sustainability

MDPI

Article

An Adaptive Offloading Method for an IoT-Cloud Converged Virtual Machine System Using a Hybrid Deep Neural Network

Yunsik Son [1,†], Junho Jeong [1,†] and YangSun Lee [2,*]

[1] Department of Computer Engineering, Dongguk University, Seoul 04620, Korea; sonbug@dongguk.edu (Y.S.); yanyenli@dongguk.edu (J.J.)
[2] Department of Computer Engineering, Seokyeong University, Seoul 02713, Korea
* Correspondence: yslee@skuniv.ac.kr; Tel.: +82-2-940-7743
† These authors contributed equally to this work.

Received: 28 September 2018; Accepted: 29 October 2018; Published: 30 October 2018

Abstract: A virtual machine with a conventional offloading scheme transmits and receives all context information to maintain program consistency during communication between local environments and the cloud server environment. Most overhead costs incurred during offloading are proportional to the size of the context information transmitted over the network. Therefore, the existing context information synchronization structure transmits context information that is not required for job execution when offloading, which increases the overhead costs of transmitting context information in low-performance Internet-of-Things (IoT) devices. In addition, the optimal offloading point should be determined by checking the server's CPU usage and network quality. In this study, we propose a context management method and estimation method for CPU load using a hybrid deep neural network on a cloud-based offloading service that extracts contexts that require synchronization through static profiling and estimation. The proposed adaptive offloading method reduces network communication overheads and determines the optimal offloading time for low-computing-powered IoT devices and variable server performance. Using experiments, we verify that the proposed learning-based prediction method effectively estimates the CPU load model for IoT devices and can adaptively apply offloading according to the load of the server.

Keywords: Internet of Things; cloud system; offloading; virtual machine; static profiler; context information; deep neural network

1. Introduction

In the developing IT environment, one challenge with the appearance of each new device is that a separate development language and environment are required. One of the solutions to this problem is to use a virtual machine (VM). However, in Internet-of-Things (IoT) environments with limited resources, it is difficult to apply this to existing virtual machines directly. Various studies on lightweight VMs have been conducted to solve this problem [1–3]. In addition, there is research on VMs for supporting Web environments such as HTML5 and JavaScript in IoT devices in addition to simply overcoming development environments [4,5]. The Secure compiler, used for developing secure IoT services based on secure software and virtual machines for services, has also been studied [6].

In recent years, resource requirements for services to be processed by IoT devices have exceeded IoT devices' performance improvements; hence, it has been difficult to process services directly from IoT devices, and IoT–Cloud environments have been studied [7,8].

The IoT–Cloud converged virtual machine system [9] can perform tasks requiring high computing power by delegating tasks that are difficult to perform in the low-performance IoT equipment

development environment to the Cloud server environment by applying an offloading technique. The context information of the work environment must be transmitted and synchronized to maintain the consistency of programs during communication between the Cloud environment and the local environment. Therefore, the VM to which the existing offloading technique is applied is a structure that transmits/receives all context information. The disadvantage of such a structure is that the amount of context information to be transmitted is increased by that which is not necessary for performing the task. Because most of the overhead costs that occur during offloading are proportional to the size of the context information transmitted over the network, these costs are increased in low-performance IoT devices.

The proposed method involves two aspects for effective offloading. First, to reduce the resource consumption rate in the context information transmitted from low-performance IoT equipment to the Cloud server, it is crucial to determine the context information required for the offloading performance function using a static profiler; thus, a context information synchronization method for sending/receiving context information was studied. A VM employing the improved context information synchronization technique reduces the overheads caused by the context information synchronization process of low-performance IoT devices, thereby increasing the offloading efficiency. Next, to determine the effective offloading time, we predict the CPU usage trend, which is one of the workload indices, through deep learning. The predicted CPU usage trend is indicative of future CPU usage information; therefore, it is an indicator of offloading execution decisions. Using the proposed method, it is possible to reduce the size of the data that needs to be synchronized between the local device and the server during offloading and determine an appropriate offloading point.

The remainder of this paper is organized as follows. In Section 2, we examine the IoT–Cloud converged VM system as an execution environment, as well as the features of an existing offloading scheme, context information, and deep learning model used in this study. In Section 3, we describe the proposed offloading method for IoT devices. In Section 4, the performance of the proposed method is verified using experiments. Finally, we present our conclusions in Section 5.

2. Related Research

2.1. IoT–Cloud Converged Virtual Machine System

The IoT–Cloud converged VM system provides the computing power of high-performance cloud servers to low-performance IoT devices using lightweight VMs, profilers, offloading techniques, and Just-in-time (JIT) compilers. This system can execute the contents written in various programming languages by applying the advantages of existing smart VMs [1,3,5] to low-performance IoT devices. Figure 1 shows the overall structure of the IoT–Cloud converged VM system. The core component is a light-weight VM, an IoT–Cloud VM, for building a platform-independent environment in low-performance IoT devices. The VM makes existing smart VMs lighter [1,3,5] to meet the requirements of low-performance IoT devices. The IoT–Cloud VM can be applied to high-performance IoT equipment by delegating high-complexity tasks to the high-performance Cloud server environment by applying the offloading technique [9].

2.2. Offloading

The offloading method is a Cloud computing method used to maximize the efficiency of resource consumption in the development environment of low-performance equipment such as IoT equipment [10–13]. The offloading technique can delegate high-complexity tasks to a high-performance Cloud server environment to be performed in a local environment, thereby reducing resource consumption due to employing low-performance equipment [14–16]. The advantage of offloading is that it can overcome low performance due to the high computational power of servers, but overhead costs are incurred during network communication between the local and server environments.

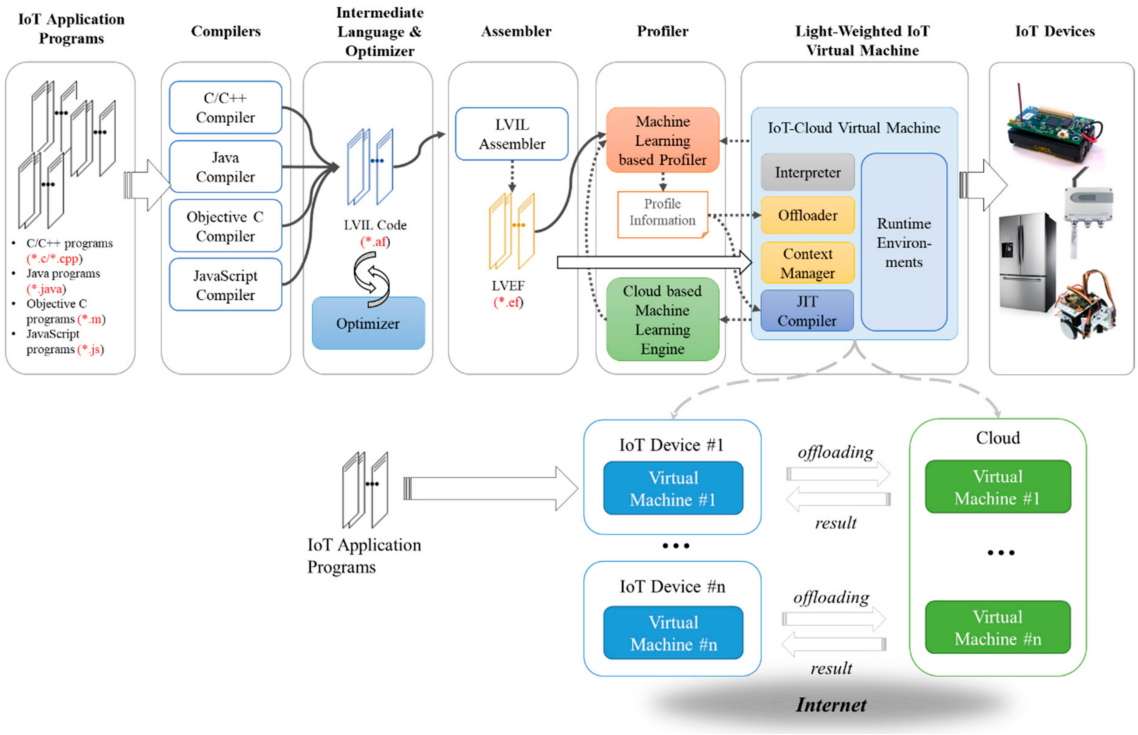

Figure 1. Overall configuration of the IoT–Cloud virtual machine system. IoT: Internet of Things.

If the overhead costs are greater than those incurred by operating in the local environment without performing offloading, the offloading performance deteriorates. Therefore, by analyzing the execution load information for each program element through profiling, the data to be offloaded must transmit and receive data with improved performance during offloading. A method of minimizing the size of data to be transmitted is required for efficient offloading because the overhead costs of network communication are proportional to the size of the transmitted and received data [17–20].

Methods for selecting computation offloading operations are largely classified into static and dynamic approaches. La et al. [16] classified offloading techniques as shown in Figure 2. The static technique reduces the execution load by selecting the part to be offloaded during program development. The static method has the advantage of a low load in terms of cost analysis at runtime. However, the cost analysis is possible only using predictable variables [17]. Meanwhile, the dynamic method selects the part to be offloaded by considering fluctuation factors during execution, such as the network state and remaining battery power. The dynamic method can accurately reflect the current state of the mobile device. Nevertheless, it is difficult to design a model that reflects all variables, and the required workload for the cost analysis is significant [18,19].

Partial offloading is a method of submitting some of the work to the Cloud. When a specific task is frequently used and cannot be performed in parallel, the communication costs and waiting times are increased. The full offloading method, on the other hand, addresses only the interaction with the user on the mobile device; it defers execution to the Cloud. For frequent interaction with a user, synchronization problems typically occur. Therefore, it is necessary to selectively assign an operation that is suitable for offloading to the Cloud.

In the proposed approach, it is difficult to reflect real-time fluctuating factors, such as mobility and communication scenarios. Therefore, the offloading decision is based on the mobile augmentation Cloud service (MACS) model [20], which estimates the code transmission cost, memory, and CPU usage for each function using the profiler at the content compilation time. The offloading object determines the unit of work based on the function.

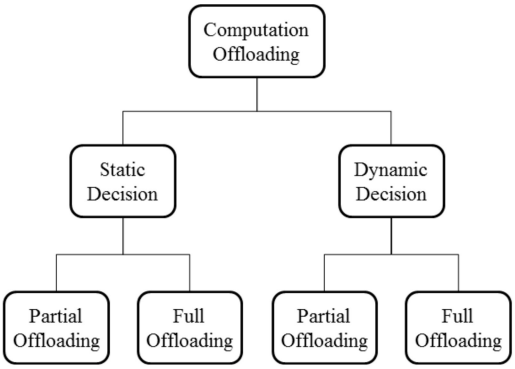

Figure 2. Classified computation offloading methods.

The IoT–Cloud converged VM system uses the computational offloading technique to provide the high-performance computing power of the Cloud server on low-performance IoT devices [9]. In this case, context synchronization must be performed because the program to be offloaded must maintain consistency between the IoT devices and the Cloud server. An efficient context synchronization technique is required because the network communication overheads linked to context synchronization can reduce the performance gain using offloading. In this study, we extract the context information required for offloading execution through static profiling for efficient context synchronization. The context synchronization method based on context information extracted through static profiling is expected to reduce network communication overheads because synchronization is only attempted on the context information necessary for the task to be offloaded.

2.3. Context Information

The context information is that required by the interpreter of the IoT–Cloud VM to execute the program. It is predominantly composed of the context information necessary for all function executions and the stack context information for managing command execution and variables. Figure 3 shows the composition of the core contextual information, which is required by all functions, and includes the instruction buffer, *gPc*, used to access the instruction buffer, *sp*, used to access the operation stack, *ep*, used to access the activation record, *SpDisplay*, used to manage the operation stack, and *ArDisplay*, used to manage the frame position information where the function starts in the activation record. The stack context information consists of an operation stack used when executing stacking or arithmetic instructions, an activation record for managing local variables, and a constant pool for managing literals and global variables. Figure 4 shows the composition of the stack context information. The operation stack terminates the function execution at the start position of the function frame, loads the return address and the lexical level of the function, and executes the operation instruction and the load instruction from the following area. The activation record, similar to the operation stack, loads the address to be returned and the lexical level of the function at the start of the function frame then subsequently manages the local variables used in the function.

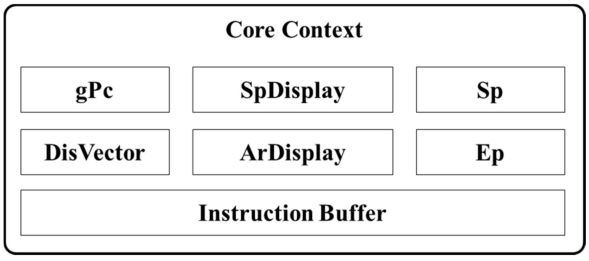

Figure 3. Core context information for executing processes on the IoT–Cloud virtual machine.

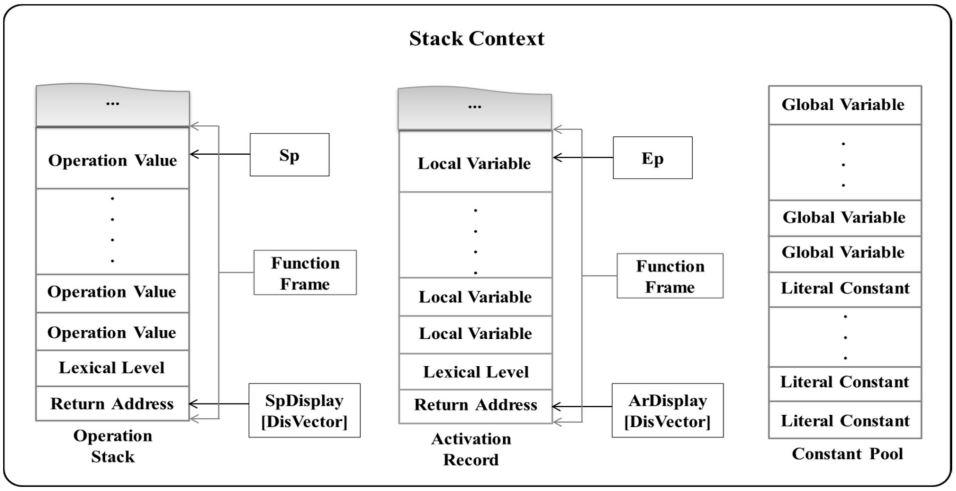

Figure 4. Context information for stack frames.

2.4. TreNet

TreNet is a neural network for the trend prediction of time series data. It is a hybrid neural network combining long short-term memory (LSTM), a convolutional neural network (CNN), and a feature fusion layer [21]. In the LSTM layer, time series data consisting of the current trend slope (l_k) and persistence (s_k) are inputted to the CNN layer, and raw data sets are inputted to determine the dependency of the current trend and pattern transition point, as well as predicting the forecast slope (l'_k) and persistence ($\acute{s}_k$) of future trends. Figure 5 shows the structure of TreNet, which is used to analyze the current characteristics of CPU usage and determine offloading timing by predicting future trends.

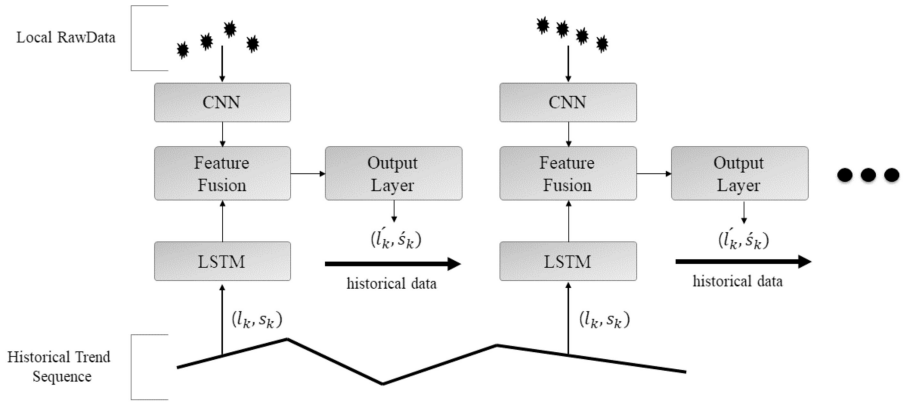

Figure 5. Structure of TreNet. CNN: convolutional neural network; LSTM: long short-term memory.

3. Adaptive Offloading Method

3.1. Static Profiler for Context Information

The context information required to execute offloading depends on the case of the execution target function. Here, we use a metric to classify contextual information that needs to be synchronized according to the offloading function to be performed, determined in a previous study [9]. Table 1 shows the context information requiring synchronization according to the type of offloading object function, which, according to defined metrics, depends on the existence of delivery parameters, the presence of delivery return values, and the use of global variables. In this study, we analyze the light-weight virtual machine intermediate language (LVIL) code of light-weight virtual machine executable format (LVEF) through static profiling to extract parameter information and return value and global variable information. Figure 6 shows the structure of a static analyzer used to analyze context information.

Table 1. Context information requiring synchronization according to function type classification.

Parameters and Return Value Types			Core Context	Stack Context			Address Context			
			gPc, gProcFrameP, sp, ep, DisVector, SpDisplay, ArDisplay	Operation Stack	Activation Record	Constant Pool	Variable Offset	Area Information Indicated by Address Value	Type Information (Type Size)	Array Information
Not Existing the parameters and return values	Not using address references	Use local variable only	O	X	X	X	X	X	X	X
		Use global Variables	O	X	X	O	O	X	O	X
	Using address references	Use global array variables	O	X	X	O	O	O	O	O
		A pointer variable refers to a global variable's address	O	X	X	O	O	O	O	X
		A pointer variable refers to a global variable's address	O	X	X	O	O	O	O	O
Existing the parameters and return values	Not using address references	The value of a local variable is passed as a parameter or a return value	O	O	X	X	X	X	X	X
		The value of a global variable is passed as a parameter or a return value	O	O	X	O	O	X	O	X
	Using address references	Local variables' references	O	O	O	X	O	O	O	X
		Local array variables' references	O	O	O	X	O	O	O	O
		Global variables' references	O	O	X	O	O	O	O	X
		Global array variables' references	O	O	X	O	O	O	O	O

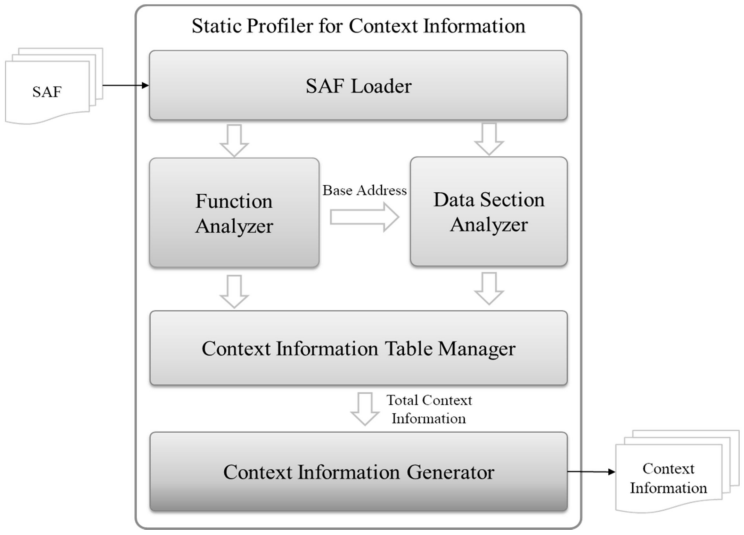

Figure 6. Structure of the static profiler for context information.

The function analyzer extracts the parameters and returns the value information. The parameter information and return value information are classified into three categories: address variables, global variables, and local variables. For local variables, the parameter and return value information do not affect the synchronization of the context information other than the operation stack. Therefore, in this case, information on the number of parameters to be loaded on the operation stack is required for operation stack synchronization, and information on the presence or absence of the return value is extracted. For an address variable, not only the operation stack but also the activation record must be synchronized. Therefore, the command code details the parameter and return value while the offset describes the location information in which the variable is managed and extracts the data size information of the variable, which is the synchronization range index of the activation record. If it is a global variable, it requests the data area analyzer to extract detailed information of global variables.

When analyzing the use of global variables in the function analyzer, the data area analyzes the literal table, and the internal symbol table of the data area of the smart assembly format (SAF) determines the offset address of the global variable. The offset address indicates the synchronization start area of the constant pool and the data size and extracts the information of the literal constant.

The context information table manager manages the information extracted from the function analyzer and the data area analyzer as a table of parameter information, return value information, and global variable information. When context information extraction is completed, the information managed in the table is generated as a JavaScript object notation (JSON) format file by the context information generator.

3.2. Context Information Synchronization

The core context information is crucial for the interpreter to execute the program and consists of an instruction buffer, a frame pointer indicating the start frame of the function, and pointers for accessing the context information. In conventional context synchronization, pointers pointing to the function start frame information of the operation stack and the activation record transmit all pointers indicating the start frame information of the function. However, as the information required for offloading is the frame information of the offloading performing function, if only the pointer pointing to the starting frame of the offloading performing function is synchronized, both the size of data and the resources consumed during communication are reduced. All pointers except for the start frame position information of the function are collected and transmitted every time offloading is performed because the value changes when the command is executed by the interpreter.

In the stack context information, unlike the constant pool, the operation stack and activation record are required context information in all function cases. The operation stack is the context information used when executing the load and computing instructions by the interpreter. The operation stack increases the start frame of the function to execute the instruction of the function at the time of the function call and loads the address to be returned after completing the function execution. Because the operation stack executes the instruction in units of functions, the area requiring synchronization is a frame area of the function to be offloaded. Therefore, the operation stack synchronizes from the start frame position of the offloading performing function to the position where the parameter value is loaded. This reduces the data size of the operation stack to collect when loading. If there is a return value after offloading, the data size of the operation stack to be collected is reduced by performing synchronization from the start frame position of the performed function to the position where the return value is loaded.

The activation record is context information which manages local variables necessary for function execution. The activation record updates the start frame position information of the function to manage the local variable of the function to be executed when the function is called. After function execution, the function frame area disappears. As the activation record manages local variables by function, the area that needs to be synchronized during offloading is the frame area of the function to be offloaded. Therefore, the activation record is synchronized from the start frame position of the function to be offloaded to the end of the function frame, thereby reducing the data size of the activation record synchronized at the time of offloading. Also, when the function is completed in the offloading environment, the function frame of the activation record disappears, so synchronization of the activation record does not proceed after offloading.

3.3. CPU Usage Trend Learning Model

The CPU usage trend learning model is based on TreNet, and the overall structure is composed of a dataset loader, data processor, and TreNet. Figure 7 shows the overall structure of the learning machine for trend prediction. Data processing involves processing a set of CPU usage data into a form suitable for trend learning; the data set loaded into the memory in the dataset loader is processed into a local dataset, a slope dataset, and a result data set. The local dataset is used to determine the pattern conversion point in a form that is normalized by the utilization distribution and bundled with the specified window size.

The slope dataset is a slope change data set of the CPU usage distribution and represents historical trend information from past to present. The data processor derives a linear function that minimizes the error in the CPU usage distribution using the least squares method to extract the gradient change data from the CPU usage distribution. The resulting dataset is used to learn the results of map learning and processed into a set of gradients indicating the future at the input data point according to the aim of this study.

As CPU usage trends represent real-time information, it is impossible to predict sustainability even by obtaining the slope. Therefore, TreNet learns the slope but not the persistence of the trend. The map learning proceeds in the following order.

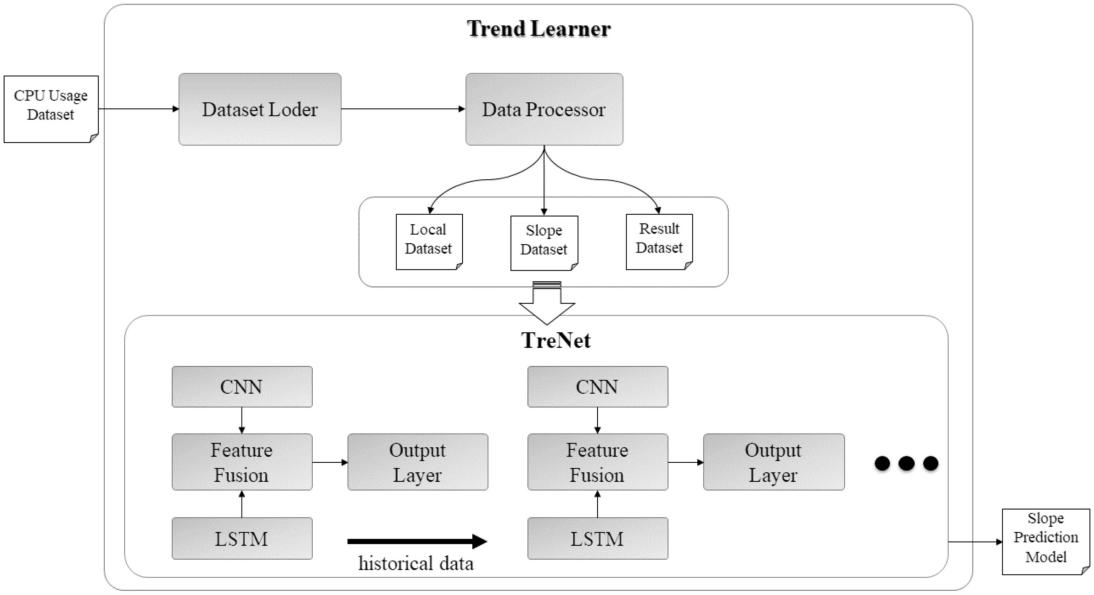

Figure 7. Structure of the learning model for performance load prediction.

(1) The local dataset is input to the CNN layer of TreNet, while the LSTM layer is input to the tilt dataset;
(2) The CNN and LSTM layers learn the dependencies in the input dataset and send the results to the feature fusion layer;
(3) The feature fusion layer fuses the results of the CNN and LSTM layers to determine the dependence of the slope and pattern transition point;
(4) The output layer derives the predictive slope through the dependency relationships and the resulting datasets learned in the feature fusion layer.

4. Experimental Results

We compare the amount of context information sent from the existing VM to confirm that it is reduced when applying the proposed context information synchronization scheme. Table 2 shows the experimental environment in which the context information synchronization technique is applied. The amount of context information to be transmitted before and after applying the synchronization technique is approximately 1.3 times greater than the core context information, 1.6 times greater than the operation stack, and twice the activation record. Therefore, it is confirmed that the amount of context information to be synchronized is reduced prior to applying the proposed context information synchronization method.

Table 2. Experimental environments for the proposed context information synchronization method.

Environments	Server	IoT Device
Processor	Intel(R) Core (TM) i7-4770K 3.50 GHz	BCM2837 64-bit QUAD Core 1.2 GHz
Memory	16 GB RAM	1 GB RAM
Operating System	Microsoft Windows 10	Raspbian

Figure 8 compares the context synchronization time required to offload the decimal, perfect, and bubble sort algorithms; the average of 1000 times offloading is displayed for each algorithm. Figure 9 compares the size of the context information synchronized during context synchronization. It is confirmed that the context synchronization method reduces the total synchronization time by approximately 10% more than the existing system (Figure 8). Furthermore, the size of the context information to be synchronized is reduced by approximately 30% from the conventional synchronization

method (Figure 9). The CPU usage pattern is required to model the real-world CPU usage distribution and analyzes and classifies the CPU usage data sets collected for the NAB (Numenta anomaly benchmark).

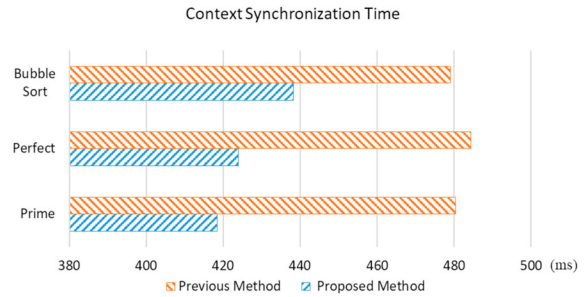

Figure 8. Context synchronization time required to run the prime number, perfect number, and bubble sort algorithms.

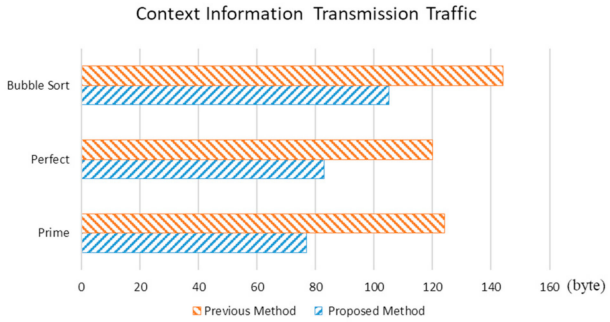

Figure 9. Context information network traffic size required to run the prime number, perfect number, and bubble sort algorithms.

The distribution of CPU usage in the dataset illustrates either stability after a surge in usage rate, stability after a steep decline, a stable usage rate, or stable usage. The peak value is classified into patterns in which the utilization rate continuously changes (Figure 10).

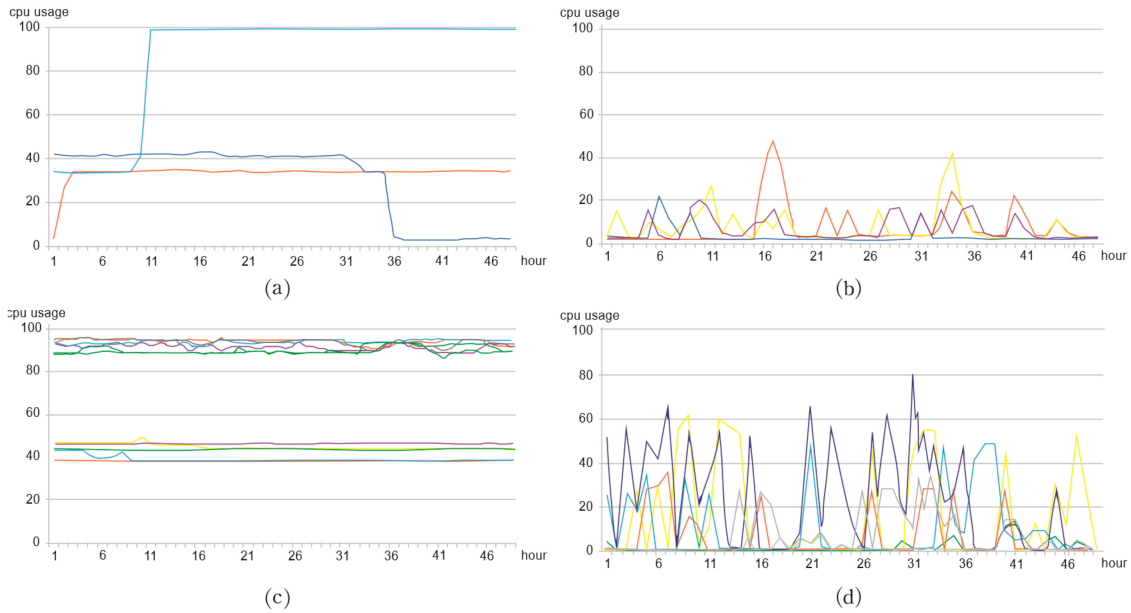

Figure 10. CPU usage classified into patterns of (**a**) rapid increases or decreases in usage rate, (**b**) peak usage rate, (**c**) stable usage rate, and (**d**) continuously changing usage rate.

The CPU usage distribution is modeled using the M/M/1 queuing system, which follows the time-variable Poisson process [17,18]. The M/M/1 queuing system is not suitable for modeling CPU usage that changes in real time because the arrival rate of work is fixed. On the other hand, time-varying M/M/1 queuing systems, which describe the workload changing in real time, can model the CPU usage rate because the arrival rate of work varies with time. The arrival rate of work for each pattern model is defined as follows.

Where the work arrival rate follows a stable upward pattern (model 1):

$$\lambda(t) = 0.04 \qquad for\ 0 \le t < 2 \tag{1}$$

$$\lambda(t) = 0.1t - 0.16 \qquad for\ 2 \le t < 3 \tag{2}$$

$$(t) = 0.14 \qquad for\ 3 \le t \tag{3}$$

Where the work arrival rate follows a stable downward pattern (model 2):

$$\lambda(t) = 0.14 \qquad for\ 0 \le t < 2 \tag{4}$$

$$\lambda(t) = -0.1t + 0.34 \qquad for\ 2 \le t < 3 \tag{5}$$

$$\lambda(t) = 0.04 \qquad for\ 3 \le t \tag{6}$$

Where the work arrival rate follows a stable and peaked pattern (model 3):

$$\lambda(t) = 0.04 \qquad for\ 0 \le t < 2 \tag{7}$$

$$\lambda(t) = 0.1t - 0.16 \qquad for\ 2 \le t < 3 \tag{8}$$

$$\lambda(t) = -0.1t + 0.44 \qquad for\ 3 \le t < 4 \tag{9}$$

$$\lambda(t) = 0.04 \qquad for\ 4 \le t \tag{10}$$

Patterns in which the arrival rate is constantly changing (model 4):

$$\lambda(t) = 0.05172t - 0.00554 \qquad for\ 0.0 \le t < 1.0 \tag{11}$$

$$\lambda(t) = 0.20536t - 0.15918 \qquad for\ 1.0 \le t < 1.5 \tag{12}$$

$$\lambda(t) = -0.0846t + 0.27576 \qquad for\ 1.5 \le t < 2.0 \tag{13}$$

$$\lambda(t) = -0.14484t + 0.39624 \qquad for\ 2.0 \le t < 2.5 \tag{14}$$

$$\lambda(t) = 0.2688t - 0.63786 \qquad for\ 2.5 \le t < 3.0 \tag{15}$$

$$\lambda(t) = -0.31804t + 1.12266 \qquad for\ 3.0 \le t < 3.5 \tag{16}$$

$$\lambda(t) = 0.098t - 0.33348 \qquad for\ 3.5 \le t < 4.0 \tag{17}$$

$$\lambda(t) = 0.01476t - 0.00052 \qquad for\ 4.0 \le t < 4.5 \tag{18}$$

$$\lambda(t) = 0.15328t - 0.62386 \qquad for\ 4.5 \le t < 5.0 \tag{19}$$

The results are verified using a simulator to show that the modeled CPU usage distribution is typically modeled according to the defined arrival rate of work (Figure 11) [22].

Figure 12 is a comparison of the CPU usage distribution modeled using the time-varying M/M/1 queuing system and the results predicted by the proposed scheme. The raw slope represents the modeled CPU usage and the predicted slope shows the predicted value of the server workload based on hybrid depth learning. First, (a) shows a steady pattern of usage rates but (b) shows a steeply decreasing pattern with a steady utilization rate distribution. Figure 12c illustrates a peak followed by

a rapid decrease and (d) shows the workload pattern of the IoT application proposed by Bell LAB [23]. From the experimental results, we can confirm that the prediction slope is similar to the CPU usage distribution slope. Therefore, the hybrid deep neural network model can effectively predict the slope of CPU usage and consequently the workload of the server. Based on these prediction results, the IoT client VM can efficiently determine the offloading execution.

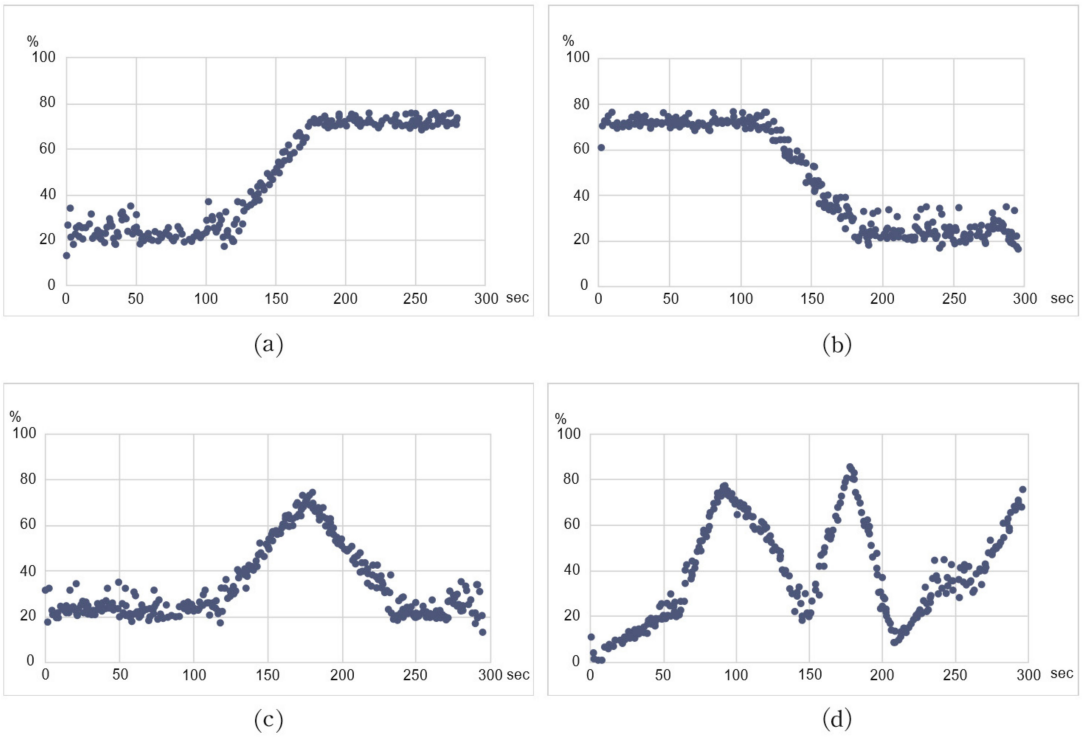

Figure 11. CPU modeling simulation results for (**a**) model 1, (**b**) model 2, (**c**) model 3, and (**d**) model 4.

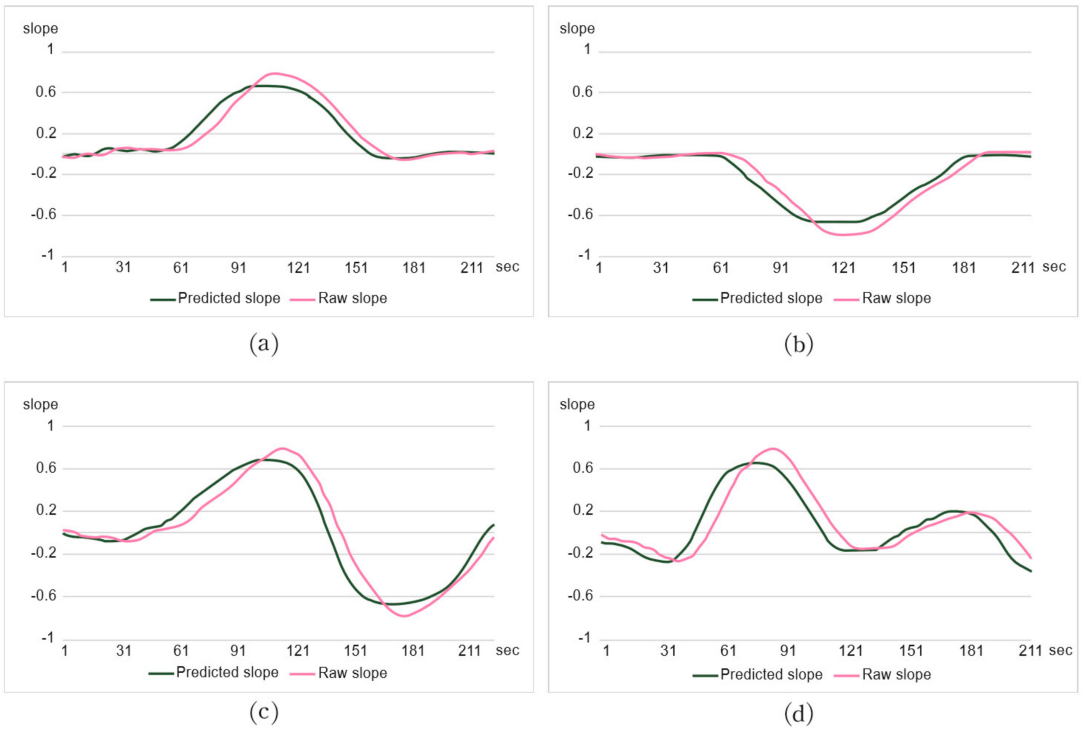

Figure 12. Estimates of server load using hybrid deep neural network for (**a**) model 1, (**b**) model 2, (**c**) model 3, and (**d**) model 4.

5. Conclusions and Further Research

The VM to which the conventional offloading technique is applied must synchronize all context information for communication between the server and the local server. As the amount of context information for transmission is large, the resource consumption rate of context information synchronization is increased in low-performance IoT equipment. In order to solve these problems, this study classified offloading functions according to the context information required to perform the offloading function and context information synchronization. We investigated core context information, the operation stack, and activation record synchronization, which represent the core contextual information for function execution. First, regarding the required context information, the data size of the pointer managing the function start frame to be transmitted was reduced. Next, the operation stack and the activation record employed the common point for managing the resources in the function frame unit, and only the frame area necessary for function execution was synchronized to reduce the respective data sizes. The VM using the proposed context information synchronization scheme can reduce the resource consumption rate of IoT equipment because the amount of data to be transmitted was reduced compared to the existing VM when context information was synchronized. From the perspective of the business community selling devices, the advantage of reducing the resources required by IoT devices is that it allows them to focus on the IoT device's functions. As a result, IoT device developers can help to build competitive advantages in uncertain environments through faster new device development [24].

We also analyzed the static profiling-based context synchronization technique to reduce the network overhead costs of the context synchronization of existing systems. The proposed scheme only synchronizes context information that is essential for function execution. When using global variables inside the offloading function, synchronization is required every time global variable values are assigned to IoT devices. Also, if a program uses reference variables such as pointers, arrays, and variable addresses, it must share the referenced memory region between the server and the local system. In order to solve this problem, we investigated the synchronization technique of context information in which global variables are managed using analyzed information by evaluating the use of global variables of functions through the static profiler. For the case where the memory area referred to in IoT equipment is offloaded, we also proposed a method to maximize the offloading efficiency by minimizing the overheads of context information synchronization in low-performance IoT devices. Also, we proposed CPU usage trends, which act as workload indicators, through deep learning for efficient offloading execution decisions. We then used CPU usage distribution data to model the workload through deep learning. The modeled CPU usage patterns reflect real-world usage patterns, despite being simple. These predicted usage trends are indicators of offloading execution decisions because they represent changes in workload. In the future, we will research memory usage and energy usage prediction by applying the CPU usage prediction model proposed in this paper and determining the offloading execution.

Author Contributions: Y.S. and J.J. contributed to writing the original draft and Y.S. also worked on the conceptualization and methodology. J.J. performed the software and validation work. Y.L. contributed to supervision and writing in the review and editing phase.

Funding: This research received no external funding.

Acknowledgments: This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning (No. 2016R1A2B4008392), by a National Research Foundation of Korea(NRF) grant funded by the Korea government (MSIP; Ministry of Science, ICT & Future Planning) (No. 2017R1C1B5018257), and by a National Research Foundation of Korea (NRF) grant funded by the Korea government (MIST)(No. 2018R1A5A7023490).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Son, Y.; Lee, Y. A study on the smart virtual machine for executing virtual machine codes on smart platforms. *Int. J. Smart Home* **2012**, *6*, 93–105.
2. Millani, C.E.; Linhares, A.; Auler, R.; Borin, E. COISA: A compact OpenISA virtual platform for IoT devices. In *Proceedings of the WSCAD'15*; Florianopolis: Santa Catarina, Brazil, 18–21 October 2015.
3. Son, Y.; Lee, Y. A study on the smart virtual machine for smart devices. *Information* **2013**, *16*, 1465–1472.
4. Gavrin, E.; Lee, S.J.; Ayrapetyan, R.; Shitov, A. Ultra lightweight JavaScript engine for internet of things. In Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, Pittsburgh, PA, USA, 25–30 October 2015; pp. 19–20.
5. Son, Y.; Kim, J.H.; Lee, Y. A design and implementation of html5 based svm for integrating runtime of smart devices and web environments. *Int. J. Smart Home* **2014**, *8*, 223–234. [CrossRef]
6. Son, Y.; Jeong, J.; Lee, Y. Design and implementation of the secure compiler and virtual machine for developing secure IoT services. *Future Gener. Comput. Syst.* **2017**, *76*, 350–357.
7. Botta, A.; De Donato, W.; Persico, V.; Pescapé, A. Integration of cloud computing and internet of things: A survey. *Future Gener. Comput. Syst.* **2016**, *56*, 684–700. [CrossRef]
8. Dupont, C.; Giaffreda, R.; Capra, L. Edge computing in IoT context: Horizontal and vertical Linux container migration. In Proceedings of the 2017 Global Internet of Things Summit (GIoTS), Geneva, Switzerland, 6–9 June 2017.
9. Son, Y.; Lee, Y. Offloading Method for efficient use of local computational resources in mobile location-based services using clouds. *Mob. Inf. Syst.* **2017**, *2017*, 1856329. [CrossRef]
10. Kumar, K. A survey of computation offloading for mobile systems. *Mob. Netw. Appl.* **2013**, *18*, 129–140. [CrossRef]
11. Yang, K.; Ou, S.; Chen, H. On effective offloading services for resource-constrained mobile devices running heavier mobile internet applications. *IEEE Commun. Mag.* **2008**, *46*, 56–63. [CrossRef]
12. Kumar, K.; Lu, Y. Cloud computing for mobile users: Can offloading computation save energy? *Computer* **2010**, *43*, 51–56. [CrossRef]
13. Shi, C. Cosmos: Computation offloading as a service for mobile devices. In Proceedings of the 15th ACM International Symposium on Mobile Ad Hoc Networking and Computing, Philadelphia, PA, USA, 11–14 August 2014; ACM: New York, NY, USA, 2014; pp. 287–296.
14. Chun, B.G. Clonecloud: Elastic execution between mobile device and cloud. In Proceedings of the 6th ACM Conference on Computer Systems, Ischia, Italy, 10–13 April 2011; pp. 301–314.
15. Dinh, H.T.; Lee, C.; Niyato, D.; Wangm, P. A survey of mobile cloud computing: Architecture, applications, and approaches. *Wirel. Comun. Mob. Comput.* **2013**, *13*, 1587–1611. [CrossRef]
16. La, H.; Kim, S. A Taxonomy of offloading in mobile cloud computing. In Proceedings of the 7th IEEE International Conference on Service-Oriented Computing and Applications, Matsue, Japan, 17–19 November 2014; pp. 147–153.
17. Wang, C.; Li, Z. A computation offloading scheme on handheld devices. *J. Parallel Distrib. Comput.* **2004**, *64*, 740–746. [CrossRef]
18. Chen, H.; Lin, Y.; Chen, C. COCA: Computation offload to clouds using AOP. In Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Ottawa, ON, Canada, 13–16 May 2012; pp. 466–473.
19. Lin, T.; Hsu, C.; King, C. Context-aware decision engine for mobile cloud offloading. In Proceedings of the IEEE Wireless Communications and Networking Conference Workshops, Shanghai, China, 7–10 April 2013; pp. 111–116.
20. Kovachev, D.; Yu, T.; Klamma, R. Computation offloading from mobile devices into the cloud. In Proceedings of the IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, Madrid, Spain, 10–13 July 2012; pp. 784–791.
21. Lin, T.; Guo, T.; Aberer, K. Hybrid neural networks for learning the trend in time series. In Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17, Melbourne, Australia, 19–25 August 2017; pp. 2273–2279.
22. CPU-Load-Generator. Available online: https://github.com/beloglazov/cpu-load-generator (accessed on 29 October 2018).

23. Nokia. *An Internet of Things Blueprint for a Smarter World: Capitalizing on M2M, Big Data and Cloud*; Strategic White Paper; Nokia: Espoo, Finland, 2015; p. 13.
24. Mikalef, P.; Pateli, A. Information technology-enabled dynamic capabilities and their indirect effect on competitive performance: Findings from PLS-SEM and fsQCA. *J. Bus. Res.* **2017**, *70*, 1–16. [CrossRef]