Technical Report

Towards Real-Time Agreement Protocols For Many-Cores

Borislav Nikolic

Stefan M. Petters

HURRAY-TR-120905

Version:

Date: 09-10-2012

Towards Real-Time Agreement Protocols For Many-Cores

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

Demands for functionality enhancements, cost reductions and power savings clearly suggest the introduction of multi- and many-core platforms in real-time embedded systems. However, when compared to uni-core platforms, the many-cores experience additional problems, namely the lack of scalable coherence mechanisms and the necessity to perform migrations. These problems have to be addressed before such systems can be considered for integration into the real-time embedded domain. We have devised several agreement protocols which solve some of the aforementioned issues. The protocols allow the applications to plan and organise their future executions both temporally and spatially (i.e. when and where the next job will be executed). Decisions can be driven by several factors, e.g. load balancing, energy savings and thermal issues. All presented protocols are analytically described, with the particular emphasis on their respective real-time behaviours and worst-case performance. The underlying assumptions are based on the multi-kernel model and the message-passing paradigm, which constitutes the communication between the interacting instances.

Towards Network-on-Chip Agreement Protocols

Borislav Nikolić and Stefan M. Petters *
CISTER/INESC-TEC, ISEP, IPP
Porto, Portugal
borni@isep.ipp.pt, smp@isep.ipp.pt

ABSTRACT

Demands for functionality enhancements, cost reductions and power savings clearly suggest the introduction of multi- and many-core platforms in real-time embedded systems. However, when compared to uni-core platforms, the many-cores experience additional problems, namely the lack of scalable coherence mechanisms and the necessity to perform migrations. These problems have to be addressed before such systems can be considered for integration into the real-time embedded domain.

We have devised several agreement protocols which solve some of the aforementioned issues. The protocols allow the applications to plan and organise their future executions both temporally and spatially (i.e. *when* and *where* the next job will be executed). Decisions can be driven by several factors, e.g. load balancing, energy savings and thermal issues. All presented protocols are analytically described, with the particular emphasis on their respective real-time behaviours and worst-case performance. The underlying assumptions are based on the multi-kernel model and the message-passing paradigm, which constitutes the communication between the interacting instances.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems

Keywords

Real-Time, Many-Core, Embedded Systems, Agreement Protocols, Worst-Case Execution-Time

*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 REPOMUC project, ref. FCOMP–01-0124-FEDER-015050, by FCT and the EU ARTEMIS JU funding, within RECOMP project, ref. ARTEMIS/0202/2009, JU grant nr. 100202 and by FCT and the ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH/BD/81087/2011.

EMSOFT'12, October 7-12, 2012, Tampere, Finland.

1. INTRODUCTION

With the current improvements in technology, there is an ever increasing need for *embedded systems*. So far these devices often had limited capacities, performed only predefined set of functions and operated in the conditions with explicitly posed constraints, such as occupied space and/or consumed power. Additionally, many embedded systems perform time-critical jobs (e.g. automotive industry, avionics), where not only the correctness of the computation is important, but also the duration of the execution itself. These systems are called *real-time embedded systems*.

1.1 Single-core → Multi-core → Many-core

Current real-time embedded systems are mostly single-core devices. However, there are several reasons which require a reassessment of this concept. Firstly, there is an increasing demand for functionality enhancements (i.e. more complex processing requires more powerful devices). Secondly, significant cost reduction can be achieved by integrating several of those devices into one. Very similar trends apply for power conservation reasons. Finally, some applications, due to their distributed nature, clearly demand extensive communication between interacting modules and can also benefit from further integration (e.g. trading systems, air traffic control).

The platforms in the server and high performance computing areas were progressing from single-cores to multi-cores and finally many-cores. The same evolution is visible in general purpose computers, although with an offset in time. Embedded systems lag even more behind the aforementioned technologies but the same trend exists.

However, the application of many-core platforms in the real-time embedded domain is far from trivial and brings new overheads. Firstly, isolation has to be provided between different executing applications (previously located on separate, independent devices). For instance, failures or misbehaving of one shouldn't influence the execution of the other applications which share the same system. Secondly, the concept of mixed criticalities has to be introduced and proper isolation has to be assured between applications of different importance.

1.2 Message-passing $\overset{scalability}{\gg}_{predictability}$ Coherency

Finally, current cache coherency mechanisms used in multi-cores are not applicable to many-core platforms, even for general purpose computers, due to scalability issues [9]. Current cache policies even cause performance drops when the number of cores increases to more than a dozen. [15] claims that future commodity systems will drop the idea of having a completely coherent system.

Sharing paradigm also imposes additional problems which are common for both uni- and many-core platforms, namely unpredictability and pessimism. Including the cache effects into the real-

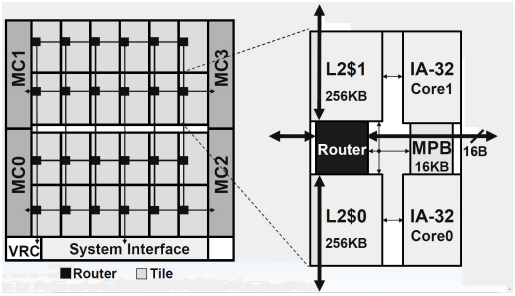

Figure 1: Intel's Single-Chip-Cloud (SCC)

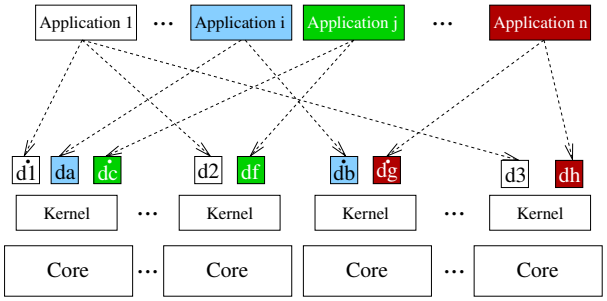

Figure 2: Architectural structure of the assumed model

time analysis presents a serious challenge even for single-cores as can be seen from the following works: [3] presents the idea how to incorporate the cache effects into the schedulability analysis by taking a probabilistic approach, while [18] governs code analysis so as to predict the cache misses. Performing the same in the many-core environment is even more complex and one solution to this outstanding problem is a methodology shift.

Although the message-passing paradigm was introduced many years ago, it was largely neglected due to the efficiency of the sharing paradigm when applied to the systems with small number of cores. [13] and [12] show that two approaches are dual and the dominance of one over another is highly dependant on the concrete purpose.

Current many-core devices, such as [20] and [10] (Figure 1) present experimental platforms which don't facilitate cache coherence mechanisms. The depicted system contains 24 tiles, each shared by two cores. The cores have private caches and communicate with the environment through hardware built-in support for message-passing called *Message-Passing Buffer* (MPB, see Figure 1).

Some operating systems, such as Barrelfish [2], advocate a *multi-kernel* approach in which every core runs a light-weight version of the OS. In such a system the kernel instances and respective applications running on top communicate with the corresponding entities located on other cores by utilising a message-passing paradigm, not only to maintain the correct system-wide state, but also to discuss temporal and spatial properties of future executions. The benefits of this approach are twofold; primarily the scalability is not an issue anymore, since it is well known from distributed and cloud computing areas that agreement protocols scale. Secondly, the message-passing model is predictable and therefore suitable for real-time analysis and, in our opinion, presents promising platform for further investigation. Secondary benefits that come with the design are the possibilities to perform load balancing, energy savings by deliberately shutting down some of the cores, thermal management, even wear out, etc.

The areas of networking, distributed and parallel computing study agreement protocols with the emphasis on either security [1], [14] or fault-tolerance [5], [7]. The performance of network-on-chip synchronisation is discussed in [21] and [6]. Present many-core scheduling algorithms [4], [11] assume instantaneous migrations with negligible overhead. However, no work was done in calculating and incorporating the communication and the migration costs into the schedulability analysis.

Based on the assumption of a non-coherent many-core platform utilising message passing as a primary primitive, our contribution is to present several agreement protocols that facilitate the task-level job migrations. The protocols provide mechanisms to derive such a decision but do not prescribe a specific policy for the migration for e.g. load balancing purposes. All protocols are analytically described and compared in terms of the amount of messages generated as well as their worst-case behaviour in successfully migrating a given job.

In the next section we give an in-detail description of the assumed model. Then, in Section 3 several agreement protocols are described with particular emphasis on their real-time characteristics. Section 4 focuses on the evaluation of the protocols. Finally, Section 5 concludes the paper with the summary of the findings, and the description of the future work.

2. MODEL

2.1 The Hardware

The platform of interest is one many-core device with network-on-chip as an interconnection network, such as [20], [10] or [19]. The system is non-coherent, which means that assuring and maintaining correct and coherent system-wide state is the responsibility of either the OS or the applications themselves. The transfer on the mesh network is packet-based, utilising wormhole routing technique [8] with the routes computed by dimensioned-ordered routing algorithm XY (the packets always travel in the X direction first) which is present in [10] and [19] platforms. Additionally we assume the classification of the bus traffic on *agreement protocol messages* (in subsequent text referred to as agreement messages) and *data messages* transporting the context of a task between cores. We assign different priorities to these two types of messages and allow preemptions on the bus between them, as is presented in [17]. Current many-core platforms, such as [10] have the facilities which can be instrumented to implement this concept (e.g. virtual channels and message classes). The purpose of message classification will be described later.

2.2 The Kernels

Every core in the system runs its own independent kernel instance. All the kernels are mutually connected and constitute the basic communication facility. The kernel exposes some of its functionalities to the applications so that they can communicate with the application instances on the other cores. The kernel instances perform local, on-core scheduling and give higher priority to protocol-related OS operations such as sending/receiving agreement/data messages, and computations related to the agreement protocol, than to the task execution which is preemptable at any point.

2.3 The Applications

The applications periodically or sporadically generate a sequence of jobs. The execution of a job has to complete on the core where it started, but the execution of the next job can commence on some other core (task-level migrations). Every application a can execute only on a subset of cores. On each of those cores, the execution code exists, constituting an entity called dispatcher d. During one

protocol instance, the dispatchers of the same application, each located on a different core, communicate between themselves and agree on the location where the next job will be executed. The dispatcher that performs the execution at current time instance is called *master dispatcher* - $\overset{\bullet}{d}$ and at any given point in time an application has exactly one master. It is also responsible for initiating the protocol once a job execution on its core is completed. The other dispatchers are considered as *slave dispatchers* - $\overset{o}{d}$ and they only participate in the protocol execution, until they are elected master, at which point in time the former master becomes a slave dispatcher. The example in Figure 2 follows the aforementioned convention.

After a job execution is completed, a master initiates the protocol; i.e. it communicates with the slaves in order to select new master. The messages exchanged by the dispatchers during this stage are called agreement messages. They are light in size and have a high priority on the mesh. The new master is selected by some criteria (least utilised core, the core with the lowest temperature, the least used core, etc.). The actual policy is immaterial for the discussion in this paper, but might be centred around e.g. load balancing or likelihood of successful execution. If the newly selected master is also the current one, then no further activities are required and the protocol stops until the master executes the next job and starts the protocol again. In the other case, when the newly selected master is different from the current one, the migration occurs. It is performed in the following way:

The current master sends the execution context to the new master. The context is the state of a task used during the execution of the next job. Its size depends on the application and may range from minimal to rather large contexts (e.g. streaming applications). For that reason the messages carrying the context are classified as data messages, have lower priority on the mesh and therefore can be preempted by agreement messages as described in [17]. The preemption points are on the granularity of one packet - the size of the agreement message m_c. Therefore the agreement message can be blocked on the mesh by data message m_d^a, belonging to some application a, for at most one packet traversal time. Similarly, the OS instructions related to the agreements messages (sending/receiving of the agreement message l_s^a, l_r^a or protocol related computation l_c) are non-preemptable, serviced in fifo order and have higher priority than the OS instructions related to the data messages (sending/receiving of the data l_s^d, l_r^d) which are preemptable at any point.

The selection of the data for the context transfer can be the responsibility of either the kernels or the applications themselves. In the first case the local kernel extracts the context from the data section of the current master process and sends it to remote kernel which stores it into the data section of the next master process. In the latter case the programmers are responsible for classifying the variables on those which are shared among all the dispatchers and those which are purely local.

In this work, we assume no dispatcher failures, cross-applications independence, the protocol is always of higher priority than the task execution, which is preemptable at any point. In addition to previously described, we use the following variables:
- a_d - the number of dispatchers belonging to application a
- $a(d)$ - the application to which dispatcher d belongs
- $c(d)$ - the core on which dispatcher d is located
- $p(d)$ - the on-core protocol overhead of dispatcher d
- $\widehat{m_a}$ - the maximum number of the agreement messages exchanged by the dispatchers of the application a during one protocol execution
- r_s - the latency of the router to switch to new port
- r_t - the latency of the router to transfer the packet

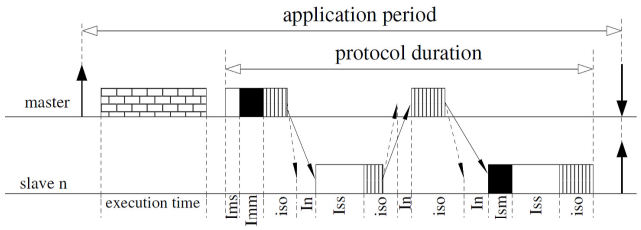

Figure 3: Master-slave protocol

Algorithm 1 MS(a) The execution algorithm of the protocol MS

Input: a
1: $\overset{\bullet}{d}.broadcast()$
2: **repeat**
3: $\quad wait()$
4: **until** $(received == a_d)$
5: $d = \overset{\bullet}{d}.calculate\_next()$
6: **if** $d! = \overset{\bullet}{d}$ **then**
7: $\quad \overset{\bullet}{d}.send\_context(d)$
8: **end if**

- w - mesh width
- $T(d)$ - the minimum inter-arrival time of the application to which dispatcher d belongs
- $C(d)$ - the execution time of one job of the application to which dispatcher d belongs
- $n_h(i)$ - the number of hops the message i takes when traversing from the source to the destination
- $\widehat{n_h(d)}$ - the maximum distance in hops between the dispatcher d and any other dispatcher of the same application

3. THE PROTOCOLS

3.1 Master-slave - MS

3.1.1 Protocol description

For easier comprehension, the protocol is illustrated with Algorithm 1. Firstly, the master dispatcher requests the statuses from all the slave dispatchers (the statuses can be system parameters such as current cpu utilisation, temperature, hardware characteristics, line 1). Every slave responds with its current status. Once the master receives all the replies (line 4) it selects the dispatcher which will execute the next job and therefore become the new master (line 5). Then, if the master is changing, the migration occurs, so the current execution context has to be transferred to the new master (lines 6-8). The aim is to calculate the worst-case protocol duration - *WCPD*.

The WCPD consists of several components, as depicted in Figure 3. It gives a graphical representation of one possible scenario, with the main objective to recognise and emphasize all the delay components. Note that in some other example the components might appear in different order, while some of them might even not exist. The first one is the execution of the protocol by the application of interest in the isolation (without any interference from the other applications) and we denote it by iso. Additionally, the master dispatcher can suffer an on-core interference caused by the other master and slave dispatchers while participating in their own protocols. These delays we denote by I_{mm} and I_{ms} respectively. Furthermore, the slave dispatchers can also suffer an on-core interference caused by the other on-core dispatchers and we recognise these interferences as I_{sm} and I_{ss}. Finally, the application of interest can suffer the interference from all the other applications within the network, noted down as I_n.

The total number of the agreement messages exchanged by the dispatchers of the application a can be calculated as the sum of the broadcast messages from the master to all the slaves and their respective responses:

$$\widehat{m_a} = 2(a_d - 1) \tag{1}$$

The master is involved in every communication step and therefore suffers the overhead of sending and receiving $a_d - 1$ messages and performs a single computation. The slaves communicate with the master (1 receive, 1 send) and perform 1 computation. It is important to emphasize that at this stage only the agreement messages are under analysis. Consequently, the protocol execution on the master core and on the slave cores causes the following overheads:

$$\overset{\bullet}{p(d)} = (a_d - 1)l_s^a + (a_d - 1)l_r^a + l_c \tag{2}$$

$$\overset{o}{p(d)} = l_s^a + l_r^a + l_c \tag{3}$$

Since all the slave dispatchers process their messages in parallel, it is sufficient to recognise only the one which causes the greatest delay, while safely assuming that all the others have already finished their processing before.

Given this, the protocol latency when run in isolation - *iso* can be described as the sum of several terms (Equation 4). The first and the second are the protocol overheads on the master and only one slave core along with the overheads of sending and receiving the context, recognised as *master delay* and *slave delay*. The traversal of the messages that two aforementioned dispatchers exchange (the status request from the master and the response from the slave) is described with the *protocol delay*. Since the master is sending out a number of requests in rapid succession, we need to conservatively assume that the slave we consider is receiving the message last and that the reply is equally received last, that is, these two messages can be blocked within the network by all the other $\widehat{m_a} - 2$ messages of the same protocol. That delay is described with the *interference delay*. In order to analyse the worst-case, in this and all the subsequent protocols it is assumed that the context transfer is needed (i.e. the migration always occurs). The overhead of performing said operation is represented with the *context transfer delay*. Additional safe assumption is that the slave of interest is located the furthest away from the master, when compared to all the other slaves belonging to that application. Therefore, the traversal of the messages that the slave of interest and the master exchange is described with the term $\widehat{n_h(\overset{\bullet}{d})}$.

$$\begin{aligned} iso \quad = \quad & \overbrace{\overset{\bullet}{p(d)} + l_s^d}^{\text{master delay}} + \overbrace{\overset{o}{p(d)} + l_r^d}^{\text{slave delay}} + \overbrace{\widehat{2n_h(\overset{\bullet}{d})} \left\lceil \frac{m_c}{w} \right\rceil (r_s + r_t)}^{\text{protocol delay}} + \\ & \overbrace{(\widehat{m_a} - 2) \left\lceil \frac{m_c}{w} \right\rceil (r_s + r_t)}^{\text{interference delay}} + \overbrace{\widehat{n_h(\overset{\bullet}{d})} \left\lceil \frac{m_d}{w} \right\rceil (r_s + r_t)}^{\text{context transfer delay}} \end{aligned} \tag{4}$$

Furthermore, a master and the slaves may suffer the on-core interference caused by the masters and the slaves of the other applications, so these values also contribute to the WCPD.

In order to calculate the interference a master dispatcher of interest $\overset{\bullet}{d}$ suffers from the other on-core master dispatcher $\overset{\bullet}{d'}$ within the time interval t, we firstly compute the maximum number of protocol executions a master dispatcher $\overset{\bullet}{d'}$ can perform during the interval t.

THEOREM 3.1. *The number of protocol executions of any application within the time interval t can be at most* $1 + \left\lceil \frac{t - C(d)}{T(d)} \right\rceil$

PROOF. The theorem is proven by contradiction. Let us assume that $2 + \left\lceil \frac{t - C(d)}{T(d)} \right\rceil$ protocol executions occurred within the time interval t. There are $\left\lceil \frac{t - C(d)}{T(d)} \right\rceil$ protocol executions surrounded by the first and the last and we refer to them as to *inner executions*. All the inner executions contribute to t with their entire application period and therefore require time interval of at least $\left\lceil \frac{t - C(d)}{T(d)} \right\rceil \times T(d)$ where only these can execute. Additionally, let us assume that ϵ is infinitesimally low but finite value representing the shortest possible duration of the protocol and that the first protocol execution with the duration of ϵ was delayed as much as possible and hence completed just before the interval of the inner executions started. Finally, the last protocol execution could not start before its application execution time $C(d)$ expires.

$$\epsilon + \left\lceil \frac{t - C(d)}{T(d)} \right\rceil \times T(d) + C(d) \geq \epsilon + \left(\frac{t - C(d)}{T(d)} \right) T(d) + C(d) = \epsilon + t \leq t \quad \square$$

The calculated value (see Theorem 3.1) is then multiplied by the overhead of a single protocol execution. Additionally, due to the higher priority that the protocol-related OS operations have over data-related, the master dispatcher of interest can suffer the interference from the context transfer performed by $\overset{\bullet}{d'}$ only once (when it tries to send its own context). The final value is represented with Equation 5.

$$I_{im}(t) = \left(1 + \left\lceil \frac{t - C(\overset{\bullet}{d'})}{T(\overset{\bullet}{d'})} \right\rceil \right) p(\overset{\bullet}{d'}) + l_s^d \tag{5}$$

If we elevate the reasoning presented in Equation 5 from the single on-core master to all the on-core masters, the interference a master $\overset{\bullet}{d}$ suffers can be calculated as:

$$\begin{aligned} I_{mm}(t) = & \sum_{\forall \overset{\bullet}{d'} \in c(\overset{\bullet}{d}) \wedge \overset{\bullet}{d'} \neq \overset{\bullet}{d}} I_{im}(t) = \\ & \sum_{\forall \overset{\bullet}{d'} \in c(\overset{\bullet}{d}) \wedge \overset{\bullet}{d'} \neq \overset{\bullet}{d}} \left(\left(1 + \left\lceil \frac{t - C(\overset{\bullet}{d'})}{T(\overset{\bullet}{d'})} \right\rceil \right) p(\overset{\bullet}{d'}) + l_s^d \right) \end{aligned} \tag{6}$$

The same logic applies for the interference caused by the on-core slaves $\overset{o}{d'}$ to $\overset{\bullet}{d}$, where I_{is} stands for the interference caused by an individual on-core slave:

$$I_{ms}(t) = \sum_{\forall \overset{o}{d'} \in c(\overset{\bullet}{d})} I_{is}(t) = \sum_{\forall \overset{o}{d'} \in c(\overset{\bullet}{d})} \left(\left(1 + \left\lceil \frac{t - C(\overset{o}{d'})}{T(\overset{o}{d'})} \right\rceil \right) p(\overset{o}{d'}) + l_r^d \right) \tag{7}$$

Similarly, for the slave dispatcher of interest $\overset{o}{d}$ we define the interference it suffers from the masters $\overset{\bullet}{d'}$ and the slaves $\overset{o}{d'}$ located on its core.

$$I_{sm}(t) = \sum_{\forall \overset{\bullet}{d'} \in c(\overset{o}{d})} \left(\left(1 + \left\lceil \frac{t - C(\overset{\bullet}{d'})}{T(\overset{\bullet}{d'})} \right\rceil \right) p(\overset{\bullet}{d'}) + l_s^d \right) \tag{8}$$

$$I_{ss}(t) = \sum_{\forall \overset{o}{d'} \in c(\overset{o}{d}) \wedge \overset{o}{d'} \neq \overset{o}{d}} \left(\left(1 + \left\lceil \frac{t - C(\overset{o}{d'})}{T(\overset{o}{d'})} \right\rceil \right) p(\overset{o}{d'}) + l_r^d \right) \tag{9}$$

Additionally, every existing application a' can cause the interference to the application of interest a within the network. We calculate the network interference delay in a very naïve and simplistic way - Equation 10, i.e. by assuming that every application can cause the interference.

$$\begin{aligned} I_n(t) \quad = \quad & \overbrace{\sum_{a' \neq a} \left(1 + \left\lceil \frac{t - C(a')}{T(a')} \right\rceil \right) \widehat{m_{a'}} \left\lceil \frac{m_c}{w} \right\rceil (r_s + r_t)}^{\text{agreement messages}} + \\ & \overbrace{\sum_{a' \neq a} \left(1 + \left\lceil \frac{t - C(a') - l_s^d - l_r^d}{T(a')} \right\rceil \right) \left\lceil \frac{m_d^{a'}}{w} \right\rceil (r_s + r_t)}^{\text{data messages (contexts)}} \end{aligned} \tag{10}$$

For every existing application, different than the application of interest, we calculate the maximum number of protocol occurrences

during the time interval t (Theorem 3.1) and multiply it by the number of the agreement messages that the dispatchers of that application produce within the time of a single protocol execution. This value represents the maximum number of agreement messages that can utilise the network in the given time and that can belong to all the applications a' different than the application of interest a. Further, we conservatively assume that every one of those messages may block the application of interest. Finally, we calculate the maximum number of context transfers that can happen in a given time interval (the proof is a slight modification of the Theorem 3.1 and is therefore omitted). The same reasoning used for the agreement messages applies here; we assume that the context transfer of every application a' different than the application of interest a may block the context transfer of a within the network and hence causes the interference.

Therefore, the total delay $WCPD$ for the master m and the slave s is represented as the sum of all the aforementioned components:

$$WCPD(m,s) = \overbrace{iso}^{\text{isolation}} + \overbrace{I_{mm}(\text{WCPD}) + I_{ms}(\text{WCPD})}^{\text{master core interference}} + \overbrace{I_{sm}(\text{WCPD}) + I_{ss}(\text{WCPD})}^{\text{slave core interference}} + \overbrace{I_n(\text{WCPD})}^{\text{network}} \quad (11)$$

Due to the space constraints, inefficiency of this protocol and general non-applicability to the real-time domain, which will be discussed in the subsequent section, the process of finding the critical slave s (one that induces the greatest latency) for which the calculation would be performed is of no importance. Additionally, the solutions to the Equations 6-9 require a concrete classification of the on-core dispatchers (i.e. the exact information about which will be assumed as masters and which as slaves), so as to produce the greatest interference. This step of finding that particular setup is also omitted. Still, this protocol is presented because, given its simplicity, it is useful for the introduction of basic analytic parameters and helps the reader to develop an intuition about the assumed model, which will be helpful when reasoning about subsequent, more complex protocols.

3.1.2 Protocol limitations

The decision made on the master core is based on the data received from every individual slave. However, in the moment the master makes a decision, there are no guarantees that the state of the system on all the slave cores is identical as in the moment of their individual observations. One extreme, yet possible scenario occurs when one slave reports very high likelihood of accommodating the next execution on its core (e.g. the core is low utilised). Additionally, many other dispatchers from the same core might have also reported low utilisation during their protocol executions. As a result many of the applications might elect that particular core for the next execution and hence overload it, i.e. the sum of the individual utilisations of the applications might exceed the capacity of the core. We recognise *the race condition* as the greatest flaw of this approach. The protocol performance will receive additional attention in Section 4.

3.2 List

3.2.1 Protocol description

One approach in solving the aforementioned problem would be to change the topology of the connections between the dispatchers of the same application. Forming a linked list of dispatchers, besides reducing the total number of the messages, presents a concept that also excludes parallel processing through atomicity and as such is not prone to race conditions. Every dispatcher communicates only with its predecessor and successor in the list. The behaviour

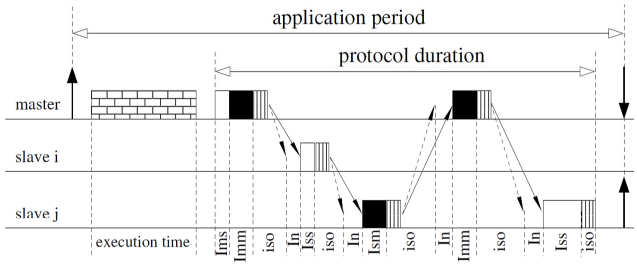

Figure 4: List protocol

Algorithm 2 LIST(a) The execution algorithm of the protocol LIST

Input: a
1: $a.scheduled = false$
2: $d = \overset{\bullet}{d}$
3: **repeat**
4: **if** $(c(d).can\_schedule(a))$ **then**
5: $a.scheduled = true$
6: **if** $(d = \overset{\bullet}{d})$ **then**
7: {master stays the same}
8: **else**
9: $d.request\_context(\overset{\bullet}{d})$
10: **end if**
11: **else**
12: $d = d.next$
13: **end if**
14: **until** $(a.scheduled = true)||(d = null)$

of the protocol is described with the Algorithm 2. When the protocol starts, firstly the master checks whether it can continue the execution and if so, stops the protocol (lines 4-7). In the other case, a master sends the message to the next dispatcher to try to schedule the application (line 12). If one of the slaves can do that, it recognizes itself as the new master and requests the context from the previous master dispatcher (lines 8-9). If during the whole traversal of the list none of the dispatchers can fit the execution on its core, the application is considered as not scheduled.

$$\widehat{m_a} = a_d \quad (12)$$

$$p(\overset{\bullet}{d}) = p(\overset{o}{d}) = l_c + l_s^a + l_r^a \quad (13)$$

The maximum number of the messages is equal to the total number of slave dispatchers $a_d - 1$ (in order to reach the tail of the list all of them have to be traversed) and 1 to request the context from the old master. The protocol overhead a master and all the slaves suffer is the same (1 send, 1 receive and 1 compute operation), however the master and only one of the slaves may additionally have to perform the transfer of the context. As stated in the description of the previous protocol, the worst-case analysis will assume that the migration occurs and therefore will incorporate that overhead into the calculation of the WCPD. Additional conservative assumption we exploit is that only the last dispatcher in the list will announce the possibility to schedule the application and hence the traversal of the entire list is required. For easier comprehension the WCPD is decomposed into several components, as depicted in Figure 4. Similarly to Figure 3, note that it illustrates only one possible scenario and that the ordering of the components is purely example specific.

$$iso = \overbrace{p(\overset{\bullet}{d}) + l_s^d}^{\text{master delay}} + \overbrace{\sum_{\forall \overset{o}{d} \in a} p(\overset{o}{d}) + l_r^d}^{\text{slaves delay}} + \overbrace{\sum_{i=1}^{\widehat{m_a}} n_h(i) \left\lceil \frac{m_c}{w} \right\rceil (r_s + r_t)}^{\text{protocol delay}} + \overbrace{\widehat{n_h(\overset{\bullet}{d})} \left\lceil \frac{m_d}{w} \right\rceil (r_s + r_t)}^{\text{context transfer delay}} \quad (14)$$

Algorithm 3 MMD($\overset{\bullet}{d}, t$) Maximum master delay over period t

Input: $\overset{\bullet}{d}, t$
Output: $delay$
1: $l_{mm} = l_{ms} = \emptyset$
2: **for all** $d \in c(\overset{\bullet}{d})$ **do**
3: **if** $(size(l_{mm}) < \widehat{d_m} - 1)$ **then**
4: ADD(l_{mm}, d)
5: **else**
6: **if** $(I_{im}(t) > min(l_{mm}))$ **then**
7: MOVE($l_{ms}, min(l_{mm})$)
8: ADD(l_{mm}, d)
9: **else**
10: ADD(l_{ms}, d)
11: **end if**
12: **end if**
13: **end for**
14: **for all** $d \in c(\overset{\bullet}{d})$ **do**
15: **if** $(d \in l_{mm})$ **then**
16: $delay+ = I_{im}(t)$
17: **else**
18: $delay+ = I_{is}(t)$
19: **end if**
20: **end for**
21: **RETURN** $delay$

Algorithm 4 MSD($\overset{o}{d}, t$) Maximum slave delay over period t

Input: $\overset{o}{d}, t$
Output: $delay$
1: $l_{mm} = l_{ms} = \emptyset$
2: **for all** $d \in c(\overset{o}{d})$ **do**
3: **if** $(size(l_{sm}) < \widehat{d_m})$ **then**
4: ADD(l_{sm}, d)
5: **else**
6: **if** $(I_{im}(t) > min(l_{sm}))$ **then**
7: MOVE($l_{ss}, min(l_{sm})$)
8: ADD(l_{sm}, d)
9: **else**
10: ADD(l_{ss}, d)
11: **end if**
12: **end if**
13: **end for**
14: **for all** $d \in c(\overset{o}{d})$ **do**
15: **if** $(d \in l_{sm})$ **then**
16: $delay+ = I_{im}(t)$
17: **else**
18: $delay+ = I_{is}(t)$
19: **end if**
20: **end for**
21: **RETURN** $delay$

The protocol overheads of the master and all the slaves contribute to the delay of the execution performed in isolation (see Equation 14). They are described with the terms *master delay* and *slaves delay* respectively and also include the on-core overheads of context transfer. Note that only one slave dispatcher (future master) receives the context. The traversal of the messages within the network is denoted by *protocol delay*. In this protocol the messages are sequentially sent and hence can not mutually interfere. Therefore the term *interference delay* described in the previous protocol does not exist here but at the same time all the messages contribute to the protocol delay with their entire traversal times. The last term accounts for the transmission of the context.

For the on-core interferences $I_{mm}, I_{sm}, I_{ms}, I_{ss}$ and for the network interference I_n the Equations 6-10 hold. Therefore, WCPD can be expressed by the Equation 15, where $\overset{\star}{d}$ presents the next master.

The **previous** and the **next master** can suffer the interference two times (see Figure 4); when performing the protocol routine and when sending/receiving the state. Note that these intervals are not of the same duration. The exact analysis, which requires the consideration of said intervals as well as the distance between them when calculating potential interferences, is cumbersome, computationally demanding and in our opinion unjustifiable approach. On the other hand, treating these intervals independently (i.e. not taking the distance into account) and assuming all the potential interferences that might occur in any of said intervals is very pessimistic and still computationally demanding. We obtain less pessimistic values and save the computation time by considering the entire WCPD as a single interval of interest during which both previous and next master may suffer the interference. The calculation performed for the current master is trivial, however in order to compute the worst-case delay of the future master, the slave which might suffer the greatest interference in the given period has to be recognised and used in further calculations.

$$WCPD(\overset{\bullet}{d}, \overset{\star}{d}) = \overbrace{iso}^{\text{isolation}} + \overbrace{I_{mm}(\text{WCPD}) + I_{ms}(\text{WCPD})}^{\text{master core interference}} + \overbrace{I_n(\text{WCPD})}^{\text{network}} + \overbrace{\sum_{\forall \overset{o}{d} \in a \wedge \neq \overset{\star}{d}} \left(I_{sm}(t') + I_{ss}(t')\right)}^{\text{interference on slave cores } / \overset{\star}{d}} + \overbrace{I_{sm}(\text{WCPD}) + I_{ss}(\text{WCPD})}^{\text{next master core interference}} \quad (15)$$

All the **other slaves** can suffer the interference from all the other dispatchers residing on their respective cores only once - when they are performing their protocol routine (receive the message from the predecessor l_r^a, unsuccessfully try to schedule the application l_c, send the message to the successor in the list l_s^a). Therefore, for every slave the greatest possible interference from the on-core dispatchers (I_{sm}, I_{ss}) within that time interval t' has to be calculated. The relationship between the time t' and the interference suffered within that time is expressed with Equation 8 and Equation 9.

Note that some of the terms that constitute the WCPD are calculated by solving the Equations 6-10, which have a recursive notion and the calculation of the exact values requires an iterative approach. Also note that these Equations assume that for all the on-core, potentially blocking dispatchers it is already known whether they are masters or slaves, which is not true at the beginning of the calculation process. In order to solve this problem we firstly introduce two helper functions $MMD(\overset{\bullet}{d}, t)$ and $MSD(\overset{o}{d}, t)$ presented by Algorithm 3 and Algorithm 4 respectively.

Algorithm 3 calculates the maximum delay a master dispatcher can suffer from the other on-core dispatchers within the time interval t: $MMD(\overset{\bullet}{d}, t) = I_{mm}(t) + I_{ms}(t)$, and similarly Algorithm 4 calculates the maximum delay a slave dispatcher might suffer from the on-core dispatchers within the time interval t: $MSD(\overset{o}{d}, t) = I_{sm}(t) + I_{ss}(t)$. Every dispatcher residing on the core of interest can be either master or slave. The aim is to find the assignment (i.e. which dispatchers should be considered as masters and which as slaves) that will lead towards the greatest possible interference suffered by the master (for MMD) or the slave (for MSD) dispatcher of interest. Both algorithms exploit the strategy of finding the dispatchers that can induce the most protocol-related overhead within observed time and assume them as the masters, while considering all the others as the slaves. Note that the only difference is that MMD already assumes one preselected master - the master of interest, while that is not the case with the MSD.

Firstly, the lists of the on-core masters l_{mm} and the on-core slaves l_{ms} are cleared (line 1). Then, the iterations cover all dispatchers residing on that core. If the number of currently assumed master dispatchers is less than the maximum allowed number of the masters per core $\widehat{d_m}$, the dispatcher is declared as one and added to the list (lines 3, 4, 5). If the master list is full, but the dispatcher,

Algorithm 5 WCPD($\overset{\bullet}{d}$) The worst-case delay of the protocol List

Input: $\overset{\bullet}{d}$
1: $delay = 0$
2: **for all** $\overset{o}{d} \in a(d)$ **do**
3: $t' = l_r^a + l_c + l_s^a$
4: **repeat**
5: $t' = MSD(\overset{o}{d}, t')$
6: **until** balanced(t')
7: **end for**
8: **repeat**
9: $MMD(\overset{\bullet}{d}, delay)$
10: $wc\_slave\_delay = 0$
11: $\overset{\star}{d} = null$
12: **for all** $\overset{o}{d} \in a(d)$ **do**
13: **if** $(MSD(\overset{o}{d}, delay) > wc\_slave\_delay)$ **then**
14: $wc\_slave\_delay = MSD(\overset{o}{d})$
15: $\overset{\star}{d} = \overset{o}{d}$
16: **end if**
17: **end for**
18: $delay = WCPD(\overset{\bullet}{d}, \overset{\star}{d})$
19: **until** fix_point($delay$)
20: **RETURN** $delay$

acting as a master, can incur the delay greater than the minimum of all the existing masters, it will automatically be assumed as one, causing the master with the minimum delay to be demoted to the slave group (lines 6, 7, 8). Otherwise, the dispatcher is added to the slave group (line 10). Therefore, the calculation of the total delay is a cumulative process and includes the summation of the master delay of all the dispatchers belonging to l_{mm} and the slave delay of the dispatchers belonging to l_{ms} (lines 14-20). The complexity of the algorithm is $O(\widehat{d_c} \times \widehat{d_m} \times log(\widehat{d_m}))$, where $\widehat{d_c}$ stands for the maximum number of the dispatchers per core (the number of iterations performed), while $\widehat{d_m} \times log(\widehat{d_m})$ presents the computational complexity of keeping the master list sorted. Algorithm 4 behaves in very similar way, has the same complexity and does not require further discussion.

Finally, the Algorithm 5 performs the calculation of the WCPD. Firstly, for all the slaves the interference they suffer during their protocol routine $MSD(\overset{o}{d}, t') = I_{sm}(t') + I_{ss}(t')$ is calculated (lines 2-6). Then, the maximum interference a master may locally suffer within the observed time is computed (line 9). In order to find the future master, the maximum interference is calculated for all the slaves and their respective cores (lines 10-17). The aim of this computation is to find the critical slave (one suffering the greatest interference within the observed time) and recognise it as the next master dispatcher, while conservatively assuming that it is positioned at the end of the list. For all the other slaves the interference they suffer from the on-core dispatchers is already calculated (lines 2-6). Then, the total delay is augmented (line 18) and fed back into the calculation of the individual terms. The procedure repeats until the equation reaches the fix point (line 19). The computational complexity of the algorithm is $O(a_d \times C \times n)$, where C stands for the complexity of the algorithm MSD (calculated above), and n corresponds to the number of performed iterations before completion.

3.2.2 Protocol limitations

The execution stops at the moment when one of the traversed dispatchers announces the possibility to perform the execution. In most cases that dispatcher might not be the optimal option, e.g. some other core yet not traversed might have better characteristics. As implicitly stated, the greatest limitation of this protocol is that

Algorithm 6 HYBRID(a) The execution algorithm of the protocol HYBRID

Input: a
1: $\overset{\bullet}{d}.broadcast()$
2: **repeat**
3: $wait()$
4: **until** $(received == a_d)$
5: $list = \overset{\bullet}{d}.calculate\_list()$
6: $a.scheduled = false$
7: $d = list.next()$
8: **repeat**
9: **if** $(c(d).can\_schedule(a))$ **then**
10: $a.scheduled = true$
11: **else**
12: $d = list.next()$
13: **end if**
14: **until** $(a.scheduled)$
15: **if** $(d = \overset{\bullet}{d})$ **then**
16: {master stays the same}
17: **else**
18: $d.request\_context(\overset{\bullet}{d})$
19: **end if**

it always traverses the list in predefined, non-intelligent order and therefore is unable to easily perform any selective scheduling for load balancing, power management or any other purpose. If those issues are of no importance, then this concept presents one of the most suitable options, due to its low complexity. The performance of the protocol is additionally analysed in Section 4.

3.3 MS + LIST = HYBRID

3.3.1 Protocol description

In order to solve the aforementioned problems, a protocol named HYBRID is presented. It combines two already covered approaches with the primary objective of gaining the benefits of their respective positive sides.

The protocol is described with the Algorithm 6. The execution can be divided into two phases. The first one is similar to the MS protocol - the master broadcasts the request for the statuses of all the slave dispatchers and waits for corresponding responses (lines 1-4). Upon receiving all the messages, the master generates the list where all the dispatchers are ordered by the preference and likelihood of accommodating the next execution (line 5). Then, the second phase begins and it is similar to the execution of the List protocol. The dispatchers are being sequentially traversed according to their position in the generated list. The first one which announces the possibility to schedule the application stops the protocol (lines 7-14). We further apply the same conservative assumption used in the List protocol; the last dispatcher in the list is the only one that announces the possibility to schedule the application, and hence the list is entirely traversed.

$$\widehat{m_a} = \overbrace{2(a_d - 1)}^{\widehat{m_a}(ms)} + \overbrace{a_d}^{\widehat{m_a}(list)} \quad (16)$$

$$p(\overset{\bullet}{d}) = (a_d + 1)l_s^a + (a_d + 1)l_r^a + 2l_c \quad (17)$$

$$p(\overset{\triangle}{d}) = 2(l_r^a + l_c + l_s^a) \quad (18)$$

$$p(\overset{o}{d}) = l_r^a + l_c + l_s^a \quad (19)$$

The total number of the messages is equal to the sum of the messages exchanged during the execution of the MS protocol: $\widehat{m_a}(ms)$ and the List protocol: $\widehat{m_a}(list)$. The number of the messages a master and all the slaves exchange is also equal to the sum of the individual terms from both algorithms. Dispatcher $\overset{\triangle}{d}$ represents the one whose first phase (master-slave part of the protocol) executed with the greatest latency when compared to all the other dispatchers and we refer to it as to the critical slave. Due to the parallel nature

of this process, for all the other slaves it can be assumed that their master-slave part finished before that of critical slave. Therefore, only the messages they exchange during the second phase (list part of the protocol) directly contribute to the delay, while the messages they exchange during the first phase contribute only indirectly (can interfere with the two messages the critical slave exchanges with the master, also in the first phase of the protocol).

The protocol execution when running in isolation is described with Equation 20. Firstly, the overhead of executing the protocol by the master, the critical slave and all the other slaves are recognised as *master delay*, *critical slave delay* and *all slaves* / $\overset{\triangle}{d}$ *delay*, respectively. Then, the latency of the two messages a master and the critical slave exchange during the first phase of the protocol is denoted by *MS delay*. Additionally, all the messages exchanged by the other dispatchers during this phase ($\widehat{m_a}(ms) - 2$) could potentially block the aforementioned two messages within the network and that delay is recognised as *interference delay*. In the second stage the messages are sequentially sent and processed so no further interferences can be caused by the messages of the same protocol. *LIST delay* stands for the latency of their traversal. Finally, the context transfer additionally augments the calculated value.

$$
\begin{aligned}
iso = {} & \overbrace{p(\overset{\bullet}{d})}^{\text{master delay}} + \overbrace{p(\overset{\triangle}{d})}^{\text{critical slave delay}} + \overbrace{\sum_{\forall \overset{o}{d} \in a \wedge \neq \overset{\triangle}{d}} p(\overset{o}{d})}^{\text{all slaves / } \overset{\triangle}{d} \text{ delay}} + \\
& \overbrace{2\widehat{n_h(\overset{\bullet}{d})} \left\lceil \frac{m_c}{w} \right\rceil (r_s + r_t)}^{\text{MS delay}} + \overbrace{(\widehat{m_a}(ms) - 2) \left\lceil \frac{m_c}{w} \right\rceil (r_s + r_t)}^{\text{interference delay}} + \\
& \overbrace{\sum_{i=1}^{\widehat{m_a}(list)} n_h(i) \left\lceil \frac{m_c}{w} \right\rceil (r_s + r_t)}^{\text{LIST delay}} + \overbrace{l_s^d + l_r^d + \widehat{n_h(\overset{\bullet}{d})} \left\lceil \frac{m_d}{w} \right\rceil (r_s + r_t)}^{\text{context transfer delay}}
\end{aligned}
\tag{20}
$$

By $\overset{\star}{d}$ we denote the future master. Although it is of no importance for the calculation of *iso*, it has a huge impact on the WCPD and has to be recognised. There are several additional factors which also contribute, namely the interferences a master and the slaves suffer from the other on-core master and slave dispatchers ($I_{mm}, I_{sm}, I_{ms}, I_{ss}$). Additionally, all the applications may block the messages of the application of interest within the network (I_n). For all of these terms the Equations 6-10 hold.

In order to calculate the interferences the **current master** $\overset{\bullet}{d}$ and the **next master** $\overset{\star}{d}$ may suffer we apply the same reasoning as for the List protocol; assume that the entire WCPD is an interval where the interferences might occur.

The **critical slave** $\overset{\triangle}{d}$ may suffer the interference at most twice (once during both phases of the protocol). Since the exact analysis is computationally demanding and the intervals are of the same length, we take an slightly modified approach: 1) calculate the interference during one interval and double it so as to assume two independent intervals of equal length, 2) calculate the interference during entire WCPD treating it as a single interval 3) take the minimum of those two, which represents a less pessimistic value.

All the **other slaves** may suffer the interference only once (during the second phase of the protocol), since the interference they suffer in the first phase of the protocol is already incorporated in the interference the critical slave suffers. The term t' has the same meaning as in the List protocol.

The solution to the Equation 21 requires an iterative approach. The calculation steps are described with the Algorithm 7. Firstly, for every slave the minimum interval t' and the interference suffered during that time are calculated (lines 2-7). Then, the delay a master suffers during the observed time interval is computed (line 9). The next master is found such that it causes the maximum possible delay within observed time (lines 10-17). Then, the critical slave is recognised (lines 18-25). For all the other slaves the maximum interference is already computed (lines 2-7). Finally, the calculation of the WCPD is performed with the selected dispatchers and their respective roles (current master, next master and critical slave, line 26). Note that in order to elaborate the scenario which causes the greatest delay, it is required to assume that the next master and the critical slave are not the same dispatcher, although in the actual execution it may happen. The process repeats until WCPD reaches fix point (line 27). The computational complexity is twice of that for List protocol.

Algorithm 7 WCPD($\overset{\bullet}{d}$) The worst-case delay of the protocol Hybrid

```
Input: d•
1:  delay = 0
2:  for all d° ∈ a(d) do
3:      t' = l_r^a + l_c + l_s^a
4:      repeat
5:          t' = MSD(d°, t')
6:      until balanced(t')
7:  end for
8:  repeat
9:      MMD(d•, delay)
10:     next_master_delay = 0
11:     d⋆ = null
12:     for all d° ∈ a(d) do
13:         if (MSD(d°, delay) > next_master_delay) then
14:             next_master_delay = MSD(d°)
15:             d⋆ = d°
16:         end if
17:     end for
18:     crit_sl_delay = 0
19:     d△ = null
20:     for all d° ∈ a(d) ∧ ≠ d⋆ do
21:         if (min{2 × MSD(d°, t'), MSD(d°, delay)} > crit_sl_delay)
            then
22:             crit_sl_delay = min{2 × MSD(d°, t'), MSD(d°, delay)}
23:             d△ = d°
24:         end if
25:     end for
26:     delay = WCPD(d•, d⋆, d△)
27: until fix_point(delay)
28: RETURN delay
```

$$
\begin{aligned}
WCPD(\overset{\bullet}{d}, \overset{\triangle}{d}, \overset{\star}{d}) = {} & \overbrace{iso}^{\text{isolation}} + \overbrace{I_{mm}(\text{WCPD}) + I_{ms}(\text{WCPD})}^{\text{master core interference}} + \overbrace{I_n(\text{WCPD})}^{\text{network}} + \\
& \overbrace{min\left\{2\left(I_{sm}(t') + I_{ss}(t')\right), \left(I_{sm}(\text{WCPD}) + I_{ss}(\text{WCPD})\right)\right\}}^{\text{latest slave } \overset{\triangle}{d} \text{ core interference}} + \\
& \overbrace{I_{sm}(\text{WCPD}) + I_{ss}(\text{WCPD})}^{\text{next master } \overset{\star}{d} \text{ core interference}} + \overbrace{\sum_{\forall \overset{o}{d} \in a \wedge \neq \overset{\star}{d} \wedge \neq \overset{\triangle}{d}} \left(I_{sm}(t') + I_{ss}(t')\right)}^{\text{slave cores interference / } \{\overset{\triangle}{d}, \overset{\star}{d}\}}
\end{aligned}
\tag{21}
$$

3.3.2 Protocol limitations

During the master-slave part of the protocol, all the dispatchers are queried for their current statuses, while during the list part they sequentially try to schedule the application. The strategy of the protocol is to firstly attempt to assign the execution to those dispatchers which reported the best possible environment for the

accommodation of the next execution. Due to its optimistic nature, when compared to the other protocols, this one has higher probabilities of completing before the WCPD, but at the expense of greater number of messages. In fact, since the most suitable candidates are checked firstly, it is reasonable to expect that they will be able to accommodate the execution and therefore stop the protocol at early stages. Note that the race condition still exists but its effect is mitigated. The behaviour of the protocol is the focus of subsequent section.

4. EVALUATIONS

The experiments were performed on the extended version of the simulator *SPARTS* [16]. The 2D-mesh characteristics have been chosen to be equivalent to those available for SCC [10], while for OS operations we assume latencies valid for present micro-kernels (OS calls for accessing the local router to send/receive the message). The aim is to observe the relations between the predicted and the measured WCPD for different protocols under a given workload. Additionally, by varying the number of dispatchers we investigate how this protocol parameter and the amount of traffic influence aforementioned relations and affect the overall protocol behaviour. We allow the mapping of the dispatchers to the cores to be a random process, since dispatcher placement is not in the scope of this paper. We test three presented protocols, each of them with synchronous and asynchronous application releases. In the former case, the idea is to trigger and observe the state where all the applications in the system try to communicate at the same time (i.e. to generate significant network contention), while the latter models a more realistic scenario.

WCET of protocol-related OS operations ($l_s^a, l_r^a, l_c, l_s^d, l_r^d$)	**100.000 cycles**
Router switch time (r_s)	**1 cycle**
Router transfer time (r_t)	**3 cycles**
2D mesh width (w)	**16 bytes**
Agreement message size (m_c)	**4 bytes**
Data message size (m_d^a)	**1024 bytes**

4.1 Observing the pessimism

We simulate the execution of 200 applications on a 10×10 platform. The applications are represented with 5 dispatchers each, have the utilisation of 25% and constitute the workload with the overall system utilisation of 50%.

In Figure 5 the horizontal axis represents the measured WCPD, expressed as the fraction of the analytical (theoretical) WCPD. The vertical axis stands for the amount of the applications which fall into given category (certain ratio between observed and calculated WCPD), expressed as the percentage of the total application-set size.

Since the **MS protocol** always generates the constant amount of messages, it induces the least amount of pessimism. As a consequence, measured WCPD represents greater fraction of calculated WCPD than in other protocols.

As expected, the **List protocol** overestimates the number of messages (always assumes the traversal of the entire list while in real cases it does not occur often) and hence induces greater pessimism.

The **Hybrid protocol** consists of the messages of the both aforementioned approaches. Since the messages exchanged in the first phase of the protocol follow the logic explained for the MS protocol (are constant), and since they constitute $2/3$ of the maximum number of messages, it is reasonable to expect that the pessimism induced by the Hybrid protocol will place it in between of two aforementioned approaches. However, the fact that the Hybrid approach sorts the dispatchers and traverses them according to desired criteria has a significant impact. As a consequence, the number of visited dispatchers is small and hence the protocol is efficient,

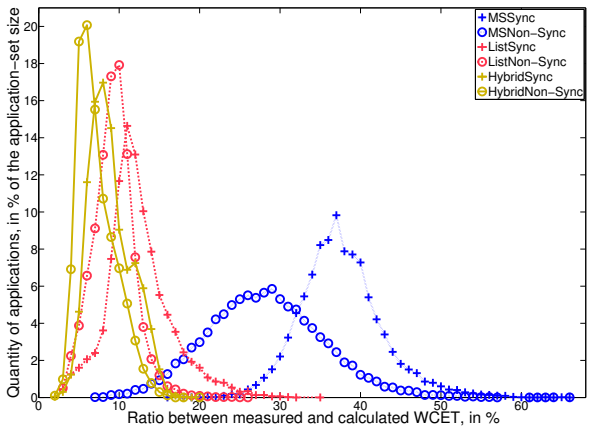

Figure 5: Distribution of pessimism

which results with low execution times and high pessimism. On the other hand, the List protocol pays the price of non-intelligent pre-determined traversing order, causing many long routes which eventually result with longer execution times and lower level of pessimism. Due to aforementioned facts, the amount of pessimism is greater in the Hybrid than in the List protocol.

As expected it holds for all the protocols that the **synchronous releases** create additional overhead as a result of the extensive amount of traffic generated in short periods of time and hence cause longer execution times and less pessimism.

4.2 Parameter variations

The aim is to observe how parameter changes influence the behaviour of the protocols. The simulation inputs are equal to ones used in the previous experiment, with the only difference that the number of the dispatchers that form an application is not constant and ranges from 2 to 15.

In Figure 6 and Figure 7 the horizontal axis represents the number of dispatchers per application. The vertical axis in Figure 6 stands for the measured WCPD, expressed as the percentage of the analytic WCPD, while in Figure 7 the absolute values of the aforementioned terms are presented.

The **MS protocol** with non-synchronised releases shows approximately constant level of pessimism on the whole domain. The increase of the pessimism in the beginning of the graph is explained with low saturated network and the overestimation of the network congestion. As the number of the dispatchers and hence messages increase, the pessimism slowly starts decreasing. Very similar reasoning applies for MS protocol with synchronised releases; fewer messages and simultaneous protocol executions cause low amount of pessimism. Until certain point, the network successfully copes with the increased amount of the dispatchers and the messages, hence causing the raise of the pessimism. Near the end of the graph, the traffic congestion becomes more significant and similar trend of slight pessimism decrease is noticeable.

As expected, both the **List protocol** runs (with and without synchronous releases) show the decrease of the pessimism when the number of the dispatchers increases. The explanation is that in many cases the dispatchers placed near the end of the list are not traversed, while the analysis always assumes the traversal of the entire list. Figure 7 demonstrates that on the most of the domain additional dispatchers cause a barely noticeable increase of the WCPD, confirming previous statement that the dispatchers positioned in the list far from the master are in most of the cases not used. However, after a certain point, the protocol starts to pay the price of predetermined, non-intelligent traversing, hence causing long routes as a re-

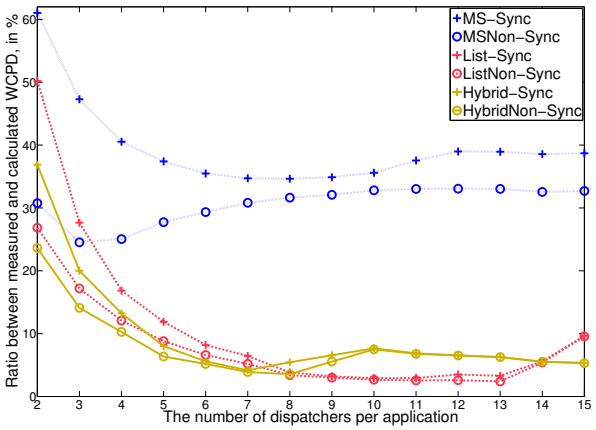

Figure 6: The impact of the number of the dispatchers on the protocol delay

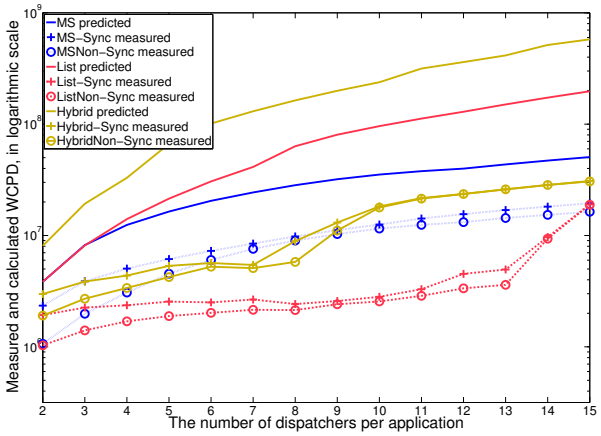

Figure 7: The impact of the number of the dispatchers on the protocol delay

sult of frequent visits to the dispatchers which can't accommodate the execution. Therefore, significant decrease of the pessimism near the end of the graph is visible, leading to a counter-intuitive conclusion that this protocol does not scale when the number of dispatchers is more than a dozen.

Finally, when compared to the List, the **Hybrid protocol** shows similar behaviour and due to the safe assumption of the entire list traversal causes steady decrease in the pessimism as the number of the dispatchers increases. However, after a certain point, the Hybrid protocol pays the price of the extensive communication, causes network congestion and shows steady but only temporary increase in WCPD in the middle of the graph. One surprising conclusion drawn from the Figure 7 is that there exists an interval (between 3 and 8 dispatchers) where the Hybrid protocol has a shorter WCPD than MS protocol, despite the fact that it always induces more messages. The explanation is that Hybrid efficiently selects the next master dispatcher, while MS protocol due to race conditions causes fragmentation and highly loaded cores, where on-core overhead of the communication becomes a predominant factor. Additional surprising fact is that the Hybrid protocol successfully copes with the network congestion by drastically minimising the duration of its second phase (i.e. efficiently finding the next master dispatcher, unlike the List protocol). As is visible in Figure 6 and Figure 7, the Hybrid protocol scales well, shows good average and worst-case performance in both relative and absolute values, but causes high pessimism which is a price paid for more complex analysis.

5. CONCLUSIONS AND FUTURE WORK

In this paper we presented the model for incorporation of many-core platforms into the real-time domain. It is based on the multi-kernel paradigm and utilises message-passing as a communication primitive. We devised several agreement protocols and analytically described their characteristics. Through simulations, we tested said protocols and compared the measured WCPD against analytical predictions, so as to evaluate the pessimism of the analysis. Finally, we gave a head-to-head comparison of all the protocols, where we compared corresponding WCPDs and elaborated on the individual potentials for scalability.

The future work can include further simulations of concrete applications and protocols, in order to observe the deviations between the WCPD and the average cases, but also for the comparison between the measured and theoretically predicted WCPD. The simulations can also be used to give head-to-head comparison of different protocols in terms of efficiency, e.g. when analysing generated traffic, power management or load balancing. The model and the respective protocol overheads can be integrated in the scheduling analysis. Furthermore, new protocols can be devised with the characteristics which fit some particular requirements: strong guarantees, good average-case behaviour, limited communication, dispatcher failure, etc. Finally, the placement of the dispatchers belonging to one application (locality) is of great importance and tighter bounds could be derived with some assumptions which are addressing that issue.

6. REFERENCES

[1] Y. Amir, Y. Kim, C. Nita-Rotaru, and G. Tsudik. On the performance of group key agreement protocols. In *22th ICDCS*, 2002.

[2] 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.

[3] G. Bernat, A. Colin, and S. M. Petters. WCET analysis of probabilistic hard real–time systems. In *24th RTSS*, pages 279–288, Austin, Texas, USA, Dec 3–5 2002.

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

[5] S. Chakravorty and L. Kale. A fault tolerance protocol with fast fault recovery. In *PDPS*, 2007.

[6] X. Chen and S. Chen. Dsbs: Distributed and scalable barrier synchronization in many-core network-on-chips. In *TrustCom*, 2011.

[7] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: a hybrid quorum protocol for byzantine fault tolerance. In *7th OSDI*, 2006.

[8] W. Dally and C. Seitz. The torus routing chip. *Distr. Comput.*, 1986.

[9] N. Eisley, L.-S. Peh, and L. Shang. In-network cache coherence. *J. Comp. Arch. Lett.*, 2006.

[10] Intel. *Single-Chip-Cloud Computer*. http://techresearch.intel.com/ProjectDetails.aspx?Id=1.

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

[12] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 1979.

[13] T. LeBlanc and E. Markatos. Shared memory vs. message passing in shared-memory multiprocessors. In *PDPS*, 1992.

[14] P. Lee, J. Lui, and D. Yau. Distributed collaborative key agreement protocols for dynamic peer groups. In *Int. Conf. Netw. Protocols*, 2002.

[15] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the intel 80-core network-on-a-chip terascale processor. In *Int. Conf. Supercomp.*, 2008.

[16] B. Nikolić, M. A. Awan, and S. M. Petters. SPARTS: Simulator for power aware and real-time systems. In *8th IEEE Int. Conf. Emb. Softw. & Syst.*, Changsha, China, Nov 2011. IEEE.

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

[18] F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *J. Syst. Arch.*, 46:339–355, Feb 2000.

[19] Tilera. *TILEPro64 Processor*. http://www.tilera.com/products/processors/TILEPRO64.

[20] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *J. Solid-State Circ.*, 2008.

[21] O. Villa, G. Palermo, and C. Silvano. Efficiency and scalability of barrier synchronization on noc based many-core architectures. In *CASES*, 2008.