Minimizing Delay in Shared Pipelines

Ori Rottenstreich, Isaac Keslassy
Technion
{or@tx, isaac@ee}.technion.ac.il

Yoram Revah, Aviran Kadosh
Marvell Israel
{yoramr, aviran}@marvell.com

***Abstract*—Pipelines are widely used to increase throughput in multi-core chips by parallelizing packet processing. Typically, each packet type is serviced by a dedicated pipeline. 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 the optimization problem of allocating cores between different packet types such that the average delay is minimized. We suggest a polynomial-time algorithm that finds the optimal solution when the packet types preserve a specific property. We also present a greedy algorithm for the general case. Last, we examine our solutions on synthetic examples, on packet-processing applications, and on real-life H.264 standard requirements.

I. Introduction

A. Background

This paper introduces the 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 [1], and application-specific systems-on-chip (AS-SoC) such as telecommunication applications [2] and high-end multiprocessors [3].

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

Letting packets go through their needed cores without flow control 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. If the k packet types are uniformly distributed, the obtained average delay is $(3+2+3)/3 \approx 2.67$ time slots.

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 $(4+2+4)/3 \approx 3.33$.

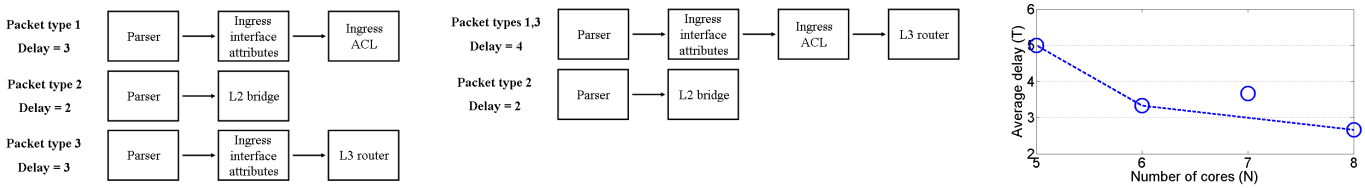
There is a clear tradeoff between the number of needed cores and the average packet delay. This tradeoff creates a capacity region in the design exploration, in the sense that a designer cannot go below some optimal bounds. This is illustrated in Fig. 1(c). The goal of this paper is to further analyze these optimality bounds.

B. Related Work

Pipeline sharing is a novel problem, and as a result there is little relevant related work.

First, a network-processor architecture with several small parallel pipelines is described in [1]. However, this architecture does not deal with dedicated cores, and also does not study our optimization problems. In addition, network processors may adopt additional alternative architectural models. For instance, the multipass NP architectural model is suggested in [4] as an alternative to pipelining. As mentioned above, while such a model may have less synchronization issues than pipeline-based models, it may also provide fewer guarantees of service.

Furthermore, several past papers have considered the problem of mapping the pipelines onto the chip cores in order



(a) Three pipelines with 8 cores (without pipeline sharing). The average delay is $T \approx 2.67$ time slots.

(b) Example of pipeline sharing: Two pipelines with 6 cores. The average delay is $T \approx 3.33$ time slots.

(c) Tradeoff between the number of cores and the average delay, defining a capacity region for the problem.

Fig. 1. Illustration of *pipeline sharing*.

to reduce energy consumption and bandwidth utilization [5]–[7]. 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.

Last, pipeline scheduling was discussed in [8]. This work deals with a simpler problem in which the cores are not configurable and each of them can perform a single predetermined task.

C. Contributions

In this paper we introduce the *pipeline sharing* problem 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 would like to reduce the average length of these pipelines.

We first present an optimal algorithm that applies when the set of required tasks for the i^{th} packet type includes the first X_i tasks out of some r tasks. We show how to recursively calculate the optimal solution for the first packet types. Based on this observation, we suggest a polynomial dynamic-programming algorithm that achieves the minimal possible delay.

Then, we describe general properties of the problem. In particular, we suggest bounds on the optimal delay as a function of the number of cores. We also calculate the exact number of possible solutions as a function of the number of packet types.

Later, we suggest a greedy algorithm for the general case. This algorithm merges pairs of pipelines, with cores servicing common tasks, to reduce the number of cores. For each possible pair of pipelines it examines the expected increase in the delay and the possible reduction in the number of cores.

We also provide detailed experimental results to examine the efficiency of the concept of pipeline sharing in general and in particular of the suggested algorithm. We first conduct experiments based on synthetic data. Later, we also examine examples such as a packet-processing application and the features of the H.264 video-compression standard. We show that in both cases the greedy algorithm obtains a delay that is often close to the optimal delay.

II. Model and Problem Definition

Let's start by introducing notations and formally defining the problem.

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 of belonging to type i, and packets of type i need to perform a fixed set of tasks S_i. For instance, an incoming packet may either be of type 1 with probability $p_1 = 1/2$ and require tasks $S_1 = \{1, 2, 3\}$; or it may be of type 2 with probability $p_2 = 1/2$ and require tasks $S_2 = \{1, 4\}$.

B. Pipeline Sharing

We are now interested in studying how the set of cores in our 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. 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. It 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 task set $S_2 = \{1, 4\}$ 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 Problem

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.

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 packet delay.

Formally, we state the 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.

III. A SIMPLE CASE OF THE REQUIRED TASKS: $S_i = [1, X_i]$

Finding an optimal solution of the problem might be difficult in the general case. In this section, we consider a property of the sets of tasks required by the different packet types $S_1, \cdots, S_k$ that when satisfied *we can 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 $S_i = \{1, \cdots, X_i\} = [1, X_i]$ for $X_i \in [1, r]$. For instance, consider the following three packet types. First, parsing is performed on all types. Two types require also L3 routing and egress resolution. In addition, access control rules have to be executed on packets of one among these two types.

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 will later enable us to suggest an efficient algorithm for finding an optimal solution. There might be more than a single optimal solution. We will try to obtain one of them.

In the rest of the section, we assume that $S_1 = [1, X_1], \cdots, S_k = [1, X_k] \subseteq [1, r]$ are ordered such that $X_a \leq X_b$ if $a < b$. We then have that $S_1 \subseteq S_2 \subseteq \cdots \subseteq S_{k-1} \subseteq S_k$. Indeed, we can show that 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. $B_i \in [1, d]$.

The properties are summarized in the following propositions.

Proposition 1. *The pipelines in an optimal solution $Q_1, \cdots, Q_d$ satisfy $(\forall j \in [1, d])\ Q_j \in \{S_1, \cdots, S_k\}$, 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, $(\forall j \in [1, d])\ Q_j$ is of the form $Q_j = [1, Y_j]$ for $Y_j \in [1, r]$.*

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 tasks of these packet types such that $i_1 < i_2 < i_{h-1} < i_h$. 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. $\bigcup_{m=1}^{h} S_{i_m} = Q_j$. This is because otherwise, cores of the pipeline could be eliminated to reduce the delay. By the assumption that $S_i = [1, X_i]$ for $X_i \in [1, r]$ and that $X_a \leq X_b$ if $a < b$, we have that $Q_j = \bigcup_{m=1}^{h} S_{i_m} = S_{i_h} = [1, X_{i_h}]$. ∎

Proposition 2. *Assume that $Q_1, \cdots, Q_d$ are ordered such that $Q_1 \subsetneqq Q_2 \subsetneqq \cdots \subsetneqq Q_{d-1} \subsetneqq Q_d$. 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 from the input. In addition, the last packet type is served by the last pipeline, i.e. the k^{th} packet type is served by $Q_{B_k} = Q_d = S_k$.*

Proof: First, such an order of $Q_1, \cdots, Q_k$ exists according to Proposition 1. Assume that the claim does not hold 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 $S_i \subseteq S_j \subseteq Q_{B_j} \subsetneqq Q_{B_i}$ and in particular $S_i \subseteq Q_{B_j}$. We can let packet type i to be served by Q_{B_j} that satisfies $|Q_{B_j}| < |Q_{B_i}|$ and reduce the average delay. A contradiction to the optimality of the solution. Finally, with the last proposition if $B_i = B_j$ (for $i < j$) we must have that $B_m = B_i = B_j$ for all $m \in [i, j]$. Finally, the set of tasks S_k is served by one of the pipelines $Q_1, \cdots, Q_d$. By Proposition 1, this pipeline has the form of $S_i \in \{S_1, \cdots, S_k\}$. The only option that satisfies $S_k \subseteq S_i$ is for $i = k$. Thus $Q_{B_k} = S_k$. By the order of $Q_1, \cdots, Q_d$ we must have that $Q_d = S_k$. ∎

Following this proposition, we would like to 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 average delay that can be achieved in serving the first i packet types with an architecture of at most n cores. If no such solution exists we define $T^i(n) = \infty$. Likewise, we denote by $B^i(n)$ (for the same possible 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 Proposition 2. We later add an example to clarify these notations. 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).

Proposition 3. (i) *For $i \geq 1$, 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 value of $T^i(n)$ in its formula. Then, there*

exists an optimal solution that satisfies

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

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

where $\cdot$ denotes the vector concatenation operation.

Proof: We first explain (i). We consider an optimal solution for the first i packet types. By Proposition 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 proposition, 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.

We now explain (ii). 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 $|Q^i(n)|$. ∎

Finally, we describe the suggested dynamic-programming algorithm. 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 $n \geq 0$. We continue to calculate in step i (for $i \in [1, k]$) the values of $T^i(n)$ for $n \in [0, N]$ according the formulas presented in Proposition 3 based on the values calculated in the previous steps. 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 all 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 we calculate the $(N + 1)$ values of optimal delays for $n \in [0, N]$. For each value, we take the minimal value out of at most k possible values. Thus the time complexity is $O(k^2 \cdot N)$.

Example 1. *Assume an input of $k = 3, N = 8$ and $(S_1, p_1), (S_2, p_2), (S_3, p_3) = ([1], 0.3), ([1, 3], 0.2), ([1, 5], 0.5)$. We would like to find a solution with at most $N = 8$ cores that minimizes the delay. We can easily see that $T^0(3) = 0, T^1(3) = 0+0.3\cdot 1 = 0.3$. Likewise, by Proposition 3 $T^2(3) = \min(T^1(0) + 0.2 \cdot 3, T^0(0) + 0.5 \cdot 3) = 1.5$.*

In addition, by the same proposition, we have that the optimal average delay satisfies $T_{OPT}(N) = T^k(N) = T^3(8) = \min(T^2(3) + 0.5 \cdot 5, T^1(3) + 0.7 \cdot 5, T^0(3) + 1 \cdot 5) = \min(1.5 + 2.5, 0.3 + 3.5, 5) = 3.8$. The minimal value among the three is the second value obtained for the value of $j = 2$ in the recursive formula from Proposition 3. The value $j = 2$ means that in an optimal solution the last two packet types are served by the last pipeline $S_3 = [1, 5]$. With this value of j, we deduce $Q^3(8) = Q^1(3) \bigcup \{[1, 5]\}$ and $B^3(8) = B^1(3) \cdot (3, 3)$. In addition, we can simply see that $Q^1(3) = \{[1]\}$, $B^1(3) = (1)$. Finally, we can obtain that $Q^k(N) = Q^3(8) = \{[1], [1, 5]\}$ and $B^k(N) = B^3(8) = (1, 3, 3)$.

IV. General Properties

We now describe some basic properties of the general problem, formally described earlier in Section II. We first present bounds on the optimal delay as a function of the number of cores. We also calculate the number of solutions for the optimization problem, i.e. the number of options for sharing pipelines. Considering these options might be a technique to find the optimal delay. We also explain that in some cases the problem can be partitioned into two smaller subproblems. These properties can be used by a designer to understand the limits of the chip design capacity region.

Proposition 4. (i) *For all $N \geq 0$, the optimal average delay $T_{OPT}(N)$ satisfies $T_{OPT}(N) \geq \sum_{i=1}^{k} \left(p_i \cdot |S_i|\right)$*
(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) *$T_{OPT}(N)$ satisfies $T_{OPT}(N) = \infty$ for $N < |\bigcup_{i=1}^{k} S_i|$.*

Proof: First, in any solution each packet type (S_i, p_i) is served by a pipeline Q_j that satisfies $S_i \subseteq Q_j$. Thus $|S_i| \leq |Q_j|$, the delay of the packet is $|Q_j|$ and the packet type (with probability p_i) contributes at least $p_i \cdot |S_i|$ to the average delay.

In addition, if the number of cores is at least the sum of the set sizes in the input, each packet type can be served separately and the minimal possible delay can be obtained.

Last, if $N < |\bigcup_{i=1}^{k} S_i|$ at least one of the packet types cannot be served since one of its tasks is not implemented by any of the N available cores. ∎

Proposition 5. *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 the additional pipeline with smaller length and reduce the average delay. ∎

Proposition 6. *Let k be the number of packet types. Then,*
(i) *Given an unlimited number of cores, the number of solutions with d pipelines $Q_1, \cdots, Q_d$ is given by $S(k, d)$, where $S(k, d) = \frac{1}{d!} \cdot \sum_{j=1}^{d} \left((-1)^{d-j} \cdot \binom{d}{j} \cdot j^k\right)$, is the Stirling number of the second kind of k, d.*
(ii) *The total number of solutions is given by $G(k) = \sum_{d=1}^{k} S(k, d)$.*

Proof: A solution with d pipelines can be described as a partition of the set of k distinct packet types into d non-empty pipelines $Q_1, \cdots, Q_d$. Each pipeline is simply the union of the corresponding sets of tasks in the corresponding packet types.

It cannot contain any additional redundant cores. The number of such possible partitions is given by $S(k,d)$.

In addition, since the number of pipelines in a solution can be any number in the range $[1,k]$, the total number of solutions is given by $G(k)$. ∎

Example 2. *Consider the example from Section I with $k=3$ packet types. The number of solutions with $d=1$ pipelines is $S(3,1)=\frac{1}{d!}\cdot\sum_{j=1}^{d}\left((-1)^{d-j}\cdot\binom{d}{j}\cdot j^k\right)=1\cdot\left((-1)^{1-1}\cdot\binom{1}{1}\cdot 1^3\right)=1$. Likewise, there are $S(3,2)=3$ solutions with 2 pipelines and another single $(S(3,3)=1)$ solution with 3 pipelines. The total number of solutions is $G(3)=\sum_{d=1}^{3}S(3,d)=1+3+1=5$. (Incidentally, in this example with its specific parameters, two of the solutions yield exactly the same tradeoff.)*

Proposition 7. *Assume that the set of tasks of the k packet types can be partitioned into two disjoint sets, i.e. they can be ordered such that $(\exists m\in[1,k])\ \left(\bigcup_{i=1}^{m}S_i\right)\bigcap\left(\bigcup_{i=(m+1)}^{k}S_i\right)=\emptyset$. Then, $(\exists N_0\in[0,N])$ s.t. an optimal solution given N cores can be obtained as the union of the two sets of pipelines in the optimal solutions for packet types $[1,m]$ with N_0 cores and for packet types $[(m+1),k]$ with $(N-N_0)$ cores.*

Proof: Any pipeline in an optimal solution cannot serve tasks from both sets $\left(\bigcup_{i=1}^{m}S_i\right),\left(\bigcup_{i=(m+1)}^{k}S_i\right)$. Otherwise, it could be partitioned into two smaller pipelines to reduce the average delay. By dividing the pipelines in an optimal solution into two subsets based on this property, we obtain the result. ∎

V. Pipeline Merging Algorithm

We would like to suggest an efficient greedy algorithm for reducing the average packet delay for a general input. 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 our 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 our multi-core chip (or when there is no clear solution).

In other words, consider a given iteration of the algorithm. Assume that its pipelines currently use n cores. If $n\le N$, 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 expected increment in the average delay if these pipelines are merged. Further let y be the corresponding decreased number of cores. Intuitively, we would like to merge the pair of pipelines that minimizes the ratio x/y.

We would like now to 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 valid 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 (ratio 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 expected 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 possible *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 possible *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.

Example 3. *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 applications for which the suggested (greedy) algorithm results in a delay that equals, in most cases, the minimal possible delay. Unfortunately, this algorithm is not necessarily optimal in the general case.

Proposition 8. *The greedy algorithm that successively merges at each step the pair of pipelines with the minimal ratio $R=x/y$ is not optimal in the general case.*

Proof: We present the following counterexample. Let $r=14$, $(S_1,p_1),(S_2,p_2),(S_3,p_3)=(\{1,2,3,4,5,6,7,8,9\},1/3)$, $(\{1,2\},1/3)$, $(\{1,2,3,10,11,12,13,14\},1/3)$ and let $N=4^2=16=9+2+8-3=|S_1|+|S_2|+|S_3|-3$.

Here, $R_{1,2}=\Big((1/3)\cdot 0+(1/3)\cdot 7\Big)/2=7/6$. Likewise, $R_{1,3}=\Big((1/3)\cdot 5+(1/3)\cdot 6\Big)/3=11/9$ and $R_{2,3}=\Big((1/3)\cdot 6+(1/3)\cdot 0\Big)/2=1$. Thus the minimal ratio is achieved for the pair pipelines serving packet types 2 and 3. If we first merge

this pair, the obtained number of cores is $|S_1| + |S_2| + |S_3| - |S_2 \cap S_3| = 9 + 2 + 8 - 2 = 17 > N$. Thus if this pair of pipelines is merged, the obtained number of cores is larger than their upper bound. Therefore, an additional merging operation with the remaining first pipeline is required. Finally, a single pipeline with $r = 14$ cores is achieved and the average delay is $T = (1/3) \cdot 14 + (1/3) \cdot 14 + (1/3) \cdot 14 = 14$.

The optimal solution for this case includes two pipelines. The first $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10 \to 11 \to 12 \to 13 \to 14$ with 14 cores serving the first and the third packet types. In addition, a second pipeline $1 \to 2$ with two cores serving packet of the second type. The obtained average delay is only $T = (1/3) \cdot 14 + (1/3) \cdot 2 + (1/3) \cdot 14 = 10$. ■

VI. Experimental Results

We would like now to conduct experiments to examine the efficiency of the suggested greedy algorithm from Section V for the general case in comparison with an optimal solution. In the experiments, we first rely on synthetic examples. In addition, we consider applications from diverse fields such as packet processing and video compression.

A. Effectiveness on Synthetic Examples

We first perform synthetic simulations and compare the performance of the greedy algorithm and the minimal possible delay obtained by considering all possible solutions. In both of the two following experiments, the sets of required tasks are created synthetically. In these experiments, we assume 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 w.p. $q = 0.5$, without any dependency between the different types and the different tasks, i.e. $(\forall i \in [1, k], j \in [1, r]) \Pr(j \in S_i) = q = 0.5$. Fig. 2 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 a *fixed probability* of $\frac{1}{k} = 0.125$. For a given set of types and a maximal value of N cores, we compare the delay obtained by the 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. 2. For example, for large enough values of N, each type has its own pipeline and the delay is approximately the average number of tasks per type, $r \cdot q = 10 \cdot 0.5 = 5$. In general, the 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 greedy delay is 8.47, larger by 4.4% than the optimal delay of 8.11. This option suggests a reduction of $(60 - 16)/60 = 73.3\%$ in the number of cores

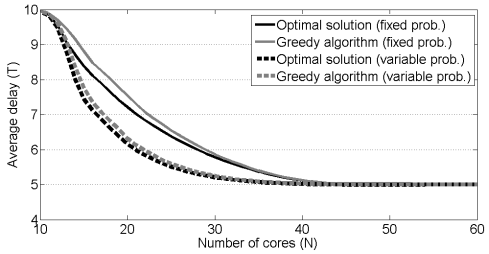


Fig. 2. Average delay (in time slots) as a function of the number of available cores (N) for the first experiment with synthetic data. Here, the probability for a packet type to require each task is 0.5. The two upper curves present the delay when the $k = 8$ types appear with a uniform distribution. 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.

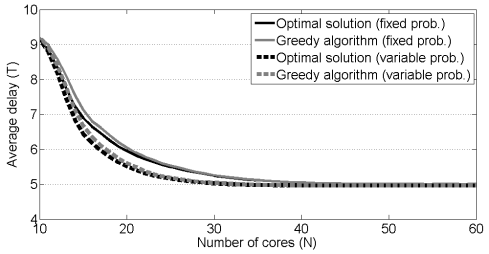


Fig. 3. The average delay (in time slots) as a function of the number of available cores (N) for the second experiment with synthetic data. Here, the probability for a packet type to require each task is either 0.9 or 0.1, according to one of three predetermined distributions. The two upper curves present 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.

with a cost of a delay larger by 70% or 62.8%. For $N = 40$ the delay of the greedy is 5.12 when the minimal possible delay (5.11) is smaller by less than 0.2%. The average difference, for $N \in [10, 60]$, between the two delays is 0.10 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 $2^{-1}, 2^{-2}, 2^{-3}, 2^{-4}, 2^{-5}, 2^{-6}, 2^{-7}, 2^{-7}$. These non-homogenous probabilities enable us to distinguish between the types to further improve the obtained 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. 2. For instance, again for $N = 16$ the observed delays are 7.35 and 7.12, respectively. Both delays are smaller 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 delay is even smaller and equals 0.07 time slots.

We also want to check whether a possible dependency between the different types can further improve the effectiveness

TABLE I
SUMMARY OF THE SYNTHETIC EXPERIMENTS

(A) Average delay (in time slots)

Experiment	Fixed Prob.		Variable Prob.	
	Greedy	Optimal	Greedy	Optimal
(i) Independent types	6.20	6.09	5.80	5.73
(ii) Distribution-based types	5.65	5.60	5.47	5.42

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

Experiment	Fixed Prob.		Variable Prob.	
	Greedy	Optimal	Greedy	Optimal
(i) Independent types	8.47	8.11	7.35	7.12
(ii) Distribution-based types	6.80	6.65	6.36	6.17

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. 3.

Informally, we expect two types selected from the same distribution to have relatively similar sets of tasks. Therefore, a possible merging operation of their pipelines is expected to result in an additional delay which is smaller than on average. We consider the same two options for the probabilities for the different types as in the first experiment (fixed and geometrically decreasing). In the first subcase, with fixed probabilities for the k types, illustrated in two upper curves, the average (over $N \in [10, 60]$) of the optimal delay is 5.60 in comparison with a corresponding average delay of 6.09 in the first experiment. Likewise, for $N = 16$ the greedy delay is 6.80 and the optimal delay is 6.65. In the second subcase, with non-homogenous probabilities, as shown in the two last curves, the average optimal delay is even smaller and equals 5.42. Here, for $N = 16$ the delay of the greedy algorithm is 6.36 and of the optimal equals 6.17 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.

B. Effectiveness on a Packet-Processing Application

We now consider an example of packet processing. Typically L2-L4 packets require a subset of the following $r = 11$ tasks:

1. Parsing - parsing the header stack, classifying its parts to specific layers and extracting the relevant fields in each layer.

2. Ingress interface attributes - assigning different attributes related to the ingress interface (port/VLAN).

3. Ingress ACL - executing access control rules on the incoming packet.

TABLE II
PACKET-PROCESSING APPLICATION: PACKET TYPES WITH THEIR REQUIRED TASKS (AS DETAILED IN SECTION VI-B)

	packet type	tasks
1	L2 unicast packet	$S_1 = \{1, 2, 4, 10\}$
2	L2 unicast packet with security control	$S_2 = \{1, 2, 3, 4, 10, 11\}$
3	L3 multicast packet	$S_3 = \{1, 2, 5, 6, 8, 9, 10\}$
4	MPLS packet	$S_4 = \{1, 2, 3, 7, 8, 10, 11\}$
5	Packet trapped to the CPU	$S_5 = \{1, 2, 3, 10\}$

4. L2 bridging - bridging based on MAC addresses and VLAN.

5. L3 routing - L3 forwarding based on IP addresses.

6. L3 replication - replicating packets to different subnets/hosts based on the router forwarding decision.

7. MPLS switching - forwarding packets based on MPLS header. Packets can be on and off LSP (Label Switching Path) as a result of the forwarding decision.

8. Header modification - modifying outgoing packet header based on forwarding (L2, L3, MPLS) decisions.

9. L2 replication - replicating packets based on the bridging forwarding decision.

10. Egress interface resolution - mapping the destination resolved by the forwarding decision to a specific interface/port in the device.

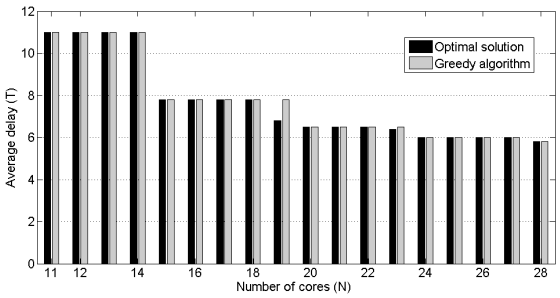
11. Egress ACL - executing access control rules on the outgoing packet.

Based on the $r = 11$ tasks from above, we can consider $k = 5$ packet types with their corresponding required sets of tasks as detailed in Table II. We assume that the probabilities for a packet to belong to each of the k types $\{1, \cdots, k\}$ are $(p_1, \cdots, p_k) = (0.25, 0.15, 0.2, 0.3, 0.1)$.

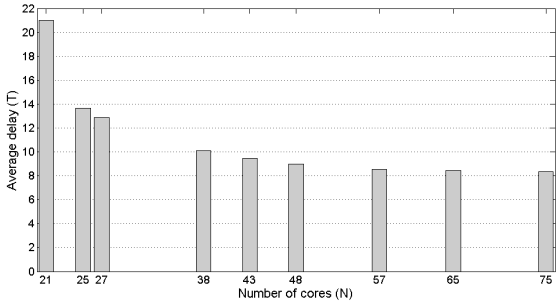
We examine how the average delay depends on the number of available cores. Since in this example the number of types $k = 5$ is relatively small, we can consider all the possible solutions and present the minimal possible delay given the number of cores. By Proposition 6, the number of solutions with $d \in [1, k = 5]$ pipelines is $S(k, d)$. For instance, there are $S(5, 2) = 15$ solutions with 2 pipelines. The total number of solutions is $G(5) = \sum_{d=1}^{5} S(5, d) = 1 + 15 + 25 + 10 + 1 = 52$.

The left black bar in Fig. 4(a) presents this minimal possible delay obtained by the exhaustive search. Given only $N = r = 11$ cores, only a single pipeline with all 11 tasks is possible and the average delay is also 11. When $N = 15$, a solution that minimizes the delay includes two pipelines. The first serves packet type 3 with $|S_3| = 7$ cores and the second serves all other types with $|S_1 \bigcup S_2 \bigcup S_4 \bigcup S_5| = |\{1, 2, 3, 4, 7, 8, 10, 11\}| = 8$ additional cores. Here, the obtained average delay is $T = p_3 \cdot 7 + (p_1 + p_2 + p_4 + p_5) \cdot 8 = 1.4 + 6.4 = 7.8$. Finally, if $N = 4 + 6 + 7 + 7 + 4 = 28$ we have a pipeline for each packet type and the average delay is $0.25 \cdot 4 + 0.15 \cdot 6 + 0.2 \cdot 7 + 0.3 \cdot 7 + 0.1 \cdot 4 = 5.8$. Here, by using pipeline sharing, we can significantly reduce the number of cores by 46% (from 28 to 15) with a cost of an increase of 34% in the average delay (7.8 in comparison with 5.8).

In addition, the right gray bar describes the behavior of the greedy algorithm. We can see that, in this example, it



(a) Packet processing



(b) *H.264 video-compression* standard

Fig. 4. The average delay (in time slots) as a function of the number of cores (N). In Fig. 4(a) we consider the packet-processing example. The delay obtained by the greedy algorithm equals in most cases the minimal possible delay. Fig. 4(b) shows the delay obtained by the greedy algorithm for $k = 9$ profiles of the *H.264 video-compression* standard. The optimal algorithm was too complex to run in that case.

obtains for most of the values of N the minimal possible delay. Since the greedy algorithm performs a series of pipeline merging operations, the number of cores in its solutions are not necessarily consecutive. Here, the greedy algorithm first merges the pipelines of packet types 2 and 5, is reducing the number of cores from 28 to 24 and increasing the average delay from 5.8 to 6. The next step is to merge the last recently achieved pipeline with the pipeline of packet type 1. A delay of 6.5 is obtained with 20 cores. Unlike the exhaustive search the greedy algorithm does not tackle the optimal solution with 19 cores that achieves a delay of 6.8. Another minor difference is achieved when $N = 23$. Finally, after performing four merging operations, a single pipeline of 11 cores is again obtained.

C. Effectiveness on a Video-Compression Application

Next, motivated by studies that discussed implementations of video compression in a multi-core chip [9], we consider a real-life application of our study for the *H.264 video-compression* standard [10]. The standard defines several profiles that support different sets of features. We consider $k = 9$ of the more important profiles and refer to the features that each of them supports as tasks required in the processing of a video of that profile. We consider the profiles CBP, BP, XP, MP, ProHiP, HiP, Hi10P, Hi422P and Hi444PP and assume they are uniformly distributed. The profiles support a total number of $r = 21$ features such as Flexible Macroblock Ordering (FMO), Interlaced Coding and CABAC Entropy Coding. Some of the profiles support different variants of Chroma formats $(4:2:0, 4:2:2, 4:4:4)$ and different sample depths $(8, 10, 12, 14)$ and we consider them as different features.

For this use-case, with a larger number of $k = 9$ profiles, we present only the results of the greedy algorithm. Since the number of solutions is relatively large ($G(9) = 21147$), we could not perform an exhaustive search. The 9 profiles include a total of 75 features (i.e. an average of 8.33 features per profile, resulting in an average delay of 8.33). Therefore, 75 cores are required to obtain the minimal possible average delay of 8.33 time slots. As shown in Fig. 4(b), an average delay of 10.11 is obtained when only 38 cores are used. Here, comparing the last two mentioned points, the average delay increases by 21% when the number of cores is roughly halved from 75 to 38.

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 average delay and described general properties of the problem. We suggested an optimal algorithm for a special case of the input and a greedy algorithm for the general case. Finally, we presented experimental results that demonstrated that the greedy algorithm often achieves an average delay that is close to optimal. Finding the optimal solution in the general case and determining when the problem is NP-hard are left as open questions for future work.

VIII. Acknowledgment

This work was partly supported by the Andrew Viterbi graduate fellowship, by the Google European Doctoral fellowship, by the Intel ICRI-CI Center, by the Japan Technion Society and Greenberg (Ottawa) Research Funds, and by the Israel Ministry of Science and Technology. Part of this work was done when Ori Rottenstreich was with Marvell Israel.

References

[1] K. Karras, T. Wild, and A. Herkersdorf, "A folded pipeline network processor architecture for 100 Gbit/s networks," in *ANCS*, 2010.

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

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

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

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

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

[7] C. A. M. Marcon *et al.*, "Exploring NoC mapping strategies: An energy and timing aware technique," in *DATE*, 2005.

[8] 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.

[9] A. Azevedo *et al.*, "Parallel H.264 decoding on an embedded multicore processor," in *HiPEAC*, 2009.

[10] G. J. Sullivan, P. N. Topiwala, and A. Luthra, "The H. 264/AVC advanced video coding standard: Overview and introduction to the fidelity range extensions," in *Optical Science and Technology, the SPIE 49th Annual Meeting*. International Society for Optics and Photonics, 2004.