Buffer Space Optimisation with Communication Synthesis and Traffic Shaping for NoCs

Sorin Manolache, Petru Eles, Zebo Peng
Linköping University, Sweden
{sorma, petel, zebpe}@ida.liu.se

Abstract

This paper addresses communication optimisation for applications implemented on networks-on-chip. The mapping of data packets to network links and the timing of the release of the packets are critical for avoiding destination contention. This reduces the demand for communication buffers with obvious advantages in chip area and energy savings. We propose a buffer need analysis approach and a strategy for communication synthesis and packet release timing with minimum communication buffer demand that guarantees worst-case response times.

1 Introduction

Networks-on-chip (NoC) have been proposed as alternatives to bus-based systems in the last few years [1, 2, 6]. A key factor to the performance of applications implemented on NoC is the synthesis of the communication. In the context of this paper, we mean by communication synthesis the mapping of data packets transmitted between communicating tasks on network links and the timing of the release of the packets on the links.

A poor synthesis of the communication may lead to a high degree of destination contention at ingress buffers of network switches. Undesirable consequences of this contention include long latency and an increased energy consumption due to repeated reads from the buffers [10]. Moreover, a high degree of destination contention runs the risk of buffer overflow and consequently packet drop with significant impact on the throughput [6]. Even in the presence of a back pressure mechanism, which would prevent packet drops, the communication latency would be severely affected by the packet contention [4]. Thus, in this paper, we concentrate on the communication mapping and packet release timing for applications implemented on NoC.

We focus on two design scenarios, namely the custom design of application-specific NoCs and the implementation of applications on general-purpose NoCs. In the former, the size and distribution of communication buffers can be tailored to precisely fit the application demands. Thus, synthesizing the communication in an intelligent manner could significantly reduce the total need of buffering. In this scenario, the optimisation objective for the communication synthesis approach that we propose is the minimisation of the overall communication buffer space.

In the second design scenario, we assume that an application has to be implemented on a given NoC, with fixed capacity for each buffer. Thus, the challenge consists in mapping the data packets such that no buffer overflow occurs. In both scenarios, it has to be guaranteed that the worst-case task response times are less than the given deadlines, and that the message arrival probability is equal or above an imposed threshold.

Our approach relies on an *analysis* of both timing behaviour and communication buffer space demand at each buffer *in the worst case*. Thus, in both design scenarios, if a solution to the communication synthesis problem is found, we are able to guarantee worst case timing behaviour and worst case buffer space demand, which means that no buffer overflows/packet drops occur.

A related approach for buffer allocation on NoC is given by Hu and Mărculescu [4]. They consider a design scenario in which an NoC is custom designed for a particular application. Hu and Mărculescu propose a method to distribute a given buffer space budget over the network switches. The algorithm is based on a buffer space demand analysis that relies on given Poisson traffic patterns of the application. Therefore, their approach cannot provide application latency guarantees.

Our approach differs in several aspects. First, in addition to buffer allocation, we perform off-line packet routing under timeliness and buffer capacity constraints. Second, we are able to *guarantee* the application latency and that no packets are dropped due to buffer overflows at the switches. Third, we propose a complementary technique that can be independently deployed for the minimisation of the buffer space demand. This technique consists of delaying the release of packets in order to minimise destination contention at the buffers. The method is sometimes referred to as traffic shaping [9].

In previous work [7], we presented an approach for communication mapping with guaranteed latency and communication energy minimisation under the assumption that the network links may fail temporarily. The approach guarantees a designer-imposed lower bound on the message arrival probability by deploying a combination of spatially and temporally redundant communication. In our current work, we focus on buffer space demand analysis and minimisation, issues which we ignored in our previous work.

The rest of the paper is structured as follows. Section 2 describes the system model and gives the problem formulation. Section 3 discusses the two techniques which we propose for solving the problems defined in Section 2.2. Section 4 presents our approach to solving the formulated problems and the buffer demand analysis procedure. Section 5 presents experimental results. Finally, Section 6 draws the conclusions.

2 System model and problem formulation

2.1 System model

We describe the system model and introduce the notation based on the example in Figure 1(a). The hardware platform consists of a 2D array of cores, depicted as squares in the figure. They are denoted with $P_{x,y}$, where x is the 0-based column index and y is the 0-based row index of the core in the array. The inter-core communication infrastructure consists of a 2D mesh network. The small circles in Figure 1(a) denote the switches, while the thick lines connecting the switches denote the communication links. Each core is connected to one switch, as shown in the figure. Each switch, except those on the borders of the 2D mesh, contains five input buffers: one for the link connecting the neighbouring core to the switch, and the rest corresponding to the links conveying traffic from the four neighbouring switches.

The application is modelled as a set of task graphs. Tasks are denoted with τ_i, $0 < i \leq N$, and are depicted by large circles in Figure 1(a). The application shown in the figure consists of three task graphs, $\Gamma_1 = \{\tau_1, \tau_2, \ldots, \tau_6\}$, $\Gamma_2 = \{\tau_7, \tau_8\}$, and $\Gamma_3 = \{\tau_9, \tau_{10}, \tau_{11}\}$. Thin arrows between pairs of tasks $\tau_i \rightarrow \tau_j$ mean that task τ_i passes a message to tasks τ_j. Task τ_j is a successor of τ_i, and τ_i is a predecessor of τ_j. A task with no predecessors is a root task, while a task with no successors is a leaf task. A task may start its execution only after it has received messages from all its predecessors.

Tasks are statically mapped to cores. The execution of tasks sharing the same core is scheduled according to static task priorities. The task execution is assumed to be preemptible. Every task τ_i is characterised by its own period π_i, deadline, priority, and worst case execution time on their corresponding core.

Communication between pairs of tasks is performed by message passing. Messages are characterised by their priority and length. Their transmission on network links is done packet-wise, i.e. the message is chopped into packets which are sent on links and reassembled at the destination core. If an output link of a switch is busy sending a packet while another packet arrives at the switch and demands forwarding on the busy link, the newly arrived packet is stored in the input buffer corresponding to the input link on which it arrived. When the output link becomes available, the switch picks the highest priority packet that demands forwarding on the output link.

Packet transmission on a link is modelled as a task, called communication task. The worst case execution time of a communication task is given by the packet length divided by the link bandwidth. The execution of communication tasks is non-preemptible.

Communication links may temporarily malfunction, with a given probability, scrambling the data of the packets that they carry at the time of the failure. The switches have the capacity to detect scrambled packets, and they do not forward them further. For each pair of communicating tasks $\tau_i \to \tau_j$, the designer may require lower bounds $B_{i,j}$ on the ratio of messages that are received unscrambled at the destination. We define the message arrival probability of the message $\tau_i \to \tau_j$ as the long term ratio $MAP_{i,j} = \lim_{t\to\infty} \frac{S_{i,j}(t)}{\lceil t/\pi_i \rceil}$, where $S_{i,j}$ is the number of messages between tasks τ_i and τ_j that arrive unscrambled at the destination in the time interval $[0,t)$, and π_i denotes the period of the sender task.

In order to satisfy message arrival probabilities imposed by the designer, temporally and/or spatially redundant communication is deployed. In order to define the mapping of redundant messages to network links, we introduced the notion of *communication supports (CS)* [7]. The communication support of a pair of communicating tasks $\tau_i \to \tau_j$, denoted CS_{ij}, is a set of tuples $\{(L,n)\}$, where L is a network link, and n is a strictly positive integer. The packets of the message $\tau_i \to \tau_j$ are sent on the links in CS_{ij}. Given a tuple $(L,n) \in CS_{ij}$, n redundant copies of each packet are sent on link L.

For a message $\tau_i \to \tau_j$, there exist several alternative CSs. Some of them can be considered as possible candidates out of which to select the particular CS_{ij} that will carry the message. They constitute the set of CS candidates for message $\tau_i \to \tau_j$. The construction of the set of CS candidates has been addressed by us in previous work [7], where the trade-offs related to communication fault handling have been taken into consideration. The buffer space minimisation problem, discussed in this paper, is orthogonal to that of CS candidate set construction. Furthermore, in order to concentrate on the buffer space minimisation issue, we will assume that no errors occur on the links throughout the motivational examples in the paper. Nevertheless, we do consider that links may fail in the experimental results section, and consequently we account for them.

2.2 Problem formulation

In this section we define the two problems that we solve in the paper. The input common to both problems consists of:

- The hardware model, i.e. the size of the NoC, and, for each link, the energy-per-bit, the bandwidth, and the probability of a packet to be successfully conveyed by the link;
- The application model, i.e. the task graphs, the mapping of tasks to cores, the task periods, deadlines, worst-case execution times, priorities and the amounts of data to be transmitted between communicating tasks;
- The packet size and message priority for each message;
- The lower bounds $B_{i,j}$ imposed on the message arrival probability $MAP_{i,j}$, for each message $\tau_i \to \tau_j$.

The constraints for both problems are:

- All message arrival probabilities satisfy $MAP_{i,j} \geq B_{i,j}$;
- All tasks meet their deadlines;

The communication synthesis problem with buffer space demand minimisation (CSBSDM) is formulated as follows:

Given the above input, for each message $\tau_i \to \tau_j$ find the communication support CS_{ij}, and determine the time each packet is delayed at each switch, such that the imposed constraints are satisfied and the total buffer space demand is minimised. Additionally, determine the needed buffer capacity of every input buffer at every switch.

The communication synthesis problem with predefined buffer space (CSPBS) is formulated as follows:

Given the above input, and additionally the capacity of every input buffer at every switch, for each message $\tau_i \to \tau_j$ find the communication support CS_{ij}, and determine the time each packet is delayed at each switch, such that the imposed constraints are satisfied and no buffer overflow occurs at any switch.

3 Motivational example

Let us consider the application depicted in Figure 1(a). We assume that each message consists of a single packet. Assuming that messages are mapped on only shortest paths (paths traversing a minimum number of switches), for each message, except the message $\tau_2 \to \tau_3$, there is only one mapping alternative, namely the shortest path. For the message $\tau_2 \to \tau_3$, however, there are two such shortest paths, namely $L_{1,1,E} \to L_{2,1,S}$ and $L_{1,1,S} \to L_{1,0,E}$.

One way to minimise buffer space demand is to intelligently map the message $\tau_2 \to \tau_3$. Let us assume that the message is mapped on path $L_{1,1,E} \to L_{2,1,S}$. Such a situation is depicted in Figure 1(b). The corresponding Gantt diagram is shown in Figure 2(a). The rectangles represent task executions (respectively message transmissions) on the processing elements (respectively communication links) to which the tasks (messages) are mapped.

Message $\tau_2 \to \tau_3$ competes with $\tau_7 \to \tau_8$ for link $L_{2,1,S}$. Message $\tau_7 \to \tau_8$ arrives at the switch connecting tile $P_{2,1}$ to the network while $\tau_2 \to \tau_3$ is conveyed on $L_{2,1,S}$. Due to the unavailability of the link, message $\tau_7 \to \tau_8$ has to be buffered. The situations in which buffering is necessary are highlighted by black ellipses. Messages that have been buffered before being transmitted, due to momentary resource unavailability, are depicted in hashed manner. The total needed buffering space is proportional to the sum of hashed areas. One more such situation occurs in Figure 2(a), caused by the conflict between messages $\tau_5 \to \tau_6$ and $\tau_9 \to \tau_{10}$ on link $L_{0,0,E}$.

We observe that message $\tau_7 \to \tau_8$ needs a relatively large buffering space, which can be avoided by choosing a different mapping alternative for $\tau_2 \to \tau_3$. This mapping is depicted in Figure 1(c), while its corresponding Gantt diagram is shown in Figure 2(b). However, while saving the buffering space required by message $\tau_7 \to \tau_8$, the new mapping introduces a conflict between messages $\tau_2 \to \tau_3$ and $\tau_5 \to \tau_6$ on link $L_{1,0,E}$. As a result, the packet from task τ_5 to task τ_6 has to be buffered at the switch S_{20} in the input buffer corresponding to link $L_{1,0,E}$. Nevertheless, because $\tau_7 \to \tau_8$ does not need to be buffered, we reduced the overall buffer space demand relative to the alternative in Figure 1(b).

As there are no other mapping alternatives, we resort to the second technique, namely traffic shaping, in order to further reduce the total amount of buffering space.

In Figure 2(b), we observe that message $\tau_5 \to \tau_6$ is buffered twice, the first time before being sent on $L_{0,0,E}$, and the second

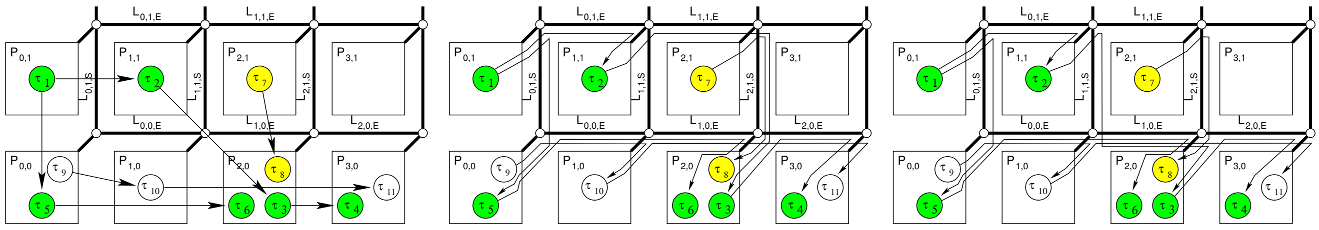


Figure 1: Example application

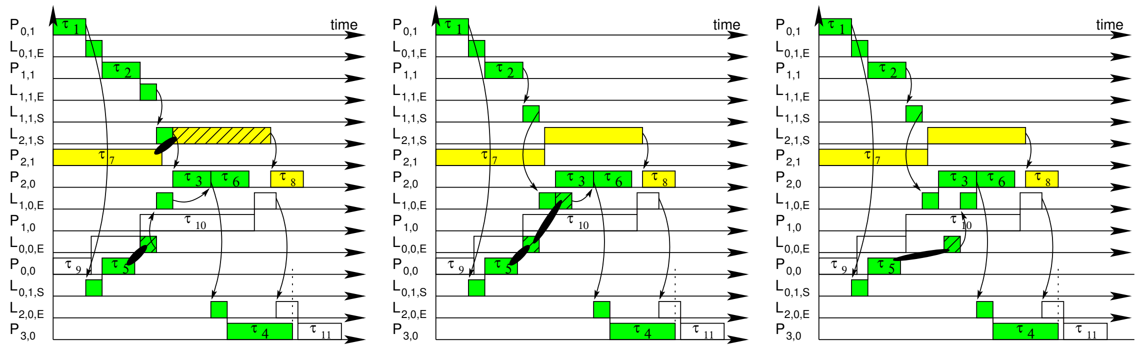


Figure 2: Impact of communication mapping and traffic shaping

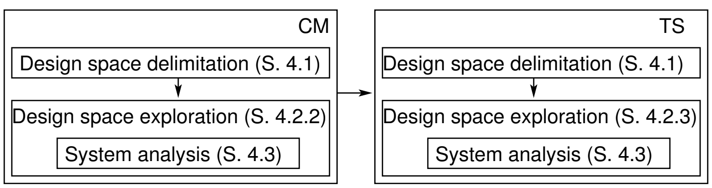


Figure 3: Approach outline

time before being sent on link $L_{1,0,E}$. If we delayed the sending of $\tau_5 \rightarrow \tau_6$, as shown in Figure 2(c), we could avoid the need to buffer the message at link $L_{1,0,E}$. In the particular case of our example, this message delaying comes with no task graph response time penalty. This is because the task graph response time is given by the largest response time among the tasks of the graph (τ_4 in our case), shown as the dotted line in Figure 2, which is unaffected by the delaying of message $\tau_5 \rightarrow \tau_6$. In general, traffic shaping may increase the application latency. Therefore, we deploy traffic shaping with predilection to messages on non-critical computation paths.

The above example demonstrates the efficiency of intelligent communication mapping and traffic shaping when applied to the problem of buffer need minimisation. Obviously, the techniques are also effective in the case of the second problem formulated in Section 2.2, the communication synthesis problem with predefined buffer space.

4 Approach outline

The solution to both problems defined in Section 2.2 consists of two components each: the set of message communication supports and the set of packet delays. Thus, each problem is divided into two subproblems, the communication mapping subproblem (CM), which determines the communication support for each message, and the traffic shaping subproblem (TS), which determines the possible delays applied to forwarding a particular packet. Depending on the actual problem, we will introduce CSBSDM-CM and CSBSDM-TS, and CSPBS-CM and CSPBS-TS respectively.

The outline of our approach is depicted in Figure 3. Solving the communication mapping as well as the traffic shaping subproblem is itself decomposed into three subproblems:

1. Delimit the space of potential solutions (Section 4.1)
2. Deploy an efficient strategy for the exploration of the design space (Section 4.2), and
3. Find a fast and accurate system analysis procedure for guiding the search (Section 4.3).

4.1 Delimitation of the design space

For the CM problem, we addressed the issue of the delimitation of the potential solution space in our previous work [7]. Including all possible CSs for each message in the space of potential solutions leads to a very large space, impossible to explore in reasonable time. Thus, we established criteria for picking only promising CS candidates which we include in the space of potential solutions. For details, the reader is referred to the cited work.

The solution space for the TS problem is constructed as follows. For each tuple $(p_{i,j}, S)$, where $p_{i,j}$ is a packet from task τ_i to task τ_j and S is a network switch on its route, we consider the set of delays $\{0, \Delta, 2\Delta, \ldots, D_j\}$, where Δ is the minimum amount of time it takes for the packet to traverse a network link, and $D_j = \delta_j - WCET_j - H \cdot \Delta$, where δ_j is the deadline of task τ_j, $WCET_j$ is the worst case execution time of task τ_j, and H is the Manhattan distance between the two cores on which tasks τ_i and τ_j are mapped. Delaying the packet $p_{i,j}$ longer than D_j would certainly cause task τ_j to break its deadline δ_j if it executed for its worst case execution time $WCET_j$.

4.2 Exploration strategy

4.2.1 Cost function

The value of the cost function that drives the design space exploration is infinite for solutions in which there exists a task whose response time exceeds its deadline.

The cost function for the CSBSDM-CM and CSBSDM-TS subproblems is $\sum_{b \in B} d_b$, where B is the set of all switch input buffers, b is a buffer in this set, and d_b is the maximum demand of buffer space of the application at buffer b.

The cost function for the CSPBS-CM and CSPBS-TS subproblems is $max_{b \in B}(d_b - c_b)$, where c_b is the capacity of buffer b. Solutions of the CSPBS problem with strictly positive cost function

```
(1)   sm = sort_messages;
(2)   for each msg in sm do
(3)       CS[msg] = select(msg, candidates[msg]);
(4)       if CS[msg] = NONE then
(5)           abort NO SOLUTION;
(6)   return CS;
      select(msg, cand_list):
(7)   cost = ∞; selected = NONE;
(8)   for each cnd in cand_list do
(9)       CS[msg] = cnd; crt_cost = cost_func;
(10)      if crt_cost < cost then
(11)          selected = cnd; cost = crt_cost;
(12)  return selected;
```

Figure 4: Heuristic for communication mapping

```
(1)   sct = sort_comm_tasks;
(2)   for each τ in sct do delay[τ] = shape(τ);
      shape(τ):
(3)   cost = ∞;
(4)   for delay[τ] = 0.0; delay[τ] < D_τ; delay[τ] ← delay[τ] + Δ
(5)       crt_cost = cost_func;
(6)       if crt_cost < cost then
(7)           best_delay = delay[τ]; cost = crt_cost;
(8)   end for;
(9)   return best_delay;
```

Figure 5: Heuristic for traffic shaping

value do not satisfy the buffer space constraint and are thus unfeasible.

4.2.2 Communication mapping

We propose a greedy heuristic for communication mapping. We map messages to CSs stepwise. At each step, we map one message and we obtain a partial solution. When evaluating partial solutions, the messages that have not yet been mapped are not considered.

The heuristic proceeds as shown in Figure 4, lines 1–6. It returns the list of communication supports for each message if a feasible solution is found (line 6) or aborts otherwise (line 5). Before proceeding, we sort all messages in increasing order of their number of mapping alternatives (line 1). Then, we iterate through the sorted list of messages *sm*. In each iteration, we select a mapping alternative to the current message (line 3).

The selection of a mapping alternative out of the list of candidates (determined in the previous step, Section 4.1) is shown in Figure 4, lines 7–12. We iterate over the list of mapping alternatives (line 8) and evaluate each of them (line 9). We select the alternative that gives the minimum cost (line 11).

The motivation for synthesizing the communication in the particular order of increasing number of mapping alternatives of messages is the following. We would like to minimise the chance that the heuristic runs into the situation in which it does not find any feasible solution, although at least one exists. If messages enjoying a large number of mapping alternatives are mapped first, we restrict the search space prematurely and gratuitously, running the risk that no feasible mapping is found for other messages among their few mapping alternatives.

4.2.3 Traffic shaping

The greedy heuristic, shown in Figure 5, determines the amount of time each communication task has to be delayed (a.k.a. shaping delay). As a first step, we sort the communication tasks according to a criterion to be explained later (line 1). Let τ be a communication task that is part of the message from task τ_i to task τ_j. Then, for all communication tasks in the sorted list we find the appropriate shaping delay (line 2). The selection of a shaping delay of a communication task is performed by the function *shape* (lines 3–9). We probe shaping delays ranging from 0 to $D_j = \delta_j - WCET_j - H \cdot \Delta$ in increments of Δ (see Section 4.1). For each probed shaping delay, we evaluate the cost of the obtained partial solution (line 5). When calculating it, we assume that the shaping delay of those tasks for which none has yet been chosen is 0. We select the shaping delay that leads to the minimum cost solution (lines 6–7).

```
(1)   Buf = 0; b = 0; t = 0; F = R_0(t); F_1 = F;
(2)   loop
(3)       t' = next_t';
(4)       F' = R_0(t');
(5)       if t' = F' then
(6)           return Buf;
(7)       b' = (F' − F) · bw + b − (t < F_1)?1 : 0;
(8)       if b' > Buf then
(9)           Buf = b';
(10)      if t' > F_1 then
(11)          b := b − (t' − max(t, F_1) − (F' − F)) · bw;
(12)      t = t'; F = F';
(13)  end loop;
```

Figure 6: Buffer space analysis algorithm

Before closing this section, we will explain in which order to perform the shaping delay selection. We observe that communication tasks on paths whose response times are closer to the deadline have a smaller potential for delaying. Thus, delaying such communication tasks runs a higher risk to break the timeliness constraints. In order to quantify this risk, we compute the worst case response time R_τ of each leaf task τ. Then, for each task τ_i we determine $\mathcal{L}(\tau_i)$, the set of leaf tasks τ_j such that there exists a computation path between task τ_i and τ_j. Then, to each task τ_i we assign the value $prt_i = min_{\tau \in \mathcal{L}(\tau_i)}(\delta_\tau - R_\tau)$. Last, we sort the tasks in decreasing order of their prt_i.[1] In case of ties, tasks with smaller depths[2] in the task graph are placed after tasks deeper in the graph. (If tasks with small depths were delayed first, their delay would seriously restrict the range of feasible delays of tasks with large depths.)

4.3 System analysis procedure

In order to be able to compute the cost function as defined in Section 4.2, we need to determine the worst-case response time of each task as well as the buffering demand at each buffer in the worst case. To do so, we extended the schedulability analysis algorithm of Palencia and González [8].

At the core of the worst case response time calculation of task τ_i is a fix-point equation of type $w_i = R_i(w_i)$. $R_i(t)$ gives the worst case response time of task τ_i when considering interference of tasks of higher priority than that of τ_i that arrive in the interval $[0,t)$. The time origin is considered the arrival time of task τ_i. Thus, evaluating R_i at two time moments, t_1 and t_2, allows us to determine the execution time demanded by higher priority tasks arrived during the interval $[t_1,t_2)$. More details regarding the calculation of the worst case response time can be found in cited work [8, 7]. Here we will concentrate on our approach to buffer demand analysis. For communication tasks, their "execution time" on their "processors" are actually the transmission times of packets on network links. This transmission time is proportional to the length of the packet. Thus, by means of the analysis of Palencia and González, that can determine the execution time demanded during a time interval, we are able to determine the buffering demand arrived during the interval.

The algorithm for the calculation of the buffer space demand of an ingress buffer of an arbitrary network link is given in Figure 6. We explain the algorithm based on the following example.

Let us consider the following scenario. Prior to time moment 0, a 400MHz link is idle. The links convey the bits of a word in parallel, with one word per cycle. At time moment 0, the first word of a 6-word packet p_1 arrives at the switch and is immediately conveyed

[1] The procedure can be easily generalised for the case in which not only leaf tasks have deadlines.

[2] The depth of a task τ is the length of the longest computation path from a root task to task τ.

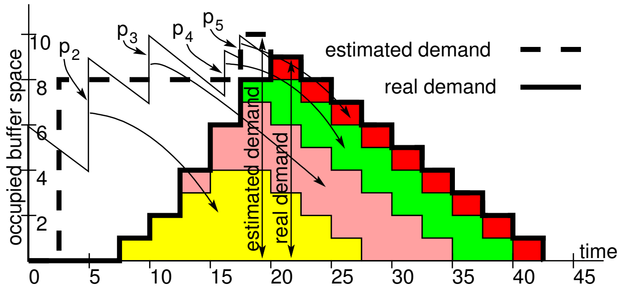


Figure 7: Waiting time and buffer demand

on the link without buffering. The following packets subsequently arrive at the switch and demand forwarding on the link: p_2, 5 words long, arrives at 5ns, p_3, 3 words long, arrives at 10ns, p_4, 2 words long, arrives at 15.25ns, and p_5, 1 word long, arrives at 17.5ns. Let us assume that a fictive packet p_0 of zero length and of very low priority arrived at time 0^+, i.e. immediately after time 0. We compute the worst case buffer space need based on the worst case transmission time of this fictive packet.

The scenario is shown in Figure 7. Time is shown on the abscissa, while the saw-teeth function shows the instantaneous communication time backlog, and the solid step function shows the instantaneous amount of occupied buffer space. The arrows pointing from the steps in the backlog line to the shaded areas show which message arrival causes the corresponding buffering.

The time interval during which the link is busy sending packets is called the busy period. In our example the busy period is the interval $[0,42.5)$, as can be seen on the figure. The main part of the algorithm in Figure 6 consists of a loop (lines 2–13). A subinterval $[t,t')$ of the busy period is considered in each iteration of the loop. In the first iteration $t=0$ while in iteration i, t takes the value of t' of iteration $i-1$ for all $i>1$ (line 12). F and F' are the times at which the link would be idle if it has to convey just the packets arrived sooner than or exactly at times t and t' respectively (lines 1 and 4). t', the upper limit of the interval under consideration in each iteration, is obtained as shown in line 3. For the moment, let us consider that $next\_t' = F$ and we will discuss the rationale and other possible choices later in the section.

For our example, only packet p_1 of 6 words is to be sent just after time 0. Hence, $R_0(0^+) = 6\text{words}/0.4\cdot 10^{-9}\text{words/sec} = 15\text{ns}$. The first iteration of the loop considers the interval $[t=0, t'=F=R_0(0^+)=15)$. We compute $F' = R_0(t'=15)$ (line 4) and we get 35ns, i.e. the 15ns needed to convey the six words of packet p_1 plus the $5\text{words}/0.4\cdot 10^{-9}\text{words/sec} = 12.5\text{ns}$ needed to convey packet p_2 plus the 7.5ns needed to convey packet p_3 (p_2 and p_3 having arrived in the interval $[0,15)$). The time by which the link would become idle if it has to convey just the packets arrived prior to $t' = 15\text{ns}$ is greater than t'. Hence, there are unexplored parts of the busy period left and the buffer space calculation is not yet over (lines 5–6). The packets that arrived between 0 and 15ns extended the busy period with $F'-F = 20\text{ns}$, hence the number of newly arrived words is $(F'-F)\times bw = 20\text{ns}\times 0.4\cdot 10^{-9} = 8\text{words}$. The algorithm is unable to determine the exact time moments when the 8 words arrived. Therefore, we assume the worst possible moment from the perspective of the buffer space demand. This moment is time t^+, i.e. immediately after time t. The 8 words are latched at the next clock period after time $t^+=0$, i.e. at 2.5ns. b', the amount of occupied buffer after latching, is b, the amount of occupied buffer at time t, plus the 8 words, minus possibly one word that could have been pumped out of the buffer between t and $t+2.5$ns. During the time interval $[0, F_1 = 15)$, where F_1 is the time it takes to convey packet p_1, the words conveyed on the link belong to p_1, which is not stored. Therefore, no parts of the buffer are freed in the interval $[0,F_1)$ (see line 7). If the required buffer space is larger than what has been computed so far, the buffer space demand is updated (lines 8–9). Because no buffer space is freed during the interval $[0,15)$, lines 10–11 are not executed in the first iteration of the loop.

The second iteration considers the interval $[t=15, t'=35)$. $F = 35$ns and $F' = 42.5$ns in this case. Hence, $(F'-F)\cdot bw = 7.5\text{ns}\times 0.4\cdot 10^{-9}\text{words/sec} = 3\text{words}$ arrived during interval $[15,35)$. The three words are considered to have arrived at the worst moment, i.e. at 15^+. They are latched at time 17.5ns when $b = 8-1$, i.e. the 8 words that are stored in the buffer at 15ns minus one word that is pumped out between 15 and 17.5ns. Thus b', the amount of occupied buffer at 17.5ns is $8-1+3=10$ (line 7). The value Buf is updated accordingly (lines 8–9). Between 15 and 35ns some words that were stored in the buffer are sent on the link and therefore we have to account for the reduction of the amount of occupied buffer. Thus, the amount of occupied buffer at 35ns is equal to 8, the amount present at 15ns, plus the 3 words that arrived between 15 and 35ns and minus the $(35-15)\times 0.4\cdot 10^{-9} = 8$ that are conveyed on the link in the interval $[15,35)$ (see lines 10–11).

The third iteration considers the interval $[35,42.5)$. As no new packets arrive during this interval, $t' = R_0(t') = 42.5$ and the algorithm has reached fix-point and returns the value of Buf.

We will close the section with a discussion on $next\_t'$, the complexity of the algorithm, and the trade-off between the algorithm execution speed and accuracy.

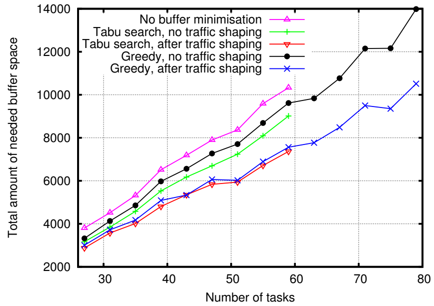
The actual amount of occupied buffer is shown as the thick solid line in Figure 7, while the amount, as estimated by the algorithm, is shown as thick dotted line. We observe that the analysis procedure produces a pessimistic result. This is due to the fact that the analysis assumes that the new packets which arrive in the interval $[t,t')$ arrive always at the worst possible moment, that is moment t^+. If we partitioned the interval in which the link is busy sending packets into many shorter intervals, we could reduce the pessimism of the analysis, because fewer arrivals would be amassed at the same time moment. However, that would also imply that we invoke function R_0 more often, which is computationally expensive. Thus, there exists a trade-off between speed of the analysis and pessimism, which is reflected in the choice of $next\_t'$ (line 3). A value closer to t would lead to short intervals, i.e. less pessimism and slower analysis, while a value farther from t would lead to longer intervals, i.e. more pessimistic but possibly (not necessarily, as shown below) faster analysis.

In our experiments, we use $next\_t' = F$, which is the finishing time of the busy period if no new packets arrive after time t. Choosing a value larger than F would incur the risk to overestimate the busy period. As a result, packets that arrive after the real finishing time of the busy period might wrongly be considered as part of the current busy period. On one hand that leads to the overestimation of the buffer space, and on the other hand it increases the time until the loop in Figure 6 reaches fix-point. In our experiments, choosing $next\_t' = 1.6\cdot F$ results in a 10.3% buffer overestimation and a 2.3× larger analysis time relative to the case when $next\_t' = F$. Conversely, choosing smaller values for $next\_t'$ lead to reductions of at most 5.3% of the buffer space estimate while the analysis time increased with up to 78.5%.

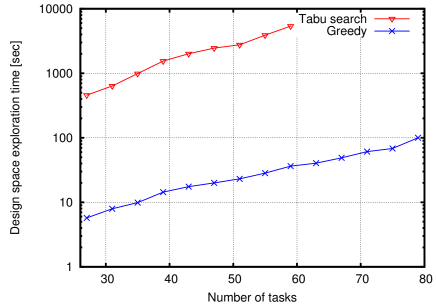
The algorithm is of pseudo-polynomial complexity due to the calculation of function R [8].

5 Experimental results

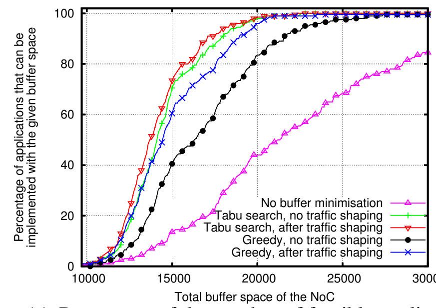
We use a set of 225 synthetic applications in order to assess the efficiency of our approach to solve the CSBSDM problem. The applications consist of 27 to 79 tasks which are mapped on a 4×4 NoC. The probability that a 110-bit packet traverses one network link unscrambled is 0.99, while the imposed lower bound on the message arrival probability is also 0.99. Due to the fact that the implementation of the packet delay capability could excessively



(a) Buffering space vs. number of tasks



(b) Run time comparison



(c) Percentage of the number of feasible applications as a function of the NoC buffer capacity

Figure 8: Experimental results

increase the complexity of the switches, we have considered that traffic shaping is performed only at the source cores. This has the advantage of no hardware overhead.

For each application, we synthesized the communication using three approaches and we determined the total buffer space demand obtained in each of the three cases. In the first case, we use the buffer space minimisation approach presented in the current paper. In the second case, we replaced the greedy heuristics described in Section 4.2 with tabu search based [3] heuristics that are assumed to generate close to optimal solutions provided that they are let to explore the design space for a very long time. In the third case, we deployed the communication synthesis approach presented by us in previous work [7] in which we do not considered buffer space minimisation. The resulting total buffer space as a function of the number of tasks is shown in Figure 8(a) as the curves labelled with “greedy”, “tabu”, and “no buffer minimisation” respectively.

First, we observe that buffer space minimisation is worth pursuing, as it results in 22.3% reduction of buffer space on average when compared to the case when buffer space minimisation is neglected. Second, traffic shaping is an effective technique, reducing the buffer space demand with 14.2% on average relative to the approach that is based solely on communication mapping. Third, the greedy heuristic performs well as it obtains results on average of only 3.6% worse than the close-to-optimal tabu search. The running times of the tabu search based and the greedy heuristic, as measured on a 1533 MHz AMD Athlon processor, are shown in Figure 8(b). The greedy heuristic performs about two orders of magnitude faster (note the logarithmic scale of the *y* axis) than the tabu search based heuristic. Thus, we are able to synthesize the communication for applications of 79 tasks in $1'40''$, while the tabu search based heuristic requires around $1\text{h}30'$ for applications of 59 tasks.

We use 50 different 4×4 NoCs in order to assess the efficiency of our approach to solve the CSPBS problem. The total buffering capacities at switches range between $9,000$ and $30,000$ bits, uniformly distributed among the switches. We map 200 applications, one at a time, each consisting of 40 tasks, on each of the 50 NoCs, and we attempt to synthesize the communication of the application such that no buffer overflows or deadline violations occur. For each NoC, we count the applications for which we succeeded to find feasible solutions to the CSPBS problem. The percentage of the number of applications for which feasible communication synthesis solutions were found is plotted as a function of the total buffer capacity of the NoC in Figure 8(c). The proposed heuristic soundly outperforms the approach that neglects the buffering aspect as the percentage of found solutions is on average 53 points higher in the former case than in the latter. Also, the deployment of traffic shaping results in leveraging the percentage of found solutions to the CSPBS problem with 18.5% compared to the case when no traffic shaping is deployed. The results of the greedy heuristic come within 9% of the results obtained by tabu search, while the greedy heuristic runs on average 25 times faster.

Finally, we applied our approach to a multimedia application [5], namely an image encoder implementing the H263 algorithm. The application is composed of 24 tasks running on a platform consisting of 6 DSPs, 2 CPUs, 4 ASICs, and 2 memory cores (organised as a 4×4 NoC with two unused tiles). The communication mapping heuristic reduced the total buffer space with 12.6% relative to the approach that synthesized the communication without attempting to reduce the total buffer space demand. Traffic shaping allowed for a further reduction of 31.8%.

6 Conclusions

In this paper, we developed an approach to the worst case buffer need analysis of time constrained applications implemented on NoCs. Based on this analysis we solved two related problems: (1) the total buffer space need minimisation for application-specific NoCs and (2) communication synthesis with imposed buffer space constraints. For both cases we guarantee that imposed deadlines and message arrival probability thresholds are satisfied. We argued that traffic shaping is a powerful method for buffer space minimisation. We proposed two efficient greedy heuristics for the communication mapping and traffic shaping subproblems and we presented experimental results which demonstrate the efficiency of the approach.

References

[1] L. Benini and G. De Micheli. Networks on chips: a new SoC paradigm. *IEEE Computer*, 35(1):70–78, 2002.

[2] J. Dielissen, A. Radulescu, K. Goossens, and E. Rijpkema. Concepts and implementation of the Philips network-on-chip. In *IP-Based SoC Design*, 2003.

[3] F. Glover. Tabu search—Part I. *ORSA J. Comput.*, 1989.

[4] J. Hu and R. Marculescu. Applicationspecific buffer space allocation for networksonchip router design. In *Proc. of the ICCAD*, 2004.

[5] J. Hu and R. Marculescu. DyAD–Smart routing for Network-on-Chip. In *Proc. of DAC*, 2004.

[6] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Öberg, K. Tiensyrjä, and A. Hemani. A network on chip architecture and design methodology. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, April 2002.

[7] S. Manolache, P. Eles, and Z. Peng. Fault and energy-aware communication mapping with guaranteed latency for applications implemented on NoC. In *Proc. of DAC*, 2005.

[8] J. C. Palencia Gutierrez and M. Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proc. of RTSS*, 1998.

[9] K. Richter and R. Ernst. Event model interfaces for heterogeneous system analysis. In *Proc. of DATE*, 2002.

[10] T. Ye, L. Benini, and G. De Micheli. Analysis of power consumption on switch fabrics in network routers. In *Proc. of DAC*, 2002.