Minimizing Delay in Network Function Virtualization with Shared Pipelines

Ori Rottenstreich, Isaac Keslassy, Yoram Revah and Aviran Kadosh

***Abstract*—Pipelines are widely used to increase throughput in multi-core chips by parallelizing packet processing while relying on virtualization. Typically, each packet type is served by a dedicated pipeline with several cores, each implementing a network service. However, with the increase in the number of packet types and their number of required services, there are not enough cores for pipelines. In this paper, we study *pipeline sharing*, such that a single pipeline can be used to serve several packet types. Pipeline sharing decreases the needed total number of cores, but typically increases pipeline lengths and therefore packet delays. We consider two novel optimization problems of allocating cores between different packet types such that the average or the worst-case delay is minimized. We study the two problems and suggest optimal algorithms that apply under different assumptions on the input. We also present greedy algorithms for the general case. Last, we examine our solutions on synthetic examples as well as on real-life applications and demonstrate that they often achieve close-to-optimal delays.**

***Index Terms*—Multicore Optimization, Network Function Virtualization, Network Processors.**

I. Introduction

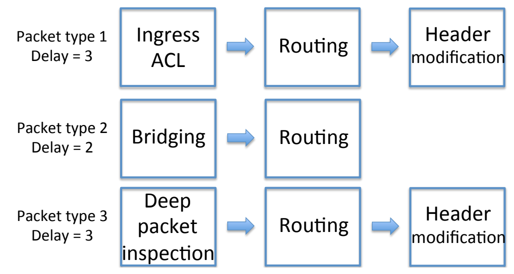
A. Background

This paper introduces the novel problem of *pipeline sharing*, which designers face when implementing the emerging class of dedicated pipeline-based multi-core chips. This group encompasses multi-core network processors [2], [3], and application-specific systems-on-chip (AS-SoC) such as telecommunication applications [4] and high-end multiprocessors [5]. Recently, NFV (Network Function Virtualization) was described as a new networking framework. It suggests to implement various network functions in identical virtual machines by relying on virtualization [6]. Practically, such virtualization must consider restrictions on the number of available cores.

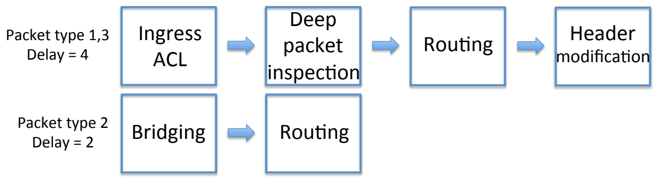
Consider a set of numbered services, and a flow of incoming packets, where each packet may need to go through a different subset of services in an increasing order. Dedicated chips used to be implemented with a single general-purpose core that could provide all the needed services using software-based algorithms. However, such a single software-based core would not be scalable. Therefore, dedicated chips have become implemented as *multi-core* chips, where each core (or engine) is specifically designed to implement a *single* needed service.

This manuscript is an extended version of “Minimizing Delay in Shared Pipelines”, which was presented in IEEE Hot Interconnects ’13, San Jose, CA, USA, August 2013 [1].

O. Rottenstreich is with Princeton University, NJ, USA (email: orir@cs.princeton.edu). I. Keslassy is with VMware, CA, USA and the Technion, Israel (e-mail: isaac@ee.technion.ac.il). Y. Revah and A. Kadosh are with Marvell, Israel (e-mails: {yoramr, aviran}@marvell.com).



(a) Three pipelines with 8 cores (without pipeline sharing). The average delay is $T \approx 2.67$ time slots and the worst-case delay is $D = 3$ time slots.



(b) Example of pipeline sharing: Two pipelines with only 6 cores. The average delay is $T \approx 3.33$ time slots and the worst-case delay is $D = 4$ time slots.

Fig. 1. Illustration of *pipeline sharing*.

Thus, each packet could go through all the corresponding cores to obtain all its needed services.

Letting packets go through their needed cores one after another would result in unpredictable queueing delays, and therefore a lack of performance guarantees. Hence, an appealing solution is to use *pipelines*. Assume that packets are divided into k different *packet types*, where each packet type needs a specific set of *services*. Then, the chip can be implemented using exactly k pipelines, where the pipeline that corresponds to each packet type includes the cores that implement its needed services. Incoming packets are simply forwarded to their appropriate pipelines. If at most one packet arrives every time slot, and each core processing takes one time slot, then it is guaranteed that each packet will be done with processing in the minimal needed time, without any potential conflict on its processing path.

For instance, Fig. 1(a) illustrates a simplified example of packet-processing chip. It accepts $k = 3$ packet types that are respectively served by $k = 3$ dedicated pipelines, with a total number of 8 cores. When a packet arrives it is assigned to a pipeline that serves its type and is first served by the first core on the pipeline. In each time slot, each core serves the packet that was served in the previous core in the previous time slot, if there was such a packet. The service of a packet is completed after a number of time slots that equals the number of cores in its pipeline. If the k packet types are uniformly distributed, the obtained average delay is $(3+2+3)/3 \approx 2.67$ time slots. Likewise, the worst-case (maximal) delay, is 3.

However, following the increase in (a) the number of packet types, and (b) the pipeline lengths, the number of needed cores does not fit multi-core chips anymore. Therefore, we need to rely on *pipeline sharing*, such that different packet types may need to go through the same pipeline. As a result, a pipeline may include more services than needed by a packet. When a packet encounters a core that it does not need, it simply does not use it, *but still spends time in it in order not to break the pipeline*. Therefore, while pipeline sharing decreases the needed number of cores, it can also increase the packet delay.

Fig. 1(b) illustrates this pipeline sharing. The first and the third packet types share the first pipeline, which includes the 4 cores required by at least one of these types. Hence, we only need 6 cores instead of 8. On the other hand, since these two types now require a larger delay of 4 time slots to go through their shared pipeline, the average delay increases to $(4+2+4)/3 \approx 3.33$ and the worst-case delay is now 4 instead of 3.

For a minimal implementation cost, the design assumes a very simple control logic which basically maps an incoming packet to the corresponding pipeline. Such a simple control logic does not support skipping pipeline steps, early pipeline termination or allowing packets to cross several pipelines.

There is a clear tradeoff between the number of needed cores and the obtained average and worst-case packet delays. This tradeoff creates capacity regions in the design exploration, in the sense that the minimal possible delay is described for a given upper bound on the number of cores. A designer cannot go below these optimal bounds. This is illustrated in Fig. 2. For instance, with at most 5 cores a single pipeline must serve the three packet types with 5 distinct services and result in an average delay and worst-case delay that both equal 5. Likewise, allowing the existence of a seventh core cannot improve the optimal delays that can be obtained with at most 6 cores. The goal of this paper is to further analyze these optimality bounds.

In this paper we introduce *pipeline sharing* in multi-core chips. A limited number of cores should be divided into pipelines, each serving several packet types. To minimize the delay of a packet we try reduce the lengths of these pipelines.

B. Related Work

Several past works have considered the problem of mapping the pipelines onto the chip cores in order to reduce energy consumption and bandwidth utilization [7]–[9]. These mapping issues are outside the scope of the paper, and the mapping solutions are complementary to our suggested algorithms, in the sense that they can be applied on the resulting pipelines.

Pipeline scheduling was discussed in [10]. This work deals with a simpler problem in which the hardware components are not configurable and each of them can perform a single predetermined task. For instance, given a formula with several additions and multiplications, they study how to schedule simultaneously several pipelines calculating the formula. They try to guarantee that in every time unit a single adder and a single multiplier are not used by more than a single pipeline.

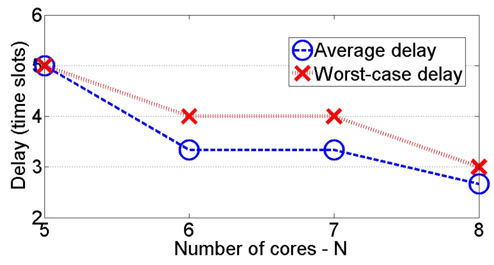


Fig. 2. Tradeoff between the number of cores and the average or worst-case delays, defining capacity regions for the two optimization problems. For a limited number of cores, the minimal possible delays are presented. A designer cannot go below these optimal bounds.

The design of multicore systems was discussed for a wider range of environments besides pipelines. [11] analyzed the number of cores that should implement each task based on its utilization. [12] studied the obtained system throughput as a function of the partition granularity of the different requirements into small tasks. Finer granularity can improve the performance but also complicates the mapping process.

In addition, network processors (NPs) may adopt additional alternative architectural models. For instance, the multipass NP architectural model is suggested in [13] as an alternative to pipelining. While such a model may have less synchronization issues than pipeline-based models, it may also provide fewer guarantees of service. A multi-queued NP architecture was suggested in [14] to serve packets with heterogeneous requirements. They study a policy that may omit a packet upon the arrival of a second packet with simpler requirements.

Our problem can also be viewed in the context of recent ongoing research on NFV, including function assignment optimization [15]. Consequences from our work can be useful also in this framework, e.g. by deciding how many cores should implement every function or what flows to serve by the same pipeline.

II. Model and Problem Definition

We start by introducing notations and formally defining the problems discussed in this paper.

A. Traffic

We consider a system where each packet needs to perform a set of required tasks among r possible tasks, $\{1, \cdots, r\}$. Each task is performed exactly once for each packet, in an increasing index order. For instance, out of $r = 10$ possible tasks, a packet may need to perform tasks $\{1, 2, 7\}$. It will successively run tasks 1, then 2, and finally 7.

We further assume that there are k types of packets. Each incoming packet has a probability $p_i > 0$ of belonging to type i, and packets of type i needs to perform a fixed set of tasks S_i. For instance, an incoming packet may either be of type 1 with probability $p_1 = 0.6$ and require tasks $S_1 = \{1, 2, 3\}$; or it may be of type 2 with probability $p_2 = 0.4$ and require tasks $S_2 = \{1, 4\}$.

Our analysis and solution are not sensitive to whether each packet of type i appears with probability p_i independently of its preceding packets or whether there are some dependencies between subsequent packets (e.g., the existence of long flows with several packets of the same type) that yield a stationary distribution with these probabilities.

We restrict the model to consider tasks that can be ordered as above. Clearly, in some systems two tasks might be required in different orders by two packets and our study does not cover them. Notice that it is impossible to share two pipelines, although they include the exact set of tasks if the required orders of one or more pairs of tasks are different. We take advantage of this restriction to enable improved performance by being able to share more pipelines. Indeed, this possibility to order tasks is common for different networking systems. A recent study [16] surveyed a wide range of middlebox applications. It describes the implementation of each application as an ordered list of processing steps. They name 10 processing steps in a specific order such that each application is associated with a subset of them according to the common order. Earlier, a methodology to profile multiprocessor and network processor applications by representing them as directed acyclic graphs (DAG) was described [17], [18]. In this representation, nodes represent computational tasks and links represent control and data dependencies. Various types might require similar variants of the application with part of the tasks, described as a path in the graph. While the graph might enable degrees of freedom in ordering the tasks, for any such DAG, our model can be applied by selecting an arbitrary legal order. The pipeline sharing performance is not affected by this selection.

Another application can be found in a simplified version of the compilation of programming languages for packet forwarding architectures such as the recently developed popular language P4 [19]. In its compilation, a DAG is used to describe dependencies between logical tables, each implementing a computation task. Assuming equal-size tables of a single matching type, this translation can follow the model while trying to implement these computation tasks by processing cores.

Some applications can benefit from having more than a single possible service path for the same packet type. Our model gives higher priority to the reduction of the delay rather than maintaining the flexibility. Keeping the cores required for that for one flow decreases the core availability for other flows and can enforce unnecessary pipeline sharing with an increased delay.

B. Pipeline Sharing

We are now interested in studying how the set of cores in a multi-core chip can be subdivided into shared pipelines that can service the different packets.

We assume that the chip holds N cores. Each core is an homogeneous general-purpose core that can be configured to serve any task. By relying on virtualization, at run-time, the chip manager configures each of the cores to serve a single task among the set of possible tasks, e.g. by loading it with a different code. Due to limitations on its memory size, the core cannot be configured to serve two or more tasks. The chip manager further subdivides the N cores into d pipelines of multiple separate cores. Let Q_i be the set of tasks served by the i^{th} pipeline.

For example, a chip with $N = 9$ cores could be subdivided into one pipeline of 2 cores that respectively deals with tasks 1 and 2; another pipeline of 3 cores that processes tasks $1, 2, 3$; and another pipeline of 4 cores that deals with tasks $1, 3, 4, 5$. Note that each of the three pipelines holds a separate core that processes task 1, i.e. several cores may be assigned the same task.

In addition, the chip manager assigns a single pipeline to each packet type, such that the pipeline contains all the tasks required by this packet type. For instance, it may assign packets of type 2 that needs the set of tasks $S_2 = \{1, 4\}$ to the last pipeline of cores, which processes tasks $1, 3, 4, 5$. In this case, note that the packets will just go through the cores of tasks 3 and 5 without any processing.

C. Optimization Problems

We model the delay of a pipeline as equal to its length, i.e. its number of cores. In our example, it will take 4 time-slots to go through the pipeline of 4 cores that processes tasks $1, 3, 4, 5$. So if a packet that needs the task set $S_2 = \{1, 4\}$ goes through this pipeline, it will still take 4 time-slots, out of which the two slots for processing tasks 1 and 4 are useful, while the two other slots are simply empty. Early pipeline termination is not allowed and a packet cannot leave the pipeline before reaching its last core due to the limited control of the system.

Thus, *there is a clear tradeoff between the flexibility of a longer pipeline, which can process more packet types, and its higher delay.* Given our set of N cores, our goal is to use this tradeoff in order to reduce the average or the worst-case packet delay.

Formally, we state our first problem as follows: Given N cores, and the k sets of tasks S_i of probability p_i, *our goal is to find shared pipelines that will minimize the average packet delay.* We denote by $T_{OPT}(N)$ this minimal possible average delay.

Likewise, in our second problem, given N cores and the k sets of tasks S_i of probability p_i, *our goal is to find shared pipelines that will minimize the worst-case packet delay*. We denote by $D_{OPT}(N)$ this minimal possible worst-case delay.

Note that we use the term *packet* for the item served in the pipelines. While this is especially relevant for network processors, in other pipeline architectures the more general term of *element* is also very common.

D. General Properties

We describe a property that can be useful for simplifying instances to one of the two optimization problems. Intuitively, if tasks can be divided into different kinds such that every packet type requires tasks of one kind, there is no point to share pipeline with tasks of different kinds. For instance, if a packet type either requires security services (e.g. deep packet inspection, firewall) or alternatively traffic shaping services (e.g., load balancing, network address translation), we can avoid having pipelines that include both security and traffic shaping tasks. The importance of this property is by helping us, later in this paper, to find optimal solutions for families of instances of the problems.

Proposition 1. *Assume that the r tasks can be divided into two categories such that each type requires tasks from one category, i.e. the k packet types can be ordered such that $\left(\bigcup_{i=1}^{m} S_i\right) \cap \left(\bigcup_{i=(m+1)}^{k} S_i\right) = \emptyset$ for some $m \in [1,k]$. Then, for each of the problems, an optimal solution given N cores can be obtained for some value $N_0 \in [0,N]$ as the union of the pipelines in optimal solutions for packet types $[1,m]$ and for types $[(m+1),k]$ with N_0 and $(N - N_0)$ cores, respectively.*

Proof: Consider a pipeline in an optimal solution that serves tasks from both sets $\left(\bigcup_{i=1}^{m} S_i\right), \left(\bigcup_{i=(m+1)}^{k} S_i\right)$. Then, it can be partitioned into two smaller pipelines to reduce the average delay without increasing the worst-case delay. Accordingly, we can distinguish between pipelines in an optimal solution according to the category of tasks they serve. ■

Similarly, the next proposition shows that if the sets of tasks (for the different packet types) served by a pipeline can be divided into two disjoint unions of tasks, then this pipeline can be divided into two shorter pipelines which will result in a shorter average delay and at most the same worst-case delay. The proof is similar to the last one and is omitted for its simplicity.

Proposition 2. *Let $S_{i_1}, \cdots, S_{i_b}$ be the sets of tasks served by one of the pipelines in a solution for one the two problems. If for some $m \in [1,b]$ we have $\left(\bigcup_{j=1}^{m} S_{i_j}\right) \cap \left(\bigcup_{j=(m+1)}^{b} S_{i_j}\right) = \emptyset$ then the pipeline can be divided into two pipelines without increasing the number of cores. This results in a shorter average delay and at most the same worst-case delay.*

III. Minimizing average delay

In this section, we study our first optimization problem of minimizing the average delay. We present an optimal algorithm that applies when the input satisfies a specific property, describe some general properties of the problem, and provide a greedy algorithm for the general case.

A. A simple case of the required tasks: $S_i = [1, Y_i]$

Finding an optimal solution of the problem that minimizes the average delay is difficult in the general case. Here, we consider a property of the sets of tasks required by the different packet types $S_1, \cdots, S_k$ that when satisfied allows us to efficiently calculate an optimal solution that minimizes the average delay.

In some applications all packet types demand several consecutive tasks starting from the first task. The different packet types differ in the number of required tasks in each of them. Then, the set of tasks of packet type i can be presented as $S_i = \{1, \cdots, Y_i\} = [1, Y_i]$ for $Y_i \in [1, r]$.

For instance, this scenario can describe a required packet processing hierarchy for packets with different required levels of security. If an Intrusion Detection System (IDS) service is partitioned into smaller tasks in which disjoint pattern sets are examined, a packet with a higher security level might require to be compared with a larger set of patterns. This can be described as additional tasks required by the packet. In particular, the Snort IDS [20] defines the notion of severity level. It enables customizable prioritization of alerts such that the severity level of 32 predefined alert categories can be modified. Then, the highest priority alerts (with lowest severity levels) are examined by installing corresponding rules of all categories with up to some specific severity level. A task can be described as examining the rules of one level such that different packet types would require running different number of the first tasks.

We show that in such cases an optimal solution has several properties such as specific forms of its pipelines as well as simplicity in the matching of the packet types to one of the pipelines. These properties enable us to suggest an efficient algorithm for finding an optimal solution.

In the rest of the section, we assume that $S_1 = [1, Y_1], \cdots, S_k = [1, Y_k] \subseteq [1, r]$ are ordered such that $Y_a \leq Y_b$ if $a < b$. We then have that $S_1 \subseteq S_2 \subseteq \cdots \subseteq S_{k-1} \subseteq S_k$. Indeed, by merging identical requirements, the set of tasks can be ordered such that $S_1, \cdots, S_k$ satisfy the required condition whenever $S_1 \subseteq S_2 \subseteq \cdots \subseteq S_{k-1} \subseteq S_k$. Let $Q_1, \cdots, Q_d$ be the set of tasks serviced by the cores in the d pipelines in an optimal solution and let $(B_1, \cdots, B_k)$ be a vector indicating the serving pipeline for each packet type s.t. $B_i \in [1, d]$. For short, we denote the range $[1, 1]$ by $[1]$.

The properties are summarized in the following lemmas. The first lemma explains that the cores in a pipeline should be those required by the packet type with the largest set of tasks among the types it serves.

Lemma 1. *The pipelines in an optimal solution $Q_1, \cdots, Q_d$ satisfy for $j \in [1,d]$ $Q_j \in \{S_1, \cdots, S_k\}$, i.e. the sets of tasks in the pipelines in an optimal solution are among the sets of tasks of the different packet types in the input. In particular, for $j \in [1,d]$ Q_j is of the form $Q_j = [1, Z_j]$ for $Z_j \in [1, r]$.*

Proof: Let h be the number of packet types served by pipeline Q_j and let $S_{i_1}, S_{i_2}, \cdots, S_{i_{h-1}}, S_{i_h}$ be the sets of tasks of these packet types such that $i_1 < i_2 < \cdots < i_{h-1} < i_h$. According to the order of $S_1, \cdots, S_k$, we have that $S_{i_1} \subseteq S_{i_2} \subseteq \cdots \subseteq S_{i_{h-1}} \subseteq S_{i_h}$. By the correctness of the solution, we have that $\bigcup_{m=1}^{h} S_{i_m} \subseteq Q_j$. In addition, by the optimality of the solution, an equality must hold i.e. $\bigcup_{m=1}^{h} S_{i_m} = Q_j$. Otherwise, cores could be eliminated to reduce the delay. Since $S_i = [1, Y_i]$ it follows that $Q_j = \bigcup_{m=1}^{h} S_{i_m} = S_{i_h} = [1, Y_{i_h}]$. ■

The next lemma presents a property of the matching of packet types to pipelines in an optimal solution.

Lemma 2. *Assume that $Q_1, \cdots, Q_d$ are ordered such that $Q_1 \subsetneq Q_2 \subsetneq \cdots \subsetneq Q_{d-1} \subsetneq Q_d$. Then, the packet types are served by an increasing order of the pipelines, i.e. $B_i \leq B_j$ for $i < j$. In particular, the packet types served by each pipeline in the solution form a subset of consecutive packet types. In particular, the last packet type is served by the last pipeline, i.e. the kth packet type is served by $Q_{B_k} = Q_d = S_k$.*

Proof: First, such an order of $Q_1, \cdots, Q_d$ exists according

Algorithm 1 Optimal Algorithm for Minimizing the Average Delay for $S_i = [1, Y_i]$ (Pseudo-code)

```
    Input:
1   Packet types with sets of tasks $S_1 = [1, Y_1], \cdots, S_k = [1, Y_k]$ and probabilities $p_1, \cdots p_k$.
2   Initialize: For $n \geq 0$: $T(0,n) = 0$. For $n < 0$: $T(0,n) = \infty$. For $i > 0, n \leq 0$: $T(i,n) = \infty$. For $n \geq 0$: $B(0,n) = ()$ (an empty vector) and $Q(0,n) = \{\}$ (an empty set of pipelines).
3   Main:
4   for $i = 1$ to $k$ do
5       for $n = 0$ to $N$ do
6           Calculate $T(i,n)$ according to Eq. (1)
7           Calculate $Q(i,n)$ according to Eq. (2)
8           Calculate $B(i,n)$ according to Eq. (3)
9   return $Q(k,N)$, $B(k,N)$
```

to Lemma 1. Assume that the claim does not hold for an optimal solution and let i, j be two indices such that $i < j$ and $B_i > B_j$. Then, based on the order of $S_1, \cdots, S_k$ and the correctness of the solution $S_i \subseteq S_j \subseteq Q_{B_j} \subsetneq Q_{B_i}$ and in particular $S_i \subseteq Q_{B_j}$. We can let packet type i be served by Q_{B_j} that satisfies $|Q_{B_j}| < |Q_{B_i}|$ and reduce the average delay of the solution in contradiction to its optimality. In addition, with the last property if $B_i = B_j$ (for $i < j$) then $B_m = B_i = B_j$ for all $m \in [i, j]$ and every pipeline serves a subset of consecutive packet types. ■

Following this lemma, we suggest a dynamic-programming algorithm to find an optimal solution to this special case of the problem. To do so, we suggest several additional definitions. We denote by $T(i,n)$ (for $i \in [0,k]$, $n \in [0,N]$) the minimal possible weighted average delay that can be achieved in serving the first i packet types given at most n cores. In the calculation of this average we consider each of the i packet types with a weight that is equal to its original probability such that for $i < k$ the sum of weights is smaller than one. If no such solution exists we define $T(i,n) = \infty$. We later explain that the optimal average delay $T_{OPT}(N)$ equals $T(k,N)$. Likewise, we denote by $B(i,n)$ (for the same values of i, n whenever $T(i,n) \neq \infty$) the vector of length i that indicates for each packet type the serving pipeline in an optimal solution for the first i packet types with at most n cores. Last, let $Q(i,n)$ denote the list of pipelines in this optimal solution. This list is ordered as assumed in Lemma 2. The next example demonstrates these definitions.

Example 1. *Assume an input of $k = 3, N = 8$ and $(S_1,p_1),(S_2,p_2),(S_3,p_3) = ([1],0.3),([1,3],0.2),([1,5],0.5)$. With at most three cores, for instance, we can serve the first type with a pipeline of a single core and thus $T(1,3) = 0.3$. Likewise, we can serve the first two types by a pipeline of three cores that results in a delay of three for these two types and accordingly, $T(2,3) = (0.3+0.2)\cdot 3 = 1.5$. For this input, an optimal solution with at most $N = 8$ cores is composed of two pipelines $Q(k,N) = Q(3,8) = \{[1],[1,5]\}$ in which the first type is served in the first pipeline in the solution $[1] = \{1\}$ while the second and third type are served by the second pipeline $[1,5] = \{1,2,3,4,5\}$. This is represented by $B(k,N) = B(3,8) = (1,2,2)$. This result in an optimal delay $T(k,N) = T(3,8) = 0.3\cdot 1 + (0.2+0.5)\cdot 5 = 3.8$. To serve the three types given at most 8 cores, we have several options that differ in the number of additional types that are served with the last type in the pipeline of 5 cores. Three cores are left for the other pipelines. This can be either only the last type itself (with probability of 0.5) or the second type together with the third (with probability of $0.2+0.5 = 0.7$) or all the three types (with probability of 1). Accordingly, $T(k,N) = T(3,8) = \min(T(2,3)+0.5\cdot 5, T(1,3)+0.7\cdot 5, T(0,3)+1\cdot 5)$.*

We present recursive formulas for the functions $T(i,n), Q(i,n)$ and $B(i,n)$ based on the idea from the example. For the correctness of the following recursive formulas, we set $T(0,n) = 0$ for $n \geq 0$, $T(0,n) = \infty$ for $n < 0$ and $T(i,n) = \infty$ for $i > 0, n \leq 0$. Likewise, for $n \geq 0$ we set $B(0,n) = ()$ (an empty vector) and $Q(0,n) = \{\}$ (an empty set of pipelines).

The intuition behind the formulas is the following. To calculate $T(i,n)$, we consider an optimal solution for the first i packet types. By Lemma 2, the i^{th} packet type (S_i, p_i) is served by the last pipeline $Q_{B_i} = S_i$ and has a delay of $|S_i|$. Additional packet types may be served by this pipeline. If we denote the total number of packets served by this pipeline by j, then $j \in [1,i]$ and by the same lemma, we must have that these are the packet types with indices $[i-(j-1), i]$. To calculate the minimal delay, we consider the best option for the value of j from the above values. Since $|S_i|$ cores are used by the last pipeline, the first $(i-j)$ packet types can be served by the other $(n-|S_i|)$ available cores. Their minimal possible contribution to the average delay is $T(i-j, n-|S_i|)$ and it can be achieved according to an optimal solution for these parameters. To calculate $Q(i,n)$ and $B(i,n)$ we look at the possible identities of the pipelines in such a solution. If there exists a solution with j packet types served by the last pipeline, we add to the pipelines in an optimal solution of the first $i-j$ packet types, an additional single pipeline of S_i. The vector B is now updated such that the last j packets are served by the last pipeline with an index of $|Q(i,n)|$. Based on the above, we can deduce the following.

Theorem 1. (i) *For $i \in [1,k]$, the variable $T(i,n)$ satisfies*

$$T(i,n) = \min_{j\in[1,i]} \left(T(i-j, n-|S_i|) + \left(\sum_{m=(i-(j-1))}^{i} p_m \right) \cdot |S_i| \right) \tag{1}$$

(ii) *Let j be the minimal value of the corresponding parameter that minimizes the right-hand side of* (i). *Then, there exists an optimal solution that satisfies*

$$Q(i,n) = Q(i-j, n-|S_i|) \cup \{S_i\}, \text{ and} \tag{2}$$

$$B(i,n) = B(i-j, n-|S_i|) \cdot \underbrace{(|Q(i,n)|, \cdots, |Q(i,n)|)}_{j \text{ times}} \tag{3}$$

where $\cdot$ denotes the vector concatenation operation.

Finally, we describe the suggested dynamic-programming algorithm. Its pseudo-code is presented in Algorithm 1. We first initialize $T(0,n) = 0$ for $n \geq 0$, $T(0,n) = \infty$ for $n < 0$, $T(i,n) = \infty$ if $i > 0$ and $n \leq 0$, $B(0,n) = ()$ and $Q(0,n) = \{\}$ for $n \geq 0$ (line 2). We continue to calculate in step i (for $i \in [1,k]$) the values of $T(i,n)$ for $n \in [0,N]$ according to the formulas presented in Theorem 1 based on the values from previous steps (lines 4-8). In the required solution, all the k packet types have to be served with N available cores. Thus the optimal average delay $T_{OPT}(N)$, the pipelines in an optimal solution and the matching of packet types to pipelines are given by $T(k,N), Q(k,N)$ and $B(k,N)$, respectively.

We now discuss the time complexity of the suggested algorithm. The algorithm is composed of k steps, in each step $i \in [1,k]$ we calculate the $(N+1)$ values of optimal delays $T(i,n)$ for $n \in [0,N]$. For each value, by relying on Eq. (1) from Theorem 1, we take the minimal value out of at most k possible values (line 6, according to the maximal value of i in this equation) or perform more simple calculations (lines 7-8). Thus the time complexity is $O(k^2 \cdot N)$. The representation of $Q(i,n)$ for some i,n includes at most k pipelines with a length that equals at most the number of distinct possible tasks r. The algorithm keeps $Q(i,n)$ for k values of the first parameter and for $(N+1)$ values of the second. The memory required for storing one value of $T(i,n), B(i,n)$ is smaller. The memory complexity of the algorithm is accordingly $O(k^2 \cdot N \cdot r)$.

B. General Properties

We now describe some basic properties of this problem of minimizing the average delay. These properties can be used by a designer to understand the limits of the chip design capacity region. The simple proof of the first of them is omitted due to space limits.

The first property discusses the value of the optimal delay as a function of the number of cores. Intuitively, for a too small number, we cannot serve some the tasks and for a large enough number, each type can have a dedicated pipeline.

Proposition 1. (i) *For all $N \geq 0$, the optimal average delay is at least as large as the (weighted) average size of a set of tasks, i.e. $T_{OPT}(N) \geq \sum_{i=1}^{k} \left(p_i \cdot |S_i|\right)$.*
(ii) $T_{OPT}(N) = \sum_{i=1}^{k} \left(p_i \cdot |S_i|\right)$ *for* $N \geq \sum_{i=1}^{k} |S_i|$.
(iii) *The number of cores should be at least the number of distinct tasks in the input, i.e. $T_{OPT}(N) = \infty$ for $N < |\bigcup_{i=1}^{k} S_i|$.*

The next property presents a condition that must be satisfied for every packet in an optimal solution.

Proposition 2. *In any optimal solution (i.e. a solution with a delay of $T_{OPT}(N)$), every packet type is served by a pipeline that has the minimal length among the pipelines that serve all the tasks of the packet type.*

Proof: Clearly, if this is not the case for one of the packet types, we can let this packet type be served by an existing pipeline with shorter length and reduce the average delay. ■

C. Pipeline Merging Algorithm for Minimizing the Average Delay

We suggest an efficient greedy algorithm for reducing the average packet delay for a general input. Intuitively, it can be efficient to serve within the same pipeline several packet types with a large number of common tasks, so that their merging would result in a small delay increment. The algorithm starts with an initial state in which there is a pipeline for each of the packet types. This initial state would need many cores to be implemented, and would typically exceed the number of available cores on the multi-core chip. Therefore, our algorithm iteratively reduces the number of needed cores. Specifically, at each iteration, it *merges* two of the remaining pipelines into a single shared pipeline. It only ends when the merged pipelines can finally be implemented using the multi-core chip (or when there is clearly no solution).

In other words, consider a given iteration of the algorithm. Assume that its pipelines currently use n cores. If $n \leq N$, the pipelines can be implemented and the current state is returned as the solution. Else, we select two pipelines and merge them. Then, packet types that make use of these pipelines might observe a larger delay after this operation.

For a given pair of pipelines, let x be the increment in the average delay if these pipelines are merged. Further let y be the corresponding decreased number of cores. Intuitively, it is better to merge a pair of pipelines with a small ratio x/y.

We calculate x, y for every pair of pipelines. Let A_i (for $i \in [1,2]$) be the set of cores in each of the two pipelines of the pair. This set of cores is the union of the cores required by the packet types served by this pipeline. Likewise, let z_i be the probability of an arbitrary packet to belong to a packet type served by this pipeline. This is the sum of the probabilities for a packet to belong to one of the packet types served by the merged pipeline.

We consider only meaningful pairs of pipelines, i.e. pairs with common cores. Merging pipelines with disjoint sets of cores cannot reduce the number of cores. For these valid pairs, we examine the three following criteria. (a) Large number of common cores, (b) Small number of non-common cores, and (c) Low probability for an arbitrary packet to belong to a type served by the merged pipelines.

The additional delay for packets previously served by the first pipeline (fraction of z_1 of all packets) is $|A_2 \setminus A_1|$. Likewise, for packets served by the second pipeline (z_2 of all packets) the additional delay is $|A_1 \setminus A_2|$. Thus the increase in the average delay if these pipelines are merged is $x = z_1 \cdot |A_2 \setminus A_1| + z_2 \cdot |A_1 \setminus A_2|$. This is the *cost*. If the two pipelines in the pair are merged, the total number of cores is reduced by the number of common cores $y = |A_1 \cap A_2| > 0$. This is the *gain* in such merging.

For this pair, we define the ratio R as the marginal cost, i.e. $R = x/y = (z_1 \cdot |A_2 \setminus A_1| + z_2 \cdot |A_1 \setminus A_2|)/|A_1 \cap A_2|$. In each step of the algorithm we simply merge the pair of pipelines with the minimal marginal cost.

Since a pair of pipelines is merged in each step, the number of outer loops of the algorithm is at most k. In each step we compare less than k^2 pairs by checking the union of sets with size of at most r. Thus the time complexity is $O(k^3 \cdot r)$. Notice that in this algorithm, keeping the current set of pipelines is required only for the last outer loop. Other variables require a smaller amount of memory. Thus the memory complexity is $O(k \cdot r)$.

Example 2. *Consider again the input from Example 1 with $k = 3, N = 8$ and $(S_1, p_1), (S_2, p_2), (S_3, p_3) = ([1], 0.3), ([1, 3], 0.2), ([1, 5], 0.5)$. For $i, j \in [1, 3]$, let $R_{i,j}$ be the value of the ratio R as defined above for the pair of packet types (S_i, p_i) and (S_j, p_j). Here, $R_{1,2} = (0.3 \cdot 2 + 0.2 \cdot 0)/1 = 0.6$, $R_{1,3} = (0.3 \cdot 4 + 0.5 \cdot 0)/1 = 1.2$ and $R_{2,3} = (0.2 \cdot 2 + 0.5 \cdot 0)/3 \approx 0.133$. Since the minimal ratio is $R_{2,3}$, the suggested algorithm merges the second and third pipelines. We then obtain a solution with two pipelines $Q_1 = [1], Q_2 = [1, 5]$ which is the optimal solution for this input.*

In Section VI, we show experiments for which the suggested (greedy) algorithm results in a delay that often equals (or is very close), to the minimal possible delay. However, as demonstrated in these simulations, the algorithm is not necessarily optimal.

IV. Minimizing worst-case delay

In this section, we consider a new objective function of minimizing *the worst-case delay*, i.e. the maximal delay obtained by one of the packet types. Here, given a limited number of cores N, a set of k packet types $S_1, \cdots, S_k$ (with positive probabilities $p_1, \cdots, p_k > 0$), we are looking for shared pipelines that minimize the worst-case delay of all packet types, i.e. the maximal length of a pipeline. We denote by $D_{OPT}(N)$ the minimal possible worst-case delay as a function of the number of cores.

A. General Properties

Our first observation is that a solution that minimizes the average delay does not necessarily minimizes also the worst-case delay. It is illustrated in the following example.

Example 3. *Let $k = 3, N = 6$ and $(S_1, p_1), (S_2, p_2), (S_3, p_3) = (\{1, 2, 3\}, 0.2), (\{3, 4\}, 0.2), (\{4, 5\}, 0.6)$. To minimize the average delay, it is better to merge the first two pipelines of types with smaller probabilities obtaining pipelines $Q_{1,1} = \{1, 2, 3, 4\}, Q_{1,2} = \{4, 5\}$ with average delay $T_1 = 0.4 \cdot 4 + 0.6 \cdot 2 = 2.8$ and worst-case delay of $D_1 = 4$. To minimize the worst-case delay, it is better to merge the two last shorter pipelines obtaining two pipelines of length 3, $Q_{2,1} = \{1, 2, 3\}, Q_{2,2} = \{3, 4, 5\}$ with average delay and worst-case delay that both equal $T_2 = D_2 = 3$.*

Another immediate observation is the independence of $D_{OPT}(N)$ of the exact values of the positive probabilities of the packet types $p_1, \ldots, p_k$. That is because the optimization function depends on the length of the longest pipeline regardless of any of the probabilities. Likewise, any of the constraints on a legal solution does not involve these probabilities. Accordingly, in this section we will characterize a packet type simply by its required set of tasks S_i.

We now describe some basic properties of this new problem, considering the worst-case delay. Next we present bounds on this delay as a function of the number of cores N. As in the case of the average delay, these bounds can be helpful for a designer to understand the dependence of $D_{OPT}(N)$ on N.

Proposition 1. (i) *For all $N \geq 0$, $D_{OPT}(N) = \infty$ or $D_{OPT}(N) \in \{1, 2, \cdots, r\}$, namely whenever the worst-case delay is finite, it has an integer value of at most r.*
(ii) *For all $N \geq 0$, the optimal worst-case delay is at least as large as the optimal average delay, i.e. $D_{OPT}(N) \geq \lceil T_{OPT}(N) \rceil$.*
(iii) *For all $N \geq 0$, the optimal worst-case delay is at least as large as the size of the largest set of tasks in the input, i.e. $D_{OPT}(N)$ satisfies $D_{OPT}(N) \geq \max_{i \in [1,k]} |S_i|$.*
(iv) *For a large enough number of cores, the worst-case delay exactly equals the size of the largest set of tasks in the input, i.e. $D_{OPT}(N) = \max_{i \in [1,k]} |S_i|$ for $N \geq \sum_{i=1}^{k} |S_i|$.*
(v) *The number of cores should be at least the number of distinct tasks in the input, i.e. $D_{OPT}(N)$ satisfies $D_{OPT}(N) = \infty$ for $N < |\bigcup_{i=1}^{k} S_i|$.*

Proof: Whenever all packet types are served, the worst-case delay is the delay of the longest pipeline in the solution. The solution is of course composed of pipelines with an integer number of cores. The length of each pipeline equals at most the number of possible tasks r. In addition, the length of the longest pipeline is clearly not shorter than the (weighted) average length of the pipelines. In any solution a packet type with a set of tasks S_i is served by a pipeline Q_j that satisfies $S_i \subseteq Q_j$. The delay $|Q_j|$ that the packet type encounters equals $|Q_j| \geq |S_i|$ and the worst-case delay among all the k packet types must satisfy $D_{OPT}(N) \geq \max_{i \in [1,k]} |S_i|$. For a large enough number of cores (satisfying $N \geq \sum_{i=1}^{k} |S_i|$), each packet type S_i is served separately by a pipeline of length $|S_i|$ and the worst-case delay is simply $D_{OPT}(N) = \max_{i \in [1,k]} |S_i|$. Last, since each of the $|\bigcup_{i=1}^{k} S_i|$ distinct tasks must be implemented by a dedicated core, any legal solution cannot be found whenever $N < |\bigcup_{i=1}^{k} S_i|$. ■

We now shortly consider the simple case of $S_1, \cdots, S_k$ described in Section III-A, for which the sets of required tasks are of the form $S_i = \{1, \cdots, Y_i\} = [1, Y_i]$ for $Y_i \in [1, r]$. It is easy to see that for this case an optimal solution for the worst-case delay is simply given by a single pipeline with the cores required by the largest set in the input. Clearly, any solution that serves all packet types requires at least that number of cores and entails such a delay.

We present additional properties of an optimal solution that minimizes the worst-case delay for a general input. The following property is useful to decrease the algorithm complexity, and will be used in Section VI of simulations. Intuitively, with regard to the worst-case, a packet type with a set of tasks S_i can always be served by a pipeline that serves a packet type with a set S_j that is a super set of S_i.

Lemma 1. *Assume that the k sets of required tasks can be ordered in such a way that for some m, each of the first m sets is a subset of one of the last $(k - m)$ sets, namely for every $S_i \in \{S_1, \cdots, S_m\}$ we can find $S_j \in \{S_{m+1}, \cdots, S_k\}$ satisfying $S_i \subseteq S_j$. Then, the set of pipelines in an optimal solution for the input $S_{m+1}, \cdots, S_k$ is also an optimal solution* *for the input $S_1, \cdots, S_k$ (with the same value of the parameter N). In extending this solution, by simply serving each pipeline in $S_i \in \{S_1, \cdots, S_m\}$ using the pipeline that serves $S_j \in \{S_{m+1}, \cdots, S_k\}$ s.t. $S_i \subseteq S_j$, we can obtain a solution that achieves $D_{OPT}(N)$.*

Proof: By considering the sets of tasks $S_{m+1}, \cdots, S_k$, a subset of the original sets of tasks, the optimal solution has a delay $D \leq D_{OPT}(N)$. Namely, this delay cannot be longer than the optimal delay $D_{OPT}(N)$ for the larger set $S_1, \cdots, S_k$. By serving each of type $S_i \in \{S_1, \cdots, S_m\}$ by a pipeline that serves type S_j s.t. $S_i \subseteq S_j$, the set of pipelines in the solution is not changed and the same delay D satisfying $D \leq D_{OPT}(N)$ is obtained for the original input. Thus, clearly $D = D_{OPT}(N)$ and the set of pipelines is optimal also for $S_1, \cdots, S_k$. ■

B. A simple case of the required tasks: $S_i = [X_i, Y_i]$

We discussed earlier a simple case of the input for which a solution achieving an optimal average delay can be found. We now present an algorithm that achieves the optimal worst-case delay for a generalized class of possible inputs. The algorithm applies when the sets of required tasks are all of the form of ranges, namely each set includes some consecutive tasks and for $i \in [1, k]$ the set S_i is of the form $S_i = [X_i, Y_i]$. Clearly, this case generalizes the set of legal inputs discussed earlier of the form $S_i = [X_i, Y_i]$.

This case is applicable for instance in a network in which a task is associated with a network layer in which it is served, e.g. *traffic segmentation* and *error correction* at layer 2, *routing* at layer 3 or *load balancing* at layer 7. If a packet type requires the tasks of several consecutive layers (for instance, the tasks of layers 2-4), the demands of the different packets can be presented in the above form by simply ordering each task according to the layer it belongs to.

Based on Lemma 1, we also assume that none of the ranges is fully contained in another range. If this is the case for some of the ranges, we could ignore them while trying to find the optimal solution. In addition, we assume that these k sets of tasks $S_1 = [X_1, Y_1], \cdots, S_k = [X_k, Y_k]$ are ordered in an non-decreasing order of the first value in each range, i.e. $X_i \leq X_{i+1}$ for $i \in [1, k-1]$. Likewise, if $X_i = X_{i+1}$, we must have in contradiction to our assumption that either $S_i \subseteq S_{i+1}$ (if $Y_i \leq Y_{i+1}$) or $S_{i+1} \subseteq S_i$ (if $Y_{i+1} \leq Y_i$). Therefore, $X_i < X_{i+1}$. By Lemma 2, we can consider an optimal solution for the worst-case delay satisfying that any of its pipelines cannot be partitioned into two pipelines.

There are three options of the form of two consecutive ranges $S_i = [X_i, Y_i]$, $S_{i+1} = [X_{i+1}, Y_{i+1}]$ with $X_i < X_{i+1}$. They are illustrated in Fig. 3. In the first, described in (a), $X_i < X_{i+1} \leq Y_i < Y_{i+1}$. In the second, in (b) $X_i \leq Y_i < X_{i+1} \leq Y_{i+1}$. A third option described in (c), in which $X_i < X_{i+1} \leq Y_{i+1} \leq Y_i$, is not possible due to our assumption that a range is not fully contained in another range.

Next, we partition the k sets of tasks $S_1, \cdots, S_k$ into blocks of consecutive sets such that two consecutive sets are in the same block if they intersect, as in case I described in Fig. 3(a). For any pair of consecutive sets in a block, we have $X_i < X_{i+1} \leq Y_i < Y_{i+1}$. In addition, there is no intersection between any two sets of different blocks. Let $\phi(i)$ (for $i \in [1, k]$) denote the block index for packet type i. We can now assume the existence of an optimal solution in which each pipeline serves only tasks from sets in one block.

The following lemma explains that there exists an optimal solution in which the packet types served by any of the pipelines are sequential.

Lemma 2. *There exists an optimal solution without pipelines that can be partitioned, satisfying that a pipeline that serves packet types a, b (for $a < b$), serves also all packet types in $[a, b]$.*

Proof Outline: In such a solution, let i (and j) be a maximal (respectively minimal) index such that all types with indices in $[a, i]$, (respectively $[j, b]$) are served in the pipeline serving types a, b. We explain that necessarily $S_i \cap S_j \neq \emptyset$ and we can let types $i+1, \cdots, j-1$ by served by that pipeline without increasing its delay. ■

Following the described properties, we can suggest a dynamic-programming algorithm that finds an optimal solution that minimizes the worst-case delay for this case of the problem. The algorithm is similar to the algorithm from Section III-A although they consider different objective functions and have different assumptions on the input.

We denote by $D(i, n)$ (for $i \in [0, k]$, $n \in [0, N]$) the minimal possible worst-case delay that can obtained in serving the first i packet types using at most n cores. If no such solution exists, $D(i, n) = \infty$. Likewise, we denote by $B(i, n)$ the vector of length i that indicates for each packet type the serving pipeline in the corresponding optimal solution for the parameters i, n and by $Q(i, n)$ the list of pipelines in this solution. While trying to serve the first i types, we will examine all options for the number of consecutive packet types that are served within the same pipeline as the i^{th} type. These types can be among the previous packet types that have the same block index as the i^{th} type. For each of these options, the last pipeline includes a number of cores that equals the total number of distinct tasks required by one of the types it serves. If this number is denoted by h, the other packet types can be served by the left $n-h$ cores. The algorithm will be based on the correctness of the recursive formulas described in the following theorem.

Theorem 2. (i) *For $i \geq 1$, the variable $D(i, n)$ satisfies*

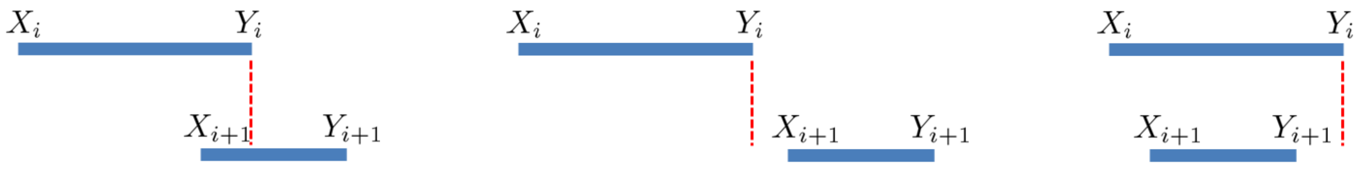
$$D(i,n) = \min_{j \in [1,i], \phi(i-j+1)=\phi(i)} \left(\max \Big(D(i-j, n-h), h \Big) \right), \quad (4)$$

$$\textit{for } h = |\bigcup_{\ell=i-j+1}^{i} S_\ell|.$$

(ii) *Let j be the minimal value of the corresponding parameter that minimizes the value of $D(i, n)$ in its formula. Then,*

$$Q(i,n) = Q(i-j, n-h) \cup \{ \bigcup_{\ell=i-j+1}^{i} S_\ell \}, \textit{ and} \quad (5)$$

$$B(i,n) = B(i-j, n-h) \cdot \underbrace{(|Q(i,n)|, \cdots, |Q(i,n)|)}_{j \text{ times}}. \quad (6)$$



(a) Case I: $X_i < X_{i+1} \le Y_i < Y_{i+1}$

(b) Case II: $X_i \le Y_i < X_{i+1} \le Y_{i+1}$

(c) Case III: $X_i < X_{i+1} \le Y_{i+1} \le Y_i$

Fig. 3. Three cases of the form of two consecutive (range) sets of tasks $S_i = [X_i, Y_i], S_{i+1} = [X_{i+1}, Y_{i+1}]$ satisfying $X_i < X_{i+1}$. In 3(a) (Case I), $S_i \cap S_{i+1} \neq \emptyset$ but $S_i \subsetneq S_{i+1}, S_{i+1} \subsetneq S_i$. In 3(b) (Case II), $S_i \cap S_{i+1} = \emptyset$. In 3(c) (Case III), $S_{i+1} \subseteq S_i$. Case III is not possible due to the assumption that a range is not fully contained in another range after processing the input.

Algorithm 2 Optimal Algorithm for Minimizing the Worst-Case Delay for $S_i = [X_i, Y_i]$ (Pseudo-code)

```
   Input:
1  Packet types with sets of tasks $S_1 = [X_1, Y_1], \cdots, S_k = [X_k, Y_k]$
2  Initialize: For $n \ge 0$: $D(0, n) = 0$. For $n < 0$: $D(0, n) = \infty$. For $i > 0, n \le 0$: $D(i, n) = \infty$. For $n \ge 0$: $B(0, n) = ()$ (an empty vector) and $Q(0, n) = \{\}$ (an empty set of pipelines).
3  Simplify Input: Sort $S_1, \cdots S_k$ with incr. values of $|S_i|$
4  for $i = 1$ to $k$ do
5      if $S_i \subseteq S_j$ for some $j \in [i + 1, k]$ then
6          Remove $S_i$
7  Main: Sort $S_1, \cdots S_k$ with incr. values of $X_i$
8  $\phi(1) = 1$
9  for $i = 2$ to $k$ do
10     $\phi(i) = \phi(i - 1)$;
11     if $S_{i-1} \cap S_i = \emptyset$ then
12         $\phi(i) = \phi(i - 1) + 1$
13 for $i = 1$ to $k$ do
14     for $n = 0$ to $N$ do
15         Calculate $D(i, n)$ according to Eq. (4) with $\phi(\cdot)$
16         Calculate $Q(i, n)$ according to Eq. (5)
17         Calculate $B(i, n)$ according to Eq. (6)
18 return $Q(k, N)$, $B(k, N)$
```

Proof: We consider all options for the number of packet types j that are served together with packet type i in the pipeline. This number of types satisfies $j \in [1, i]$ and these j types must be in the same block as type i. In this case the delay (number of cores) of the last pipeline is $h = |\left(\bigcup_{\ell=i-j+1}^{i} S_\ell\right)|$ and the worst-case delay is the maximum of this delay and the delay obtained in the optimal solution for the first $(i - j)$ types with $(n - h)$ cores. For a given j, the last pipeline (with index $|Q(i, n)|$) supports tasks $\left(\bigcup_{\ell=i-j+1}^{i} S_\ell\right)$ and serves types $[i - j + 1, i]$. ■

A pseudo-code of the dynamic-programming algorithm is given in Algorithm 2. We set the initial values of $D(0, n) = 0$ for $n \ge 0$, $D(0, n) = \infty$ for $n < 0$, $D(i, n) = \infty$ if $i > 0$ and $n \le 0$, $B(0, n) = ()$ and $Q(0, n) = \{\}$ for $n \ge 0$ (line 2). We then calculate in step i (for $i \in [1, k]$) recursively the values of $D(i, n)$ for $n \in [0, N]$ based on Theorem 2 (line 15) as well as of $D(i, n), B(i, n)$ (lines 16-17). Finally, the optimal worst-case delay $D_{OPT}(N)$, the pipelines in an optimal solution and the matching vector for the packet types are given by $D(k, N), Q(k, N)$ and $B(k, N)$, respectively. It takes $O(k^2)$ to look for subsets that are included in others (lines 4-6). Since we have k types, $n \in [0, N]$ and we consider at most k options to calculate a value of the recursive formula of $D(i, n)$ (line 15), the time complexity of the algorithm is $O(k^2 \cdot N)$. The memory complexity is $O(k^2 \cdot N \cdot r)$ since we keep $O(k \cdot N)$ values of $Q(i, n)$ each with a memory complexity of $k \cdot r$.

Example 4. *We consider the set of tasks from Example 3 with $k = 3$, $(S_1, p_1), (S_2, p_2), (S_3, p_3) = (\{1, 2, 3\}, 0.2), (\{3, 4\}, 0.2), (\{4, 5\}, 0.6)$ and $N = 6$ cores. Of course, the exact values of $p_1, p_2, p_3 > 0$ have no influence on the optimal worst-case delay. Since $S_1 \cap S_2 \neq \emptyset$ and $S_2 \cap S_3 \neq \emptyset$, all these three sets are in the same block.*

Here, $D(0, 0) = D(0, 1) = 0$, $D(1, 3) = 3$. Based on Theorem 2, $D(2, 4) = \min\big(\max(D(1, 2), 2), \max(D(0, 0), 4)\big) = \min\big(\max(\infty, 2), \max(0, 4)\big) = 4$. In the first option of this formula, type 2 is served by a dedicated pipeline, while in the second option, type 1 is served by the same pipeline as type 2. Likewise, $D(3, 6) = \min\big(\max(D(2, 4), 2), \max(D(1, 3), 3), \max(D(0, 1), 5)\big) = \min\big(\max(4, 2), \max(3, 3), \max(0, 5)\big) = 3$. The optimal delay of $D(k, N) = D(3, 6) = 3$ is obtained for the second option among the three above, using $j = 2$. Thus in the optimal solution, the last $j = 2$ types S_2, S_3 are served in the same pipeline. Finally, $Q(k, N) = Q(3, 6) = Q(1, 3) \cup \{S_2 \cup S_3\} = \{\{1, 2, 3\}, \{3, 4, 5\}\}$ and $B(k, N) = B(3, 6) = B(1, 3) \cdot (|Q(3, 6)|, |Q(3, 6)|) = (1, 2, 2)$. We can see that this solution is indeed the optimal solution for minimizing the worst-case delay as described in Example 3.

C. Pipeline Merging Algorithm for Minimizing the Worst-Case Delay

We now describe a greedy algorithm for trying to minimize the worst-case delay. In Section VI, we examine also this algorithm and show that it can often obtain the minimal possible worst-case delay, although it is not optimal in the general case. We later refer to this algorithm as the second greedy algorithm.

The first step of the algorithm is to check for each set of tasks in the input whether it is a subset of another set. If this is the case for some of the sets, by Lemma 1 we can reduce the number of types that have to be considered by ignoring the corresponding types.

We remind that the number of possible tasks is r. By Proposition 1, whenever it is finite, the value of the optimal worst-case delay satisfies $D_{OPT}(N) \in [\max_{i\in[1,k]} |S_i|, r]$. The second step of the algorithm considers all these possible values for the delay. For each possible value, it starts with a state in which all the types (that have not been eliminated in the first step) have dedicated pipelines. As long as pairs of pipelines that can be merged without increasing the assumed worst-case delay can be found, the algorithm merges a pair among these pairs that its two packet types have the maximal number of tasks in common. Otherwise, the pair is selected among all pairs according to the same criteria till satisfying the constraint on the number of cores. Finally, the solution is selected as the best one for the different values of the delay.

The time complexity of the algorithm is $O(k^3 \cdot r^2)$. It first takes $O(k^2 \cdot r)$ to examine the possibility of simplifying the input. The number of outer loops (values of D) equals at most the maximal value of the worst-case delay r. In each step we perform $O(k)$ merging operations. For each such operation we compare at most k^2 pairs, with a union of size of at most r. Its memory complexity is $O(k \cdot r)$ to keep the best solution found so far.

Example 5. *Let $k = 4$, $S_1 = \{1, 2, 3, 4\}$, $S_2 = \{2, 3, 5\}$, $S_3 = \{3, 5\}$ and $S_4 = \{5, 6\}$ with $N = 7$. Since $S_3 \subseteq S_2$, we can neglect S_3 and later serve it by the pipeline serving S_2. We consider value in $[\max_{i\in[1,k]} |S_i|, r] = [4, 6]$ for the worst-case delay. 9 cores are required if S_1, S_2, S_4 are served separately. First, we try to achieve a worst-case delay of 4. We can only merge S_2, S_4 by avoiding having a pipeline with more than 5 cores. Then. we have the two pipelines $\{1, 2, 3, 4\}$ and $\{2, 3, 5, 6\}$ that must be merged to satisfy the constraint on the number of cores obtaining a single pipeline with a delay of 6. While trying to achieve a worst-case delay of 5, we first merge S_1, S_2 that have 2 tasks in common, more than any other pair. We have the pipelines $\{1, 2, 3, 4, 5\}$ and $\{5, 6\}$ with 7 cores and an improved worst-case delay of 5. We get a similar solution in the next option for the delay. Thus the greedy achieves a worst-delay of 5 with pipelines $\{1, 2, 3, 4, 5\}$ serving packet types 1,2,3 and $\{5, 6\}$ serving packet type 4. While this is also the optimal solution for this input, we demonstrate later that the algorithm is not necessarily optimal.*

V. Model Generalization

Our model includes various assumptions as described in Section II. We discuss how we can easily support relaxation of some of them.

Tasks with different processing delays. If the tasks have non-identical processing delays, the clock period of a pipeline can be modeled as the longest task it implements. For $j \in [1, r]$, let τ_j be the delay of task j. The clock period of a pipeline $Q \subseteq [1, r]$ is accordingly $\tau_Q = \max_{j\in Q}(\tau_j)$. Then, a packet observes a delay that equals the number of cores in a pipeline times its clock period, i.e. $|Q| \cdot \tau_Q$ for a packet served by pipeline Q. In this scenario, when considering the merging of two pipelines, the clock period of each should also be taken into account. This change affects both average and worst-case delay. We demonstrate that for the average delay. Consider for instance three packet types with $S_1 = \{1, 4, 5\}, S_2 = \{2, 4\}, s_3 = \{3, 4, 6\}$, $p_1 = p_2 = 0.3, p_3 = 0.4$ and $N = 7$ available cores. If the processing delays of all tasks are identical (i.e. $\tau_1 = \tau_2 = \tau_3 = \tau_4 = \tau_5 = \tau_6$), a solution that minimizes the average delay has two pipelines $\{1, 2, 4, 5\}$ (for the first two packet types) and $\{3, 4, 6\}$ (for the third type) and is obtained by merging the pipelines of the two types with the smaller probability. Assume now that the processing time of task 2 is $\tau_2 = 2\mu$s while for all other tasks ($j \in \{1, 3, 4, 5, 6\}$) it is only $\tau_j = 1\mu$s. Then, the clock period of a pipeline that contains task 2 is 2μs while the clock period of a pipeline without this task is 1μs. By merging the first two pipelines, we have one pipeline with four cores $Q_{1,1} = \{1, 2, 4, 5\}$ serving $0.3 + 0.3 = 0.6$ of the traffic and a second with three cores $Q_{1,2} = \{3, 4, 6\}$ serving 0.4 of the traffic. The clock periods yield an average delay of $(p_1 + p_2) \cdot |Q_{1,1}| \cdot \tau_{Q_{1,1}} + p_3 \cdot |Q_{1,2}| \cdot \tau_{Q_{1,2}} = 0.6 \cdot 4 \cdot 2 + 0.4 \cdot 3 \cdot 1 = 6\mu$s. An optimal solution is achieved by merging the pipelines of the first and the third packet types, obtaining $Q_{2,1} = \{1, 3, 4, 5, 6\}$, $Q_{2,2} = \{2, 4\}$. This results in a delay of $(p_1 + p_3) \cdot |Q_{2,1}| \cdot \tau_{Q_{2,1}} + p_2 \cdot |Q_{2,2}| \cdot \tau_{Q_{2,2}} = 0.7 \cdot 5 \cdot 1 + 0.3 \cdot 2 \cdot 2 = 4.7\mu$s. To support this generalization, we can simply change our greedy algorithms to take into account the possible change in the clock period in the calculation of the additional delay for each possible pipeline merging.

Generalized traffic arrival patterns. The assumption that at most a single packet arrives at each time slot guarantees that no more than one packet enters a shared pipeline. With this assumption, a large ratio of the cores, that depends on the number of existing pipelines, remains idle at each time slot. Of course, by sharing pipelines, the ratio of idle cores decreases. The assumption can be relaxed in three ways. First, by assuming that packet types are divided into families, namely each packet type $i \in [1, k]$ is associated with a family id β_i, such that in every time slot at most one packet arrives in each family. Then, pipelines can be shared while satisfying the above guarantee according to the restriction that a shared pipeline serves packet types of the same family, i.e. satisfying $\beta_i = \beta_j$ for packet types i, j. In a second possible generalization, at each time slot at most one packet appears for each packet type while more than a single packet can appear for different types. Likewise, we can assume a shared pipeline can serve up to h packets in a time slot for some $h \geq 1$. Accordingly, a shared pipeline is restricted to serve at most h packet types and two pipelines serving together more than h types cannot be merged. A third generalization enables queueing packets waiting for the service of a pipeline. Even while serving only a single packet per time slot in a pipeline, with the cost of memory and additional queueing delay, packets arriving at the same time can be stored with the assumption that the total server rate of the pipeline is larger than the arrival of all types it serves.

VI. Experimental Results

We conduct experiments to examine the efficiency of the suggested greedy algorithms from Section III and Section IV for minimizing the average and the worst-case delays, respectively. When possible, we compare them with the optimal solutions. In the experiments, we rely on synthetic examples. For space reasons, additional experiments with real-life applications can be found in the Appendix. We denote the number of possible solutions for sharing k packet types without a restriction on the core number by $G(k)$. This equals the number of options to divide the k types into up to k pipelines. For a small k, we can calculate the optimal delays by considering all possible solutions.

We start by assuming a variety of $k = 8$ types with tasks among $r = 10$ possible tasks, $\{1, \cdots, r = 10\}$. The tasks required by each type are selected randomly. The results are based on the average of 10^3 simulations.

In our first experiment, each type requires a specific task with probability $q = 0.5$, without any dependency between the different types and the different tasks, i.e. for every $i \in [1, k], j \in [1, r] \Pr(j \in S_i) = q = 0.5$. Fig. 4 presents the obtained average delay (in time slots) as a function of the number of cores.

We assume a first subcase in which all types have *identical probabilities* of $\frac{1}{k} = 0.125$. For a given set of types and a maximal value of N cores, we compare the average delay obtained by the (first) greedy algorithm with the minimal possible delay found while considering all possible solutions satisfying the constraint on the number of cores. Here, the total number of solutions for each input is $G(k = 8) = 4140$. Clearly, the minimal value of N that guarantees that all tasks can be satisfied is $N = r = 10$. In addition, the maximal observed total number of tasks in the $k = 8$ types is smaller than 60. Thus, we examine the values of $N \in [10, 60]$. The results are presented in the first two upper curves in Fig. 4. For example, for large enough values of N, each type has its own pipeline and the average delay equals the average number of tasks per type, $r \cdot q = 10 \cdot 0.5 = 5$. In general, the average delay obtained by the greedy algorithm is relatively close to the optimal delay and it becomes even closer for larger N. For instance, for $N = 16$ the average greedy delay is 8.470, larger by 4.4% than the optimal delay of 8.110. This option suggests a reduction of $(60 - 16)/60 = 73.3\%$ in the number of cores with a cost of a delay larger by 70% or 62.8%. For $N = 40$ the delay of the greedy is 5.122 when the minimal possible average delay (5.112) is smaller by less than 0.2%. The average difference, for $N \in [10, 60]$, between the two delays is 0.103 time slots.

For the presented scheme of the selection of the tasks, we examine also a second subcase in which the types appear with *variable probabilities*. We set the probabilities to be geometrically decreasing such that the $k = 8$ probabilities of the k types are $\alpha \cdot 2^{-1}, \alpha \cdot 2^{-2}, \alpha \cdot 2^{-3}, \alpha \cdot 2^{-4}, \alpha \cdot 2^{-5}, \alpha \cdot 2^{-6}, \alpha \cdot 2^{-7}, \alpha \cdot 2^{-8}$ for $\alpha = 256/255$. These non-homogenous probabilities enable us to distinguish between the types to further improve the obtained average delay. For instance, we might prefer to have a dedicated pipeline for the most common type containing only cores for its required tasks. The observed delays, again by the greedy algorithm and the exhaustive search are illustrated in the two additional curves in Fig. 4. For instance, again for $N = 16$ the observed average delays are 7.310 and 7.090,

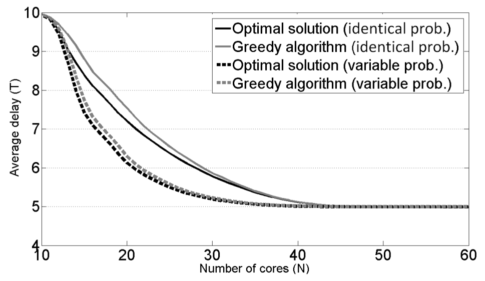


Fig. 4. Average delay (in time slots) as a function of the number of available cores (N) for the first experiment with synthetic data. Here, the probability for a packet type to require each task is 0.5. The two upper curves present the delay when the $k = 8$ types appear with a uniform distribution. The two bottom curves examine the case where the types appear with geometrically-decreasing probabilities. The results are based on the average of 10^3 experiments. The greedy algorithm performs relatively close to the optimal one.

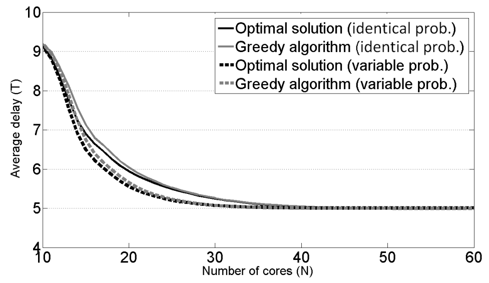


Fig. 5. The average delay (in time slots) as a function of the number of available cores (N) for the second experiment with synthetic data. Here, the probability for a packet type to require each task is either 0.9 or 0.1, according to one of three predetermined distributions. The two upper curves present the delay when the $k = 8$ types appear with a uniform distribution. The two bottom curves examine the case where the types appear with geometrically-decreasing probabilities. The results are based on the average of 10^3 experiments. Again, the greedy algorithm achieves close-to-optimal results.

respectively. Both delays are shorter by approximately a single time slot than the corresponding delays in the homogenous case. Here, the average difference between the average delay of the greedy algorithm and the minimal possible average delay is even shorter and equals 0.061 time slots.

We also want to check whether a possible dependency between the different types can further improve the effectiveness of our approach. In a second experiment, the tasks required by the types are selected in a different manner. We first randomly produce three task distributions. Each distribution randomly defines for each of the r tasks whether it will be required with a high probability of 0.9 or only with a smaller probability of 0.1. For each task, both options are obtained with equal probabilities of 0.5. Next, each of the $k = 8$ types is assigned with a distribution and its tasks are randomly selected accordingly. In this experiment, the probability of a type to require a task is $0.5 \cdot 0.9 + 0.5 \cdot 0.1 = 0.5$, as in the first experiment. The results are displayed in Fig. 5.

Informally, we expect two types selected from the same distribution to have relatively similar sets of tasks. Therefore, a merging operation of their pipelines will result in an additional delay that is relatively small. We consider the same two options for the probabilities for the different types as in the first experiment (identical and geometrically decreasing). In the first

TABLE I
SUMMARY OF THE SYNTHETIC EXPERIMENTS FOR THE AVERAGE DELAY FOR $N \in [10, 60]$

(A) Average value of the delay (in time slots)

Experiment	Identical Prob.		Variable Prob.	
	Greedy	Optimal	Greedy	Optimal
(i) Independent types	6.197	6.094	5.788	5.726
(ii) Distribution-based types	5.652	5.603	5.515	5.469

(B) Delay for $N = 16$ (in time slots)

Experiment	Identical Prob.		Variable Prob.	
	Greedy	Optimal	Greedy	Optimal
(i) Independent types	8.470	8.110	7.310	7.090
(ii) Distribution-based types	6.802	6.649	6.401	6.223

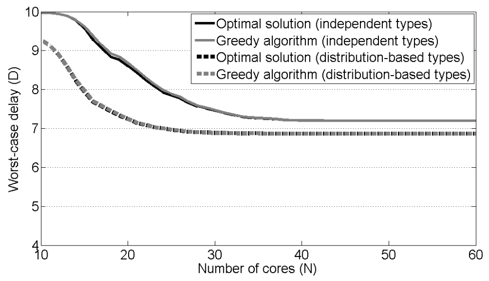


Fig. 6. The worst-case delay (in time slots) as a function of the number of available cores (N) (with the same scale as in Fig. 4 and Fig. 5). The worst-case delay is independent of the probabilities of the different packet types and accordingly we do not distinguish between the two options of these probabilities. The two upper curves present the delay when the $k = 8$ types are drawn independently (as in Fig. 4). The two bottom curves examine the case where the types are drawn based on one of three task distributions (as in Fig. 5). In all cases, the worst-case delay is not shorter (and in most cases longer) than the average delay in the corresponding experiment. In this experiment, the delays obtained by greedy algorithm are even closer to the optimal delays.

subcase, with identical probabilities for the k types, illustrated in two upper curves, the average (over $N \in [10, 60]$) of the optimal delay is 5.603 in comparison with a corresponding average delay of 6.094 in the first experiment. Likewise, for $N = 16$ the greedy delay is 6.802 and the optimal delay is 6.649. In the second subcase, with non-homogenous probabilities, as shown in the two last curves, the average optimal delay is even shorter and equals 5.469. Here, for $N = 16$ the delay of the greedy algorithm is 6.401 and of the optimal equals 6.223 time slots. The results, in both experiments, of the average delay (over $N \in [10, 60]$) and the obtained delays for $N = 16$ are summarized in Table I.

We now examine the worst-case delay in the same synthetic simulations. We again compare the results of the relevant greedy algorithm with the optimal solution. Since the probabilities for the different packet types do not influence the worst-case delay, we do not distinguish between the two options discussed above of identical or geometrically decreasing probabilities. However, the dependency between the sets of tasks (selected independently or according to one of three task distributions) is still important.

The results are based again on the average of 10^3 experiments and are presented in Fig. 6. The two upper curves present the delay when the $k = 8$ types are drawn independently (as in Fig. 4). The two bottom curves examine the case where the types are drawn based on one of three task distributions (as in Fig. 5). For this metric of the worst-case delay, the results of the (second) greedy algorithm are even closer to the optimal solution. For the independent types, the average value of the optimal worst-case delay is 7.820 while the greedy algorithm obtains an average value of 7.838 (larger by only 0.23%). The obtained values of the worst-case delays are of course larger than the corresponding values of the average delays discussed earlier. When the sets of tasks are selected based on one of predetermined distributions, the delays are shorter and equals 7.158 and 7.162, respectively. Considering again the independent sets of tasks, the value of the optimal worst-case delay when $N = 16$ equals 9.273 time slots. When the number of cores is $N = 60$, the $k = 8$ pipelines can be served separately and the value of the worst-case delay equals the average value of the maximal number of tasks per type, 7.196. A short summary of the results is given also in Table II.

TABLE II
SUMMARY OF THE SYNTHETIC EXPERIMENTS FOR THE WORST-CASE DELAY FOR $N \in [10, 60]$

(A) Average value of the delay (in time slots)

Experiment	Greedy	Optimal
(i) Independent types	7.838	7.820
(ii) Distribution-based types	7.162	7.158

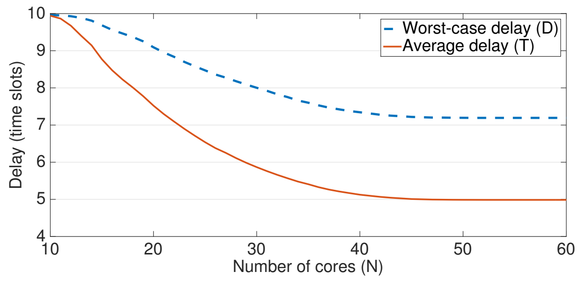
(B) Delay for $N = 16$ (in time slots)

Experiment	Greedy	Optimal
(i) Independent types	9.369	9.273
(ii) Distribution-based types	7.699	7.678

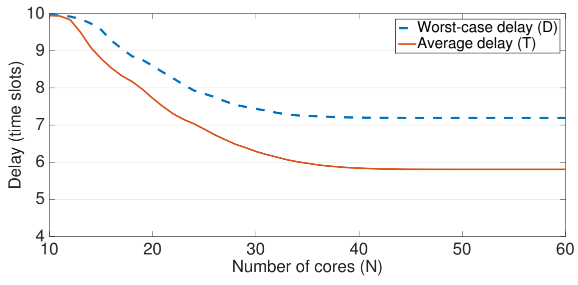
As demonstrated in Example 3, there often exists a tradeoff between the two metrics of the average delay and the worst-case delay. We study this tradeoff by comparing both delays of the solutions for the two greedy algorithms. The first algorithm that tries to minimize the average delay, might achieve far-from-optimal worst-case delay while the second algorithm that optimizes the worst-case delay can result in a relatively high average delay. We examine this for the same instance from Fig. 4 ($k = 8$ types with $r = 10$ distinct tasks, each required with probability of 0.5 by each type, identical type probabilities, 10^3 experiments). The results are illustrated in Fig. 7 and Fig. 8.

Fig. 7(a) shows the result of the first algorithm for minimizing average delay and Fig. 7(b) the results of the second algorithm for minimizing worst-case delay. For $N \in [10, 60]$, in (a) the average value of the average delay is 6.197 and that of the worst-case delay is 8.05. In (b), the average value of the average delay is 6.709 and that of the worst-case delay is 7.811. As expected, they are much closer in (b) that optimizes the worst-case delay, which is always larger.

This is demonstrated also in Fig. 8(a) that examines the ratio of the worst-case delay by the average delay in the results for the two algorithms. While in both algorithms, the worst-case and the average delays have similar values for small number of

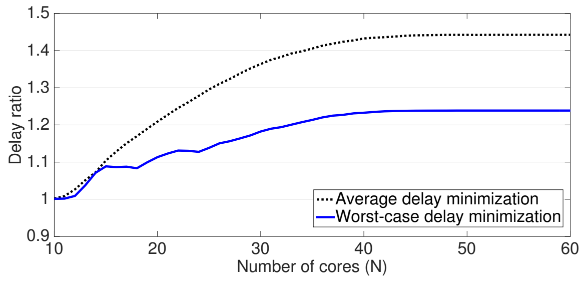


(a) Algorithm for minimizing the *average* delay

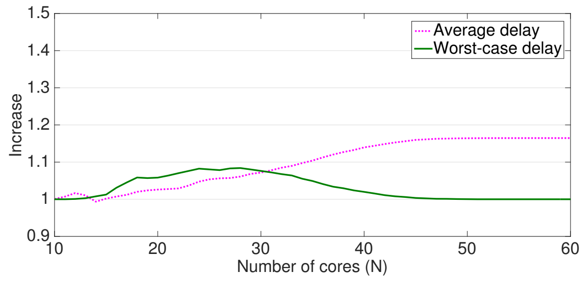


(b) Algorithm for minimizing the *worst-case* delay

Fig. 7. Algorithm comparison: The worst-case and average delay in the results of the two algorithms. In (a), the algorithm for minimizing average delay. In (b), the algorithm for minimizing worst-case delay. For both algorithms, minimizing one delay results in an increase for the other.

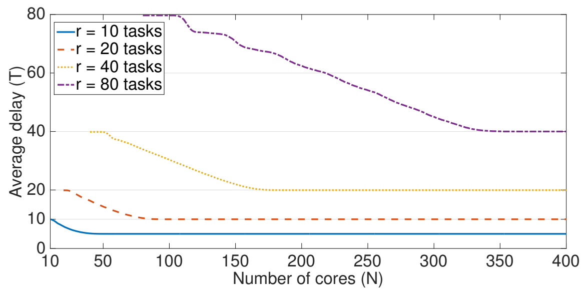


(a) Ratio of worst-case delay and average delay

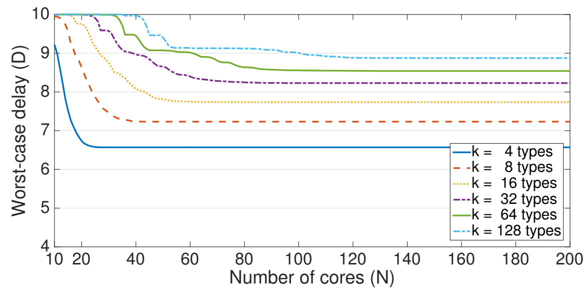


(b) Increase in each of the delays by minimizing the other

Fig. 8. Algorithm comparison: (a) presents the ratio of the worst-case delay and the average delay in both algorithms. The ratio is smaller when the worst-case delay is minimized. (b) illustrates the relative increase of each of the delays as a result of minimizing the other.



(a) Average-case delay vs. number of distinct tasks (r)



(b) Worst-case delay vs. packet types number (k)

Fig. 9. Algorithm scalability: (a) presents the average delay as a function of the number of distinct tasks (r) (with k=8 packet types). (b) describes the worst-case delay as a function of the number of packet types number (k) (with r=10 distinct tasks). Results are calculated based on the corresponding greedy algorithms.

cores (where the solution is often composed of a single pipeline with all cores), for larger values of $N \in [10, 60]$ we can see different ratios in the two algorithms. In the first algorithm the ratio grows up to 1.443, while in the second the maximal value of the ratio is 1.239. Fig. 8(b) describes the increase in each of the delays achieved by running the other algorithm. The worst-case delay increases by up to 8.4% by using solutions for the first algorithm. For a large enough number of cores, the solution of the first algorithm serves each type separately and also often achieves an optimal worst-case delay that equals the largest demand in the input. For the average delay, the increase is larger and can equal up to 16.5%. This larger increase follows an inherent property of the algorithm for minimizing the worst-case delay that simplifies its input by colliding two types when the required tasks of one is a subset of the demand of another. Thus a type can often be served unnecessarily by a longer pipeline. While this never influences the worst-case delay, it can increase the average delay. Accordingly, even for a large number of cores, we can see an increase in the average delay considering solutions of the second algorithm.

In Fig. 9, we examine the scalability of the results by considering larger values of the distinct tasks number r or the number of types k. We again let a task be required by a type with a probability of 0.5. In (a), we examine the dependency in r (with $k = 8$ types). Longer delays are obtained for larger demands and the minimal number of cores equals of course r. The number of required cores to obtain the minimal delay by separating each type in a dedicated pipeline, is correlated to (and larger than) the expected number of required tasks ($0.5 \cdot k \cdot r$). It equals 55,98,189 and 365 for $r = 10, 20, 40, 80$, respectively. We also compare the results of the greedy algorithm to the optimal delays obtained by considering all possible solutions. We find out that these results are also close-to-optimal even for larger r. For each value of r, we compare the two delays for $N \in [r, 0.5 \cdot k \cdot r]$, values for which the dependency in N is strong. The average difference in the delay was 0.163, 0.353, 0.665 and 1.162 time slots for the different values of $r = 10, 20, 40, 80$, respectively. We are encouraged by the fact the average relative increase in the delay is 2.24%, 2.38 %, 2.24% and 1.98%, i.e. relatively small and not increasing as a function of r.

In (b), we consider various values of types number k (with $r = 10$) and examine the worst-case delay. With more types, the probability to have at least one large demand is increasing and for a given number of cores enforces to share the pipelines of more types. For instance, for $k = 16$ it takes $N = 79$ cores to obtain a minimal worst-case delay of 6.569, while for $k = 128$ a delay of 8.874 is obtained with 170 or more cores. Generally, for large k this number is smaller than the expected number of required tasks ($0.5 \cdot k \cdot r$) since a single pipeline for each type is not necessarily required to minimize the worst-case delay. We cannot compare these results to the optimal ones to the increasing number of solutions for larger k.

VII. Conclusion

In this paper we introduced the *pipeline sharing* problem in multi-core chips. We explained the tradeoff between the number of needed cores and the obtained delay. We studied two optimization problems of minimizing the average delay or alternatively the worst-case delay given a limited number of cores. We suggested optimal algorithms for each of the problems that apply under different assumptions on the input. We also described greedy algorithms for the two problems for the general case. Finally, we presented experimental results that demonstrated that the greedy algorithms often achieve delays that are close to optimal. They also showed the often-existing tradeoff between the two metrics.

References

[1] O. Rottenstreich, I. Keslassy, Y. Revah, and A. Kadosh, "Minimizing delay in shared pipelines," in *IEEE Hot Interconnects*, 2013.

[2] P. G. Paulin, F. Karim, and P. Bromley, "Network processors: a perspective on market requirements, processor architectures and embedded S/W tools," in *IEEE DATE*, 2001.

[3] Y. Qi, B. Xu, F. He, B. Yang, J. Yu, and J. Li, "Towards high-performance flow-level packet processing on multi-core network processors," in *ACM/IEEE ANCS*, 2007.

[4] F. Clermidy *et al.*, "Reconfiguration of a 3GPP-LTE telecommunication application on a 22-core NoC-based system-on-chip," in *IEEE/ACM NOCS*, 2011.

[5] Q. Yu, J. Cano, J. Flich, and P. Ampadu, "Transient and permanent error control for high end multiprocessor systems-on-chip," in *IEEE/ACM NOCS*, 2012.

[6] *Network Functions Virtualisation Introductory White Paper*. ETSI, 2012.

[7] J. Subhlok and G. Vondran, "Optimal latency-throughput tradeoffs for data parallel pipelines," in *ACM SPAA*, 1996.

[8] J. Hu and R. Marculescu, "Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints," in *IEEE DATE*, 2004.

[9] C. A. M. Marcon, N. L. V. Calazans, F. G. Moraes, A. A. Susin, I. M. Reis, and F. Hessel, "Exploring NoC mapping strategies: An energy and timing aware technique," in *IEEE DATE*, 2005.

[10] K. S. Hwang, A. E. Casavant, C.-T. Chang, and M. A. d'Abreu, "Scheduling and hardware sharing in pipelined data paths," in *IEEE International Conference on Computer-Aided Design*, 1989.

[11] Q. Wu and T. Wolf, "Runtime task allocation in multicore packet processing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 10, 2012.

[12] T. Wolf and N. Weng, "Runtime support for multicore packet processing systems," *IEEE Network*, vol. 21, no. 4, pp. 29–37, 2007.

[13] I. Keslassy, K. Kogan, G. Scalosub, and M. Segal, "Providing performance guarantees in multipass network processors," *IEEE/ACM Trans. Networking*, vol. 20, no. 6, pp. 1895–1909, 2012.

[14] K. Kogan, A. López-Ortiz, S. I. Nikolenko, and A. Sirotkin, "Multi-queued network processors for packets with heterogeneous processing requirements," in *IEEE COMSNETS*, 2013.

[15] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015.

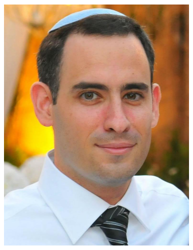
[16] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: Enabling innovation in middlebox applications," in *ACM SIGCOMM HotMiddleboxes*, 2015.

[17] E. S. H. Hou, N. Ansari, and H. Ren, "A genetic algorithm for multiprocessor scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, no. 2, pp. 113–120, 1994.

[18] N. Weng and T. Wolf, "Profiling and mapping of parallel workloads on network processors," in *ACM SAC*, 2005.

[19] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *USENIX NSDI*, 2015.

[20] J. Koziol, *Intrusion Detection with Snort*. SAMS, 2003.



Ori Rottenstreich is a Postdoctoral Research Fellow at the department of Computer Science, Princeton university, working with Prof. Jennifer Rexford. He received the B.S. in Computer Engineering (summa cum laude) and Ph.D. degree from the Electrical Engineering department of the Technion, Haifa, Israel in 2008 and 2014, respectively. He is a recipient of the Rothschild Yad-Hanadiv postdoctoral fellowship, the Google Europe PhD Fellowship in Computer Networking, the Andrew Viterbi graduate fellowship, the Jacobs-Qualcomm fellowship, the Intel graduate fellowship and the Gutwirth Memorial fellowship. He also received the Best Paper Runner Up Award at the IEEE Infocom 2013 conference.

Isaac Keslassy (M'02, SM'11) received his M.S. and Ph.D. degrees in Electrical Engineering from Stanford University, Stanford, CA, in 2000 and 2004, respectively. He is currently an associate professor in the Electrical Engineering department of the Technion, Israel. His recent research interests include the design and analysis of high-performance routers and multi-core architectures. He is the recipient of the European Research Council Starting Grant, the Alon Fellowship, the Mani Teaching Award and the Yanai Teaching Award.

Yoram Revah received the B.S. (cum laude) and M.S. degrees in Electrical Engineering from the Technion, Israel in 1997 and 2001, respectively. He worked as a VLSI Engineer and a research assistant in the Electrical Engineering department at the Technion until 2003. He received his Ph.D (summa cum laude) from the Communication Systems Department at Ben Gurion University of the Negev, Israel in 2008, where he held a Kreitman Foundation Fellowship. Currently, he is working in Marvell Israel in a position of Research & Academic Relationship engineer.

Aviran Kadosh received his B.S. degree in computer engineering (cum laude) from the Electrical Engineering department of the Technion, Haifa, Israel in 1999. With over 15 years of experience in VLSI development and networking technologies, Aviran has served in various positions throughout the development cycle. His areas of interest is fabric networks, traffic management and network-on-chip technologies.