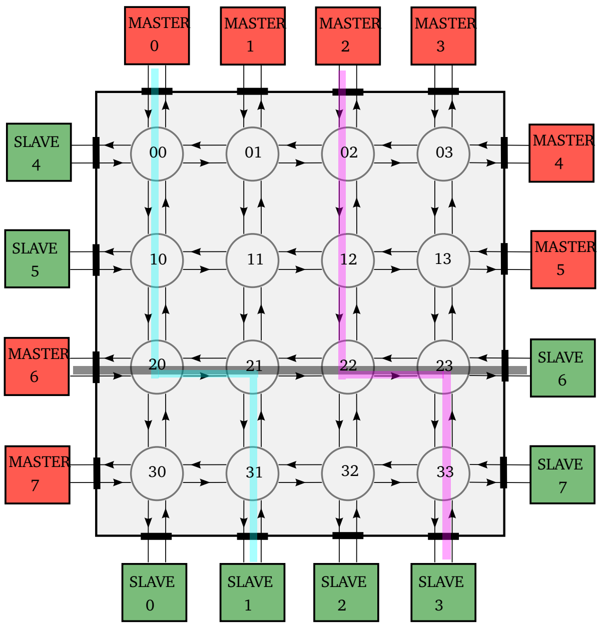

Figure 6.11: Case study for utilization bound on network latency.

The next case study isolates the network latency effect and it is shown in figure 6.11. Also in this case we observe slack under low utilization, in figure 6.12, zero slack under critical conditions, in figure 6.13, and deadline violations under overloading conditions, in figure 6.14.

6.4 Results at application-level

In order to validate our theorems we have experimented the software we have written on synthetic models. Subsequently we have applied them to real world case studies. We have reported some interesting syntetic results in section 6.4.1 and the final outcome of our work in section 6.4.2.

6.4.1 Syntetic results

Figure 6.15 presents three random distributions of delays which affect a nominal transport throughput. These three distributions bear the same mean value and different variances.

The folded gaussian distribution has the lowest variance and the expovariate distribution (the exponential distribution derived from Poisson arrivals) has the largest variance.

The service curves obtained from the transport of a nominal throughput, when affected by a random delay distribution, are shown in figure 6.16. As expected, the calculation has shown that the folded gaussian distribution offers the best responsive behavior while the expovariate distribution the worst one.

The plots on figure 6.16 show that the service curves can be approximated by simpler latency-rate curves. The detailed amount of latency can be observed in figure 6.17, which reproduces the detail of figure 6.16.

For example, the service curve caused by the shown expovariate distribuition can be well approximated by a latency of 1000 time units and a rate of 2 data units per time unit.

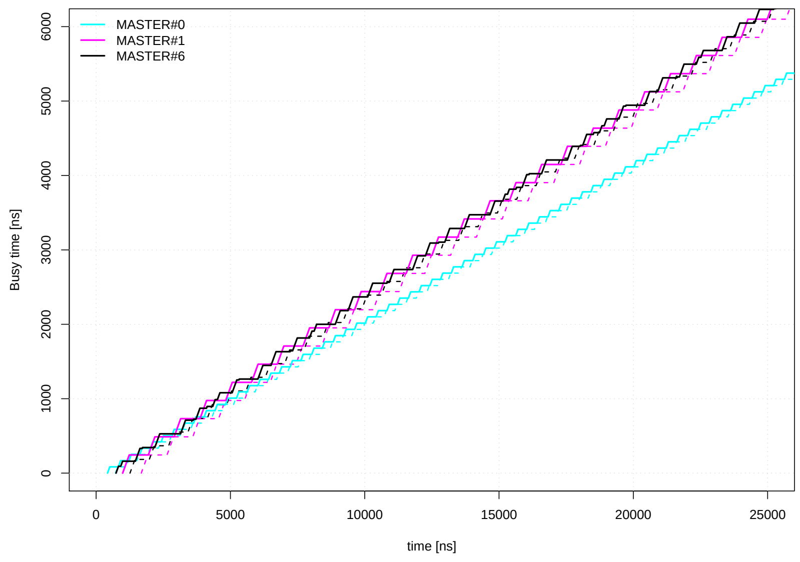

Figure 6.12: Low utilization for figure 6.11: interference with slack.

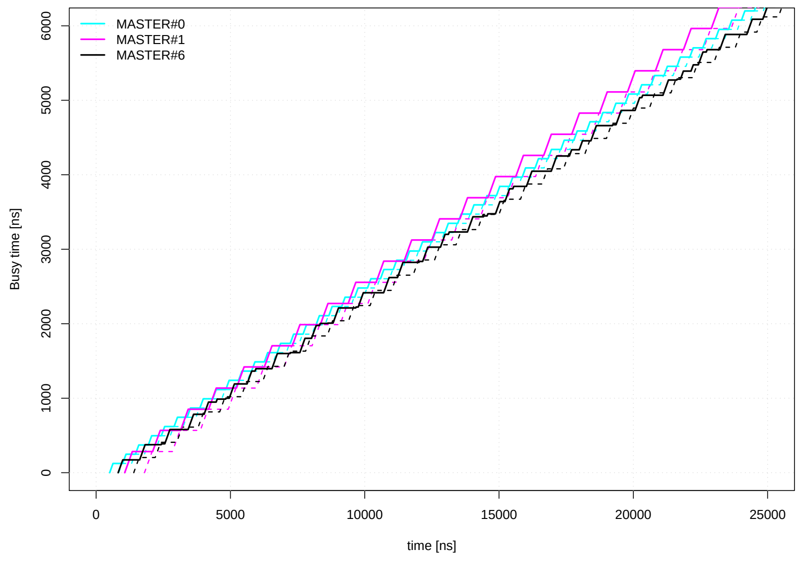

Figure 6.13: Critical utilization for figure 6.11: interference at critical point.

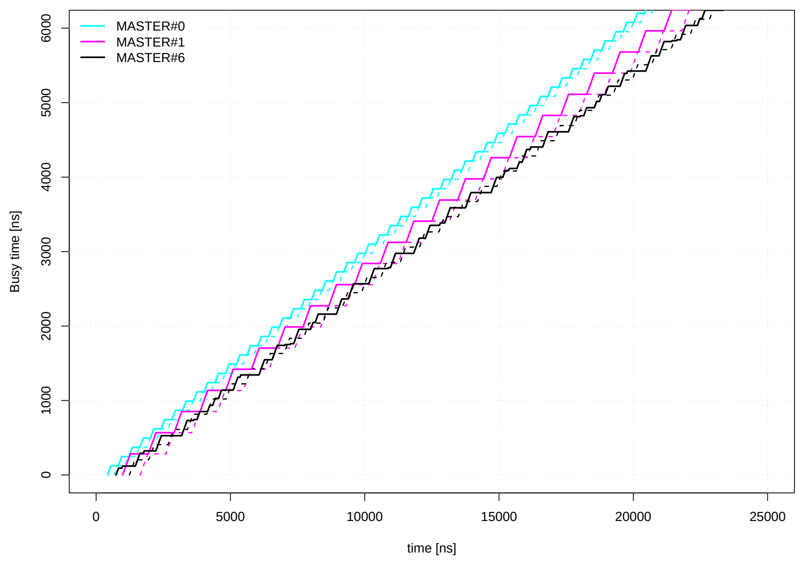

Figure 6.14: Overload condition for figure 6.11: interference under overload.

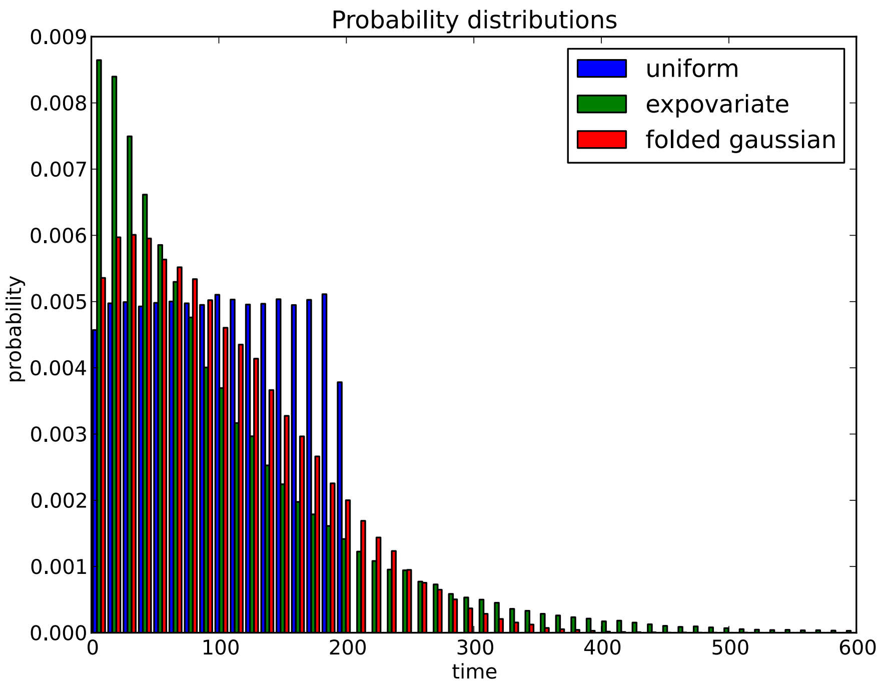

Figure 6.15: Random delay distributions with same average

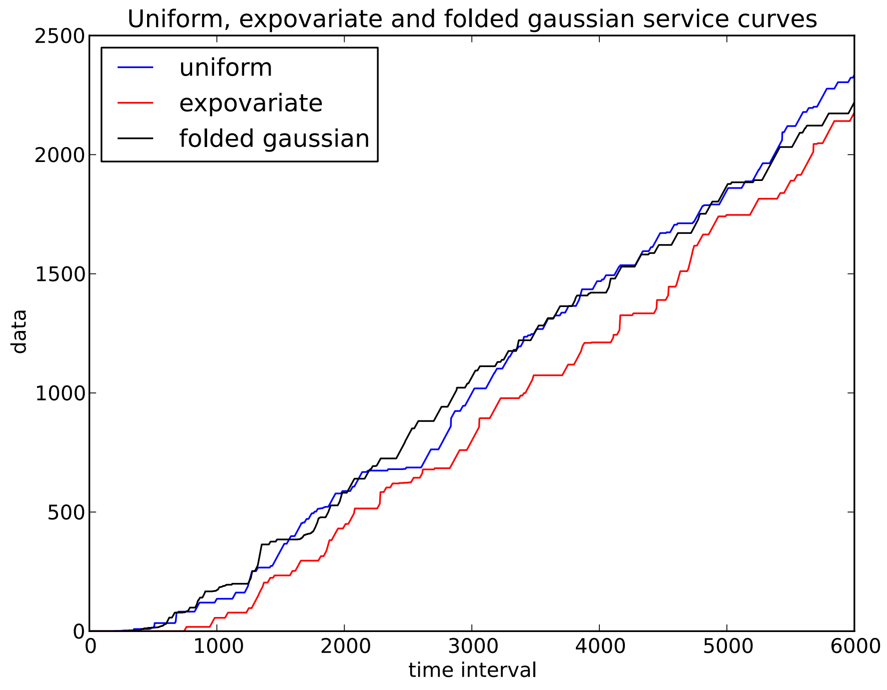

Figure 6.16: Service curves obtained for random delay distributions with same average

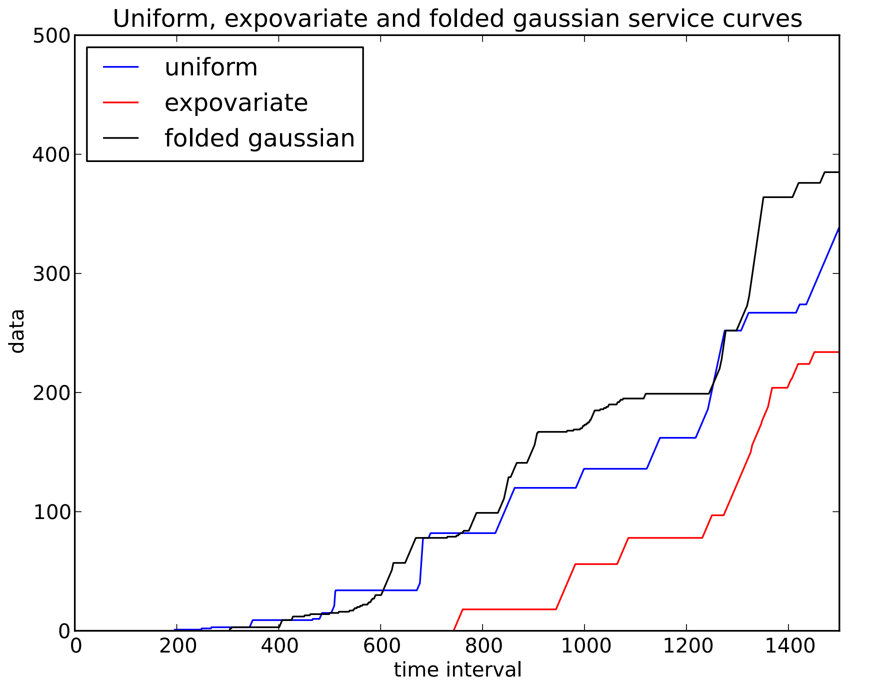

Figure 6.17: Detail of the service curves shown in figure 6.16

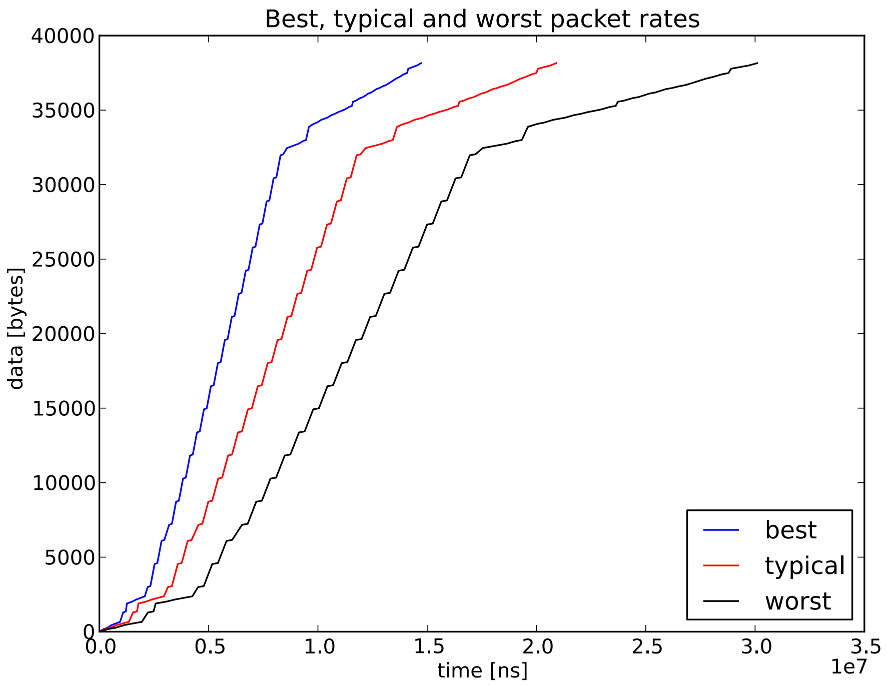

Figure 6.18: Data transport to be achieved under increased latency guarantee in the NoC: the goal is to be compliant with the worst case

6.4.2 Real-world results

In the real-world use case a processor executing the OpenDPI classification on a packet stream is observed while the NoC provides increasing latency guarantees, employing our results on rate and deadline monotonic scheduling.

Figure 6.18 shows the the target service to be achieved by the processor in terms of processed packet backlog over time.

This service is achieved by the processor in worst case assumptions, with a performance drop shown in figure 6.19.

The corresponding service curve obtained on the processor toward the application is reported in 6.20.

This service turns into the service of the application toward the packet stream shown in figure 6.21.

The slack profile which is available to the NoC for enforcing guarantees is shown in figure 6.22. This plot shows that up to 4 ms of program execution (vertical axis) there is a slack of up to 50 ns time available with very few violations (horizontal axis). From 4 ms to 12 ms the available slack drops almost linearly from 50 ns to 10 ns with no violations. Above 12 ns of program execution there are always critical points which leave no available slack, but it is possible to define an amount of acceptable violations to relax the timing, for example we see from the plot that a large number critical instants allow more than 20 ns of slack, so the violations of this deadline could be acceptable from a given application.

The implication of this result is that it is now possible to define soft real-time constraints with an arbitrary degree of “softness” because the amount of possible violations can be established upfront.

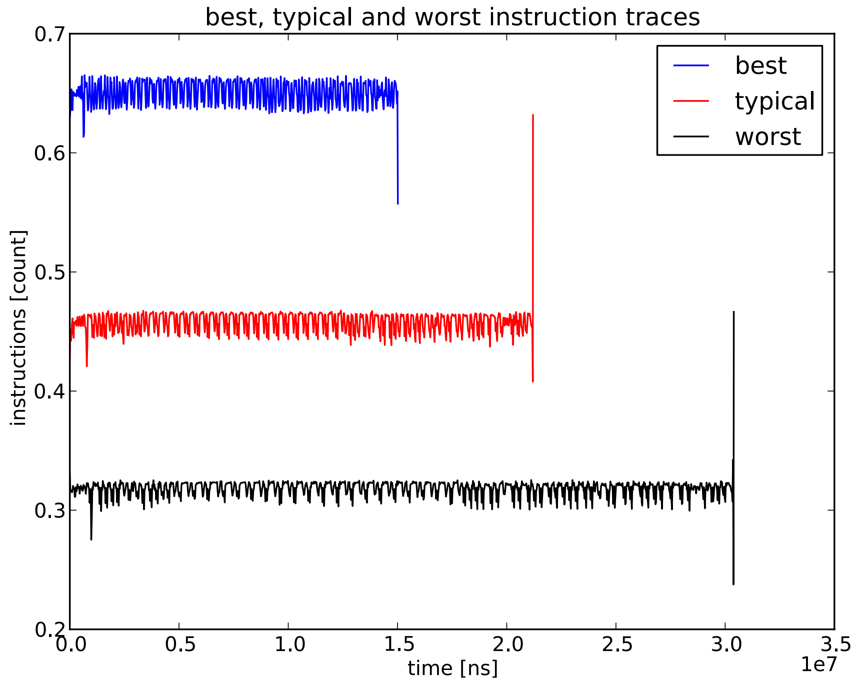

Figure 6.19: Performance of a processor core under increasing latency guarantee on NoC traffic

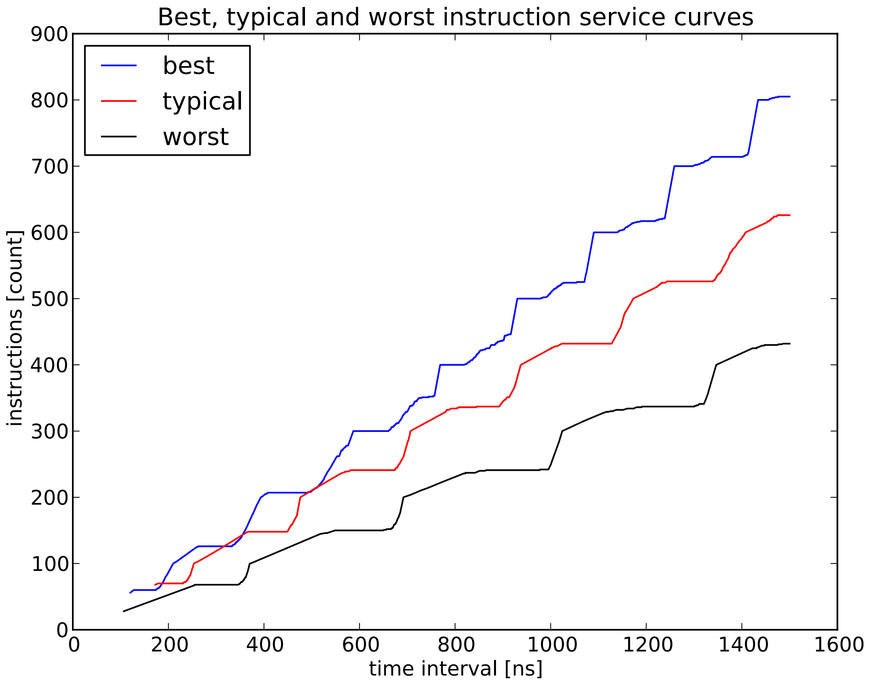

Figure 6.20: Service provided by the processor to the software application under increaded latency guarantee in the NoC

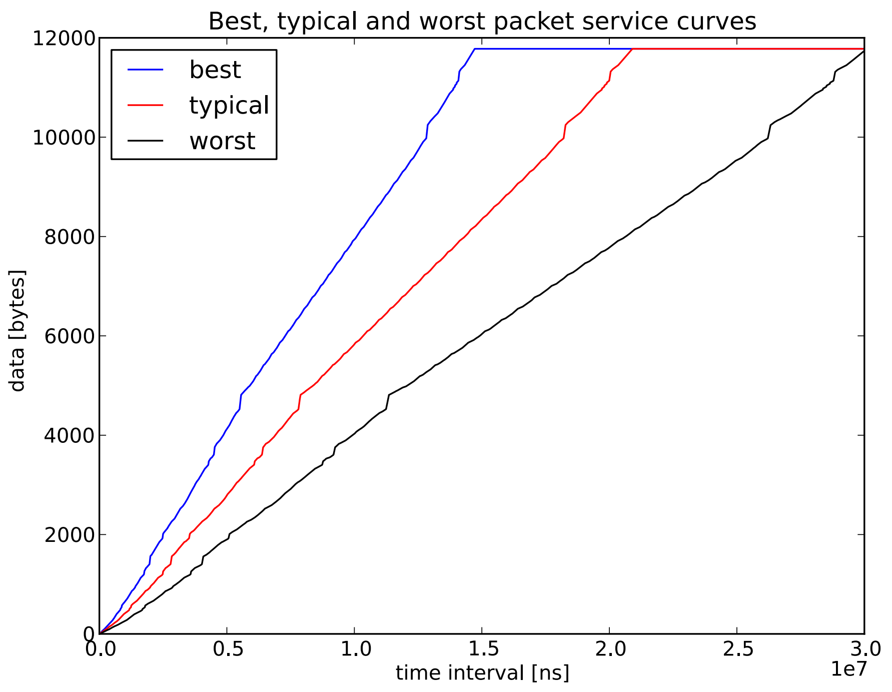

Figure 6.21: Service provided by the application to the packet stream under increased latency guarantee in the NoC

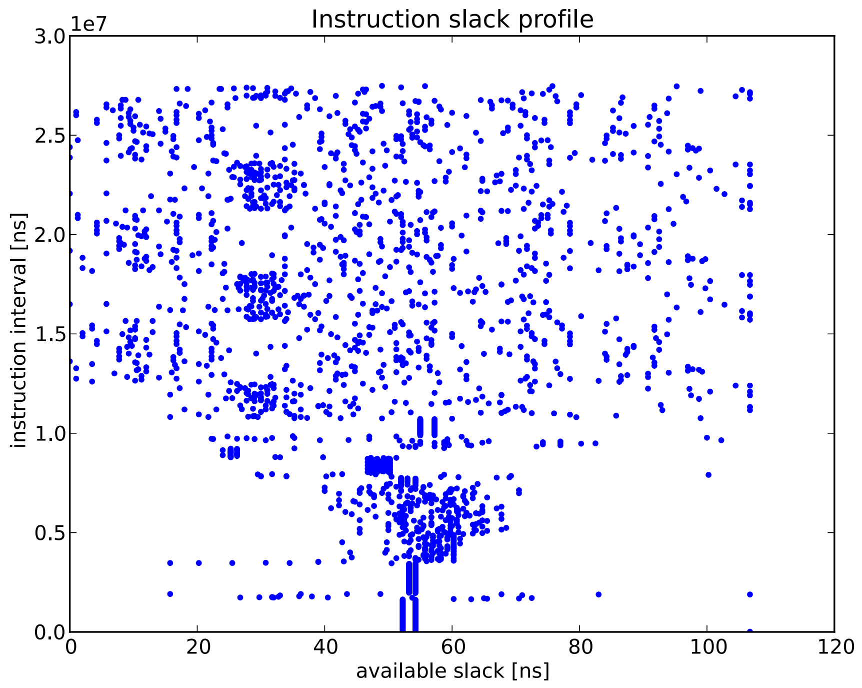

Figure 6.22: Tolerable slack profile obtained on the processor's capacity toward the worst case service to be provided to the packet stream

Bibliography

[1] Alaa R. Alameldeen and David A. Wood. Variability in architectural simulations of multi-threaded workloads. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, page 7, Washington, DC, USA, 2003. IEEE Computer Society.

[2] Neil C. Audsley. Deadline monotonic scheduling. Technical Report September, University of York, 1990.

[3] Neil C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *IEEE Workshop on Real-Time Operating Systems and Software*, 1991.

[4] Enrico Bini, G.C. Buttazzo, and G.M. Buttazzo. Rate monotonic analysis: the hyperbolic bound. *IEEE Transactions on Computers*, 52(7):933–942, 2003.

[5] Enrico Bini and Giorgio C. Buttazzo. The space of rate monotonic schedulability. *Real-Time Systems Symposium, IEEE International*, 0:169, 2002.

[6] Rene L Cruz. A Calculus for Network Delay, Part {I}: Network Elements in Isolation. *{IEEE} Transactions on Information Theory*, 37(1):114–131, 1991.

[7] Rene L Cruz. A Calculus for Network Delay, Part {II}: Network Analysis. *{IEEE} Transactions on Information Theory*, 37(1):132–141, 1991.

[8] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, volume 2050 of *Lecture Notes on Computer Science*. Springer-Verlag, March 2004.

[9] J.Y. Le Boudec. *Network calculus: a theory of deterministic queuing systems for the internet*. Online, 2001.

[10] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[11] Yue Qian, Zhonghai Lu, and Qiang Dou. Qos scheduling for nocs: Strict priority queueing versus weighted round robin. In *ICCD*, pages 52–59. IEEE, 2010.

[12] H Sariowan and RL Cruz. SCED: a generalized scheduling policy for guaranteeing quality-of-service. *IEEE/ACM Transactions on Networking*, 7(5), 1999.

[13] Hanrijanto Sariowan, Rene L Cruz, and George C Polyzos. Scheduling for Quality of Service Guarantees via Service Curves. In *Proceedings of the International Conference on Computer Communications and Networks (ICCCN)*, pages 512–520, September 1995.

[14] Hanrijanto Sariowan, R.L. Cruz, and G.C. Polyzos. Scheduling for quality of service guarantees via service curves. In *icccn*, page 0512. Published by the IEEE Computer Society, 1995.

[15] L. Sha, Tarek Abdelzaher, K.E. Å rzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and A.K. Mok. Real time scheduling theory: A historical perspective. *Real-time systems*, 28(2):101–155, 2004.

[16] Zheng Shi and Alan Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '08, pages 161–170, Washington, DC, USA, 2008. IEEE Computer Society.

[17] Sören Sonntag, Matthias Gries, and Christian Sauer. SystemQ: Bridging the gap between queuing-based performance evaluation and systemc. *Design Automation for Embedded Systems*, 11(2):91–117, September 2007.

[18] Lothar Thiele, Samarjit Chackraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings to the International Symposium on Computer Architectures and Systems (ISCAS 2000)*, pages 101 – 104, May 2000.