Technical Report

A Tighter Analysis of the Worst-Case End-to-End Communication Delay in Massive Multicores

Vincent Nelis

Dakshina Dasari

Borislav Nikolic

Stefan M. Petters

HURRAY-TR-111207

Version:

Date: 12-09-2011

A Tighter Analysis of the Worst-Case End-to-End Communication Delay in Massive Multicores

Vincent Nelis, Dakshina Dasari, Borislav Nikolic, Stefan M. Petters

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail:

http://www.hurray.isep.ipp.pt

Abstract

"Many-core" systems based on the Network-on-Chip (NoC) architecture have brought into the fore-front various opportunities and challenges for the deployment of real-time systems. Such real-time systems need timing guarantees to be fulfilled. Therefore, calculating upper-bounds on the end-to-end communication delay between system components is of primaryinterest. In this work, we identify the limitations of an existing approach proposed by [1] and propose different techniques to overcome these limitations.

A Tighter Analysis of the Worst-Case End-to-End Communication Delay in Massive Multicores

Vincent Nelis, Dakshina Dasari, Borislav Nikolić, Stefan M. Petters
CISTER-ISEP Research Centre
Polytechnic Institute of Porto, Portugal
Email: {nelis, dndi, borni, smp}@isep.ipp.pt

***Abstract*—"Many-core" systems based on the Network-on-Chip (NoC) architecture have brought into the fore-front various opportunities and challenges for the deployment of real-time systems. Such real-time systems need timing guarantees to be fulfilled. Therefore, calculating upper-bounds on the end-to-end communication delay between system components is of primary interest. In this work, we identify the limitations of an existing approach proposed by [1] and propose different techniques to overcome these limitations.**

I. Introduction

The current trend in the embedded industry is towards a strong push for integrating previously isolated functionalities into a single-chip. Multicores are becoming ubiquitous, not only for general purpose systems, but also in the embedded computing area. This trend reflects the steadily increasing demands on the processing power of contemporary embedded applications. Also, advancements in the semiconductor arena have paved the way for the introduction of the "many-core" (or massive multicore) era and we are witnessing the emergence of chips with up to 1024 cores. The Tile64 from Tilera [2], Epiphany from Adapteva and the 48-core Single-Chip-Cloud computer from Intel are a few examples of such many-core systems. The immense computing capabilities offered by these chips, coupled with a power efficient design, make them potential candidates for use in real-time embedded systems. If real-time guarantees can be obtained for tasks executing on these cores, then many safety-critical real-time applications could benefit from such an architecture.

Besides offering enhanced computational capabilities compared to the traditional multicore platforms, the internal architecture of many-core platforms is fundamentally different: of particular interest is the Network-on-Chip [3] (NoC) communication framework which serves as a communication channel amongst the cores and between the cores and the main memory. System designers realized that the traditional shared bus/ring (see left plot of Figure 1) would not scale beyond a limited number of cores, because it would result in an non-negligible increase of the access time to main memory and other cores due to contention on the bus/ring. Therefore, the presence of many cores necessitated a shift in the earlier design paradigm of using a shared bus/ring as an interconnection network. Each core of a massive multicores architecture is typically a part of a more general device called "tile". Each tile is composed of a core, a private cache and a network switch connected to its neighbors (see right plot of Figure 1).

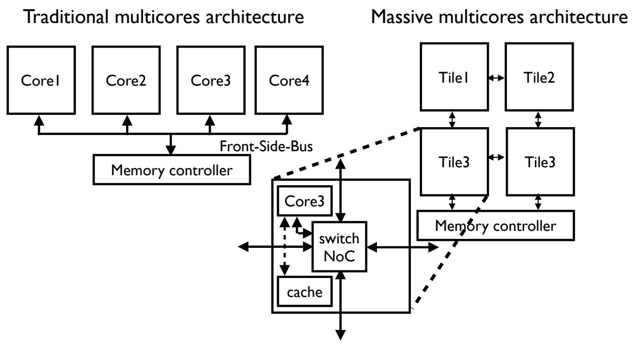


Fig. 1. Traditional vs. massive multicores architecture

It is important to find an upper bound on the delay introduced by the interconnection network in a system on which real-time applications must be deployed. In a scenario involving data transfers (amongst cores or from cores to memory), the execution time of a task running on a given core is increased because the core stalls waiting for data to be fetched over the underlying network. This waiting time can lead to a non-negligible increase in the execution time when the traffic on the network increases due to the congestion. Specifically when the analyzed task is required to meet some strict timing guarantees, this extra delay must be upper-bounded.

There has been extensive research aimed at providing guaranteed timing requirements for NoCs. Some of these use the support of special hardware mechanisms [4], using priority based mechanisms [5], time-triggered systems [6] and time division multiple access [7]. Among the most relevant studies, one can also cite [8] and [9]. The list of contributions is extensive and an entire survey is beyond the scope of this short paper. Here, we aim to identify the limitations in the work done by Ferrandiz et. al [1] and suggest improvements to provide tighter upper-bounds on the end-to-end communication delay between the system components (cores and memory). First, we identify the sources of pessimism in their approach and then, we propose techniques to tighten the upper bounds.

II. System model

We assume a platform model as illustrated in Figure 1 (right-hand plot). Each tile is composed of one core and one switch. The tiles are arranged as a $m \times m$ grid and are connected to only one memory controller. The entire grid is modeled by a directed graph $\mathcal{G}(\mathcal{N}, \mathcal{L})$, where (i) $\mathcal{N} = \{\mathrm{n}_1, \mathrm{n}_2, \ldots, \mathrm{n}_{2m^2+1}\}$ is the set of $2m^2 + 1$ nodes composed of m^2 switches, m^2 cores, and the memory controller, and (ii) $\mathcal{L}$ is the set of edges, i.e., the channels that interconnect the switches to the cores, to other switches or to the memory controller. A bi-directional channel is modeled by using two edges in opposite directions. For a given channel $l \in \mathcal{L}$, we denote by $\mathrm{src}(l)$ and $\mathrm{dest}(l)$ the origin and destination node of the directed channel, respectively. Hereafter, we will sometimes use the term *link* to refer to a channel. All the links have the same capacity denoted by C. We assume the presence of bidirectional links (full-duplex transmission) with the interpretation that request and response packets can be simultaneously sent across a tile and will not contend amongst each other for the link. Packets are switched between routers using the wormhole switching technique [10], where the arbitration in the switch is done using round-robin arbitration protocol as in Tilera architecture [11]. Within wormhole routing schemes, every packet sent over the network is split into smaller irreducible units called flits (FLow control digITS).

The tasks are periodic and non-preemptive in nature. Regarding task assignment, we assume that a single task is assigned to a given core and that there is a 1:1 mapping between a task and a core.

A communication between a task and a destination node (the memory controller or any other core) is modeled by a flow. That is, each task can generate several flows, each modeling a communication with the memory controller or with another core. Each flow f goes though a pre-defined *path*, which is defined by an ordered list of links noted $\mathrm{path}(f)$. We denote by $\mathrm{first}(f)$ the first link of $\mathrm{path}(f)$. Note that the first link of every path always connects the core and its associated switch. In addition, given a link l, we use the notations $\mathrm{prev}(f, l)$ and $\mathrm{next}(f, l)$ to refer to the links directly *before* and *after* the link l in $\mathrm{path}(f)$. If l is the first link of $\mathrm{path}(f)$ then $\mathrm{prev}(f, l)$ returns null and similarly, if l is the last link of $\mathrm{path}(f)$ then $\mathrm{next}(f, l)$ returns null. Finally, we denote by $\mathrm{psize}(f)$ the maximum size of a packet in the communication modeled by f (which includes the protocol headers).

Packets are routed statically using a deadlock free algorithm. Although adaptive routing patterns are more efficient as the route taken by a packet is decided at run-time by taking into account a global-view of the congestion in the network, they are non-deterministic and hence we adopt a static routing algorithm.

In contrast with the work presented in [1], we assume that *there can be only one outstanding request packet from a task at any given time*, i.e., the core running the task *stalls* waiting for the packet to be sent and for the response to be received (the response can be a simple ACK). This implies that the delay incurred by sending the packet and waiting for the response is added to the resulting execution time of the task running on the core. Also, we denote by $\mathrm{CR}(f, t)$ an upper-bound on the number of packets that the flow f can generate in a time

Algorithm 1: $d(f, l)$ [1]

```
input : a flow $f$,
        a link $l$,
output: an upper-bound on the end-to-end delay of $f$, starting
        from link $l$, to the destination.
1 begin
      /* there cannot be any contention on the first
         link.                                              */
2     if $l = \mathrm{first}(f)$ then return $d(f, \mathrm{next}(f, l))$ ;
      /* If the first flit of the packet has reached
         the destination, then the whole packet can
         transit.                                           */
3     if $l = \mathrm{null}$ then return $\frac{\mathrm{psize}(f)}{C}$ ;
      /* Determine the set of links connected to the
         input ports of the switch $\mathrm{src}(l)$.        */
4     $U \leftarrow \{l_{\mathrm{in}} \in L \mid l_{\mathrm{in}} \neq \mathrm{prev}(f, l) \text{ and } \mathrm{dest}(l_{\mathrm{in}}) = \mathrm{src}(l)\}$ ;
5     foreach $l_{\mathrm{in}} \in U$ do
          /* Determine the set of flows $f_{\mathrm{in}}$ passing
             through $l_{\mathrm{in}}$ and $l$).            */
6         $F_{l_{\mathrm{in}}} \leftarrow \{f_{\mathrm{in}} \in F \mid l_{\mathrm{in}} \in \mathrm{path}(f_{\mathrm{in}}) \text{ and } \mathrm{next}(f_{\mathrm{in}}, l_{\mathrm{in}}) = l\}$ ;
7     cumul_delay $\leftarrow \sum_{l_{\mathrm{in}} \in U} \max_{f_{\mathrm{in}} \in F_{l_{\mathrm{in}}}} \{d_{\mathrm{sw}} + d(f_{\mathrm{in}}, \mathrm{next}(f_{\mathrm{in}}, l))\}$
      return cumul_delay $+ d_{\mathrm{sw}} + d(f, \mathrm{next}(f, l))$ ;
```

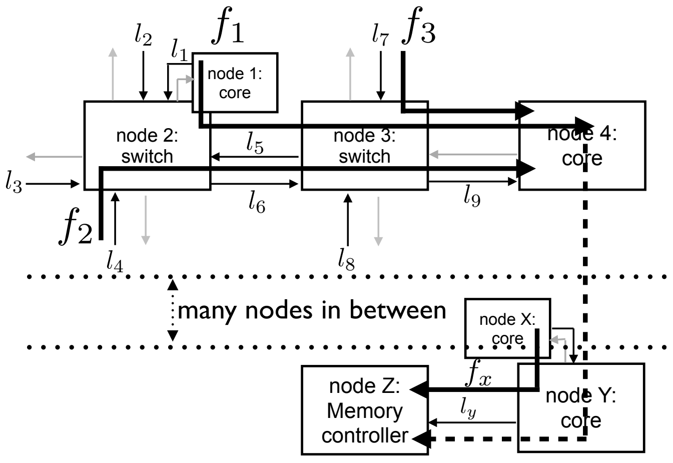


Fig. 2. Example of flows.

interval of length t, when the task initiating this flow f is run in isolation. We assume that these communication patterns are derived using either static analysis or by measurement.

III. The approach proposed in [1]

A. Description of the method

In [1], the authors propose a recursive equation to compute an upper bound $d(f, l)$ on the delay needed to deliver a packet of flow f, starting from the moment at which the packet tries to access the link l. For sake of readability, we have rewritten this recursive equation as an pseudo-code given by Algorithm 1 and the reasoning behind this procedure is explained through the following simple example.

Let us consider the part of Figure 2 above the horizontal dotted lines (the part below these dotted lines will be considered later). There are four nodes: a core n_1, two switches

n_2 and n_3, and another core n_4, and three flows: f_1 models a communication between n_1 and the core n_4, whereas the source nodes of f_2 and f_3 are not specified; f_2 transits through the switches n_2 and n_3, and f_3 passes only across the switch n_3 before ending in n_4. Suppose that we study the worst-case end-to-end delay of f_1 by calling $d(f_1, l_1)$ (see Algorithm 1).

Since $l = l_1$ is the first link of $f = f_1$, f could be blocked only if other flows generated by its source (here, the cpu n_1) have to transit first. The authors of [1] propose a particular procedure to deal with this specific case. In contrast, we assume that the cores stall while waiting for a given packet transmission to be completed before initiating a new transmission. Therefore, under this assumption, a flow f issued from one core can never be blocked by another flow issued from the same core and the first flit of the packet is directly transferred to the input port of the switch n_2. Thus, the algorithm directly calls the function $d(f, \mathrm{next}(f, l)) = d(f_1, l_6)$ at line 2.

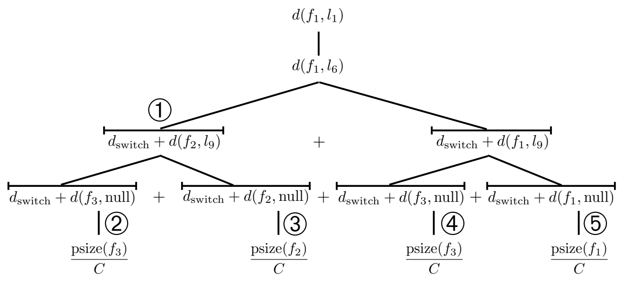
With the input $\langle f, l\rangle = \langle f_1, l_6\rangle$, the algorithm starts at line 4. At this stage, the flow f_1 is coming from l_1 and it has to pass through l_6 via the node n_2. The algorithm first computes the set U of links connected to the other input ports of n_2 (i.e., the links different from l_1). Here, $U = \{l_2, l_3, l_4, l_5\}$. Then, for each of those links $l_{\mathrm{in}} \in U$, the algorithm determines the set $F_{l_{\mathrm{in}}}$ of flows f_{in} such that f_{in} passes through l_{in} and l_6. Here, $F_{l_2} = \phi, F_{l_3} = \phi, F_{l_4} = f_2$ and $F_{l_5} = \phi$. Note that one and only one flow of each set $F_{l_{\mathrm{in}}}$ might block f_1 since the switch arbitration rule is assumed to be Round-Robin. At line 7, the algorithm sums the maximum delay that every flow f_{in} in each $F_{l_{\mathrm{in}}}$ can generate. Finally at last line returns this cumulative delay, plus the time needed to make the flow f_1 progress through n_2 (i.e., d_{sw}), plus the delay incurred by f_1 in the next hop (i.e., $d(f_1, \mathrm{next}(f_1, l_6)) = d(f_1, l_9)$).

Notice that at line 3, if $l = \mathrm{null}$, then it means that the flow f has reached its destination. In this case, the packet of f is totally transmitted, which takes in the worst-case $\frac{\mathrm{psize}(f)}{C}$ time units.

B. Sources of pessimism

Although the computation presented in the previous section is correct and terminates within a reasonable computation time (as shown in [1]), we identified two main sources of pessimism in this computation. In order to highlight this pessimism, one can construct the computation tree followed by Algorithm 1, in which each recursive call to the function $d(f, l)$, with $f \neq \mathrm{null}$, is a node of the tree and each call to $d(f, \mathrm{null})$ is a leaf (see Figure 3). Algorithm 1 traverses this computation tree in a *preorder depth-first* manner, i.e., the node is visited, and then each of the subtrees, from the left to the right.

As it can be seen in this figure, the order in which the leafs of the computational tree are reached reflects the following scenario. The flow f_1 is delayed because f_2 goes first (step ①). f_2 is then blocked by f_3 in node n_3. Once f_3 has reached the core n_4, its whole packet is transferred to n_4, hence adding $\mathrm{psize}(f_3)/C$ to the delay (step ②, the first "leaf"). Then f_2 flows and also reaches the core n_4 (step ③), followed by f_1 which passes through n_2 but gets blocked by another flow of



$d(f_1, l_1)$
$d(f_1, l_6)$
①
$d_{\mathrm{switch}} + d(f_2, l_9)$
+
$d_{\mathrm{switch}} + d(f_1, l_9)$
$d_{\mathrm{switch}} + d(f_3, \mathrm{null})$
+
$d_{\mathrm{switch}} + d(f_2, \mathrm{null})$
+
$d_{\mathrm{switch}} + d(f_3, \mathrm{null})$
+
$d_{\mathrm{switch}} + d(f_1, \mathrm{null})$
②
③
④
⑤
$\frac{\mathrm{psize}(f_3)}{C}$
$\frac{\mathrm{psize}(f_2)}{C}$
$\frac{\mathrm{psize}(f_3)}{C}$
$\frac{\mathrm{psize}(f_1)}{C}$

Fig. 3. Computation tree of $d(f_1, l_1)$.

f_3 in n_3. This second flow of f_3 passes first (step ④), and finally f_1 can progress to its destination (step ⑤).

As a conclusion, the scenario considered by this computation of $d(f_1, l_1)$ supposes that f_3 can block the flow f_1 twice before it reaches the core n_4, which may not be possible for several reasons that we call "source of pessimism" in the computation.

a) Network-level pessimism: Algorithm 1 can lead to situations in which two consecutive calls to $d(f, l)$ (with the same f and l) are too close in time so that they do not reflect a possible scenario, i.e., a scenario in which a packet of f arrives to a switch before the previous packet of the same flow f could have carried out a round-trip from its source to its destination. To understand this first source of pessimism, let us focus on the node n_3, and in particular on what it does in the scenario described above. When the flow f_3 progresses to the core n_4 for the first time, n_3 transmits every flit of the packet p_3 of f_3. During this time, both flows f_1 and f_2 are blocked in n_2. Then, right after transmitting the last flit of p_3, n_3 starts transmitting all the flits of the packet p_2 of f_2. Finally, directly after transmitting the last flit of p_2, n_3 transmits another packet of p_3 before transmitting p_1. However, this scenario is possible only if the second packet of f_3 can arrive at the switch n_3 before the packet p_2 has reached its destination, i.e., if $d_{\mathrm{sw}} + \frac{\mathrm{psize}(f_2)}{C} > \mathrm{RTT}(f_3)$, where $\mathrm{RTT}(f)$ denotes a lower-bound on the delay needed to (i) deliver a packet of flow f, from its source to its destination, and (ii) carry the response back to its source. We will address the computation of $\mathrm{RTT}(f)$, for all flows f, in future work.

In order to detect this kind of situation, in which two consecutive calls to $d(f, l)$ (with the same f and l) are too close in time so that they do not reflect a possible scenario, we propose to extend Algorithm 1 as follows. A "timestamp" $\mathrm{ts}(f, l)$ is associated to each call to the function $d(f, l)$ during the traversal of the computation tree. Basically, the computation starts with a counter count set to 0 and updates it as follows: the counter is increased by d_{sw} times units whenever it encounters a d_{sw} term in the computation (while visiting the nodes of the tree) and it is increased by $\frac{\mathrm{psize}(f)}{C}$ whenever it reaches a leaf $d(f, \mathrm{null})$ of the tree. Then, *after* visiting a node $d(f, l)$ for the first time (i.e., at the return from the first call to the function $d(f, l)$), the algorithm sets the corresponding timestamp $\mathrm{ts}(f, l)$ to the current value of count. Whenever it has to enter a node $d(f, l)$ that has been already visited, the algorithm compares the value of its corresponding timestamp $\mathrm{ts}(f, l)$ to the current value of count. If $\mathrm{count} - \mathrm{ts}(f, l) \leq \mathrm{RTT}(f)$ then it is impossible for the flow f to have another packet in the current switch at this time count, given that the last packet of f in that switch was considered at time $\mathrm{ts}(f, l)$. Therefore, the node $d(f, l)$ does not have to be traversed and can be pruned, together with all its subtrees. Since some nodes can be potentially excluded from one of the top levels of the tree, we believe that this improvement can lead to a *considerable* reduction of the pessimism of the returned upper-bound.

b) Task-level pessimism: Suppose that the function $d(f, l)$ is called multiple times with the same input parameters f and l, and suppose that every pair of consecutive calls to this function are separated in time by more than $\mathrm{RTT}(f)$ time units (thus, this scenario is valid according to the previous improvement). Let Δt denote the time between any two calls to $d(f, l)$, and let x denote the number of times that $d(f, l)$ has been called during these Δt time units (including the two calls occurring at the boundary of this time interval). It might be the case that the task generating the flow f is not able to generate x packets in a time interval of length Δt. Therefore, upon any call to the function $d(f, l)$, the algorithm should check whether the total number of calls performed to this function (including the current call) does not exceed the maximum number of packets that can be generated by the task generating f.

We propose to reduce this source of pessimism in the same way as the network-level pessimism. Basically, instead of associating one timestamp to each node $d(f, l)$ we associate a *list* of timestamps $\mathrm{list}(f, l) = \{\mathrm{ts}_1(f, l), \mathrm{ts}_2(f, l), \ldots, \mathrm{ts}_k(f, l)\}$. Whenever the computation returns from a call $d(f, l)$, it inserts a new timestamp at the end of the list (the value of the newly inserted timestamp is set to the current value of count (see the first improvement). That is, given one node $d(f, l)$, the length k of its associated $\mathrm{list}(f, l)$ gives the number of times that the function $d(f, l)$ has been called from the beginning of the computation. During the computation, before entering a node $d(f, l)$ of the tree for the $(k+1)^{\mathrm{th}}$ time ($k = 2, \ldots, \infty$), the algorithm should check, for every element $\mathrm{ts}_i(f, l)$ ($i = 1, \ldots, k$) in the associated list, whether the task generating f is capable of generating $(k+1) - i$ packets in a time interval of length $\mathrm{count} - \mathrm{ts}_i(f, l)$, i.e., $\mathrm{CR}(f, \mathrm{count} - \mathrm{ts}_i(f, l)) \leq (k+1) - i$. If this condition is satisfied for all $i = 1, \ldots, k$, then it might be possible for the task generating f to emit another $(k+1)^{\mathrm{th}}$ packet. Otherwise, the node $d(f, l)$ does not have to be traversed and can be pruned together with all its subtrees, hence further reducing the pessimism.

We believe that this second improvement can also drastically reduce the pessimism involved by Algorithm 1. The intuition is given in the example of Figure 2. Suppose now that the destination of f_1 is the memory controller denoted by the node n_Z, and suppose that this node n_Z is located far away from n_1 in terms of number of hops. In addition, suppose that there is a flow f_x from the core n_X to n_Z, where n_X is located just a few hops next to n_Z. Chances are high that Algorithm 1 will call the function $d(f_x, l_y)$ a significant number of times since this node $d(f_x, l_y)$ will belong to many subtrees. Furthermore, $\mathrm{RTT}(f_x)$ is very low since the distance between n_X and n_Z is short. In this case, this second improvement will enable not to account for $d(f_x, l_y)$ an excessive amount of times, hence reducing the pessimism.

IV. Conclusion

We present the intuition to improve the approach presented in [1], and we believe that this improvement will drastically reduce the pessimism involved in the computation of the end-to-end communication delay in massive multicores. Obtaining such a tighter upper-bound on these delays will in turn decrease the time-overhead added to the worst-case execution time (WCET) of the tasks, which will ultimately propagate in a cascading manner through the upper-layer analyses (such as tasks worst-case response time and schedulability analysis) which are built on top of the WCET analysis.

Acknowledgement

This work was supported by the RePoMuC project, ref. FCOMP-01-0124-FEDER-015050, funded by FEDER funds through COMPETE (POFC - Operational Programme 'Thematic Factors of Competitiveness) and by National Funds (PT) through FCT – Portuguese Foundation for Science and Technology – and the RECOMP project, funded by National Funds, through the FCT - Portuguese Foundation for Science and Technology, under grant ref. ARTEMIS/0202/2009, as well as by the ARTEMIS Joint Undertaking, under grant agreement no. 100202.

References

[1] T. Ferrandiz, F. Frances, and C. Fraboul, "A method of computation for worst-case delay analysis on spacewire networks," in *Proceedings of the IEEE International Symposium on Industrial Embedded Systems*, 2009, pp. 19–27.

[2] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, pp. 15–31, 2007.

[3] L. Benini and G. De Micheli, "Networks on chips: a new soc paradigm," *Computer*, vol. 35, no. 1, pp. 70 –78, jan 2002.

[4] J. Diemer and R. Ernst, "Back suction: Service guarantees for latency-sensitive on-chip networks," in *Networks-on-Chip (NOCS), 2010 Fourth ACM/IEEE International Symposium on*, may 2010, pp. 155 –162.

[5] Z. Shi and A. Burns, "Real-time communication analysis for on-chip networks with wormhole switching," in *Networks-on-Chip, 2008. NoCS 2008. Second ACM/IEEE International Symposium on*, april 2008, pp. 161 –170.

[6] C. Paukovits and H. Kopetz, "Concepts of switching in the time-triggered network-on-chip," in *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08. 14th IEEE International Conference on*, aug. 2008, pp. 120 –129.

[7] K. Goossens, J. Dielissen, and A. Radulescu, "Aethereal network on chip: concepts, architectures, and implementations," *Design Test of Computers, IEEE*, vol. 22, no. 5, pp. 414 – 421, sept.-oct. 2005.

[8] Y. Qian, Z. Lu, and W. Dou, "Analysis of worst-case delay bounds for on-chip packet-switching networks," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 29, pp. 802 –815, may 2010.

[9] D. Rahmati, S. Murali, L. Benini, F. Angiolini, G. De Micheli, and H. Sarbazi-Azad, "A method for calculating hard qos guarantees for networks-on-chip," in *Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, nov. 2009, pp. 579 –586.

[10] W. Dally and S. C., "The torus routing chip," *Distributed Computing*, vol. 1, pp. 187 –196, 1986.

[11] "Tile processor: user architecture manual," May 2011. [Online]. Available: http://www.tilera.com/scm/docs/UG101-User-Architecture-Reference.pdf