Trade-Offs in the Configuration of a Network on Chip for Multiple Use-Cases

Andreas Hansson[1] and Kees Goossens[2,3]
[1]Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands
[2]Computer Engineering, Delft University of Technology, Delft, The Netherlands
[3]Research, NXP Semiconductors, Eindhoven, The Netherlands
m.a.hansson@tue.nl, kees.goossens@nxp.com

***Abstract*— Systems on chip (SoC) are becoming increasingly complex, with a large number of applications integrated on the same chip. Such a system often supports a large number of use-cases and is dynamically reconfigured when platform conditions or user requirements change.**

Networks on Chip (NoC) offer the designer unsurpassed run-time flexibility. This flexibility stems from the programmability of the individual routers and network interfaces. When a change in use-case occurs, the application task graph and the network connections change. To mitigate the complexity in programming the many registers controlling the NoC, an abstraction in the form of a configuration library is needed. In addition, such a library must leave the modified system in a consistent state, from which normal operation can continue.

In this paper we present the facilities for controlling change in a reconfigurable NoC. We show the architectural additions and the many trade-offs in the design of a run-time library for NoC reconfiguration. We qualitatively and quantitatively evaluate the performance, memory requirements, predictability and reusability of the different implementations.

I. Introduction

Systems on Chip (SoC) grow in complexity with an increasing number of processors, memories and accelerators integrated on a single chip. These heterogeneous high-complexity chips are programmable and integrate a rich set of applications [1]–[4], e.g. PDA phones with mp3 players, cameras, radios and gaming. The application dynamism is increasing, both between applications, through run-time addition or removal [4], and within applications, for example the variation on bandwidth and computation demand in MPEG-4 [2].

Networks on Chip (NoC) have emerged as the design paradigm for scalable on-chip communication architectures, providing better structure and modularity while allowing good wire utilisation through sharing [5]–[8]. The programmability of the NoC enhances flexibility and reuse in both the long term, over platform generation, and short term, when switching between use-cases on a milli-second granularity [1], [4], [7]. Internally, programming the NoC involves the individual routers and network interfaces (NI). To mitigate the complexity of reconfiguration, the abstraction level must be raised and provide an interface between NoC implementations and applications [2], [9].

The system typically moves from one configuration to the next as a result of user interaction [4], change in environment, e.g. signal reception, or resource availability, such as the battery level. A switch in use-cases upon such an event causes changes in the application task graph and subsequently the network connections [2], [10]. NoC reconfiguration requires: 1) the means for performing changes, i.e. a programmable NoC architecture [11], [12], 2) the means to specify and compute configurations, either at run time [1], [2] or at design time [10], and 3) the facilities for controlling change, that is, orchestration of the actual reconfiguration.

When performing the latter task, it is crucial that the changes are applied in such a way as to leave the modified SoC in a *consistent state*, that is, a state from which the system can continue processing normally rather than progressing towards an error state [13]. The Intellectual Property (IP) modules in the SoC move from one consistent state to the next by issuing and serving *transactions*. These transactions modify state and while in progress, have transient state distributed in the system, including also the NoC. Simply updating the NI registers could cause out-of-order delivery or even no delivery, with an erroneous behaviour, e.g. deadlock, as the outcome. Should a transaction be corrupted or not even allowed to finish due to inconsistent NoC reconfiguration, it is unlikely that the application will recover. Consider for example a situation where the response to a read transaction never arrives, causing the IP module that initiated the transaction to stall indefinitely. Similarly, a write transaction that is only partially delivered might cause the target IP to stall, waiting for the outstanding data elements. These events are catastrophic to the SoC and must be avoided or recovered from.

In this paper we address the actual *configuration process*. We introduce the *Æthereal Run-Time (ART)* library for NoC reconfiguration with a basic set of functions, through which we can connect the IP ports in a graph structure and dynamically add, remove or modify the interconnections, thus changing the task-graph topology in a consistent way. We show: 1) the hardware support required by the library to ascertain transaction consistency, and 2) the trade-offs (performance, memory requirements, predictability, flexibility, reusability) in implementing the library functions.

The remainder of this paper is organised as follows. Section II introduces related work and is followed by a problem description and delimitation in Section III. Then, Section IV discusses the architectural support and the different operations that are used to reconfigure the NoC. The details of the

ART library implementation are given in Section V before we present the results in Section VI. Finally, we conclude in Section VII.

II. Related work

A comprehensive model of dynamic change management is given in [13]. The work discusses the requirements of the configuration management and the implications of evolutionary change of software components. A more practical approach to dynamic application reconfiguration in multi-processor SoCs is presented in [4], [14]. Reconfiguration of the interconnect is, however, not addressed.

A methodology for deriving procedures for deadlock-free reconfiguration between routing functions is introduced in [15]. The work shows how *reconfiguration-induced deadlock*, caused by additional dependencies in the transition between the old and new routing function, can be avoided. In this work we assume a fixed network topology and employ turn-prohibited routing [16] with a fixed set of prohibited turns. Hence, reconfiguration-induced deadlock cannot occur.

Much work is focused on complete NoC design and compilation flows [9], [17], [18]. While the works suggest automated generation of configuration code [17] and standardised NoC application programming interfaces [9], no details are given as how to facilitate the dynamic changes.

NI architectures are presented in [3], [7], [11], [19], [20]. The interfaces in [11], [19], [20] are OCP compliant, whereas [7] presents an NI with support for DTL and AXI. A specialised NI, tightly coupled to an accelerator, is implemented and evaluated in [3]. All these works focus on *providing programmability* and only briefly discuss how to employ it in practice.

Methodologies for derivation of network configurations are presented in [1], [10], [21]–[24]. Design-time algorithms limited to a single use-case [22]–[24] are plentiful. In [10], multiple use-cases are supported by computing a number of hierarchical configurations at design-time. The works in [1], [21] add even more flexibility by deciding the resource binding at run-time and allowing task migration. While showing how to *determine NoC configurations*, none of the works detail how to deploy it.

Concluding, existing work addresses the problem of providing hardware run-time programmability and deriving a configuration to be programmed. This paper investigates the actual detailed configuration process and presents the necessary hardware additions and the trade-offs in designing a library for consistent NoC reconfiguration.

III. Problem description

NoCs comprise two components: routers (R) and network interfaces (NI). The routers can be randomly connected amongst themselves and to the NIs (i.e., there are no topology constraints). The routers transport packets from one NI to another.

The NIs enables end-to-end services [7] to the IP modules and are key in decoupling computation from communication [11], [25]. The NI is responsible for (de-)packetisation, for implementing the *connections* and services, and allows the designer to simplify communication issues to local point-to-point transactions at IP module boundaries, using protocols natural to the IP (e.g., AXI or OCP) [25].

The IP modules, or rather their ports, acts as either masters or slaves [7]. Masters initiate transactions by issuing requests. One or more slaves receive and execute each transaction. Optionally, a transaction also includes a response, returning data or an acknowledgement from the slave to the master.

The term *connection* is used throughout this paper to denote a bidirectional peer-to-peer inter-connection between a master and a slave IP. As shown in Figure 1, a connection comprises a *request channel*, from master to slave, and a *response channel* in the reverse direction. Every connection is bound to four unique queues: one in the sending NI and one in the receiving NI, for both the request and the response channel. The Quality of Service, best-effort (BE) or guaranteed service (GS), is determined on a per-connection basic. Guarantees are provided by time-division multiplexed (TDM) virtual circuits [7].

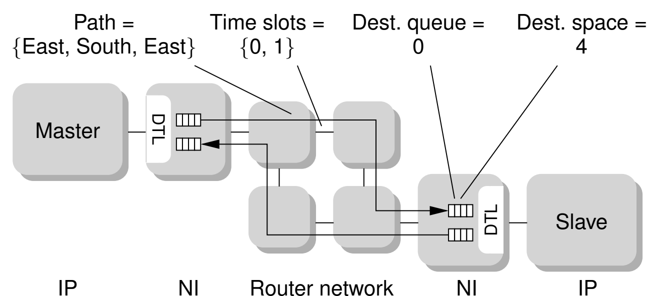


Fig. 1. A connection with request and response channel on a 2×2 mesh.

To set up a channel, only the source NI requires programming [7], [11]. Hence, the target NI is always ready to receive data and does not require any programming to do so.

As indicated in Figure 1, the information required is:

- a *path* through the router network,
- a *queue* identifier, selecting a queue in the destination NI,
- the *end-to-end flow control credits*, reflecting the size of the destination queue, and, in the case of a GS connection,
- a set of *TDM time-slots* determining when the channel may use the network links.

The NI also offers the possibility to set a lower limit for when data and credits may be sent. These threshold values are initialised to zero and therefore require no modification if the functionality is not used.

We assume that the configuration is already computed, either off-line [10] or on-line [1], [21]. Similar to [1], [4], [7], [21], the configuration is done by a general-purpose CPU (in our case an ARM) and is initiated by events in the system, caused by e.g. a user command to start or stop an application, an embedded resource manager [26] or mode switches in the incoming streams [4].

When modifying or closing channels, both the NoC and the IPs must be left in a consistent state, that is, a state from which the system can continue processing normally rather than

progressing towards an error state [13]. Simply updating the NI registers could cause out-of-order delivery or even no delivery, with an erroneous behaviour, e.g. deadlock, as the outcome.

Reconfiguration of the tasks running on IP modules [4], [14], [27] is an important task that lies outside the scope of this paper. However, the techniques shown here are also applicable at the higher IP level.

While parts of the presented algorithms are specific to Æthereal, the concepts described apply to NoCs in general and we give suggestions to alternative implementations throughout the text.

IV. Network configuration

The configuration of the network is hidden from the application programmer by an application-level configuration API, such as C-HEAP [27], that switches between use-cases and leaves the actual NoC reconfiguration to the ART library software. This approach hides the underlying implementation and eases integration as well as modifications [9]. Thereby, it is the responsibility of the configuration management system, not the user, to determine the specific ordering of actual change operations applied [13].

A. Configuration registers

The network connections are configured at run time via a memory-mapped configuration port on the NI, denoted *Config* in Figure 2. The configuration port offers access to the NI registers through normal read and write transactions [12].

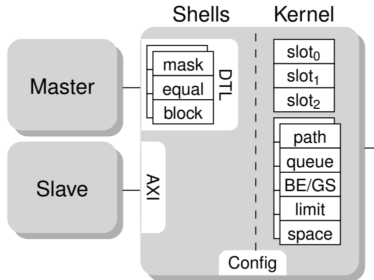


Fig. 2. Network interface registers.

As seen in Figure 2, the registers are divided into three different blocks with three different address ranges. First, the *path*, *queue*, *BE/GS*, *limit* and *space* registers, with one instance per outgoing channel. This block controls, in order, the path through the router network (source routing), the queue to which data is delivered in the destination NI, the assigned service level (BE/GS), lower limits for when data and credits may be sent, and the flow-control space counter that is used to track the occupancy of the destination queue. Second, the slot table, in this example with only three slots. These registers determine when the GS channels are scheduled and may inject flits into the router network. Third, the registers in the protocol shells [7]. These registers are optional and depend on the functionality of the specific shell instantiations. The DTL shell connected to the master in Figure 2 has *mask*, *equal* and *block* registers. The first two are used to implement narrowcast connections [7], whereas the latter is used when inactivating and closing connections, further discussed in Section IV-E.

B. Configuration infrastructure

We assume that there exists one or more *configuration master* modules that execute the configuration operations. Such a module, typically a general-purpose CPU, only has one data bus and needs an infrastructure to facilitate access to the NIs in the platform.

By connecting the configuration port back to a (DTL) initiator port on the NI, as done on NI_m and NI_s in Figure 3, the configuration data is carried by the NoC itself [7] instead of an additional control network, as advocated in [2]. By reusing the existing infrastructure, low-bandwidth control traffic is allocated resources just like any user-generated traffic. In Section IV-F we show that it is still possible to separate user traffic from control traffic by allocating TDM slots also for the latter.

To reach NIs other than NI_c from the configuration master we set up *configuration connections* over which the configuration data is read and written. As illustrated in Figure 3(a), the first step is to open a *configuration request channel* from NI_c to NI_s. The opening of this channel only involves manipulation of the R_c registers in NI_c, as indicated by the arrow marked 1. The configuration request channel makes the R_s register in NI_s accessible via the *DTL* port on NI_c. To enable communication of responses back to NI_c, we also establish a *configuration response channel*, from NI_s back to NI_c. This is done by writing to the R_s registers of NI_s, indicated by the arrow marked 2 in Figure 3(a). The response channel is left unused but in place when moving on to the initialisation of NI_m (arrows 3 and 4).

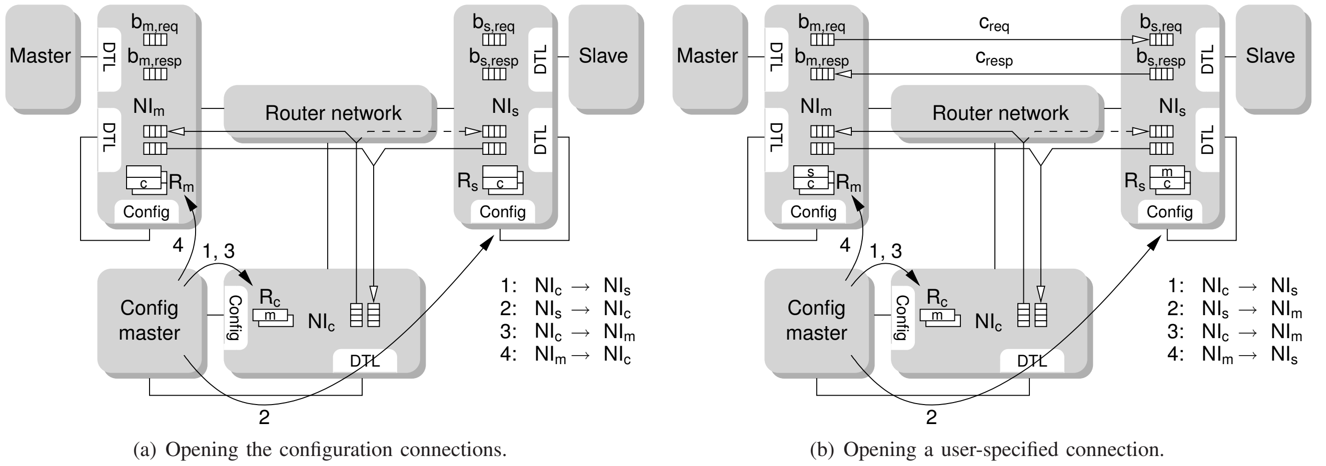
The configuration connections, allocated resources just like any other connection [10], are all set up in the configuration initialisation phase, further covered in Section V. Hence, the configuration response channels are set up *once* and are already in place when the first user-specified connection is opened. The configuration request channel, however, is inactivated and re-opened for every NI_t.

There are two opportunities for adding parallelism in the configuration infrastructure. First, having more than one configuration master, e.g. hierarchically distribute the control to different subsystems. Second, by using more ports on NI_c for accessing remote NIs and thereby enabling one master to configure multiple NIs in parallel. In our experiments we use a single configuration master, and only one port for remote configuration.

C. Configuration operations

Configuration of the network relies on three top-level operations, closely related to the *channel_create*, *channel_reconfigure* and *channel_destroy* primitives of C-HEAP [27].

- Open a new connection.
- Modify an existing connection.
- Close an existing connection.



(a) Opening the configuration connections.

(b) Opening a user-specified connection.

Fig. 3. Using the NoC itself as configuration infrastructure.

1) Opening a connection: Algorithm IV.1 describes how a channel is opened on a target NI, hereafter denoted NI_t.

Algorithm IV.1 Open channel emanating from NI_t

1) Open a configuration request channel to NI_t.
   a) Set the path, the destination queue in NI_t (the queue of the looped-back initiator port), the initial space, and, if using GS for configuration, also the time slots by writing to NI_c via the *Config* port.
   b) Enable the queue in NI_c for scheduling by the NI kernel scheduler.
2) Open the channel emanating from NI_t.
   a) Set the path, the destination queue, the initial space, the thresholds, and time-slots in NI_t by writing to the *DTL* port on NI_c.
   b) Enable the queue in NI_t for scheduling.
3) Close the configuration request channel to NI_t.
   a) Await the completion of the writes to NI_t by reading the *Config* port of NI_c until no data is left in NI_c and all credits have returned from NI_t (i.e. the transactions are consumed and executed by NI_t).

In Step 2, with the configuration channel in place, the registers of NI_t are written. Thereafter, in Step 3a, the configuration master waits until all transactions are delivered to the *Config* port on NI_t. As seen in Step 3a this is done by busy-waiting on the local status registers in NI_c until all data has been sent and all credits have returned. In a NoC that does not employ end-to-end flow control this can be implemented with acknowledged writes or by reading back the value of the last register written.

Using the primitive for channel opening, Algorithm IV.2 lists the steps necessary to open a connection between a master and a slave port. Recall that the configuration response channels are already in place, as shown in Figure 3(b). The three writes required to set the registers in a target NI are posted and delivered in order as they use the same configuration channel. When switching between different target NIs, the configuration request channel must be closed. Therefore, Step 3 waits until NI_s has consumed and hence executed the transactions (Step 3a in Algorithm IV.1).

Algorithm IV.2 Open connection between NI_m and NI_s

1) Open a configuration request channel from NI_c to NI_s.
2) Open the response channel from NI_s to NI_m.
3) Close the configuration request channel from NI_c to NI_s.
4) Open a configuration request channel from NI_c to NI_m.
5) Open the request channel from NI_m to NI_s.
6) Close the configuration request channel from NI_c to NI_m.

As we will see in Section VI, the time required to open a connection varies, although the operation only involves writes to the NI registers.

D. Modify a connection

When the source and destination remain the same (NI as well as queue), but the properties of the connection change, then we say the connection is modified. This is for example done to adapt to quality changes in a scalable algorithms [26], redistribute resources (links and time slots) between the connections, or to move connections from one path to another so that routers can be powered down [9]. For the individual channel, the modification can reflect a change in:

- the data or credit threshold,
- the allocated bandwidth,
- the guaranteed latency, or
- the path through the network.

The first three modifications require only minimal updates. Time slots can be added and removed and threshold values changed even while the channel is being used. Modifying the path, however, is more complicated as we must preserve (loss-less) in-order delivery. A shorter path means flits can arrive earlier than those already sent on the longer, or more congested, path and thereby invalidate the in-order delivery.

For a GS channel, the transmission delay is known and directly proportional to the path length, as contention is avoided by means of *pipelined virtual circuits* [7]. Hence, if the new path is longer, then the update can be done without disabling and later re-enabling the queue. Should the path be shorter, then it is possible to only disable scheduling for δ flit cycles where δ is the reduction in path length. Note, however, that this additionally requires the configuration to be done over a GS connection as the inter-arrival time of the disable and re-enable operations at NI_t must not be shorter than δ flit cycles. If configuration is done with BE connections this is not possible to guarantee.

Modifying the path of a BE channel, with a non-deterministic delay, is comparable to adaptive routing, where a flit $i+1$ may overtake flit i, due to a shorter or less congested route [28]. In Algorithm IV.3, we show the general (and safe) procedure for modification of any (BE or GS) channel.

Algorithm IV.3 Modify channel emanating from NI_t

1) Open a configuration request channel from NI_c to NI_t.
2) Reconfigure NI_t.
   a) Disable scheduling of the queue.
   b) Await the return of all outstanding credits.
   c) Update the path and slots.
   d) Enable scheduling of the queue.
3) Close the configuration request channel from NI_c to NI_t.

The channel modification can safely be done as soon as the router network has delivered all the flits to the destination NI. Consider for example modification of the request channel, c_{req} in Figure 3(b). When the router network is empty there is no longer any risk for out-of-order delivery. However, NI_m has no notion of what data is received, only what data is consumed, and this in the form of end-to-end flow control. Hence, to ascertain that no flits are in flight, Step 2b not only waits until the router network has delivered all the flits to the destination NI, but also until the data is consumed and the credits returned. This corresponds to emptying both the router network and $b_{s,req}$ in Figure 3(b).

As a consequence of the above, channel modification requires the destination of the channel, exemplified by the slave in Figure 3(b), to sink all outstanding transactions. This might require progress on the reverse channel and due to this, care must be taken when ordering the configuration operations.

E. Close a connection

The crux in closing a connection is to ensure that the master, slave and the network are left in a consistent state after the change, with no transient state distributed anywhere along the request-response path. Transactions, e.g. a DTL or AXI read, are initiated by the master and accepted by NI_m. NI_m subsequently engages in a number of transactions with NI_s through the exchange of flits and credits. Then, NI_s acts as a master to the slave and delivers the request and accepts the response. Thereafter, the response goes through the network and is presented to the master IP module.

The key observation is that transactions take place both on the IP (reads/writes) and on the network level (flits/credits). When *quiescence* is reached on both levels, a change can take place with a consistent state as the outcome. Using the definition of [13], a module is *quiescent* if: 1) it will not initiate new transactions, 2) it is not currently engaged in a transaction that it initiated, 3) it is not currently engaged in servicing a transaction, and 4) no transactions have been or will be initiated which require service from this module.

1) Requirements: To achieve quiescence, a number of actions must be taken. Starting at the master in Figure 3(b), it is necessary to force the module into a state where it is not initiating any new transactions, but still continues to accept and service transactions that are outstanding. In terms of requests and responses, this translates to:

1) No new requests must enter the request buffer $b_{m,req}$, but an ongoing request (e.g. a write burst) must be allowed to finish.
2) Requests already accepted into $b_{m,req}$ must be delivered, such that $b_{m,req}$, c_{req} and $b_{s,req}$ are empty.
3) All initiated transactions that require a response must be allowed to finish. Thereby, also $b_{s,resp}$, c_{resp} and $b_{m,resp}$ are empty.

Step 1 requires knowledge of how command/address/data is accepted from the master and what timing relations are allowed between these groups of signals. Typically, a valid-accept handshake first takes place on the command and address signals with the master offering a request to the NI by driving the *valid* signal high. The NI in turn indicates that it has accepted the request by driving the *accept* signal high. Thereafter, the two parties engage in a similar handshake on the potential data words involved in the transaction.

Step 2 arises due to the fact that the NoC acts as a slave, thus giving the master the impression that a transaction involving only a request is finished already when delivered to the NI shell (if posted).

Step 3 requires knowledge of the various types of transactions allowed by the protocol to determine whether a request also gives rise to a response. This functionality is implemented in protocol shells, separating protocol specific functionality from the NI kernel.

2) Hardware implementation: The master protocol shell requires one input signal to initiate the close operation, *passivate*, and one output signal to assert that a quiescent state is reached, *is_quiescent*. These two signals are read and written from the NI kernel through the *block* control register, shown in Figure 2.

When a request channel is passivated, any ongoing request is allowed to finish, whereafter the *accept* signal to the master is gated and kept low, thus preventing it from initiating any new transaction[1]. Before the *is_quiescent* is asserted, all ongoing transactions must also finish. We implement this check with a counter inside the shell. The value is incremented for every accepted request that gives rise to a response, such as a read,

[1]This infrastructure can also be used for debugging purposes [29].

and decremented for every response delivered back to the master. A signal *activate* is used to bring the shell back to an active state where new transactions can be initiated.

3) Algorithm: With the above functionality implemented in the protocol shells, a connection is closed according to Algorithm IV.4. Steps 7, 8 and 12 are carried out by repeatedly polling a remote register. These operations can be transformed to local busy-waits by using the concept introduced in [30].

Algorithm IV.4 Close connection between NI_m and NI_s

1) Open a configuration request channel to NI_s.
2) Set the thresholds for both data and credits to zero.
3) Close the configuration request channel to NI_s.
4) Open a configuration request channel to NI_m.
5) Set the thresholds for both data and credits to zero.
6) Passivate the connection by writing to the *block* registers in the shell.
7) Await the quiescent state from the shell.
8) Await the outgoing request queue to be emptied and the return of all outstanding credits (plus the sending of any potential credits back to NI_s).
9) Clear the slot table reservation.
10) Close the configuration request channel to NI_m.
11) Open a configuration request channel to NI_s.
12) Await all outstanding credits.
13) Clear the slot table reservation.
14) Close the configuration request channel to NI_s.

Algorithm IV.4 assumes *independent transactions*, where completion of a transaction does not depend on any other transactions with other IPs [13]. If such dependencies do exist, they must be addressed by the user of the ART library, as the change then is on a granularity larger than a single connection. The library provides functions for manipulation of individual channels or even individual registers to facilitate such use.

Note that in NoCs that do not employ end-to-end flow control, quiescence in the router network has to be implemented by e.g. inserting a special tagged message as an end-of-stream marker [21].

F. Guaranteed-service configuration

When NoC configuration is done via BE channels, no timing guarantees can be given on the operations performed. The benefit is that no resource reservation has to be made for the configuration connections, that are both seldom used, and have very low bandwidth requirements. The drawback is the uncertainty in when the configuration operations are executed, and potentially long delays in case of congestion.

Although resource reservations for configuration connections might initially seem wasteful, it is, however, possible to reuse the same resources (TDM slots) for all the configuration connections, wherever two or more configuration channels share a common link. This is due to the fact that we use the configuration request and response channels in a mutually exclusive manner, where one is always passivated before another one is used. Hence, it suffices to reserve a single time slot per link for all the configuration connections. In a network with n TDM slots this bounds the reserved resources to $\frac{1}{n}$ of the total ingress/egress bandwidth. The actual use of these time slots is multiplexed by the configuration master at run time.

V. LIBRARY IMPLEMENTATION

The API shown in Listing V.1 constitutes the foundation of the ART library for manipulation of the NoC configuration. The *art_config_init* is performed as a part of the bootstrap procedure, before any calls to the other functions. It instantiates the configuration response channels, from every NI back to NI_c.

Listing V.1 Top-level configuration operations.

```
/* Initialise the config response channels. */
void art_config_init();

/* Open, modify and close a connection. */
void art_open_conn(const conn_t* const conn);

void art_modify_conn(const conn_t* const old_conn,
                     const conn_t* const new_conn);

void art_close_conn(const conn_t* const conn);
```

The last three operations all work on the granularity of a connection, represented by a data structure carrying all the information required to configure the NIs (and optionally also the routers). The C implementation of this structure is shown in Listing V.2.

Listing V.2 Connection structure.

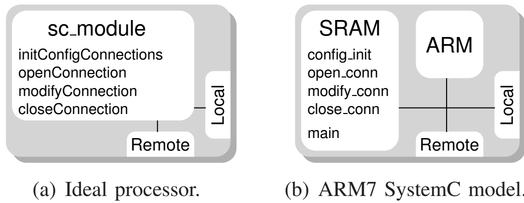
```
typedef struct {
  /* Location of master and slave */
  ni_t master_ni;
  unsigned char master_queue_id;
  ni_t slave_ni;
  unsigned char slave_queue_id;

  /* Quality-of-Service, GS/BE */
  service_t qos;

  /* Settings for the master NI */
  path_t master_slave_path;
  unsigned char master_space;
  unsigned char master_data_limit;
  unsigned char master_credit_limit;
  slot_t master_slot_list;
  unsigned char master_channel_id;
  unsigned int narrowcast_equal;
  unsigned int narrowcast_mask;

  /* Settings for the slave NI */
  path_t slave_master_path;
  unsigned char slave_space;
  unsigned char slave_data_limit;
  unsigned char slave_credit_limit;
  slot_t slave_slot_list;
} conn_t;
```

The top-level operations in turn call a number of internal functions for encoding of the struct fields into 32-bit words,



(a) Ideal processor. (b) ARM7 SystemC model.

Fig. 4. Configuration master instantiations.

e.g. *art_encode_path*, opening and closing of configuration connections, *art_open_config_conn* and *art_close_config_conn*, or reading/writing specific registers, such as *art_set_slots* and *art_get_transaction*. The final outcome is calls to DTL read and write operations, which is the only processor-specific part of the library. This allows for easy porting to other platforms [31].

The library functions can also be used together with a power management library as it implements the functionality to close and modify connections. Together with hardware support, e.g. clamping of the network link signals, this affords safe power down of parts of the NoC. Note though that modifications of the network topology may necessitate modifications of the BE routing function to preserve full connectivity. In such a situation it is imperative to address also reconfiguration-induced deadlock [15].

VI. RESULTS

In our experiments, we use two different configuration-master implementations. First, a SystemC module where computation is done without progressing the simulation time, i.e. infinitely fast. Only the read and write operations contribute to the configuration time. This option, depicted in Figure 4(a) is referred to as the *ideal processor*.

Second, the implementation shown in Figure 4(b), comprising a SWARM SystemC model of an ARM7 [32], clocked at 100 MHz. The core has a von Neumann architecture, and a bus within the tile determines whether the read and write transactions go to the local memory, through the *Local* port to the corresponding *Config* port of NI_c, or via the *Remote* port to a remote NI.

To assess the impact of the processor execution time, we compile two different binaries for the ARM: one with the connection structures and the ART library, as described in Section V, and one where the configurations are pre-compiled into plain read and write calls, thus removing all computation and minimising the amount of function calls. Both binaries are compiled using *arm-elf-gcc* version 3.4.3 with the compiler options *-mcpu=arm7 -Os*.

Unless indicated otherwise, configuration is done using BE connections. The configuration-related traffic thus contends with connections that are already open.

The first measure we study is the time required to initialise the configuration connections and how this grows with the number of NIs. Figure 5(a) shows the effect of increasing the mesh size from 1×4 to 5×4, constantly having two NIs per router. It is clear that the NoC is not the limiting factor, as the ARM is consistently more than 8 times slower than the ideal processor for the read/write implementation, and 70 times slower using the library. All three implementations show a linear growth, as expected with a constant work per NI. No user-specified traffic is yet occupying the NoC.

Figure 5(b) shows the effect on the cumulative setup time when varying the number of connections on a fixed 4×4 mesh with two NIs per router. The setup time is measured from the point when initialisation of all the configuration response channels is completed. The impact of the latter is shown in Figure 5(a) (the scenario with 32 NIs). The user-specified connections have a total bandwidth of 6.2 Gbyte/s, uniformly spread across the NoC. With an average burst size of 8 words and a flit size of 3 words this roughly amounts to 20% of the total ingress/egress bandwidth. All connections belong to the GS class of traffic and are thus subjected to non-work-conserving arbitration.

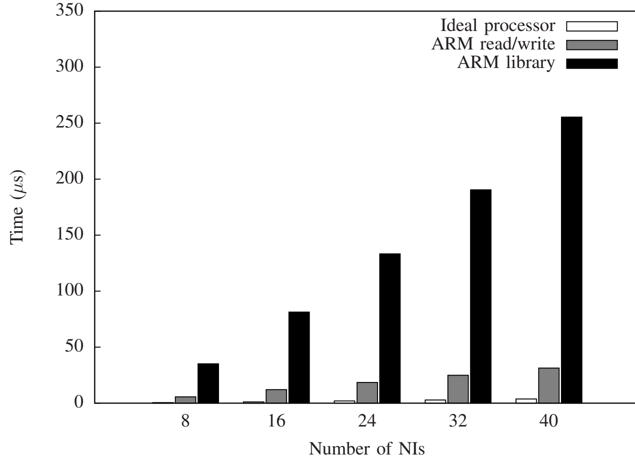
Also in this experiment the ARM is roughly 10 times slower than the ideal processor when using read/write calls, and 60 times slower with the library implementation. The time required grows linearly with the number of connections, only showing some minor fluctuations. The worst case setup times for a single connection are 0.46, 3.04 and 16.72 μs for the three implementations. Again, we conclude that the computation time is far greater than the time spent on communication and we can see that the contention from already opened connections is hardly noticeable.

We also evaluate the possibility to exploit locality when multiple channels share the same source NI and there are no ordering constraints between the channels, e.g. when opening connections. In contrast to Algorithm IV.2, we iterate over the NIs rather than over the connections when opening a set of channels. Then, for every NI, the configuration request channel only has to be opened and closed once. Applying this strategy, the cumulative time still grows linearly with the number of connections. However, for the ideal processor, the time required to open all connections is consistently less than half or what is achieved in Figure 5(b). Similarly for the ARM implementations, the total setup time is roughly 40% less using this technique.

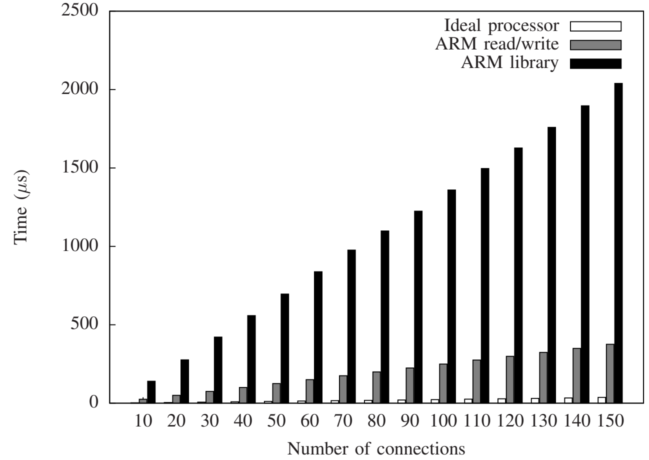
Next, we assess the memory requirements of the ARM binary. To reduce the memory footprint, we also specify the bit widths of the different fields in the connection structure. Note though, that bit members generally worsen the execution time as many compilers generate inefficient code for reading and writing them.

The binary size for a varying number of NIs is shown in Figure 6(a). The various NoC instantiations correspond to the same mesh networks as in Figure 5(a), here together with a 40 connection use-case. Expanding the library functions to read and write calls roughly doubles the size. This is to be compared with the ten-fold speedup observed in Figure 5(a).

Figure 6(b) shows the corresponding scaling with the number of connections. Here, the difference between the library and the read/write implementation becomes more obvious as the number of connections grows. As an example, the expansion to read/write calls increases the size with 170%

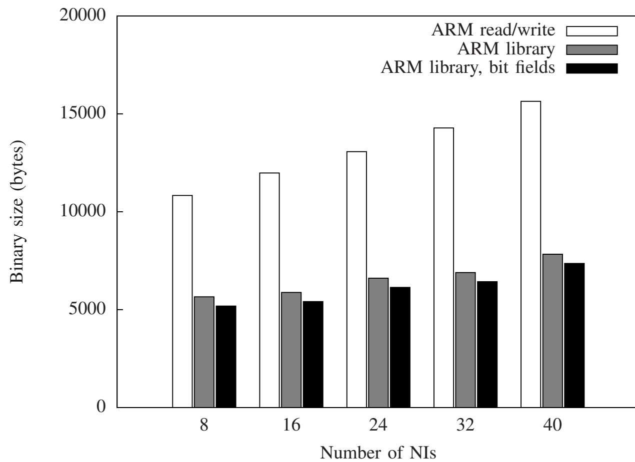


(a) Initialisation time for different number of NIs.

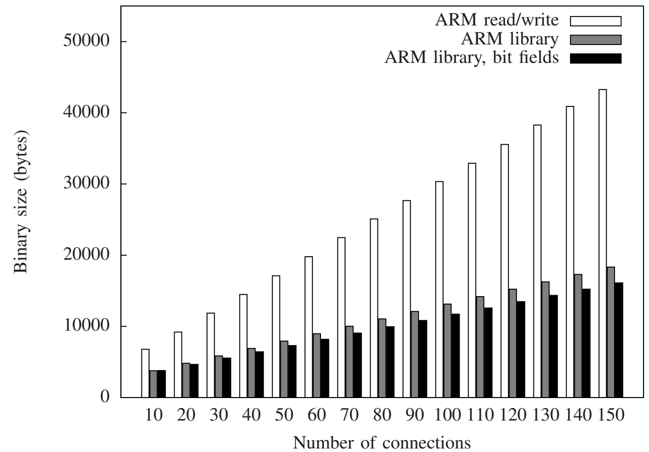


(b) Setup time for different number of connections.

Fig. 5. Execution time.



(a) Binary size for different number of NIs.



(b) Binary size for different number of connections.

Fig. 6. Binary size.

for the 150 connection case.

We conclude that the effect of bit members is only a few percent reduction of the binary size. Moreover, as expected, we observe an execution time that is slightly worsened, although the measured increase is a mere 1.2%. Taking both these facts into account it is hardly justifiable to employ bit members unless memory footprint is absolutely critical.

Table I summarises the analytical approximations of the graphs in Figure 5 and Figure 6. Note that the table does not show the constant term.

It is clear from our experiments that the quantitative differences between the ARM library and read/write implementation are both in performance, where the library is roughly a factor six slower, and size with only half the binary size. Qualitatively, the library offers a flexibility and reusability that is impossible to achieve without it. Taking these facts into account, we propose to pre-compute the read/write calls for any use-case that is fixed and do not require any run-time flexibility. The configuration channels themselves constitute a good example of such a use-case. Here, we can enjoy a six-fold speed up without sacrificing any flexibility. Similarly, for applications that frequently change between a set of pre-defined operation points, e.g. due to QoS levels, it is possible to pre-compute the instruction sequences and thereby speedup the transitions.

TABLE I

ANALYTICAL APPROXIMATIONS OF THE LINEAR TERM IN FIGURE 5 AND 6

	Ideal processor	ARM read/write	ARM library
Initialisation time / NI (ns)	95	784	6385
Setup time / conn. (ns)	246	2506	13594
Tear-down time / conn. (ns)	647	2305	9575
Binary size / NI (bytes)	-	149	67
Binary size / conn. (bytes)	-	262	104

While the effect of contention on connection instantiation time is hardly noticeable in Figure 5(b), Figure 7 shows a far more adverse scenario. A network similar to the one shown in Figure 3, with only one router between the NIs is configured with varying load on the links between NI_m, and NI_s. The raw link bandwidth is 2000 MB/s. A total of 40 IPs are active in the setup, communicating over 20 connections, of which 10 are BE. The TDM table has a size of 11, leaving one slot unreserved. Note though that the work-conserving BE scheduling also may use unoccupied GS slots.

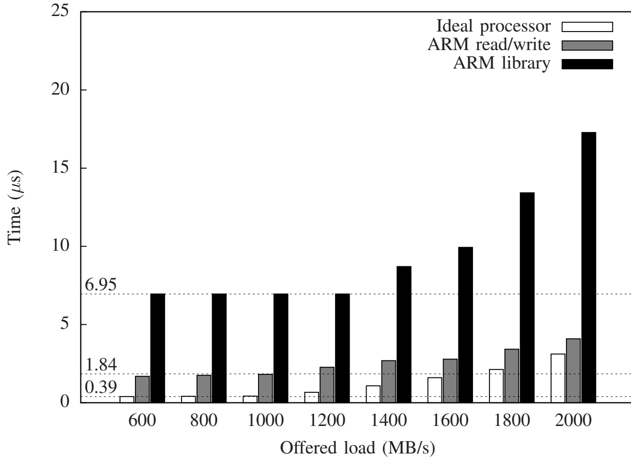


Fig. 7. Connection setup time with varying link contention.

When configuration is done with GS connections, the end result is a configuration time of 0.39, 1.84 and 6.95 μs for the three different implementations. As seen in Figure 7 this corresponds to the BE configuration time when link contention is low. We can thus conclude that the non-work-conserving nature of the TDM scheduling only marginally increases the time required. Furthermore, it is clear from the experiment that the offered load must be high (more than 60%) before there is any noticeable effect on the configuration time. Moreover, the impact varies greatly for the three implementations. The ideal processor is slowed down 8 times by the contention whereas the ARM implementations are in the order of 2.5 times slower. The latter are less affected by the increased communication time as it constitutes a smaller part of the total execution time.

Note that using GS connections for configuration enables us to give timing bounds on use-case transitions, if only the IP modules are guaranteed to react in a bounded time. When opening connections this is not an issue, as the IPs per construction are in an idle state. However, as we shall see, closing time is largely dependant on the IP reaction time.

Similar to Figure 5(b), Figure 8 shows the cumulative execution time required to tear down a use-case with a varying number of active connections. In contrast to an open operation, a tear down also involves waiting for in-flight transactions to finish. Here, we use an ideal model of the IP that is able to stop after every single transaction. Typically, *reconfiguration points* occur much more infrequently [21]. Even with the ideal model, the IP modules play an important part in determining the time required to carry out a tear down operation. This also causes a large variation across the different connections. For the ARM the fastest tear-down operation is completed in a little more than 1 μs while the library implementation requires 4.8 μs. The ideal processor does the corresponding action in 12 ns. As the IPs are involved it is not trivial (or even possible) to bound the time required to tear down a connection. The measured worst case, however, is 12 μs for all three methods.

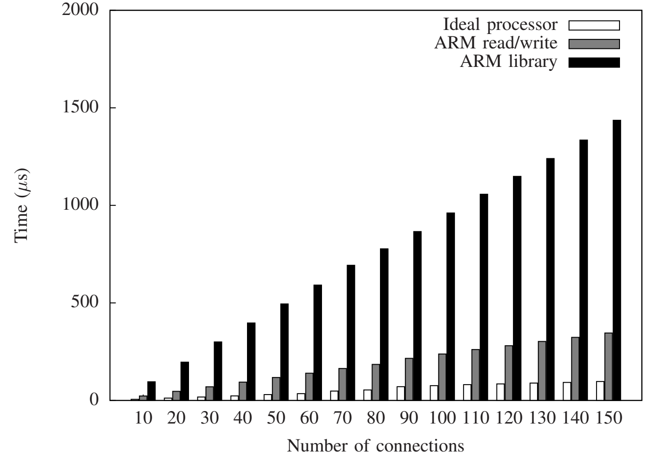


Fig. 8. Tear-down time.

Figure 9 shows the distribution of setup and tear-down times for the different connections, using the ideal processor and BE connections for configuration. As expected, the setup times are less varying, with a narrow distribution around 250 ns. The only source of variation is due to scheduling and contention in the network. The tear-down times are far more varied with a long tail to the right.

To study the impact of the remote polling used in the close operations we also implement a *polling engine*, similar to the *Synchronisation-Operation Buffer* in [30]. This minuscule hardware module is added just before the *Config* port of the NIs and transforms the remote busy-wait to a local busy-wait. Instead of reading the remote register and then check for a certain value, the desired value is first programmed in the polling engine, whereafter a subsequent read only returns once this value is read. This addition reduces the average tear-down time with 8%. More important, it reduces the amount of remote reads with 65%, but introduces one extra write for each connection as the desired value must be set. The total amount of remote transactions and data traffic is reduced with 30%.

VII. Conclusion and future work

In this paper we present the facilities for controlling change in a reconfigurable NoC. To mitigate the complexity in programming the NoC, we advocate an abstraction in the form of a run-time configuration library. Such a library must leave the modified system in a consistent state, from which normal operation can continue.

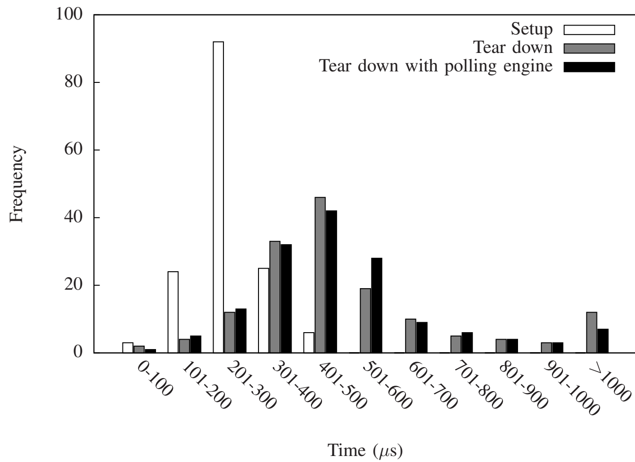


Fig. 9. Setup and tear-down time distribution

We introduce a library for NoC reconfiguration with a basic set of functions, through which we can connect the IP ports in a graph structure and dynamically add, remove or modify the interconnections. We show the architectural additions and the many trade-offs in the design of the run-time library and evaluate the performance, memory requirements, predictability and reusability for the various options.

From our experiments we can conclude that the NoC configuration has implications on the practical management policies of multiple use-cases. Even with hardware polling engines, the tail in tear-down times might cause severe predictability problems when applications are deactivated and freed resources have to be reallocated to new applications. In some cases the problem can be avoided by allocating mutually exclusive resources to the various applications, but in general this problem must be addressed on the application level. Furthermore, as the network is not the performance bottleneck, adding more ports and hence more parallel configuration connections is of little or no use. However, a centralised approach might soon reach serious scalability limitations. The time required to reconfigure the network is already substantial and this does not include the process of computing a new configuration and the reconfiguration of tasks running on the IP modules. A first step is to distribute the work to multiple configuration masters. Further decentralisation is possible by employing the distributed programming model.

In our future work, we plan to explore the coupling with a run-time mapper.

REFERENCES

[1] L. T. Smit *et al.*, "Run-time mapping of applications to a heterogeneous reconfigurable tiled system on chip architecture," in *Proc. FPT*, 2004.

[2] T. Marescaux *et al.*, "Networks on chip as hardware components of an os for reconfigurable systems," in *Proc. FPL*, 2003.

[3] M. D. van de Burgwal *et al.*, "Hydra: an energy-efficient and reconfigurable network interface," in *Proc. ERSA*, 2006.

[4] M. Rutten *et al.*, "Dynamic reconfiguration of streaming graphs on a heterogeneous multiprocessor architecture," *IS&T/SPIE Electron. Imag.*, vol. 5683, 2005.

[5] L. Benini and G. de Micheli, "Networks on chips: A new SoC paradigm," *IEEE Comp.*, vol. 35, no. 1, 2002.

[6] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Proc. DAC*, 2001.

[7] A. Rădulescu *et al.*, "An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming," *IEEE Trans. on CAD of Int. Circ. and Syst.*, 2005.

[8] M. Sgroi *et al.*, "Addressing the system-on-a-chip interconnect woes through communication-based design," in *Proc. DAC*, 2001.

[9] Z. Lu and R. Haukilahti, "NOC application programming interfaces: high level communication primitives and operating system services for power management," in *Networks on chip*, A. Jantsch and H. Tenhunen, Eds. Kluwer Academic Publishers, 2003, ch. 12.

[10] A. Hansson *et al.*, "Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip," in *Proc. DATE*, 2007.

[11] T. Bjerregaard *et al.*, "An OCP compliant network adapter for GALS-based SoC design using the MANGO network-on-chip," in *Proc. SOC*, 2005.

[12] J. Dielissen *et al.*, "Concepts and implementation of the Philips network-on-chip," in *IP-Based SOC Design*, 2003.

[13] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Trans. on Soft. Eng.*, vol. 16, no. 11, 1990.

[14] J. Kang *et al.*, "An interface for the design and implementation of dynamic applications on multi-processor architecture," in *Proc. ESTImedia*, 2005.

[15] O. Lysne *et al.*, "A methodology for developing deadlock-free dynamic network reconfiguration processes. part ii," *IEEE Trans. on Par. and Distr. Syst.*, vol. 16, no. 5, 2005.

[16] 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.

[17] K. Goossens *et al.*, "A design flow for application-specific networks on chip with guaranteed performance to accelerate SOC design and verification," in *Proc. DATE*, 2005.

[18] D. Bertozzi *et al.*, "NoC synthesis flow for customized domain specific multiprocessor systems-on-chip," *IEEE Trans. on Par. and Distr. Syst.*, vol. 16, no. 2, 2005.

[19] L. Ost *et al.*, "MAIA: a framework for networks on chip generation and verification," in *Proc. ASP-DAC*, 2005.

[20] S. Stergiou *et al.*, "×pipes lite: A synthesis oriented design library for networks on chips," in *Proc. DATE*, 2005.

[21] V. Nollet *et al.*, "Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles," in *Proc. DATE*, 2005.

[22] J. Hu and R. Mărculescu, "Exploiting the routing flexibility for energy/performance aware mapping of regular NoC architectures," in *Proc. DATE*, 2003.

[23] S. Murali *et al.*, "Mapping and physical planning of networks on chip architectures with quality of service guarantees," in *Proc. ASP-DAC*, 2005.

[24] K. Srinivasan *et al.*, "An automated technique for topology and route generation of application specific on-chip interconnection networks," in *Proc. ICCAD*, 2005.

[25] D. Wingard, "Socket-based design using decoupled interconnects," in *Interconnect-Centric design for SoC and NoC*, J. Nurmi, H. Tenhunen, J. Isoaho, and A. Jantsch, Eds. Kluwer, 2004.

[26] R. J. Bril *et al.*, "Multimedia QoS in consumer terminals," in *Proc. SIPS*, 2001.

[27] A. Nieuwland *et al.*, "C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems," *Design Automation for Embedded Systems*, vol. 7, no. 3, 2002.

[28] J. Hu and R. Marculescu, "Dyad: Smart routing for networks-on-chip," in *Proc. DAC*, 2004.

[29] K. Goossens *et al.*, "Transaction-based communication-centric debug," in *Proc. NOCS*, 2007.

[30] M. Monchiero *et al.*, "An efficient synchronization technique for multiprocessor systems on-chip," in *Proc. MEDEA*, 2005.

[31] A. Kumar *et al.*, "An FPGA design flow for reconfigurable network-based multi-processor systems on chip," in *Proc. DATE*, 2007.

[32] M. Dales, "SWARM – Software ARM," 2000, http://www.cl.cam.ac.uk/˜mwd24/phd/swarm.html.