Online allocation for contention-free-routing NoCs

Radu Stefan
Eindhoven University of Technology
R.Stefan@tue.nl

Ashkan Beyranvand Nejad
Delft University of Technology
A.BeyranvandNejad@tudelft.nl

Kees Goossens
Eindhoven University of Technology
K.G.W.Goossens@tue.nl

ABSTRACT

Time-division-multiplexed networks based on the contention-free routing model represent an attractive high-performance and low-cost solution for on-chip communication thanks to their low buffer requirements at the router level. Traditionally, allocating the slots for each connection in the network TDM tables was performed at design time, thus requiring prior knowledge of the application communication demands and as a consequence making this approach unfeasible to certain classes of problems. In this paper we propose performing the slot allocation on demand, at run time. While this approach is not new, we improve upon the state-of-the-art in terms of speed by more than one order of magnitude, while at the same time requiring less memory space.

1. INTRODUCTION

Networks on Chip (NoC) [2] have been proposed as a scalable solution for on-chip communication, as the traditional bus-based interconnects began showing their limits. Among NoC implementations, the circuit-switched, time-division-multiplexed (TDM) NoCs based on the contention-free routing model represent an attractive choice as they use only minimal buffering at the router level which translates into reduced NoC cost [4]. Circuit-switching was also proven to be more energy-efficient than packet-switching [6, 1].

The contention-free routing model assumes that packets travel through the network without having to wait for each other at the intermediate nodes (routers). This is enforced by and using strict timing for the allowed packet insertion time and predefined routes, according to a global, collision-free schedule. Because the arbitration delays are eliminated and packets do not need to wait for each other inside the network, network traversal times are reduced and latencies are predictable.

The disadvantage of the approach is that the computation of the collision-free schedule is resource-intensive and as a result it is customarily performed at design time. This is possible when the communication behavior of the application is known beforehand but that limits the usability of these networks to certain classes of problems. In this study we propose a solution for computing the communication schedule at run time, according to application demands. Our solution allocates network connections one at a time using an minimal path exhaustive search algorithm that is able to deal simultaneously with allocation in the space and time domains.

Our algorithm is implemented in software, and we measure its performance running in FPGA, on an embedded processor. Compared to similar solutions proposed in the past it provides a significant advantage in terms of speed and memory requirements.

INA-OCMC '12, January 25, 2012, Paris, France

2. RELATED WORK

The closest proposal to ours is [8] which performs allocation in a TDM NoC based on the contention-free routing model. Our solution has several advantages compared to this study. Instead of performing explicit graph-splitting we store the graph with one node per router and we infer the split edges. We use a precomputed table of distances to know beforehand what the distance to destination is, therefore eliminating the need for iterative deepening used in the previous study. We eliminate routes that lead away from the destination at an early stage. We perform allocation of multiple slots simultaneously. Overall, these choices result in much lower allocation times.

In [10], the authors propose runtime mapping of applications on a multi-core design supported by the Æthereal NoC. The path finding algorithm employs as well a graph-splitting method. The algorithm running time is not presented.

A hardware accelerated NoCManager for performing path-finding and allocation is presented in [14]. The network architecture employed in that case is simpler, not making use of TDM. By comparison our solution is a purely software one, although we are currently looking into hardware accelerated computation.

In [13], the authors discuss routing in the context of run-time application mapping. Their experiments show that the failure in assigning tasks to specific locations in the system has high probability resulting from a failure in computing proper routing.

The assignment of virtual channels (or VCs) at run time in a NoC is discussed in [6]. Although the platform is different from ours, the approach is similar. A central authority assigns network resources (this time virtual channels instead of time-slots) to connections requiring service guarantees. The problem is solved using a simple path-finding algorithm as it does not have to deal with the more complex time-domain allocation we encounter in our target NoC implementation which is Æthereal [4].

Several studies propose hardware accelerated solutions for similar problems: wire routing [3], reachability and shortest path [9]. Hardware accelerated slot allocation is part of our current and future work.

The state-of-the-art in solving the same allocation problem at design time is found in [5]. The approach is based on

a branch-and-bound algorithm unapplicable here due to the high memory requirements. Another design-time allocation flow for a similar network model is presented in [7].

3. IMPLEMENTATION

In this section, we first introduce the data structure that is required to implement the TDM based routing technique. Secondly, we propose the run-time path finding algorithm and the methods for computation of the available bandwidth.

3.1 Data structures

A routing algorithm requires knowledge of the underlying network topology and available network resources. To allow an efficient implementation we opted for simple data structures with minimal memory footprint and we selected data types with the minimum bit-width that allows storing the necessary values. The given data structures are able to describe any network topology.

Both links and network nodes are identified by numeric IDs. Network nodes are sorted by type, first network interfaces (NIs), then routers. The algorithm does not make a distinction between NIs and routers except that the source and the destination are always NIs. Links are considered to be unidirectional (bidirectional links are stored as two separate unidirectional links). Links are sorted by their source node. One table (*dest* in Figure 1) stores the destination of each link. Another table (*start*) stores the first link in the table of links that belongs to each IP. The last entry in this table marks the end of the links table. This corresponds to the Compressed Row Format [11] for storing sparse matrices. The set of available TDM slots for each link is stored as bits packed in an integer. This allows manipulating sets of up to 32 slots at once.

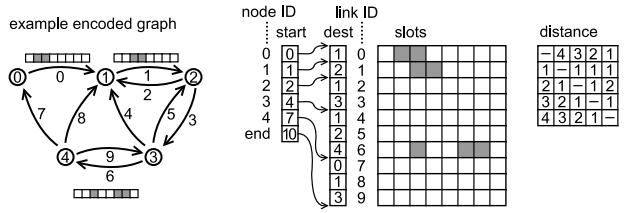

Figure 1: An example graph representation.

The same numeric link ID used as an index in the *dest* table is also used as index in the *slots* table (Figure 1). Depending on the usage scenario, we can have multiple slot tables, for example for offline (guaranteed) allocated slots and for best-effort paths.

3.2 The path finding algorithm

Our allocation algorithm consists of an exhaustive search of minimal length paths based on the backtracking method. The reader is referred to [12] for more details as the space here does not allow us an in-depth explanation.

A formal description of the algorithm is given in Algorithm 3.1. Instead of using recursion we simulate a stack using the arrays *solution[]*, *solLink[]* and *avSlots*. The *level* variable represents the top of the stack, and it also represents the number of segments in the currently explored path. We found this finite-state-machine-like implementation to be more efficient than an implementation using recursion.

firstLink[*node*] and *lastLink*[*node*] are used as more suggestive names for the *start*[] array. In fact *lastLink*[*node*] = *start*[*node* + 1] − 1. We use *crt* as an abbreviation in the variable names representing the current hop and *nxt* for the variables representing the next hop.

Algorithm 3.1: Non-recursive exhaustive pathfinding

Data: *source* and *destination* nodes
requiredBw the required Bandwidth
dist[] precomputed distance from all nodes to *dest*.
allowedDistance allowed path length, equal to precomputed minimum distance *source-destination*
Result: Path from *source* to *destination* which satisfies the bandwidth constraints
will be found stored in $solLink[1..level]$

```
1  level ← 1;
2  crtNode ← source;
3  crtSlots ← S;
4  crtLink ← start[crtNode];
5  while level > 0 do
7      if crtNode = destination then
8          found solution;
9          break;
10     end
11     nxtSlots ← shift(crtSlots) and not slots[crtLink];
12     crtDest ← dest[crtLink];
13     slotsOK ← bw(nxtSlots) ≥ requiredBw;
15     if dist[crtDest] ≤ allowedDistance − level ∧ slotsOK
       then
16         solution[level] ← crtNode;
17         solLink[level] ← crtLink;
18         avSlots[level] ← crtSlots;
19         level ← level + 1;
20         crtSlots ← nxtSlots;
21         crtNode ← crtDest;
22         crtLink ← firstLink[crtDest];
23         continue;
24     end
25     crtLink ← crtLink + 1;
27     if crtLink ≥ lastLink[crtNode] then
28         level ← level − 1;
29         crtNode ← solution[level];
30         crtLink ← solLink[level] + 1;
31         crtSlots ← avSlots[level];
32     end
33 end
```

The algorithm starts with an empty path at the source node. The algorithm tries to add links to the current path (the *if* statement in line 15) as long as the slots available on the path provide sufficient bandwidth (*slotsOK*) and the link brings us closer to the destination (which is ensured by the *dist*[*crtDest*] ≤ *allowedDistance* − *level* condition).

When all links departing from one node have been exhausted (the *if* statement in line 27) the algorithm falls back to the previous node by reading the value from the top of the stack. When the condition in line 7 is met the destination was reached and the solution (list of links forming the path) can be read from the *solLink*[] array.

3.3 Computation of the available bandwidth

The path finding algorithm is very efficient because it performs operations on an entire slot table at a time: a shift operation for advancing time as well as the bit-wise "and" and "not" operations. One task that is more difficult however is the computation of the bandwidth delivered by a particular set of slots. This computation potentially needs to take into account the overhead of network headers. In particular in the Æthereal implementation which we used in our study, a header is inserted at the beginning of each group of consecutive slots as well as every 3 slots afterwards in the larger groups (Figure 2).

In the following we will discuss alternatives for computing

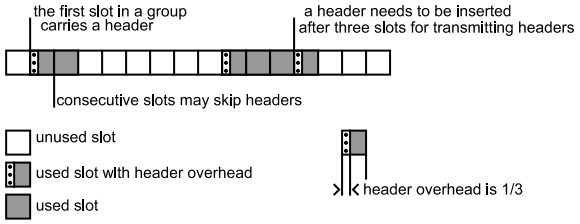

Figure 2: Header overhead in Æthereal.

the available bandwidth efficiently.

Exact bandwidth computation

The most straightforward method for determining the available bandwidth is to iterate over the entire slot table, checking which slots are available and which slots are not. In addition, we need to keep track of how large are the groups of available slots and how many headers need to be used. This approach is however very time consuming.

To avoid the computational overhead of the previous method, we can employ a look-up table storing the bandwidth provided by each combination of available and unavailable slots. The fact that the slot table is already stored as bits packed in an integer value means that this value can be stored directly as an index into the lookup table, providing the result in a single operation. The drawback of this method is that the size of the lookup table increases exponentially with the number of slots (it has memory complexity of $O(2^n)$). For a realistic size of the slot table of 16 slots, the cost of the table would be prohibitive.

Bandwidth approximation using lookup tables

A less memory intensive solution would be to split the slot table into groups of slots of reasonably small size and perform a lookup operation for each group. The difficulty is that in the networks that employ headers (e.g. Æthereal) we would also need to keep track of groups of consecutive slots that span multiple groups which would increase the computation time. Instead, we prefer a solution which produces an approximate result.

It is possible to assume conservatively that all the slots have headers, which results in underestimating the available bandwidth by 15.5% on average. A more accurate solution is to assume that the first slot in a group always has a header (Figure 3). This results an underestimate of 3.3% of the real value and never produces an overestimate which is important for the correctness of the solution. We will prefer thus this method as an alternative to exact computation.

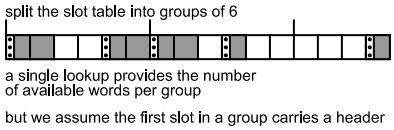

Figure 3: Approximate computation of the available words.

4. EXPERIMENTAL RESULTS

In this section we evaluate the performance and memory requirements of our online allocation algorithms. We measure the speed and memory requirements of the allocation algorithm proposed in Section 3, implemented in the C language and running on an embedded Microblaze processor in an FPGA prototype. The target NoC platfrom of the experiments is an Æthereal 4x4 mesh network on chip. We use both the exact and approximate bandwidth computation methods and we evaluate the decrease in the number of successful allocations resulting from the approximation.

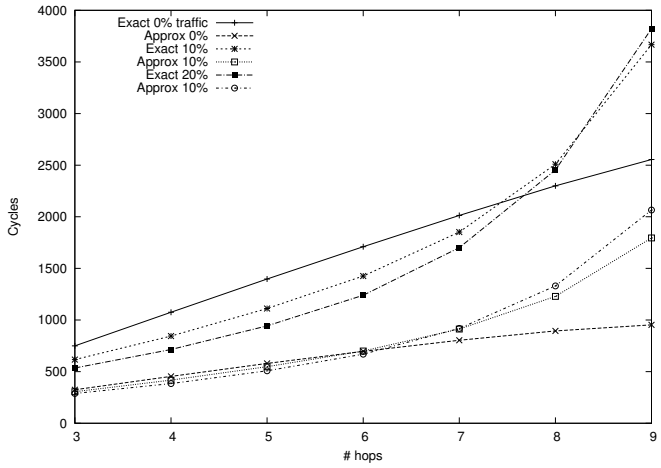

Figure 4: Allocation time vs. path length using 0%, 10% and 20% background traffic.

Allocation speed

We first generate an amount of random background traffic that uses some of the 16 TDM slots. We then measure the speed of path allocations between all pairs of 2 nodes and all requested bandwidths. Our performance evaluation metric is the number of cycles that the algorithm needs before it can either find a solution or determine that an allocation is not possible. We perform exhaustive search of minimal paths as described in Algorithm 3.1. It is possible to perform a search of longer paths by increasing the *allowedDistance* variable but that may lead to an unacceptable increase in the execution time. It would also be possible to bound the computation time by requiring the algorithm to give up after a certain number of attempted paths. This can be achieved by forcing an exit out of the loop 5-30 in the same algorithm.

The main factor affecting the duration of path computation is the distance between the source and destination. When the network load is zero or close to zero it is expected that the first path attempted produces a successful allocation. The algorithm running time will then increase linearly with the path length. This behavior is confirmed by the experimental results in Figure 4. Under higher network load, the execution time of the algorithm increases and the dependency on path length becomes exponential instead of linear.

Furthermore, the combination of background traffic load and requested bandwidth has an important effect on the execution time. When the requested bandwidth is very low, a path will be found early. If the requested bandwidth is much higher than the one that could be accommodated by the network, the algorithm will also determine quickly that no route is possible. Long running times are obtained when paths are neither too easy nor straight impossible to find.

Previous work [8] reports allocation times of 1000 cycles/hop/allocated slot in a 4x4 mesh. The speed of our solution is not directly dependent on the number of allocated slots as operations are performed on an entire table of slots at the same time, but there is an indirect dependence as the requested bandwidth affects the number of paths that will be examined. In our case, the highest runtime is 6394

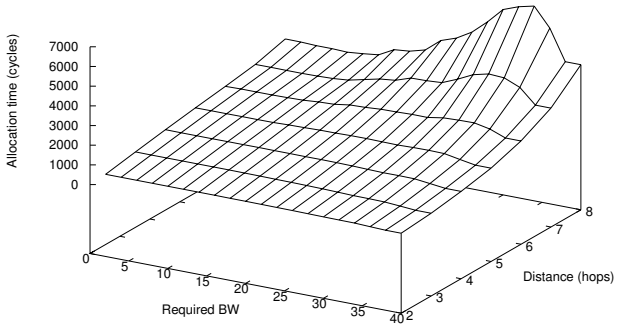

Figure 5: Allocation time vs. path length and requested bandwidth, Exact method, 10% background traffic.

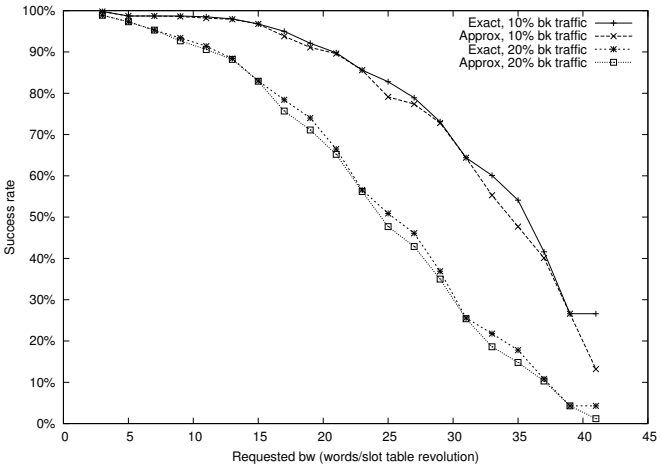

Figure 6: Success rate vs. requested bandwidth with 10% and 20% background traffic.

cycles/13 slots/path length 8 = 61.5 cycles/slot/hop, thus more than one order of magnitude lower.

Success Rate

The previous experiments indicate that using approximations in the bandwidth computation offers a significant speed advantage. On the other hand, as we mentioned in Section 3.3 the approximate method for bandwidth computation underestimates the available bandwidth on a given path by 3.3% on average. This means that some paths that would otherwise provide valid solutions are considered as having insufficient bandwidth. Figure 6 shows that the success rate of the approximate method as a function of requested bandwidth is decreased by 1.7% comparing with the rates of the exact method.

Memory requirements

The data structure presented in Section 3.1 allows us to provide a complete description of the network topology with memory complexity $O(n+m)$ where n is the number of nodes and m is the number of links. The array of distances has nevertheless memory complexity of $O(n * n)$. As only one line of this table is used during one path-search (the array of distances from each node to one destination), it could be possible to also compute these values before each allocation. In the case of a 4x4 mesh network the memory size used by the topology description is:

	80 *links* × 1 *byte/link*	(link destinations)
+	80 *links* × 2 *byte/link*	(slot tables)
+	32 *IPs* × 1 *byte/IP*	
+	13 *bytes (scalar data)*	
+	256 *bytes (distance table)*	
+	8 *stack entries* × (2+1+1) *bytes/stack entry*	
=	573 *bytes*	

This is also lower than the 1.5 kbytes reported in [8] for a slot table size of 16.

5. CONCLUSIONS

In this study we have proposed an efficient run-time path allocation algorithm for TDM based contention-free NoCs. The allocation algorithm has low computation overhead. We find our implementation to be more than one order of magnitude faster than the state of the art and to require less memory.

6. REFERENCES

[1] A. Banerjee *et al.* An energy and performance exploration of Network-on-Chip architectures. *Transactions on VLSI*, 17(3):319–329, 2009.

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

[3] A. DeHon *et al.* Hardware-assisted fast routing. In *FCCM*, 2002.

[4] K. Goossens *et al.* The Æthereal network on chip after ten years: Goals, evolution, lessons, and future. In *DAC*, June 2010.

[5] A. Hansson *et al.* A unified approach to mapping and routing on a network-on-chip for both best-effort and guaranteed service traffic. *VLSI Design*, 2007.

[6] N. Kavaldjiev *et al.* Providing QoS guarantees in a NoC by virtual channel reservation. *Reconfigurable Computing: Architectures and Applications*, 2006.

[7] Zhonghai Lu *et al.* Slot allocation using logical networks for TDM virtual-circuit configuration for network-on-chip. In *ICCAD*, 2007.

[8] T. Marescaux*et al.* Dynamic time-slot allocation for QoS enabled networks on chip. In *ESTIMedia*, 2005.

[9] O. Mencer *et al.* HAGAR: efficient multi-context graph processors. *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, 2002.

[10] O. Moreira *et al.* Online resource management in a multiprocessor with a network-on-chip. In *SAC*, Seoul, Korea, 2007.

[11] Sergio Pissanetsky. *Sparse Matrix Technology.* Academic Press, London, 1984.

[12] R. Stefan *et al.* A tdm slot allocation flow based on multipath routing in nocs. *Microprocessors and Microsystems*, 2010.

[13] T.D. ter Braak *et al.* Run-time spatial resource management for real-time applications on heterogeneous MPSoCs. In *DATE*, 2010.

[14] M. Winter *et al.* A Network-on-Chip channel allocator for Run-Time task scheduling in Multi-Processor System-on-Chips. In *DSD*, 2008.