CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Technical Report

Worst-Case Memory Traffic Analysis for Many-Cores using a Limited Migrative Model

Borislav Nikolic
Patrick Meumeu Yomsi
Stefan M. Petters

CISTER-TR-130902
Version:
Date: 09-13-2013

Worst-Case Memory Traffic Analysis for Many-Cores using a Limited Migrative Model

Borislav Nikolic, Patrick Meumeu Yomsi, Stefan M. Petters

CISTER Research Unit

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.cister.isep.ipp.pt

Abstract

The ratio between the number of cores and memory subsystems (i.e. banks and controllers) in many-core platforms is constantly increasing, leading to non-negligible latencies of memory operations. Thus, in order to study the worst-case execution time of an application, it is no longer sufficient to only take into account its computational requirements, but also have to be considered latencies related to its memory operations.

In this paper we study a limited migrative model applied upon many-core platforms. This approach is based on a multi-kernel paradigm – a promising step towards scalable and predictable many-cores, which are essential prerequisites for the integration of such systems into the real-time embedded domain. Under that assumption, we present two analytical methods to obtain the worst-case memory traffic delays of individual applications. Through experiments we test the applicability of the proposed approaches to different scenarios, and draw practical conclusions concerning routing mechanisms and a distribution of memory operations across memory controllers.

Worst-Case Memory Traffic Analysis for Many-Cores using a Limited Migrative Model

Borislav Nikolić, Patrick Meumeu Yomsi and Stefan M. Petters
CISTER/INESC-TEC, ISEP, IPP, Porto, Portugal
Email: {borni, pamyo, smp}@isep.ipp.pt

***Abstract*—The ratio between the number of cores and memory subsystems (i.e. banks and controllers) in many-core platforms is constantly increasing, leading to non-negligible latencies of memory operations. Thus, in order to study the worst-case execution time of an application, it is no longer sufficient to only take into account its computational requirements, but also have to be considered latencies related to its memory operations.**

In this paper we study a *limited migrative model* applied upon many-core platforms. This approach is based on a *multi-kernel paradigm* [3] – a promising step towards scalable and predictable many-cores, which are essential prerequisites for the integration of such systems into the real-time embedded domain. Under that assumption, we present two analytical methods to obtain the worst-case memory traffic delays of individual applications. Through experiments we test the applicability of the proposed approaches to different scenarios, and draw practical conclusions concerning routing mechanisms and a distribution of memory operations across memory controllers.

I. Introduction

The miniaturisation process in the semiconductor technology hit the limit [19]. In order to cope with the constantly increasing requirements for more powerful computational devices, chip manufacturers took a design paradigm shift [14], [26]; instead of enhancing the capabilities of single-core devices, they opted to integrate multiple cores within a single chip. Platforms consisting of several cores (*multi-cores*) and more than a dozen of cores (*many-cores*) have became nowadays the mainstream in many scientific areas, most notably *high performance computing*, while are the new frontier technology in others like *real-time embedded systems*. This paper focuses on the latter.

In order to apply many-core platforms into the real-time embedded domain, an OS paradigm is required such that it allows to exploit the full potential of the underlying architecture, and yet assures scalability and predictability, which are essential prerequisites for deriving execution guarantees. *Limited Migrative Model* (LMM) [22] builds on top of the foundations and principles of the *multi-kernel paradigm* [3], which offers both scalability and predictability, and hence promises to be a suitable approach towards deriving execution guarantees for many-cores.

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within SMARTS project, ref. FCOMP-01-0124-FEDER-020536 and by FCT and ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH/BD/81087/2011.

Development trends associated to many-cores suggest that the ratio between the number of cores and available memory subsystems (i.e. banks and controllers) is constantly rising. This causes an increase in contentions, both within memory controllers and the interconnect medium. Consequently, the latencies of memory operations are rising as well. Thus, in order to derive the worst-case execution times of applications, it is no longer sufficient to only analyse computational requirements, but latencies of memory operations also have to be considered [23].

In many-cores, Network-on-Chip architecture (NoC [4]) became a predominant interconnect medium, due to its scalability potential [16]. The majority of currently available NoC-based many-core platforms (e.g. [14], [26]) perform data transfer via *wormhole switching technique* [20], due to its good throughput and small buffering requirements [16].

In this paper we study the worst-case memory traffic delays of applications residing within a NoC-based, wormhole-switched many-core platform encompassing LMM. Specifically, we propose two methods to obtain upper bound estimates on the worst-case traversal delays of memory operations related to individual applications. This approach allows to divide the problem and study only the contention of memory operations within NoC, while abstracting away the latencies occurring within memory controllers, which are out of scope of this paper, and were extensively studied (e.g. [23], [27]). Through experiments we investigate the suitability of the proposed methods to specific scenarios. Additionally, we draw practical conclusions concerning routing policies and a distribution of memory operations across memory controllers, which may impact the design of future real-time many-core systems, from both hardware and software perspectives.

II. Motivation and concepts of LMM

Scalability, unpredictability and pessimism are some of the issues which make real-time analysis of many-cores a challenging subject. Thus, before these architectures can be integrated into the real-time embedded domain, an OS paradigm is needed, which will allow to exploit the full potential of the underlying platform, and yet assure predictability and scalability, such that the execution guarantees can be derived.

Current state-of-the-art approaches which address many-cores from the real-time perspective can be broadly classified into two categories: *Non-Migrative Approaches* and *Migrative Approaches*. We firstly introduce these categories and then position LMM in that context.

Non-migrative approaches [18] are in the literature also known as *Fully-Partitioned Approaches*. Each application is

migrationless, and statically assigned to a specific core where it has to execute. Such schemes are inflexible, and can be very inefficient in scenarios with substantial load changes where run-time load balancing is required for energy and thermal management, performance enhancements or fault tolerance reasons.

Migrative approaches are further classified into *Semi-Partitioned Approaches* and *Global Approaches*. The first group ([5], [15]) assumes a static assignment of an application to a particular core (or cores if it migrates). A migrative application also obeys to design-time decisions, i.e. always has to execute the prescribed fraction of work on each of assigned cores. Hence, these methods are very similar to fully-partitioned approaches and the same observations regarding inflexibility to perform run-time load balancing also hold in this context. Conversely, global approaches [2] allow every application to execute on any core within the platform. However, this amount of flexibility comes with the price. Due to necessity to maintain global structures (e.g. ready-queue) and have a centralised entity, scalability issues arise [3], and serious challenges occur when attempting real implementations [12].

LMM [22] is an approach that builds on top of the fundamental concepts of the multi-kernel paradigm [3], which presents a promising step towards deriving execution guarantees for many-cores. One well known example of a multi-kernel OS is *Barrelfish* [3], and many LMM concepts are inspired by it. LMM poses a realistic constraint that each application may execute only on a subset of cores, which are decided during design time. During runtime, an application has the flexibility to migrate between candidate cores and execute on any of them. This removes the necessity to maintain global structures, which contributes the scalability [3], and yet gives the possibility to perform run-time load balancing.

III. Related Work

The wormhole switching technique [20] is not a novelty in academia, nor industry, but was introduced more than twenty years ago. However, it was neglected by chip manufacturers, as an alternative - *store and forward switching technique* was providing satisfactory results [16]. Recently, buffering within routers became more challenging [16], due to the constant increase in the amount of data that has to be transferred. This brought wormhole switching back into focus, resulting in the commodity of present many-core architectures (e.g. [14], [26]) employing that technique. Additionally, in the aforementioned platforms the packets are predominantly routed via a static, dimension-ordered *XY routing policy* and a round-robin arbitration scheme is used within routers.

If a platform provides a single virtual channel, complex interference patterns may occur [17]. Several methods have been proposed to obtain upper-bounds on worst-case latencies ([8], [10], [11]), but they yield either pessimistic results [10], [11], or have complexity and scalability issues [8]. Conversely, if a platform contains multiple virtual channels ([6], [7]), the benefits are twofold: (i) by avoiding idle routers, the performance of the wormhole switching technique is significantly improved ([6], [7]) and (ii) preemptions can be implemented [24]. Shi and Burns employ this, and several additional assumptions, namely *per-priority virtual channels* ([6], [7]), *flit-level preemptions* [25] and per-packet distinctive priorities,

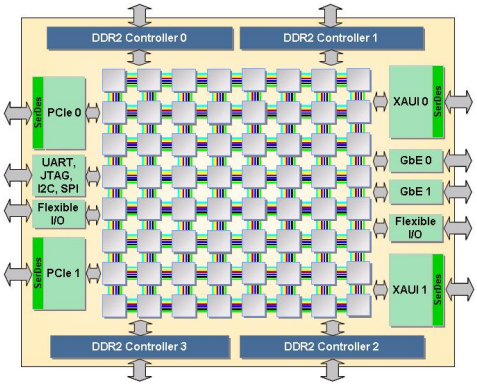

Fig. 1: TILEPro64 Platform

and also present the worst-case analysis for NoCs with wormhole switching [24].

All the aforementioned analyses provide upper-bounds on the latency of a single packet occurrence, or what we further call *per-packet analysis*. However, LMM brings additional challenges due to path uncertainty (discussed later), thus rendering none of the proposed methods directly applicable to the model. We recognised that problem, introduced placement constraints and reroutings to make paths more deterministic, and proposed a per-packet analysis for intra- and inter-application traffic [21], which can be efficiently applied to LMM.

However, in practical scenarios, instead of a single occurrence of a packet, sometimes it is more important to bound the latency of the sequence of packet occurrences. In that vein, for intra- and inter-application traffic we presented a path-abstracting method of computing the cumulative delay of all packet occurrences, happening within minimum inter-arrival period of an application [22]. We call this *per-pattern analysis*, where pattern in this case corresponds to an application's minimum inter-arrival period.

Although these two per-packet and per-pattern approaches were initially derived for intra- and inter-application traffic (i.e. core-to-core communication), they can be also applied to core-to-memory traffic. However, in many cases applications communicate with memory controllers far more frequent than with other applications. Thus, in order to be considered as applicable, a prerequisite is that an approach is deemed scalable as well. As will be discussed later, that is not the case with the previously mentioned approaches. Therefore, in this paper we firstly formulate modifications and restrictions which are necessary in order to make the per-packet and per-pattern approaches applicable to memory traffic, and consequently propose such methods.

IV. Model

A. Hardware

The system under consideration is a NoC-based many-core platform, comprising of $n \times n$ tiles $\mathcal{T} = \{t_1, t_2, ..., t_{n^2-1}, t_{n^2}\}$, and 4 memory controllers $\mathcal{M} = \{mc_0, mc_1, mc_2, mc_3\}$, where the first two are accessible from the topmost row of tiles, while the other two from the bottommost (see Figure 1). Additionally, each controller provides a concurrent access to $\frac{n}{2}$ tiles from its access row, e.g. top-left controller to $\frac{n}{2}$ leftmost tiles of the topmost row, etc. Memory accesses on a depicted platform (Figure 1) are organised in this manner.

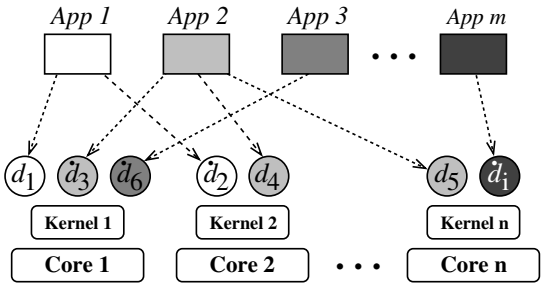

Fig. 2: Limited Migrative Model - LMM

Although in some platforms a tile contains multiple cores [14], without loss of generality in this paper we assume that each tile comprises of a single core and a single router, like in the illustrated system (Figure 1). Furthermore, the platform employs a XY routing policy, which is deadlock and livelock free [13]. Such policy assumes that packets firstly traverse along the x-axis, and upon reaching the x-coordinate of the destination, traverse along the y-axis. Data transfer involves wormhole switching. This means that each data packet that has to be transferred is divided into small elements of fixed size, called *flits* [20]. The first flit establishes the route, and the other flits follow in a pipeline manner. Packets are prioritised, and there exist per-priority virtual channels. These are used to perform flit-level preemptions [24]. Virtual channels are implemented as per-priority flit-sized buffers within each router. Furthermore, core-to-core and core-to-memory traffic use separate physical links [14], [26], thus mutually do not contend, i.e. the existing intra- and inter-application communication does not influence the memory traffic analysis.

B. Software layers in LMM

As mentioned in Section II, LMM encompasses a multi-kernel paradigm. Thus, each core runs an independent micro-kernel instance. Kernels are mutually interacting and form the basic communication infrastructure. Each kernel exposes some of its functionalities to applications residing on its core and allows them to communicate with applications located on the same, or other cores. Furthermore, each kernel is responsible for the scheduling process on its core.

As discussed (Section II), an application may execute only on a subset of cores which are decided at design-time. On each of corresponding cores, the execution code of that application exists, encapsulated within an entity called *dispatcher*. Dispatchers of the same application (each located on a different core) communicate among themselves via agreement protocols [22], and discuss temporal and spatial properties of the application's next execution, i.e. will the migration occur, and if so, which core (dispatcher) will be the destination.

Applications are single-threaded and implemented as tasks. Multi-threaded applications can be implemented as multiple applications with dependencies. A dispatcher elected via agreement protocol is termed *master dispatcher*, and it is responsible for releasing the job of its application on its core. A job has to complete the execution on the master's core, it can not migrate (i.e. job-level migrations are not allowed). Other dispatchers that participate in the protocol are called *slave dispatchers*. After a job execution is completed, the master initiates another instance of the agreement protocol. If the outcome is that a migration occurs – the master changes; the previous master becomes the slave, while the newly elected slave becomes the master. Perceived from application's perspective, its dispatchers exchange one master token, which position is decided during runtime. Thus, being a master is a temporary role for a dispatcher. We call this *master volatility phenomenon* and it has several implications which will be discussed in Subsection V-B. The architectural structure of LMM is illustrated in Figure 2, where master dispatchers are distinguished by dots over their names.

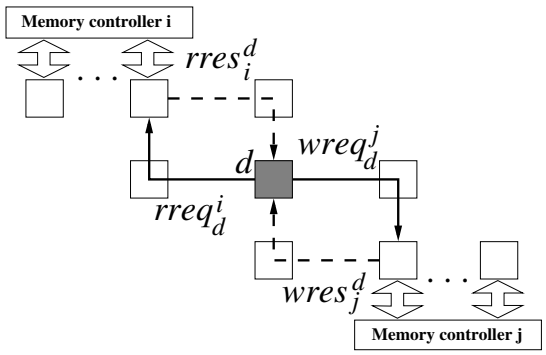

Fig. 3: Example of memory operations

An application-set $\mathcal{A}$ comprises of m applications $\mathcal{A} = \{a_1, a_2, ..., a_{m-1}, a_m\}$, where each application $a\langle T_a, C_a, W_a, P_a, \mathcal{D}_a, \mathcal{R}_a\rangle$ is characterised by its minimum inter-arrival period T_a, the execution time $C_a \leq T_a$, the worst-case protocol duration $W_a \leq T_a$, a unique priority P_a, a set of dispatchers $\mathcal{D}_a$ and a set of memory operations $\mathcal{R}_a$.

C. Memory operations

A memory operation $r_a\langle req, res, o\rangle \in \mathcal{R}_a$ of an application a consists of a request req sent from its master dispatcher d to one memory controller mc, and a consequent response res from mc to d. req can be a read request $rreq_d^{mc}$, or a write request $wreq_d^{mc}$. Similarly, res can be a read or a write response – $rres_{mc}^d$ and $wres_{mc}^d$, respectively. $o(r_a)$ defines the maximum number of occurrences of a given memory operation within T_a, i.e. one minimum inter-arrival period of a.

Requests and responses are implemented as *packets*. Each packet $p_i^j \in \mathcal{P}$ traverses a static, XY routed path from the source i to the destination j in a wormhole-switched manner. For requests, i is a dispatcher and j is a memory controller, while for responses i denotes memory controller, and j symbolises a dispatcher. Each packet is characterised by the priority it inherits from its application, the amount of transferred content – $size(p)$, the distance between i and j expressed in hops – $nhops(p)$, the traversed path – $path(p)$, and the maximum number of occurrences $o(p)$, inherited from its memory operation. Memory operations are executed sequentially, so at any time instant an application can have only one packet within the network. This prevents a concurrent existence of same-priority packets and their mutual blocking.

A request is considered as delivered when all flits of a packet reach the tile that provides an access to the targeted memory controller. As said in Section I, contentions occurring inside memory controllers are out of scope of this paper. Similarly, the response is considered as delivered when all flits of a packet reach the tile of the master dispatcher which issued

that memory request. The destination tile of the request packet is the source tile of the response packet, and vice versa.

An illustrative example is given if Figure 3, where a master dispatcher d (located on the shaded core) performs a read operation with the controller i and a write operation with the controller j. Solid lines depict requests, while dashed lines symbolise responses.

The latency of an individual occurrence of a packet p when traversing in isolation – $l(p)$ is in the literature known as *basic network latency* [9]. It is equivalent to the time needed for the first flit to reach the destination router, augmented by the processing rate of all flits at the destination router (Equation 1). d_{sw} and d_t are mesh latencies to switch the crossbar and route the flit, while $size(f)$ represents the size of the flit.

$$l(p) = nhops(p) \times (d_{sw} + d_t) + \left\lceil \frac{size(p)}{size(f)} \right\rceil \times d_t \quad (1)$$

V. Proposed Approach

A. Interferences

Equation 1 expresses the traversal latency of a packet in isolation. Additionally, a packet may suffer interference from other higher priority packets encountered on its path. In order to provide an upper-bound on the worst-case traversal delay of a packet, all possible interferences have to be considered.

Definition 1 (Directly interfering packet): If a packet p' of an application a' shares a part of the path with the packet under analysis p of an application a, and has a higher priority than p, it is considered as a *directly interfering packet* and belongs to the set $\mathcal{P}_D(p)$. Formally:

$$\forall p' \in \mathcal{P} : p' \in \mathcal{D}_{a'} \wedge P_{a'} > P_a \wedge path(p') \cap path(p) \neq \emptyset \Rightarrow p' \in \mathcal{P}_D(p) \quad (2)$$

The packet under analysis p can be preempted by a directly interfering packet p', and consequently suffer interference. The delay caused to p by a single occurrence of p' is equivalent to the traversal latency of p' (Equation 3).

$$I_p(p', 1) = l(p') = nhops(p') \times (d_{sw} + d_t) + \left\lceil \frac{size(p')}{size(f)} \right\rceil \times d_t \quad (3)$$

Definition 2 (Indirectly interfering packet): If a packet p'' of an application a'' does not share a part of the path with the packet under analysis p of an application a, but shares it with some other packet p' of an application a', which is either directly or indirectly interfering packet of p, and has a higher priority than p', it is considered as an *indirectly interfering packet* and belongs to the set $\mathcal{P}_I(p)$. Formally:

$$\forall p'' \in \mathcal{P} : p'' \in \mathcal{D}_{a''} \wedge p'' \notin \mathcal{P}_D(p) \wedge \exists p' \in \mathcal{D}_{a'} \wedge p' \in \{\mathcal{P}_D(p) \cup \mathcal{P}_I(p)\} \wedge P_{a''} > P_{a'} \wedge path(p'') \cap path(p') \neq \emptyset \Rightarrow p'' \in \mathcal{P}_I(p) \quad (4)$$

When the NoC infrastructure contains only a single virtual channel, complex interference patterns may occur [17], causing the packet under analysis to be blocked by both *directly* and *indirectly* interfering packets. However, when per-priority virtual channels and flit-level preemptions exist, a packet can suffer interference only from directly interfering packets (see Theorem 1).

Theorem 1: In wormhole-switched NoCs with per-priority virtual channels and flit-level preemptions, any packet p with a unique priority can not suffer interference from indirectly interfering packets.

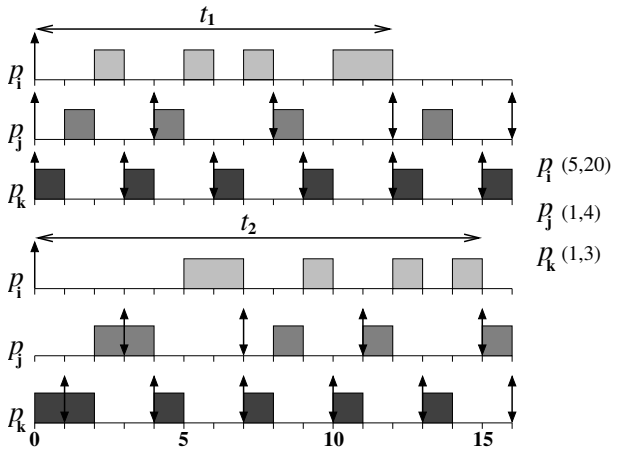

Fig. 4: Example of interferences

Proof: Proven by contradiction. Consider that p'' is a directly interfering packet of p', which is a directly interfering packet of p, and that p'' and p do not share a common part of the path. Thus, by Definition 2, p'' is an indirectly interfering packet of p. Assume that p'' can cause an interference to p. By initial assumption, p is preempted. Furthermore, as p'' causes an interference it is traversing. Due to the traversal of p'', a packet p' is either preempted or does not exist at that time instant. In either case, it is not progressing. Thus, the routers on the path of p are idle. Due to per-priority virtual channels, p can uninterruptedly reach its destination, even though preempted packets of p' may exist on its path (which would not be the case in a scheme with a single virtual channel, where p would have to wait until p' passes). Contradiction reached. ■

In spite of the fact that indirectly interfering packets can not cause interference, they can influence occurrence patterns of directly interfering packets. Thus, considering periodic appearances of directly interfering packets is an unsafe assumption. This is illustrated with an example given in Figure 4, where a packet p_i suffers direct interferences from two packets – p_j and p_k. Values in brackets represent a traversal time of a packet and a minimum inter-arrival period of its application, respectively. For the sake of simplicity, consider that each packet occurs only once in the inter-arrival period of its application, i.e. $o(p_i) = o(p_j) = o(p_k) = 1$. In case of periodic appearances of all packets (upper part of Figure 4), the worst-case traversal delay of p_i is 12 time units. However, if p_j and p_k suffer interference from their directly interfering packets, they may exhibit occurrence patterns like in the lower part of Figure 4. Consequently, the delay of p_i can additionally increase.

Note, that occurrences of the same packet p associated to distinctive minimum inter-arrival periods of its application a have to be separated by at least W_a, which is the duration of the agreement protocol that has to be executed between two job instances. For clarity reasons, $W = 0$ was assumed in this example, for all depicted packets and their applications. Therefore, in order to compute the worst-case traversal delay of a packet, it is necessary to assume the worst-case occurrence pattern for each of its directly interfering packets. Shi and Burns provide a method [24], as a function of all direct and indirect interferences. However, that approach is not applicable to LMM, as it includes hundreds of applications, and hence packets, which can cause combinatorial explosion and render the analysis intractable when constructing interference trees. We dramatically reduce the complexity by taking a more

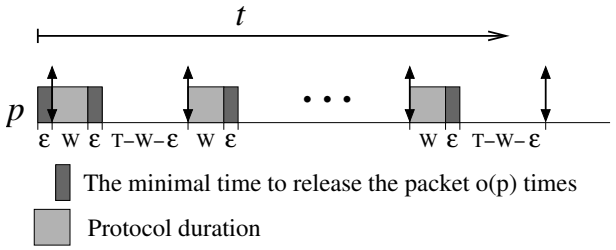

Fig. 5: Worst-case occurrence pattern

pessimistic approach; we assume the theoretically worst-case occurrence pattern for every directly interfering packet, and make the analysis independent of indirect interferences. Specifically, we consider that the first occurrence of every directly interfering packet happened as late as possible, while all others happened as early as possible. An illustrative example is given in Figure 5. ϵ represents infinitesimally small, but finite value, during which all $o(p)$ occurrences of a given packet p may happen. The safe assumption is that $\epsilon \to 0$.

In order to compute the maximum interference that a packet under analysis might suffer within the observed time interval t, firstly the maximum number of occurrences of every directly interfering packet within t has to be derived (Theorem 2).

Theorem 2: The number of occurrences of a packet p, belonging to an application a, within the time interval t can be *at most* $o(p) \times \left(1 + \left\lceil \frac{t-W_a}{T_a} \right\rceil\right)$.

Proof: Proven by contradiction. Let us assume that in total $o(p) \times \left(1 + \left\lceil \frac{t-W_a}{T_a} \right\rceil\right) + 1$ packet occurrences happened within the time interval t. As $o(p)$ represents the maximum number of occurrences of p within one minimum inter-arrival period (MIP) of its application, according to the initial assumption $2 + \left\lceil \frac{t-W_a}{T_a} \right\rceil$ MIPs occurred within t. Consider $\left\lceil \frac{t-W_a}{T_a} \right\rceil$ MIPs, surrounded by the first and the last and we refer to them as to *inner MIPs*. All the inner MIPs contribute to t with their entire duration, therefore require time interval of at least $\left\lceil \frac{t-W_a}{T_a} \right\rceil \times T_a$ where only these can execute, and where only $o(p) \times \left\lceil \frac{t-W_a}{T_a} \right\rceil$ packet occurrences could happen. Additionally, assume that ϵ is infinitesimally small value, representing the shortest possible fraction of MIP, when all $o(p)$ packet occurrences can happen. Consider that the first such interval was delayed as much as possible and hence completed just before the interval of the inner MIPs started. Finally, the last packet occurrence could not happen before the protocol of duration W_a completed (see Figure 5).

$$\epsilon + \left\lceil \tfrac{t-W_a}{T_a} \right\rceil \times T_a + W_a \geq \epsilon + \left(\tfrac{t-W_a}{T_a} \right) T_a + W_a = \epsilon + t > t$$

Contradiction reached. ■

This coincides with Theorem 1 of our previous work [22], where we computed the maximum number of protocol occurrences within a given time interval, which is a problem orthogonal to the one we solve here.

Now we can compute the maximum interference that a packet under analysis p might suffer from a directly interfering packet p' within the time interval t (Equation 5). It is equal to the product of the maximum number of occurrences of p' within t (Theorem 2) and the latency of a single interference (Equation 3). $a_{p'}$ stands for the application of p'.

$$I_p(p', t) = \overbrace{I_p(p', 1)}^{\text{single interference}} \times \overbrace{o(p') \times \left(1 + \left\lceil \frac{t - W_{a_{p'}}}{T_{a_{p'}}} \right\rceil\right)}^{\text{maximum number of occurrences}} \quad (5)$$

We extend this reasoning, and compute the maximum interference a packet p might suffer from all directly interfering packets $\mathcal{P}_D(p)$ within the time interval t (Equation 6).

$$I_p(t) = \sum_{\forall p' \in \mathcal{P}_D(p)} I_p(p', t) \quad (6)$$

Due to flit-level preemptions, a packet of interest can be additionally blocked by one lower priority packet within each router on its path for at most one flit traversal time ($d_{sw} + d_t$). The blocking delay of a single occurrence of a packet p is expressed with Equation 7.

$$B_p = nhops(p) \times (d_{sw} + d_t) \quad (7)$$

B. Challenges of LMM: Pessimism and Intractability

Each memory operation between an application and a particular memory controller renders 2 packets – a *request* and a *response*. If both read and write operations are included, the number of packets is 4. Each additional memory controller may render 4 new packets. However, in LMM, applications do not have fixed locations, but a set of dispatchers. Thus, the total number of distinctive packets associated with each application can be at most $4 \times |d| \times |mc|$, where $|d|$ stands for the number of dispatchers of an application, while $|mc|$ represents the number of accessed memory controllers. Moreover, as only one dispatcher can be a master at any time instance, all packets can not exist during the same inter-arrival period and some of them are mutually exclusive. An illustrative example of a 4-dispatcher application accessing one memory controller, only with a read operation, is given in Figure 6. Of all 8 depicted packets, during any application's minimum inter-arrival period, only 2 may exist (one read request and one read response, involving the dispatcher which is the master during the observed interval). Performing the analysis on such a model can be either overly pessimistic (considering concurrent existence of all mutually exclusive packets), or computationally expensive (analysing all possible scenarios arising from the fact that every dispatcher can be a master).

C. Inapplicability of the existing method

The problem mentioned in Section V-B also occurs when, instead of memory traffic, intra- and inter-application traffic is considered [21]. In that case, the number of distinctive packets was significantly reduced by enforcing placement and communication constraints, formally introduced below.

Constraint 1: [21] Dispatchers of an application can be placed only on the edges of a rectangular $a \times b$ structure, such that no corner is left unoccupied and $a, b \in \mathbb{N}$. The special case is a line-like shape, which occurs when one or both dimensions of a shape are equal to "1".

Constraint 2: [21] Intra-application messages travel only on the edge of the shape its application is forming, and re-routing occurs where needed to comply with the global XY routing policy. An individual message rotation (i.e. clockwise or counterclockwise) is chosen such that the traversal distance is minimised.

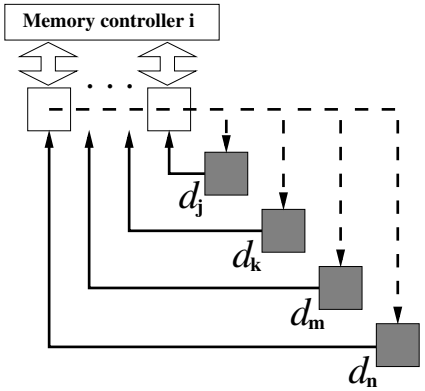

Fig. 6: Unconstrained operations

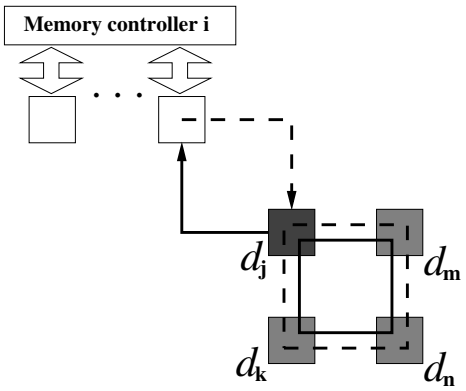

Fig. 7: Example with proxies

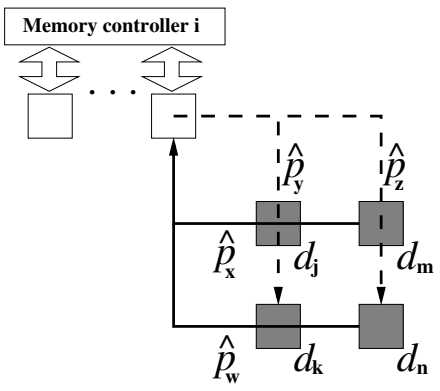

Fig. 8: Example with superpackets

Assuming said restrictions, similar approach could be applied to memory traffic. One dispatcher is chosen for each memory controller that an application communicates with, called *proxy dispatcher*. It is responsible for mediating the communication between the current master dispatcher and the memory controller. An example is depicted in Figure 7, where proxy dispatcher d_j is emphasized with a darker color. In this scenario, a master sends the request (read or write) to its proxy. Then, proxy issues the request to the memory controller on behalf of its master. Upon receiving the response, the proxy forwards it to the master. Even though there exist hardware mechanisms which can be instrumented to efficiently perform rerouting operations (e.g. Hardwall™ feature of Tilera platforms [26]), the number of memory requests that an application issues within its minimum inter-arrival period can reach thousands, which would create a substantial overhead on proxy cores and lead towards poor performance. Thus, although applicable to intra- and inter-application communication, this approach does not have sufficient scalability potential to be applied to memory traffic as well.

D. Placement constraints, access constraints and superpackets

In order to make the analysis tractable, in this subsection we propose the method to (i) decrease the total number of considered packets and (ii) solve the problem of mutually exclusive packets. We assume dispatcher placement constraints given by Constraint 1. Furthermore, we define constraints regarding tiles via which packets access memory controllers.

Constraint 3: If an application accesses a memory controller, all dispatchers access it from the same tile, which is (i) for the leftmost controllers (i.e. top and bottom) chosen such that it is either in the column of the application's leftmost dispatcher, or left from it, and (ii) for the rightmost controllers, chosen such that it is either in the column of the application's rightmost dispatcher, or right from it.

In other words, on a platform with $n \times n$ tiles, if an application's leftmost and rightmost dispatchers occupy columns i and j respectively, then the leftmost controllers can be accessed via columns $1 : i$ and the rightmost via columns $j : n$. This was done with an intention to cause overlapping paths of mutually exclusive packets (notice the visual difference between Figure 8 and Figure 6). Now we define *superpackets* – a generalisation of overlapped, mutually exclusive packets.

Definition 3 (Superpacket): Superpacket $\widehat{p}$ is a packet that exists for each memory operation that an application performs, and for each row and column of its shape, where it has dispatchers. It connects a memory controller, and a dispatcher from a given row/column, with the furthest distance from it.

The per-row superpacket is directed from the tile to the memory controller, and represents a generalisation of overlapped, mutually exclusive requests from its row. The per-column superpacket is directed from the memory controller to the tile, and depicts a generalisation of overlapped, mutually exclusive responses from its column. An example is given in Figure 8. An application with 4 dispatchers has 4 superpackets: 2 per-row – $\widehat{p_x}$ and $\widehat{p_w}$ and two per-column – $\widehat{p_y}$ and $\widehat{p_z}$.

The benefits of superpackets can be shown by considering all possible read request packets that might be generated from the row w (bottom row in the example of Figure 8), i.e. the ones generated by dispatchers d_k and d_n – $rreq_k^i$ and $rreq_n^i$. These read requests are mutually exclusive, as they belong to different dispatchers of the same application. Additionally, both share a part or entirely the path with the superpacket $\widehat{p_w}$. Thus, assuming that $\widehat{p_w}$ has identical packet characteristics as $rreq_k^i$ and $rreq_n^i$ (i.e. size, priority and number of occurrences), the maximum delay of $rreq_k^i$ and $rreq_n^i$ has an upper-bound, which is equivalent to the maximum possible delay of the $\widehat{p_w}$. Therefore, a superpacket can substitute all mutually exclusive read request packets from the same row, targeting the same controller. The same applies for write requests. Similarly, a superpacket can substitute all mutually exclusive read and write responses from the same controller, targeting the same column (e.g. $\widehat{p_y}$ can substitute $rres_i^j$ and $rres_i^k$). Instead on packets, the worst-case analysis can be performed on superpackets, which number is smaller. If a superpacket fulfils posed time constraint, all substituted packets also fulfil it.

In order to estimate the reduction in the computational complexity, let us compute the amount of superpackets. Consider a single memory operation of an application with a and b dispatchers on horizontal and vertical edges of its shape, respectively. Thus, an application has $2 \times (a+b-2)$ dispatchers and hence $4 \times (a+b-2)$ packets, but only $a+b$ superpackets. In the aforementioned example in Figure 8, an application with $a = b = 2$ and 4 dispatchers has 8 packets, but only 4 superpackets. On the application-set level, the total number of superpackets is $\left(\frac{4\times(a+b-2)}{a+b}\right)^m \approx 4^m$ times less than the number of packets, where m symbolises the number of applications in the application-set. That is, the analysis on superpackets is $O(4^m)$ times computationally less complex

than the analysis on packets. In many practical cases this can make a difference between the tractability and intractability.

Note, it may appear that the analysis on superpackets inherently brings pessimism, since many packets may share only a fraction of the path with the superpacket, which delay they will assume as the corresponding upper-bound. But it is not so. Timing constraints are posed on groups of mutually exclusive packets and not individual packets. Thus, the only matter is whether all possible mutually exclusive packets (arising from different dispatcher positions) meet a certain constraint, while the tightness of the estimate of individual packets is irrelevant.

Also note, a superpacket exists for every memory operation. Therefore, if an application performs both read and write operations with a memory controller, similarly to distinctive read and write packets, there will also be distinctive superpackets, for both operations.

E. Solution to mutually exclusive superpackets

Superpackets substitute mutually exclusive requests from the same row, and mutually exclusive responses from the same column. However, mutually exclusive superpackets still exist. For instance, in Figure 8, $\widehat{p_x}$ and $\widehat{p_w}$ are mutually exclusive, the same applies for $\widehat{p_y}$ and $\widehat{p_z}$.

In this subsection we propose the method to solve this problem. Let $\widehat{\mathcal{P}_E}(\widehat{p})$ be a set of all mutually exclusive superpackets of $\widehat{p}$, including $\widehat{p}$. As during one minimum inter-arrival of a only one dispatcher can be a master, consequently only one of mutually exclusive superpackets may exist. Observe that all of them have the same packet properties (i.e. size, priority, number of occurrences), only differ in the traversal latency, and hence interference delay they can cause per single occurrence (Equation 3). Thus, when a superpacket has multiple mutually exclusive superpackets in its list of directly interfering superpackets, only one of them with the maximum traversal latency can be assumed. The conclusion reached for $\widehat{\mathcal{P}_E}(\widehat{p})$ is also valid for every set of mutually exclusive superpackets of every application.

F. Approach one: Per-packet analysis

In this approach, of interest is the maximum delay of a single occurrence of a superpacket $\widehat{p}$. This delay consists of a sum of: an isolation latency (Equation 1), a blocking delay (Equation 7), and an interference delay (Equation 6). However, as discussed in Subsection V-E, not all directly interfering superpackets $\mathcal{P}_D(\widehat{p})$ should be considered, as some of them are mutually exclusive. Thus, we define a reduced set of directly interfering superpackets as follows. If two or more superpackets from $\mathcal{P}_D(\widehat{p})$ are mutually exclusive, only one with the highest traversal latency is added to the set $\mathcal{P}_R(\widehat{p})$.

Formally:

$$\forall \widehat{p'} \in \mathcal{P}_D(\widehat{p}) : \nexists \widehat{p''} \in \mathcal{P}_D(\widehat{p}) \wedge \widehat{p''} \in \widehat{\mathcal{P}_E}(\widehat{p'}) \wedge l(\widehat{p''}) > l(\widehat{p'}) \Rightarrow \widehat{p'} \in \mathcal{P}_R(\widehat{p}) \tag{8}$$

Equation 9 presents the maximum delay of a single occurrence of a superpacket $\widehat{p}$. Note, that Equation 9 has a recursive notion, hence is solved iteratively, until reaching a fixed conversing point (if one exists).

$$D_{\widehat{p}}(1) = \overbrace{l(\widehat{p})}^{\text{isolation}} + \overbrace{B_{\widehat{p}}}^{\text{blocking}} + \overbrace{\sum_{\forall \widehat{p'} \in \mathcal{P}_R(\widehat{p})} l(\widehat{p'}) \times o(\widehat{p'}) \times \left(1 + \left\lceil \frac{D_{\widehat{p}}(1) - W_{a_{\widehat{p'}}}}{T_{a_{\widehat{p'}}}} \right\rceil\right)}^{\text{interference}} \tag{9}$$

The total delay of a superpacket $\widehat{p}$ within a minimum inter-arrival period of its application is equivalent to the delay of a single occurrence, multiplied by the number of occurrences $o(\widehat{p})$ (Equation 10).

$$D_{\widehat{p}} = D_{\widehat{p}}(1) \times o(\widehat{p}) \tag{10}$$

Upon obtaining the worst-case delays of superpackets within application's minimum inter-arrival period, it is possible to derive the worst-case delay of an entire application within that time. For an application a we define the reduced set of superpackets $\mathcal{P}_R(a)$ as follows. From each set of mutually exclusive superpackets of a only one with the maximum delay is added to $\mathcal{P}_R(a)$. Formally:

$$\forall \widehat{p} \in \mathcal{R}_a : \nexists \widehat{p'} \in \widehat{\mathcal{P}_E}(\widehat{p}) \wedge D_{\widehat{p'}} > D_{\widehat{p}} \Rightarrow \widehat{p} \in \mathcal{P}_R(a) \tag{11}$$

Theorem 3: The worst-case memory traffic delay of an application within one minimum inter-arrival period is defined with Equation 12.

$$D_a = \sum_{\forall \widehat{p} \in \mathcal{P}_R(a)} D_{\widehat{p}} \tag{12}$$

Proof: Follows directly from Equations 8-11. ■

G. Intermediate step: Partial per-pattern analysis

Obtaining the worst-case delay of a single occurrence of a superpacket, and consequently multiplying it with the number of occurrences might be a beneficial approach for computation-intensive applications, where few memory requests occur. However, memory-intensive applications have hundreds, if not thousands of memory requests per job execution. Thus, assuming the worst-case for every occurrence of a memory operation might be a very pessimistic approach. In this subsection we present the method to compute the worst-case delay of superpackets, not per single occurrence, but per group of occurrences, i.e. per pattern. Specifically, the goal is to compute the worst-case delay of a superpacket, but assuming all occurrences that happen within one minimum inter-arrival period of an application.

Similarly to the previous approach, for the superpacket under analysis $\widehat{p}$, we define a reduced set of directly interfering superpackets $\mathcal{P}_R(\widehat{p})$, i.e. a set where only one of mutually exclusive superpackets exists. Consequently, the worst-case delay of $\widehat{p}$ is given by Equation 13. Note that, as occurrences of $\widehat{p}$ might be spread within the entire minimum inter-arrival period of its application, excluding protocol duration, the interference has to be computed within that period: $T_{a_{\widehat{p}}} - W_{a_{\widehat{p}}}$. Due to that fact, Equation 13 does not have a recursive notion.

$$D_{\widehat{p}} = \overbrace{l(\widehat{p}) \times o(\widehat{p})}^{\text{isolation}} + \overbrace{B_{\widehat{p}} \times o(\widehat{p})}^{\text{blocking}} + \overbrace{\sum_{\forall \widehat{p'} \in \mathcal{P}_R(\widehat{p})} l(\widehat{p'}) \times o(\widehat{p'}) \times \left(1 + \left\lceil \frac{T_{a_{\widehat{p}}} - W_{a_{\widehat{p}}} - W_{a_{\widehat{p'}}}}{T_{a_{\widehat{p'}}}} \right\rceil\right)}^{\text{interference}} \tag{13}$$

Now, Equations 11-12 are used to compute the worst-case memory traffic delay of an application.

H. Approach two: Full per-pattern analysis

We refer to the previous method as *partial per-pattern analysis*. It is strictly worse than or equal or this approach, so we consider it as an intermediate step. Here we present a method which we call *full per-pattern analysis*.

As described in Section V-D, distinctive superpackets exist for each memory operation. Therefore, if an application performs read and write operations with a memory controller, distinctive superpackets will be generated for a read request, a read response, a write request and a write response. Although superpackets of read and write requests have different sizes and number of occurrences, they have the same priority and some of them share same paths. In such cases they have the same list of directly interfering superpackets. Therefore, the idea behind this approach is to merge superpackets forming read and write requests which share the same path, and compute their grouped delay. The same observation hold for read and write responses.

Consider $\widehat{p_1}$ and $\widehat{p_2}$, which are superpackets of a read and a write request of an application $a_{\widehat{p}}$, and which share the same path. Thus, their joined restricted list of directly interfering superpackets $\mathcal{P}_R(\widehat{p_{12}})$ is equivalent to their individual lists, i.e. $\mathcal{P}_R(\widehat{p_{12}}) = \mathcal{P}_R(\widehat{p_1}) = \mathcal{P}_R(\widehat{p_2})$. Their worst-case grouped delay $D_{\widehat{p_{12}}}$ can be expressed by Equation 14. Due to the same reasons as for the partial per-pattern analysis, Equation 14 does not have a recursive notion. Note that, this approach behaves identically as the partial per-pattern analysis in cases where no superpackets that can be merged exist, i.e. Equation 14 becomes Equation 13. On the other hand, intuitively, this approach should provide tighter estimates than the partial per-pattern analysis in cases where superpackets can be merged, i.e. both read and write operations are performed with the same controller. This is further investigated in Section VI.

$$D_{\widehat{p_{12}}} = \overbrace{l(\widehat{p_1}) \times o(\widehat{p_1}) + l(\widehat{p_2}) \times o(\widehat{p_2})}^{\text{isolation}} + \overbrace{B_{\widehat{p_1}} \times o(\widehat{p_1}) + B_{\widehat{p_2}} \times o(\widehat{p_2})}^{\text{blocking}} + \overbrace{\sum_{\forall \widehat{p'} \in \mathcal{P}_R(\widehat{p_{12}})} l(\widehat{p'}) \times o(\widehat{p'}) \times \left(1 + \left\lceil \frac{T_{a_{\widehat{p}}} - W_{a_{\widehat{p}}} - W_{a_{\widehat{p'}}}}{T_{a_{\widehat{p'}}}} \right\rceil\right)}^{\text{interference}} \quad (14)$$

Again, Equations 11-12 can be used to derive the worst-case memory traffic delay of an application.

VI. Evaluations

In this section we evaluate the efficiency of the proposed approaches. Specifically, we compare the tightness of the results derived with all three methods, and investigate how these trends change with different application parameters, e.g. priority, minimum inter-arrival period, number of memory operations. Furthermore, we draw practical conclusions concerning routing policies and a distribution of memory accesses across memory controllers.

A. Analysis parameters

Analysis parameters are given in Table I. An asterisk sign denotes a randomly generated value, assuming a uniform distribution.

TABLE I: Analysis parameters

Platform size	**8×8**
Application-set size	**200**
Router switch latency	**1 cycle**
Router transfer latency	**3 cycles**
Mesh width	**16B**
Control packet size (read request and write response)	**32B**
Content packet size (read response and write request)	**1kB**
Minimum inter-arrival periods of applications	**[30-1000]* mS**
Dispatchers per application	**[2-10]***
Memory requests of computation-intensive applications	**[1-50]***
Memory requests of memory-intensive applications	**[100-1000]***
Different memory controllers accessed by one application	**[1-4]***

B. Experiments

Experiment 1: Overall comparison

In this experiment we conducted the overall comparison of the proposed approaches. Each application of an application-set was randomly mapped on the grid, assuming dispatcher placement constraints (Constraint 1). Consequently, packets of memory operations were generated. Then, the worst-case memory traffic delay was computed for each application, with all three methods. Finally, we compared the obtained values. The process was repeated for 1000 application-sets.

Figure 9 shows the improvements of the partial per-pattern analysis over the per-packet analysis. It is visible that the per-packet analysis renders tighter estimates only for 3.64% of applications. This receives additional attention in Experiment 4. In 0.53% of the cases, both methods derive the same results, while in all other cases the partial per-pattern analysis performs better. Specifically, for more than half of applications, the improvements are greater than 90%, which means that the estimates are tighter 10 times or more.

Figure 10 compares the full per-pattern analysis and the partial per-pattern analysis. For 4.20% of applications both methods derive the same values. As concluded in Subsection V-H, these are the cases when an application performs only one (read or write) operation with every memory controller that it communicates with. In the rest of scenarios, the full per-pattern analysis provides improvements, which in this case grow until 50%. Additional conclusion is that the full per-pattern analysis dominates the partial per-pattern analysis (strictly better or equal). This coincides with the intuitive assumption from Subsection V-H.

Experiment 2: Distribution of accesses across controllers

In this experiment we investigated how the distribution of memory accesses across memory controllers might impact the analysis. Thus, we assume that all data that an application needs is fetched from only one memory controller. Similarly, we derived worst-case delays for each application of the application-set, and repeated the process for 1000 application-sets, assuming the full per-pattern analysis. We compared the obtained values with the results from the previous experiment for the full per-pattern analysis, where every application may access multiple memory controllers.

The improvements achieved by a scheme where each application accesses only a single controller are demonstrated in Figure 11. Surprisingly, this approach does not always yield better results. In fact, for 9.27% of applications this scheme renders worse estimates. The explanation is that, when

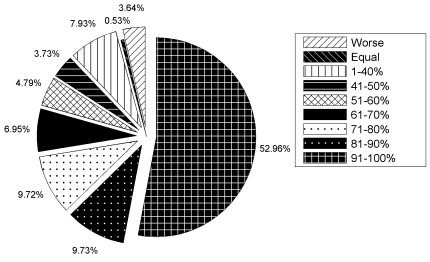

Fig. 9: Partial per-pattern v. per-packet

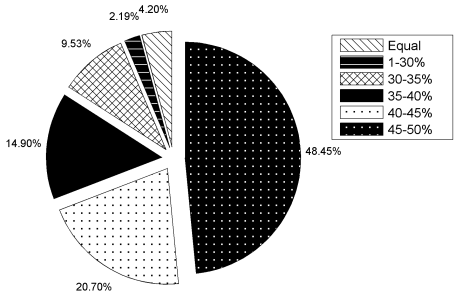

Fig. 10: Full per-pattern v. partial per-pattern

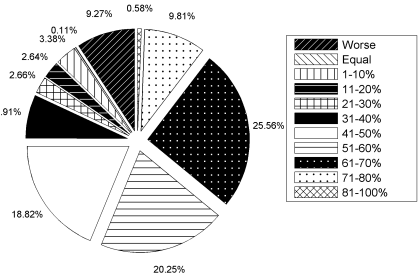

Fig. 11: Imp. when accessing a single controller

an application accesses only a single controller, the path is "greedily" consumed by its entire traffic. Consequently, some links of the NoC infrastructure are extensively used, causing a substantial interference to any lower priority application which utilises that path. Oppositely, when each application accesses multiple memory controllers, the traffic is more equally distributed, thus making it possible for lower priority applications to suffer less interference. However, the rest of applications benefit from the new scheme and have tighter estimates. Thus, we conclude that the distribution of memory accesses plays an important role in the analysis and we consider it as a potential topic for the future research.

Experiment 3: Different memory operations and packets

In this experiment the objective is to quantify the fraction of the total application delay spent in each individual component, namely read requests, read responses, write requests and write responses. In order to investigate that, we performed the per-packet analysis. Of interest were only applications that contain all 4 components, i.e. perform both read and write operations with the memory controller. The analysis was performed for each such application in the application-set, and repeated for 1000 application-sets. Upon obtaining the delays of single packet occurrences, we estimated the contributions of individual delays in the cumulative delay of all 4 packets.

The results are presented in Figure 12. It is visible that read and write request packets are almost overlapping. Furthermore, both packets share the same path and only differ in the packet size. The same is also true for both response packets. Thus, the first conclusion is that the packet size almost does not have any influence on the delay. Additionally, responses have significantly higher delays than the requests, each averaging at 27% of the total cumulative delay. Requests consume less, around 23% each. The explanation for this surprising conclusion is that XY routing experiences problems with memory responses [1]. Specifically, each response packet is injected into the NoC either in the topmost row or in the bottommost. Firstly, a packet traverses on the x-axis. However, all other responses also do the same. This can cause large amount of contention within the topmost and the bottommost row. The effects in Figure 12 are substantially mitigated, due to the existence of per-priority virtual channels which prevent indirect interferences. However, in scenarios with a single virtual channel, this can cause more significant impacts on delays of response packets, thus further motivating the research in the area of routing mechanisms. We also see this topic as a potential future work.

Experiment 4: Computation-intensive applications

In this experiment we investigated the applicability of the proposed methods to computation-intensive applications. For each such application we varied the number of memory operations and the minimum inter-arrival period. Consequently, we performed the per-packet and the full per-pattern analysis, and compared the obtained values. The computation was repeated for 1000 application-sets.

The results are depicted in Figure 13. It is visible that the per-packet analysis renders tighter estimates for applications with few memory operations and long periods. As the number of operations increases, the improvements over the full per-pattern analysis decline. Additional increase favours the full per-pattern analysis, which outperforms the per-packet analysis on the rest of the domain. Similar trends apply to minimum inter-arrival periods of applications, where any decrease also decreases the benefits of the per-packet analysis.

Experiment 5: Memory-intensive applications

It is obvious that the full per-pattern analysis is the most suitable method for memory-intensive applications, thus that aspect is not investigated. In this experiment we again focus on the distribution of memory accesses across memory controllers. Experiment 2 demonstrated the positive and negative sides of having the data of each application fetched from a single memory controller. However, in some cases an application has to be mapped on a platform which already has an existing application-set. In such cases, dedicating only one memory controller to it might be a good decision from the perspective of that application, but might have a negative impact on the already existing workload (as shown in Experiment 2). Thus, in order to minimise the effects of the new application on the existing system it might be more beneficial to distribute its memory accesses across multiple memory controllers. Motivated by this reasoning, in this experiment we quantified the individual, per-application overheads of having its memory content fetched from multiple memory controllers. For each memory-intensive application we varied the priority and performed the full per-pattern analysis assuming its memory operations are equally spread across: (i) only one controller, (ii) two controllers, (iii) three controllers and (iv) four controllers. Consequently, we estimated the penalty of accessing multiple controllers, when compared to the schemes where only one controller is accessed. The experiment was repeated for all 1000 application-sets.

Figure 14 shows the penalty of having the data fetched from multiple memory controllers. Surprisingly, the priority has a very small impact on the results. For higher priorities (smaller numbers), applications suffer fewer contentions, thus the penalty of accessing multiple controllers is predominantly composed of increased traversal distances. As priority decreases, schemes with multiple controllers suffer the interfer-

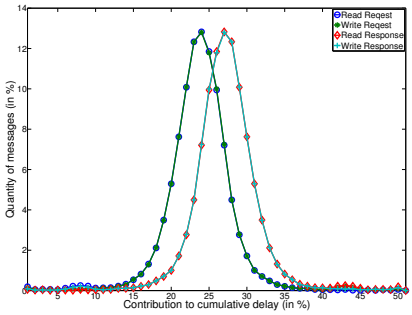

Fig. 12: Delays of different packets

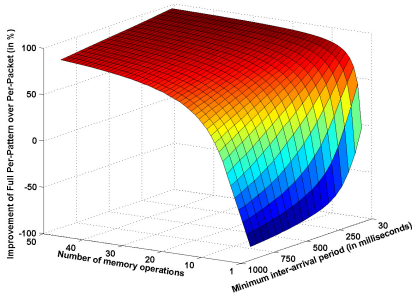

Fig. 13: Full per-pattern v. per-packet

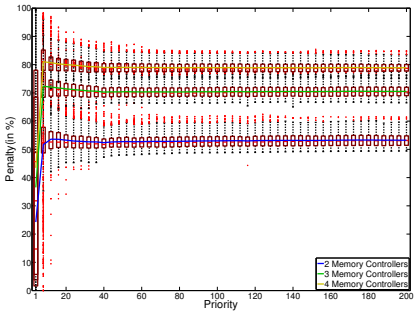

Fig. 14: Penalty of accessing multiple controllers

ence, due to the spread traffic across the grid. Oppositely, scenarios with single controller accesses consume less NoC infrastructure, and in many cases still manage to avoid the interference. This causes the increase in the penalty. Around the priority level 10, the interference becomes predominant, resulting in significant delays even for single controller accessing schemes, thus a slight drop in the penalty is visible. Finally, the penalty stays constant on the rest of the domain. As is visible, the same trends apply for all scenarios involving accesses to multiple controllers. Moreover, the results suggest that the distribution of application's data accesses among multiple controllers is very "expensive". Thus, one of the strategies when adding new application might be to distribute its accesses among as much different memory controllers as possible, such that its temporal constraints are still fulfilled. In this way the impact on the existing system would be minimised. This falls into the domain of application mapping, which is also a very investigated area and promising for the future work.

VII. Conclusions and Future Work

In this paper we elaborate on a *Limited Migrative Model*, which is a promising approach towards integrating many-core platforms into the real-time embedded domain. Assuming such model, we propose two methods to obtain the upper-bound estimates on the worst-case memory traffic delays of individual applications. Through experiments we investigated the applicability of the proposed methods to specific application types, and also drew practical conclusions regarding routing policies and a distribution of memory accesses across memory controllers. The future work was already mentioned, we aim to investigate (i) different routing policies, (ii) memory access distribution strategies and (iii) application mapping strategies.

References

[1] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson, and M. H. Lipasti. Achieving predictable performance through better memory controller placement in many-core cmps. In *36th ISCA*, pages 451–461, 2009.

[2] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: a notion of fairness in resource allocation. In *25th annual ACM symposium Theory computing*, New York, NY, USA, 1993.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new os architecture for scalable multicore systems. In *SOSP*, 2009.

[4] L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. *The Comp. J.*, 35(1):70 –78, jan 2002.

[5] K. Bletsas and B. Andersson. Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. In *30th RTSS*, 2009.

[6] W. Dally. Virtual-channel flow control. *Trans. Parall. & Distr. Syst.*, 3(2):194 –205, Mar 1992.

[7] W. Dally and C. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *Trans. Computers*, 1987.

[8] D. Dasari, B. Nikolić, V. Nelis, and S. M. Petters. A tighter analysis of the worst-case end-to-end communication delays in massive multicores. In *WIP Session 33rd RTSS*, 2012.

[9] J. Duato, S. Yalamanchili, and N. Lionel. *Interconnection Networks: An Engineering Approach*. M.K. Publishers, 2002.

[10] T. Ferrandiz, F. Frances, and C. Fraboul. A method of computation for worst-case delay analysis on spacewire networks. In *IEEE Int. Symp. Industrial Emb. Syst.*, 2009.

[11] T. Ferrandiz, F. Frances, and C. Fraboul. Using network calculus to compute end-to-end delays in spacewire networks. *SIGBED Rev.*, 2011.

[12] P. L. Holman. *On the implementation of pfair-scheduled multiprocessor systems*. PhD thesis, The University of North Carolina at Chapel Hill, North Carolina, United States, 2004.

[13] J. Hu and R. Marculescu. Energy-aware mapping for tile-based noc architectures under performance constraints. In *8th ASPDAC*, 2003.

[14] Intel. *Single-Chip-Cloud Computer*. www.intel.com/content/www/us/en/research/intel-labs-single-chip-cloud-computer.html.

[15] S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *21st ECRTS*, 2009.

[16] N. K. Kavaldjiev and G. J. M. Smit. A survey of efficient on-chip communications for soc. In *Symp. Emb. Syst.*, 2003.

[17] B. Kim, J. Kim, S. Hong, and S. Lee. A real-time communication method for wormhole switching networks. In *1998 Int. Conf. Parall. Processing*, Aug 1998.

[18] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 1973.

[19] M. Lundstrom. Moore's law forever? *Science*, 2003.

[20] L. M. Ni and P. K. McKinley. A survey of wormhole routing techniques in direct networks. *The Comp. J.*, 26, 1993.

[21] B. Nikolić and S. M. Petters. Real-time application mapping for many-cores using a limited migrative model. Technical report. Available at: http://www.cister.isep.ipp.pt/people/Borislav+Nikolic/publications/.

[22] B. Nikolić and S. M. Petters. Towards network-on-chip agreement protocols. In *12th EMSOFT*, 2012.

[23] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *47th DATE*, 2010.

[24] Z. Shi and A. Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *Int. Symp. Netw.-on-Chip*, 2008.

[25] H. Song, B. Kwon, and H. Yoon. Throttle and preempt: a new flow control for real-time communications in wormhole networks. In *1997 Int. Conf. Parall. Processing*, Aug 1997.

[26] Tilera. *TILE64™ Processor*. www.tilera.com/products/processors/TILE64.

[27] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *24th ECRTS*, 2012.