IEICE TRANS. INF. & SYST., VOL.E101–D, NO.9 SEPTEMBER 2018

PAPER

Proxy Responses by FPGA-Based Switch for MapReduce Stragglers

Koya MITSUZUKA[†a)], ***Nonmember***, **Michihiro KOIBUCHI**[††], ***Senior Member***, **Hideharu AMANO**[†], ***Fellow***, ***and*** **Hiroki MATSUTANI**[†], ***Member***

SUMMARY In parallel processing applications, a few worker nodes called "stragglers", which execute their tasks significantly slower than other tasks, increase the execution time of the job. In this paper, we propose a network switch based straggler handling system to mitigate the burden of the compute nodes. We also propose how to offload detecting stragglers and computing their results in the network switch with no additional communications between worker nodes. We introduce some approximate techniques for the proxy computation and response at the switch; thus our switch is called "ApproxSW." As a result of a simulation experiment, the proposed approximation based on task similarity achieves the best accuracy in terms of quality of generated Map outputs. We also analyze how to suppress unnecessary proxy computation by the ApproxSW. We implement ApproxSW on NetFPGA-SUME board that has four 10Gbit Ethernet (10GbE) interfaces and a Virtex-7 FPGA. Experimental results shows that the ApproxSW functions do not degrade the original 10GbE switch performance.

key words: *FPGA, MapReduce, straggler*

1. Introduction

As the data sets grow rapidly in size, parallel processing frameworks such as MapReduce [1] are becoming more important. MapReduce consists of two types of nodes: worker nodes (i.e., compute nodes) that perform parallelized tasks and a master node that manages the entire system including worker nodes. The parallel processing performance is often limited by only a few worker nodes that process given tasks with low performance due to machine troubles and/or excessive workloads. These workers are called stragglers.

Although various techniques have been invented to handle the stragglers, they mostly impose a burden on master node to monitor the progress of all the worker nodes. In Backup Task [1], for example, when a straggler is detected, the delayed task is assigned to worker node that has been completing its task faster than the others. Backup Task that reruns delayed tasks on fast worker nodes can always return correct results even with stragglers. The master node is in charge of monitoring all the worker nodes for Backup Task. However, as the number of worker nodes increases, the management overhead of the master node increases, resulting in a new performance bottleneck in massively parallel processing. Because such management tasks by the master node are reluctantly introduced for the fault tolerance, spending a lot of CPU times for the management is a waste of CPU resources. If these resources are effectively used for computation, the performance will be improved. Please note that the ratio of delayed tasks over all the tasks is quite small in the case of a large degree of parallelism [2]. Depending on applications, especially for those that do not require exact results, additional costs for Backup Task is not justified. Although replication of master nodes can distribute the management overhead, a more efficient approach is required.

As an alternative approach, in this paper, we propose to move such straggler management burden from master node to network switch that connects the master and worker nodes. More specifically, the proposed network switch monitors output packets from Map tasks to detect stragglers. When detected, the proposed switch generates a response instead of the straggler based on the outputs of the other Map tasks, so that Reduce tasks can be started without delay*. Network traffic for straggler management is mitigated because network switch is the nearest device to stragglers. We introduce some approximate techniques for the proxy computation and response at the switch; thus our switch is called "ApproxSW." We implement ApproxSW on NetFPGA-SUME board that has four 10Gbit Ethernet (10GbE) interfaces and a Virtex-7 FPGA, and demonstrate that the ApproxSW functions do not degrade the original 10GbE switch performance.

The rest of this paper is organized as follows. Section 2 overviews related work. Section 3 proposes ApproxSW architecture and Sect. 4 illustrates its implementation on NetFPGA-SUME board. Section 5 shows experimental results and Sect. 6 concludes this paper.

2. Related Work

2.1 MapReduce Processing Flow

MapReduce [1] is one of the most widely-used parallel processing framework that processes large data sets with number of compute nodes. We illustrate the MapReduce processing flow which is a push-based implementation as

Manuscript received September 9, 2017.
Manuscript revised May 5, 2018.
Manuscript publicized June 15, 2018.

[†]The authors are with Graduate School of Science and Technology, Keio University, Yokohama-shi, 223–8522 Japan.
[††]The author is with National Institute of Informatics, Tokyo, 101–8430 Japan.
a) E-mail: koya@arc.ics.keio.ac.jp

DOI: 10.1587/transinf.2017EDP7287

*An early stage of this work has been accepted as a poster in FPL 2017 [3]. This paper is an extended version of [3].

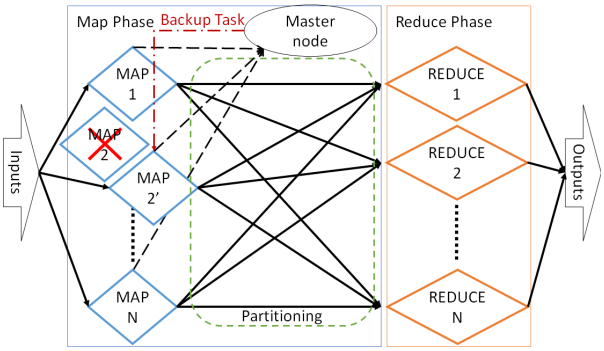

Fig. 1 A behavior of MapReduce framework

shown in Fig. 1 for simplicity but our approach is applicable to a pull-based implementation.

The cluster consists of a master node and worker nodes. The master node assigns Map tasks to the worker nodes when a job is started. The number of Map tasks should be less than or equal to the number of workers so that all the tasks can be executed in parallel. The master node divides input files into fixed-size chunks and feeds them to the worker nodes evenly. Outputs of Map tasks are formed as key-value pairs and will be sent to appropriate Reducer nodes through the network during the Map phase. The way to partition the Map outputs can be defined by the user. For example, a hashed value of each key in the output is used to determine the destination Reducer node as in Apache Hadoop. Each Map task notifies the master node when it completes its assigned task. The proposed ApproxSW monitors such output key-value pairs and completion notifications from the Map tasks in order to deal with stragglers. After receiving completion notifications from all the Map tasks, the master node assigns Reduce tasks to the workers.

During the Reduce phase, worker nodes process Map outputs received in the Map phase and generate the final results. A MapReduce job is completed when the master node receives completion notifications from all the Reduce tasks.

2.2 Backup Task

Backup Task [1] is the conventional solution for the straggler problem in MapReduce framework. With Backup Task, the master node monitors all the task progress and detects delayed ones. Then the master node reschedules them speculatively to faster workers. If the rerunning tasks overtake the original tasks and are completed faster, waiting time for the delayed tasks can be eliminated. Sophisticated scheduling algorithms for Backup Task have been proposed in [4], [5]. They focus on detecting stragglers as early and correctly as possible and thus they impose more burden on the master node to collect more detailed progress report from worker nodes. A distributed scheduling algorithm for large-scale clusters is also proposed in [6]. However, it does not discuss how to monitor the tasks in parallel in detail.

Although Backup Task can obtain accurate results while mitigating the effect of stragglers, it imposes more burden on the master node and network to monitor progress of a number of worker nodes. In addition, speculative rerunning consumes extra power and compute resources. Our ApproxSW is completely different from these prior works but is a natural approach because all the information goes through the switch. A prototype of ApproxSW is implemented on NetFPGA-SUME board.

2.3 MapReduce Acceleration by Approximation

Approximate computing improves the compute performance and the energy efficiency in exchange for acceptable degradation on computation accuracy [7], [8]. ApproxHadoop [9] adopts an approximation technique that consists of input data sampling and task dropping in MapReduce. The input data sampling computes a partial result based on a part of input data and estimates the entire result based on the partial result. Task dropping reduces the computation cost and execution time by dropping a part of tasks. Especially, tasks that take longer time frames to complete compared to the others and those that have not been started are dropped, in order to reduce the execution time. Our ApproxSW also employs the dropping technique in the network switch and implements it on the NetFPGA-SUME board.

2.4 MapReduce Acceleration by FPGA

FPGA-based acceleration for various computations improves both the performance and energy efficiency. Especially, commonly-used operations for a wide range of applications have been offloaded to FPGA-based accelerators. For example, generating key-value pairs in Map phase, their sorting based on keys in Map phase, and merging data in Reduce phase are offloaded to FPGA-based accelerators in [10]. Application-specific computations in Map and Reduce phases are also offloaded to FPGA-based accelerators. High-level synthesis is used to implement such computation tasks on FPGAs [11]. As our ApproxSW focuses on the management burden of master node for straggler handling, it is orthogonal to these FPGA-based accelerations on Map and Reduce phases and can be combined with them to further improve the performance.

3. Design

3.1 Motivation

The master node is in charge of 1) task scheduling that decides which worker node executes a given task and 2) task monitoring to detect stragglers. As the number of worker nodes increases, monitoring all the tasks becomes a burden. One approach to improve the scalability is to employ multiple master nodes. A simple solution for the multiple master nodes is parallel monitoring, in which each master node is in charge of scheduling and monitoring of a part of worker nodes. Worker nodes can be monitored by multiple master nodes so that fault tolerance of master nodes can be ad-

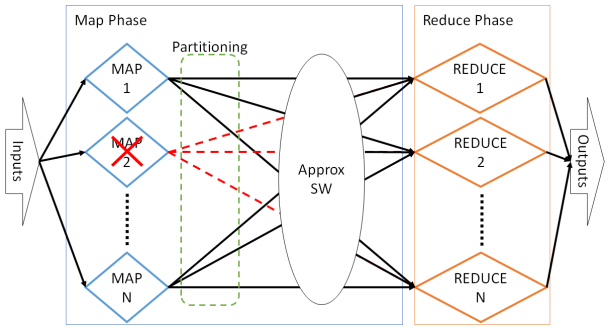

Fig. 2 Behavior of MapReduce with ApproxSW

dressed. In this case, however, each worker node must know where to send its progress report and also the network workload increases due to duplicated progress report messages. On the other hand, in the case of a large degree of parallelism, the number of stragglers and thus tasks assigned to such stragglers are quite limited since tasks are more likely assigned to fast worker nodes. In this case, dropping the tasks executed on stragglers may not severely impact the final results. This approach is called "Drop" in ApproxSW.

3.2 ApproxSW Overview

This paper proposes ApproxSW, a network switch based solution for the straggler problem to eliminate a burden of master/worker nodes to handle stragglers. Figure 2 shows the ApproxSW overview. In general, a straggler solution consists of two stages: detection and proxy response. A simple network switch based solution is to just detect delayed tasks and request the master node to reschedule them as in Backup Task. On the other hand, ApproxSW generates proxy responses instead of detected stragglers; thus no computation and communication overheads are imposed in any master/worker nodes to handle stragglers. Although there are Map and Reduce tasks, in this paper we focus on stragglers of Map phase for the proxy response by a network switch.

3.3 Straggler Detection at Network Switch

ApproxSW monitors Map outputs to check the progress of each task and detect stragglers. More specifically, ApproxSW counts the number of key-value pairs from each Map task, and the task whose counter value is less than θ (Slow Task Threshold) from the average is detected as a delayed one. This assumes that each Map task processes almost the same number of keys. The other cases can be also handled by adjusting θ appropriately. The delays are detected and made up for each time when the packets arrive at the network switch from Map tasks. Please note that the proposed ApproxSW can work correctly if Combiner function is used in the Map phase. Since Combiner function aggregates the Map task results within the Map phase, in this case ApproxSW is modified to monitor the Combiner outputs to detect stragglers.

3.4 Proxy Response at Network Switch

ApproxSW does not notify the master node about the detected stragglers but generates proxy responses instead of the stragglers. That is, ApproxSW is in charge of the proxy computations and completion notifications instead of delayed tasks. Completion notification is to inform a completion of a task to master node for starting the next phase. Generating it by proxy omits the waiting time due to stragglers, and thus it is mandatory to handle stragglers and improve the performance. However, it introduces some uncompleted tasks and degrades the accuracy of the final results. Here, we propose proxy computation for the delayed tasks to compensate the negative impact on accuracy.

3.4.1 Proxy Computation

The proxy computation for stragglers by a network switch is not a trivial job. It is difficult for network switches to access input files and process them accordingly to complete the computation. In the case of a large degree of parallelism and quite small number of stragglers, such costs cannot be justified. Therefore, we adopt an approximate computing for the proxy computation to strike a balance between a burden for handling stragglers and the accuracy of the final results.

A unique characteristic of network switch based solutions is that the outputs from the other tasks are available at a network switch since the outputs go through the switch. We thus propose to use the outputs from the other tasks for proxy computation. A simple implementation of this approach is to replicate the outputs of the other tasks. This method can generate data which have a possibility of being generated without considering the application and the input dataset. In this paper we introduce the following three proxy computation methods.

- Drop method: Completion notifications are sent by proxy but no proxy computations are performed.
- Random method: In addition to proxy completion notifications, proxy computations that copy the normal task outputs in the probability of $\frac{1}{n}$ are performed, where n is the number of Map tasks.
- Similarity method: In addition to proxy completion notifications, proxy computations that copy the normal task outputs in the probability calculated by Eq. (1) and Algorithm 1 are performed.

In the following, the Similarity method, the most sophisticated one among them, is introduced.

ApproxSW utilizes Map outputs sent by the other workers connected to the switch in order to generate similar outputs of delayed tasks. Since it is not feasible to store all the Map outputs in the limited memory capacity of the network switch, we propose to use only the recently-arrived Map outputs (called "new data") and statistical information based on all the past Map outputs. Every time ApproxSW

Algorithm 1 Similarity counting function for proxy computation

```
a ⇐ task sending new data
key ⇐ key included in new data
for all the other tasks i do
  if i has sent key and a sent key for the first time then
    S[a][i] ⇐ S[a][i] + 1
    S[i][a] ⇐ S[i][a] + 1
  end if
end for
```

receives new data, it updates the statistical information and then generates outputs of delayed tasks by proxy. A similarity between tasks is used as the statistical information. We employ a modified version of cosine similarity as shown in Algorithm 1. Cosine similarity is used to measure a similarity between two documents [12]. Similarity counters of two tasks are incremented when the two tasks output the same key during a certain time window. If some specific keys are sent many times by almost all the tasks (e.g., "the" and "and" for word counting), the similarities between these tasks become uniformly high and the accuracy of the proxy computation becomes worse. To avoid this, the increment of similarity is done only once for the same key.

After updating the similarity, ApproxSW performs a proxy computation for each delayed task based on the output of the normal task which has a high similarity to the delayed task. Please note that delayed tasks can be detected by the method illustrated in Sect. 3.3. In other words, the proposed ApproxSW assumes that the outputs of a delayed task are similar to those of other tasks that have a high similarity to the delayed task. The similar data are selected from the outputs of normal tasks based on a probability determined by the degree of their similarity. That is, ApproxSW copies outputs of a similar task with the probability P in order to perform a proxy computation for the delayed task. Equation (1) shows how to calculate P based on the similarity S.

$$P = \frac{{S_{ab}}^4}{\sum_{i=0}^{n-1} {S_{ai}}^4}, \tag{1}$$

where n is the number of tasks, a is the delayed task, b is the other task, and S_{ab} is their similarity corresponding to $S[a][b]$ in Algorithm 1. P is a value obtained by normalizing the similarity. We empirically use the fourth power of similarity as the evaluation function, in order to filter out task pairs with low similarities since every task pair has a low similarity due to frequently appearing words, such as "a" and "the."

ApproxSW sends the proxy computation results to appropriate destination Reducer nodes as soon as the computation is completed. In this approach, the proxy computations of delayed tasks are performed in parallel with Map tasks so that they do not increase the total execution time.

3.4.2 Completion Notification

If ApproxSW has not received completion notifications from all the tasks yet, it sends the completion notifications instead of delayed tasks when a pre-determined time has passed since it received the first completion notification. After sending completion notifications by proxy, ApproxSW communicates with delayed worker nodes to terminate their delayed Map tasks.

4. Implementation

We employ the NetFPGA-SUME board as an FPGA-based switch that has four 10GbE interfaces. NetFPGA-SUME Reference Switch Lite design [13] is used as a baseline 10GbE switch, and we implement our ApproxSW to handle stragglers by extending the baseline switch.

Figure 3 shows the ApproxSW implementation overview. Packets from remote machines via four 10GbE interfaces (MAC RX0-RX3) and host machine via PCIe (PCIe DMA) are merged into a single packet stream by Input Arbiter module. Output Port Lookup module selects the output port for each incoming packet and adds the routing information. According to the information, Output Queues module delivers the packets to one of the four 10GbE interfaces (MAC TX0-TX3) or the host machine via PCIe (PCIe DMA). ApproxSW module is inserted between Output Port Lookup and Output Queues modules to monitor and generate the packets for proxy response.

4.1 Network Switch Part

A packet stream goes through the switch in 256-bit per cycle based on the implementation of AXI (Advanced eXtensible Interface) on the Reference Switch Lite. When the network switch receives a packet, the packet is classified into a Map output, a completion notification, or the other application packet. In ApproxSW, UDP packets destined to specific port numbers are identified as the Map outputs or completion notifications. Figure 4 shows their packet formats. A payload of these packets consists of *type, id, key,* and *value* fields.

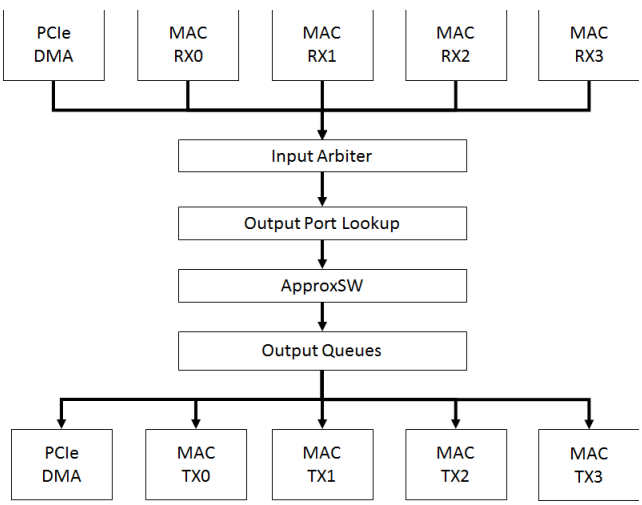

Fig. 3 Entire hardware architecture overview implemented on FPGA

The type field is used to identify a packet as either a Map output or a completion notification. The id field is used to identify the task. A Map output consists of a pair of key and value. As shown in Fig. 4, the second part of a packet (i.e., 256-bit to 511-bit) is processed in the second cycle. The second part contains all the information needed for updating the similarities which are also required for the proxy calculation.

The network switch provides a dedicated hash table to each task. A hash table manages whether each key has been sent from a task. That is, a hashed value of a key is used for an index of the table where the flag (sent or not) of the key is stored. When a Map phase starts, all the flag values of the hash tables are invalidated. When the network switch receives a packet that includes a key, the hash table for the task that sent the packet is updated and the flag value is validated. To simplify reseting all the values, the network switch changes the value which indicates "sent" so that the old "sent" values are invalidated.

The bit width of the hash table index is 15-bit and thus the number of hash table entries is 32,678. If more hardware resources are available, the hash table size can be extended in order to reduce the possibility of hash collisions. A hash collision causes an accidental increase in the similarity (if words on different tasks conflict) or an accidental decrease (if words on the same task conflict and both the words appear in another task) by one word. If the number of unique keys is far smaller than the hash table size, hash collisions rarely occur and the penalty would be small.

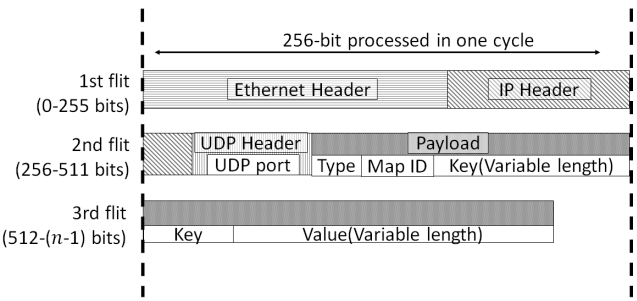

Fig. 4 Packet format sent by worker nodes

Please note that when ApproxSW receives the other packets, such as ARP (Address Resolution Protocol) requests and replies, they are simply passed to the original L2 processing module in the network switch as regular packets.

4.2 Straggler Handling Part

Figure 5 shows the control logic of ApproxSW. When the network switch receives a packet, the packet is classified according to its IP protocol, destination UDP port, and type fields. If it is a UDP packet which goes to a dedicated port, it is classified into a Map output or a completion notification according to its type field. In the other cases, it is classified into the others. If it is a Map output, the count of key-value pairs given by the task corresponding to the id field is incremented. The similarities between the sender task and all the other tasks are also updated. Then, the replication judgement described in this section is done to detect the delay at the time and determine which tasks require the replication. According to the result of the replication judgement, the packet is replicated for the delayed tasks. If the packet is a completion notification and the Complete Buffer has not been written yet, the packet is replicated to the Complete Buffer for proxy response of completion notifications. In all the cases, the original packet is passed to the original L2 switch module. When a predetermined time has passed since the first completion notification was received, proxy responses of the completion notifications are performed.

Figure 6 illustrates a packet passing flow of ApproxSW. All packets are processed when they arrive at the network switch. A received packet is first buffered in the Input FIFO

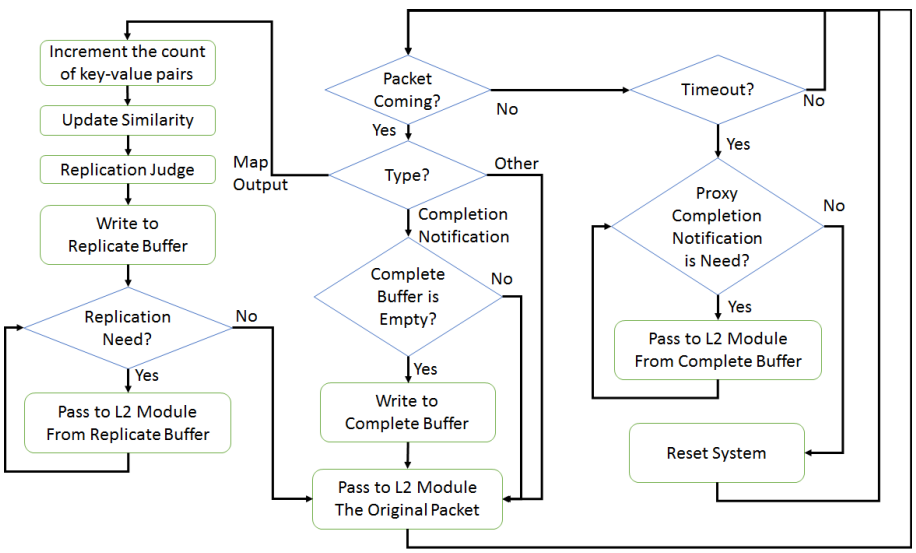

Fig. 5 Packet control logic of ApproxSW

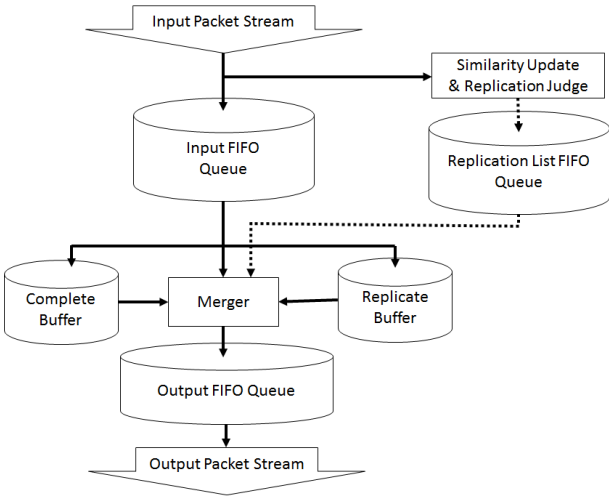

Fig. 6 Packet passing flow in ApproxSW

Queue and, according to its type field, classified into three types: Map output, Completion notification, and the others. If it is an output from a Map task, the task similarities and the Map output amounts are updated based on its id and key fields. Also, based on their Map output amounts as illustrated in Sect. 3.3. Equation (2) is a condition for detecting the delayed tasks.

$$(x + \theta) < \frac{y}{n} \tag{2}$$

n is the number of tasks, x is the number of occurrences of a key given by each task, y is the total sum of x for all the tasks, and θ is Slow Task Threshold mentioned in Sect. 3.3. θ is a constant value given by the user. The division by the number of Map tasks is implemented with a right shift operation based on an assumption that the number is a power of two.

As mentioned in Sect. 3.4.1, a received packet is replicated for a delayed task with a probability P. More specifically, the received packet is replicated when the following relation is satisfied.

$$P = \frac{{S_{ab}}^4}{\sum_{i=0}^{n-1} {S_{ai}}^4} > r \tag{3}$$

n is the number of tasks, a is a delayed task, b is the task that sent new data, and S_{ab} is their similarity. r is a random number in range $[0, 1)$. For ease of implementation, r is generated by using a 7-bit counter value c incremented in every cycle; thus $r = c/2^7$. To eliminate the division operation in Relation 3, it is transformed as follows. That is, the packet is replicated when the following relation is satisfied.

$$2^7 \times {S_{ab}}^4 > c \times \sum_{i=0}^{n-1} {S_{ai}}^4 \tag{4}$$

Figure 7 shows a pipelined processing for updating the similarities and detecting the delayed tasks for which the proxy computation is required. The similarity of each task

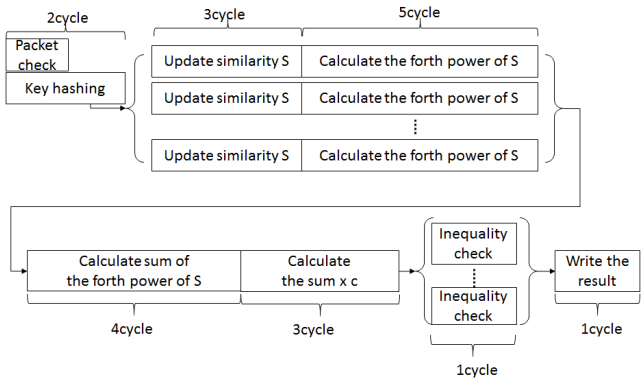

Fig. 7 Pipeline for similarity updating and Map output replication

is updated and its fourth power is calculated in parallel. The pipeline processing takes 20 cycles for each packet. The depth of the 256-bit FIFO queue is more than 20 (e.g., 32) in order to buffer all the input data coming during the 20 cycles.

After the replication judgement in the pipeline, the packet is removed from the Input FIFO Queue and passed to the Output FIFO Queue. If the result of the replication judgement indicates that the packet is a Map output which will be replicated for one or more delayed tasks, it is replicated and stored in a Replicate Buffer. Then the proxy computation is done for the delayed tasks. More specifically, a Map output is copied from the Replicate Buffer and then it is modified so that its id field is changed to one of the delayed tasks and passed to the Output FIFO Queue. These steps are performed for each of the delayed tasks to which the Map output is replicated. If the packet is a completion notification, it is also replicated and stored in a completion buffer. When a predetermined time has passed since the first completion notification was received, a proxy response of the completion notification is performed for all the delayed tasks detected. A completion notification is copied from the Complete Buffer and then it is modified so that its id field is changed to one of the delayed tasks and passed to the Output FIFO Queue. These steps are performed for all the delayed tasks. The packets in the Output FIFO Queue are continuously removed and passed to the original L2 switch module.

5. Evaluations

5.1 Accuracy of Proxy Computation

We simulate ApproxSW in terms of the accuracy of proxy computation on the final result. Figure 8 shows the evaluation setting. On this simulation, Map tasks and a Reduce task are executed as processes on a Linux machine (CentOS 6.8). The Reduce process also works as the role of collecting completion notifications for simplicity. An ApproxSW process provides the functions of straggler detection and proxy response and is executed on the same machine. Map outputs and completion notifications are sent from the Map processes to the Reduce process through the

ApproxSW process. The processes perform the socket communication with each other using local loopback address. When the ApproxSW process generates proxy completion notifications, it sends signals to the delayed Map processes in order to terminate them. Word count is employed as a target application in the experiments. In the word count workload, a Map output is a key-value pair where the key is a word and the value is always 1 (e.g., "apple, 1"). We used three input datasets listed in Table 1 and evaluate the three proxy computation methods: Drop, Random, and Similarity methods. Each dataset has 16 files. Dataset1 has 16 identical files. Dataset2 has 8 pairs of 2 identical files. Dataset3 has 16 different files. In this setting, all the tasks have high similarities to each other in Dataset1, each task has a high similarity to another task in Dataset2, and all the tasks have low similarities to each other in Dataset3. Each file size included in the datasets is approximately 10,000 Bytes. Each Map task is in charge of a single file called a chunk. The amount is small compared to popular MapReduce use cases, e.g., a default chunk size of Hadoop is 128MB [14]. If input data becomes larger while the hash table size is the same, more hash collisions in the ApproxSW will occur because the number of unique keys in the word count application is increased. Thus, larger datasets can be applied to ApproxSW in response to available FPGA resources to implement larger hash tables. In Table 1, "Proportion of words causing hash collisions" shows ratio of words which caused hash collisions to all the unique words in the datasets when the bit width of the hash table index is 15-bit, 20-bit, and 24-bit, respectively. We implemented the hash tables whose index width is 15-bit in order to meet the timing constraint of 200MHz. When the bit width is 24-bit, no hash collisions occur in these datasets.

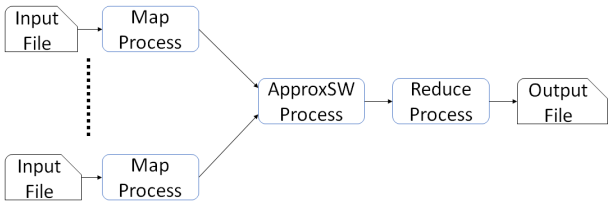

Fig. 8 Overview of the simulation for accuracy evaluation

5.1.1 Accuracy vs. Proxy Computation Methods

In this experiment, the number of Map tasks is set to 16 and the number of stragglers is set to 2. The processing times for a single word in healthy and delayed tasks are 1,000 μsec and 2,000,000 μsec, respectively. First, we executed the job without any stragglers to obtain the perfect results. Then, we executed the same job with proxy computation methods and stragglers. Finally, we compared the latter results to the former ones and calculated the accuracy.

Figure 9 shows the accuracy of proxy computations in ApproxSW with two proxy methods: Random and Similarity methods. The left bar at Similarity method shows the result with hash tables whose index width is 15-bit and the right bar shows that with hash tables whose index width is 24-bit. In these figures, "error" and "correct" represent the wrong and correct results generated by the proxy computation, respectively. In Figs. 9 (a), 9 (b) and 9 (c) the total numbers of proxy computations are almost the same in all the datasets because the probabilities are normalized and the probability distribution among the tasks does not directly affect the number of proxy responses. Both the Random and Similarity methods work well for Dataset1, in which all the tasks have the largest similarity. Furthermore, the Similarity method achieves a high accuracy for Dataset2, in which at least a single pair of tasks has a high similarity as shown in Fig. 9 (b). Such situations would be more likely in real workloads where at least a single pair of tasks has a high similarity. For example, when a file is split into multiple chunks, tasks which process one of the chunks may have high similarity to each other. On the other hand, when all

Table 1 Three datasets

	File configuration	Total number of words	Words per task	Proportion of words causing hash collisions		
				15-bit index	20-bit index	24-bit index
Dataset1	1 file × 16	25,803	1,612	0.67%	0.00%	0.00%
Dataset2	8 files × 2	24,859	1,554	8.33%	0.47%	0.00%
Dataset3	16 files × 1	24,850	1,553	15.44%	5.09%	0.00%

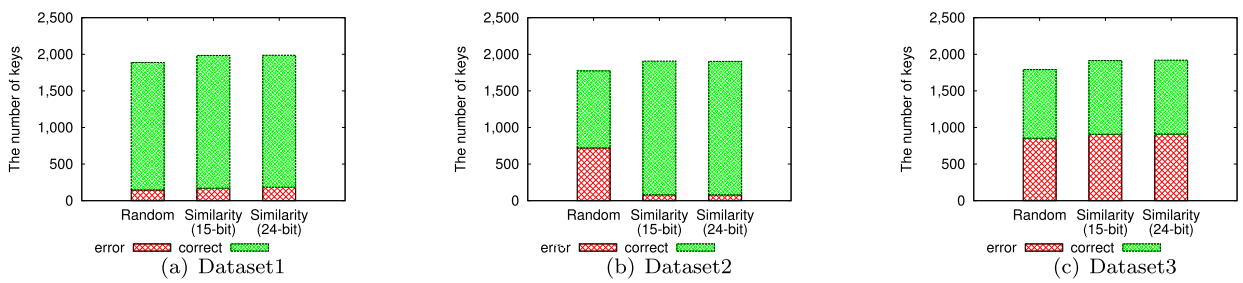

Fig. 9 Accuracy of proxy computation results only

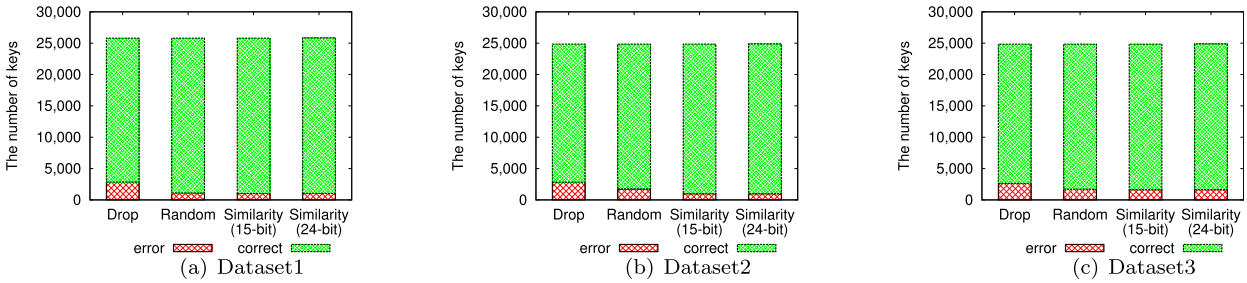

Fig. 10 Accuracy of total results including proxy computation results

the tasks have uniform similarity at all, the Random method is better because of simplicity. Such situations occur when the input data has no locality. For Dataset3, both the Random and Similarity methods achieve almost the same accuracy because the input data has no locality. Compared to the result of Dataset1, the accuracy of Dataset3 becomes lower because the input file for each task in Dataset3 has lower similarities to each other. In summary, our findings are as follows. In terms of the accuracy, Similarity method is equal to or better than Random method. If the input data has a strong locality (similarities are non-uniform) like Dataset2, Similarity method can find out the similar task and improve the accuracy of proxy computation. On the contrast, in Dataset2 and Dataset3, the accuracy of Random method is not improved due to a few similar tasks as shown in Figs. 9 (b) and 9 (c). In the specific situation where all the tasks have a uniform similarity like Dataset1 and Dataset3, the accuracy of both methods becomes almost the same. The reason why the accuracies of these methods are not exactly the same for Dataset1 and Dataset3 is that the Similarity method estimates the similarities incrementally and thus the probability slightly fluctuates during a job execution unlike Random method. In these datasets, the hash collisions do not impact significantly to the accuracy as shown in Fig. 9.

Figure 10 shows the accuracy of the entire computation results including not only proxy responses but also outputs from the tasks. The left bar at Similarity method shows the result with hash tables whose index width is 15-bit and the right bar shows that with hash tables whose index width is 24-bit. Please note that these figures include entire results of Map tasks while Figure 9 accounts only for the results generated by proxy computation. In these figures, "correct" represents the results which are matched to the correct results and "error" represents the results which are necessary but not generated by the proxy computation or unnecessary but generated by the proxy computation. As shown in Fig. 10, negative impacts on the proxy computation in terms of accuracy are small because the proportion of stragglers is small.

5.1.2 Accuracy vs. Slow Task Threshold

Here we discuss the relationship between accuracy and Slow Task Threshold parameter, which is given by a user as mentioned in Sect. 3.3. Slow Task Threshold determines the sensitibility to the delay and affects the amount of the proxy computation. If the Slow Task Threshold parameter is too small, the system becomes too sensitive to the delay and tends to make unnecessary proxy responses which introduce low accuracy. On the other hand, the parameter is too large, the system does not detect any delay and make no proxy responses like the Drop method. Again the number of tasks is set to 16 and the number of stragglers is set to 2. We use hash tables whose index has 24-bit width in this experiment. To investigate the effect of different delay distributions (uniform and non-uniform), we evaluate the following two cases: homogeneous environment and heterogeneous environment. In the homogeneous environment, all the tasks except for stragglers process words at the same speed. The processing times are the same as the previous evaluation in the homogeneous environment. In the heterogeneous environment, the processing times are added by (Task ID $\times$ 200) μsec so that non-uniform delay distributions are imposed. Slow Task Threshold parameter is varied from 0 to 1,000 Dataset2 is used as the input data.

Figure 11 shows the results when the Random and Similarity proxy methods are used, respectively. In these figures, the left and right bars show the results in the homogeneous environment and the heterogeneous environment for each threshold, respectively. "error" and "correct" represent the wrong and correct results generated by the proxy computation, respectively. In the homogeneous environment, the probability to incorrectly detect a healthy task as a straggler is small no matter how small Slow Task Threshold parameter is. In this environment, thus the Slow Task Threshold parameter does not degrade the accuracy. In the heterogeneous environment, if Slow Task Threshold parameter is improperly low, a few slightly-slow healthy tasks, which do not require proxy responses, may be detected as stragglers and generate unnecessary proxy responses. This is the reason the error ratio becomes high when θ is too low in the heterogeneous environment. On the other hand, when the threshold is high enough, the proxy computations can be suppressed. We can tune the Slow Task Threshold properly according to the situation (i.e., application, environment, input data size, etc) to obtain good accuracy. For example, in this environment, when θ is equal to 300, the number of errors by the excessive proxy responses is reduced while keeping the number of correct proxy responses as shown in Fig. 11.

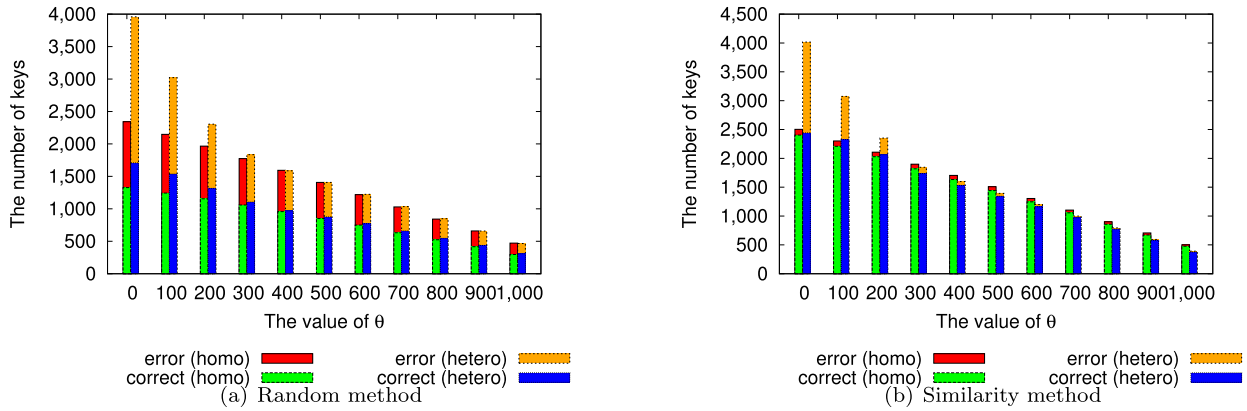

Fig. 11 Accuracy vs. Slow Task Threshold

Table 2 FPGA utilization of the implementation

Module	LUTs	BRAMs	DSPs	GTHE2_CHANNELs
System Signal	15	0	0	0
PCIe Controller	17,794	74	0	8
10GbE Interface (x4 in total)	25,736	58	0	4
Input Arbitor	2,328	30	0	0
Output Port Lookup	1,104	6	0	0
Output Queues	2,554	22.5	0	0
ApproxSW Core	29,394	274	1,344	0
Total	78,974	468.5	1,344	12
Available	433,200	1,470	3,600	36

Table 3 FPGA utilization detail of the ApproxSW Core module

Module	LUTs	BRAMs	DSPs
Complete Buffer	265	6	0
Replicate Buffer	742	6	0
Replication List FIFO	385	0	0
Hash Table (x16 in total)	384	256	0
$sum \times r$ (x16 in total)	288	0	64
$similarity^4$ (x256 in total)	0	0	1,280
Input FIFO Queue	1,166	6	0
ApproxSW Core Total	29,394	274	1,344

5.2 FPGA Utilization

Table 2 and Table 3 show the resource utilization of ApproxSW that implements the Similarity method. The design tool is Xilinx Vivado Design Suite 2014.4. The target FPGA device is Virtex-7 XC7VX690T. The bit width of a hashed value is 15-bit and thus the number of hash table entries is 32,768. A similarity is represented as a 16-bit value. The depth of the 256-bit FIFO queue is 32. The number of managed tasks is 16. The operating frequency is 200MHz.

5.3 Throughput

We evaluate the throughput of ApproxSW by using Open Source Network Tester (OSNT) [15] on NetFPGA-10G board [13]. We assume word count as an application in the experiment. Map and Reduce tasks of the word count application are executed as CPU processes and their communications go through the network switch. Based on this assumption, word count traffic is generated by the packet generators in order to measure the maximum throughput. The OSNT packet generator is implemented on Machine B and C (see Table 4) to generate 10Gbps packet streams. ApproxSW, which is based on Reference Switch Lite offered by NetFPGA project [13], is implemented on Machine A (see Table 4). Figure 12 shows the experimental settings. ApproxSW has four 10GbE interfaces and two OSNT packet generators are connected to two ports each. Machines B

Table 4 Machines used for measuring throughput

	CPU	OS	NIC
MachineA	Intel Core i5-3470S (2.9GHz, 6GB, 4cores)	Ubuntu 14.04 LTS	NetFPGA-SUME
MachineB	Intel Core i5-4460 (3.2GHz, 8GB, 4cores)	CentOS 6.7	NetFPGA-10G
MachineC	Intel Core i5-3470S (2.9GHz, 8GB, 4cores)	CentOS 6.8	NetFPGA-10G

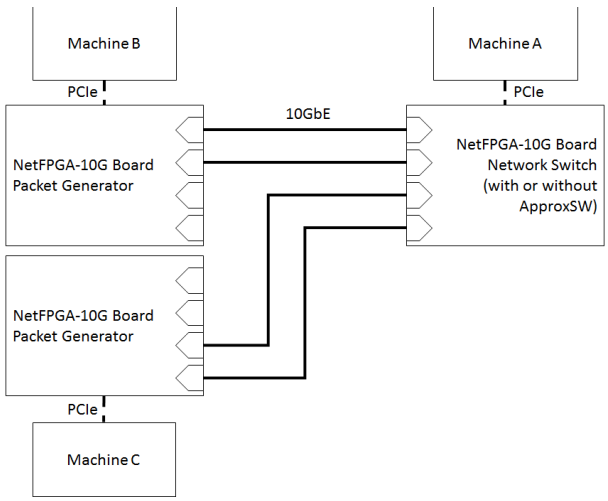

Fig. 12 Throughput evaluation settings

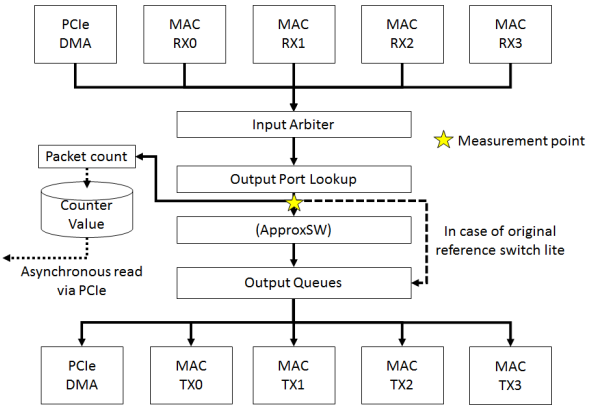

Fig. 13 Packet counting module

and C set the generated packet into the packet generator and make it start generating the packet stream via PCIe. While the packet generators are running, the NetFPGA-SUME board transfers the received packets and counts their number. To count the number of incoming packets to measure the throughput, the counting logic is added just after the Output Port Lookup module. Figure 13 shows the measurement point of throughput. Machine A reads the counter value of incoming packets asynchronously via PCIe in every 500msec to measure the throughput. We measured the throughputs of the original Reference Switch Lite and ApproxSW ten times and calculate the average values. To verify that the packet size does not make the performance worse, we measured the throughputs with two packet sizes, 512-bit and 1024-bit. When the packet size is set to 512-bit, the original Reference Switch Lite processes packets at 28.92Gbps and ApproxSW achieves 28.95Gbps. When the packet size is 1,024-bit, the original Reference Switch Lite achieves 33.49Gbps and ApproxSW achieves at 33.51Gbps. As shown, there are no significant differences between the throughputs of the original Reference Switch Lite and ApproxSW. Thus, performance overhead of ApproxSW is negligible.

6. Conclusions

To eliminate the burden to handle stragglers by the master node in MapReduce applications, in this paper we proposed ApproxSW which is a network switch based straggler detection and proxy computation mechanism, because all the information goes through the switch. Thus, by introducing ApproxSW, the master node no longer has to monitor the progress of each task and detect stragglers. Also, fast workers no longer have to rerun delayed tasks speculatively as in Backup Task. However, the proxy computation for delayed tasks by a network switch is not a trivial job due to its limited resource; thus, we adopted an approximate proxy computation that replicates Map outputs of healthy tasks which have a high similarity to the delayed tasks. Based on information from all the tasks, the ApproxSW can estimate how many proxy computations should be done, what data should be generated, and when the Map phase should be finished without no dedicated communications or rerunning tasks. We described how to implement the mechanism as a packet processing engine of a network switch. The proxy computations of delayed tasks are performed in parallel with Map tasks so that it does not increase the total execution time. ApproxSW was implemented on NetFPGA-SUME board that has Xilinx Virtex-7 FPGA and four 10GbE interfaces. It achieved the same performance as the original Reference Switch Lite.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplied Data Processing on Large Clusters," Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI'04), pp.137–149, Dec. 2004.
[2] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica, "The Case for Tiny Tasks in Compute Clusters," Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS'14), May 2013.
[3] K. Mitsuzuka, A. Hayashi, M. Koibuchi, H. Amano, and H. Matsutani, "In-Switch Approximate Processing: Delayed Tasks Management for MapReduce Applications," Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL'17), Sept. 2017. http://www.arc.ics.keio.ac.jp/~matutani/mitsuzuka_fpl2017.pdf
[4] M. Zaharia, A. Konwinski, A.D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environment," Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI'08), pp.29–42, Dec. 2008.
[5] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the Outliers in MapReduce Clusters using Mantri," Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI'10), pp.1–16, Oct. 2010.
[6] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, Low Latency Scheduling," Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'13), pp.69–84, Nov. 2013.
[7] M. Shafique, W. Ahmad, R. Hafiz, and J. Henkel, "A Low Latency Generic Accuracy Configurable Adder," Proceedings of the Annual Design Automation Conference (DAC'15), pp.86:1–86:6, June 2015.
[8] M. Shafique, R. Hafiz, S. Rehman, W. El-Harouni, and J. Henkel, "Invited - Cross-layer Approximate Computing: From Logic to Architectures," Proceedings of the Annual Design Automation Conference (DAC'16), pp.1–6, June 2016.
[9] I. Goiri, R. Bianchini, S. Nagarakatte, and T.D. Nguyen, "ApproxHadoop: Bringing Approximations to MapReduce Frameworks," Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15), pp.383–397, March 2015.
[10] T. Honjo and K. Oikawa, "Hardware Acceleration of Hadoop MapReduce," Proceedings of the International Conference on Big Data (BigData'13), pp.118–124, Oct. 2013.
[11] D. Diamantopoulos and C. Kachris, "High-level Synthesizable Dataflow MapReduce Accelerator for FPGA-coupled Data Centers," Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), pp.26–33, July 2015.
[12] G. Salton, Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer, Addison-Wesley, 1989.
[13] "The NetFPGA Project." http://netfpga.org/

[14] “The Apache Hadoop Project.” http://hadoop.apache.org/
[15] “OSNT 10G Home.” https://github.com/NetFPGA/OSNT-Public/wiki/OSNT-10G-Home

Koya Mitsuzuka received the BE degree from Keio University in 2017. He is currently a master course student in Keio University.

Michihiro Koibuchi received the BE, ME, and PhD degrees from Keio University, Yokohama, Japan, in 2000, 2002 and 2003, respectively. Currently, he is an associate professor in the Information Systems Architecture Research Division, National Institute of Informatics and the Graduate University of Advanced Studies, Tokyo, Japan. His research interests include the area of high-performance computing and interconnection networks. He is a member of the IEEE and a senior member of IEICE and IPSJ.

Hideharu Amano received the PhD degree from the Department of Electronic Engineering, Keio University, Japan in 1986. He is currently a professor in the Department of Information and Computer Science, Keio University. His research interests include the area of parallel architectures and reconfigurable systems.

Hiroki Matsutani received the BA, ME, and PhD degrees from Keio University in 2004, 2006, and 2008, respectively. He is currently an associate professor in the Department of Information and Computer Science, Keio University. His research interests include the areas of computer architecture and interconnection networks.