2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip

2-Step Power Scheduling with Adaptive Control Interval for Network Intrusion Detection Systems on Multicores

Lau Phi Tuong
Department of Computer Science and Engineering,
Green Computing Systems Research and Development Center,
Waseda University
Tokyo, Japan
laulpt@kasahara.cs.waseda.ac.jp

Keiji Kimura
Department of Computer Science and Engineering,
Green Computing Systems Research and Development Center,
Waseda University
Tokyo, Japan
kimura@apal.cs.waseda.ac.jp

Abstract—**Network intrusion detection system (NIDS) is becoming an important element even in embedded systems as well as in data centers since embedded computers have been increasingly exposed to the Internet. The demand for power budget of these embedded systems is a critical issue in addition to that for performance. In this paper, we propose a technique to minimize power consumption in the NIDS by 2-step power scheduling with the adaptive control interval. In addition, we also propose a CPU-core controlling algorithm so that our scheduling technique can preserve the performance for other applications and NIDS assuming the cases of multiplexing NIDS and them simultaneously on the same device such as a home server or a mobile platform. We implement our 2-step algorithm into Suricata, which is a popular NIDS, as well as a 1-step algorithm with the adaptive interval, and a simple fixed-interval algorithm for evaluations. Experimental results show that our 2-step scheduling with both the adaptive and the fixed 30-millisecond interval achieve 75% power saving comparing with the Ondemand governor and 87% comparing with the Performance governor in Linux, respectively, without affecting their performance capability on four ARM Cortex-A15 cores at the network traffic of 1,000 packets/seconds. In contrast, when the network traffic reaches to 17,000 packets/seconds, our 2-step scheduling and the Ondemand as well as the Performance governor can maintain the packet processing capacity while the fixed 30-milliseconds interval processes only 50% packets with two and three cores, and about 80% packets on four cores.**

Keywords- Network Intrusion Detection System (NIDS); Dynamic Voltage Frequency Scaling (DVFS); Suricata; Data Center

I. INTRODUCTION

The number of Internet users is increasing rapidly because of benefits from searching and sharing information, social network services, open education services and other useful activities. There is a demand for extending data centers to be able to process a huge amount of information in responding to the requirements from Internet users. Because of recent sophisticated external attacks from the Internet, companies and organizations are always required to maintain their security on server systems.

In addition, embedded devices such as wireless sensor nodes and smart phones have been widely used in people's life. For instance, many people enjoy services in smart phones such as convenient communication services and a variety of entertainment activities via the Internet. Moreover, Implantable Medical Device (IMD) is a breakthrough technology merged into patients demanding for 24-hours service to keep track of health status. A caretaker can monitor the health status of patients, diagnose symptoms, and download personal health data in a hospital via wireless network system.

Such smart devices may introduce several vulnerabilities. For instance, an unauthorized reader can exploit them to collect private information illegally from the hospital or health center. Therefore, Intrusion Detection Systems (IDSs), which can detect suspicious activities from outside users via the Internet, are becoming important even in embedded platforms.

Network Intrusion Detection Systems (NIDS) have been widely used in servers. They keep track of traffic in real time and identify attacks coming from and going to network devices. Then, they issue alerts such as detecting viruses and Trojan horse to security administrators before causing serious damages to the system. In this paper, we utilize Suricata [16], which is a popular multithreaded NIDS as an evaluation platform.

The performance improvement has been the first priority issue for NIDSs. Many previous researches have been carried out from the point of this view [8], [9], [10]. However, power consumption in NIDSs is becoming a critical issue especially in the embedded area along with increasing the number of devices connected to Internet as mentioned in the previous paragraphs. Thus, current NIDSs are required to provide sufficient performance to process as many packets as possible in the real-time manner while keeping their power consumption as low as possible.

Dynamic voltage and frequency scaling (DVFS) has been widely used to control power consumption in today's computer systems. In order to maintain the required performance for the system while keeping its power consumption as low as possible by utilizing DVFS, there are two issues to take into account; the one is how to decide the appropriate clock frequency, and the other is how to decide the control interval of DVFS. If the clock frequency is inappropriate, it causes performance violation or wastes of power usage of the system. Similarly, if the control interval is inappropriate, it introduces long control overhead or long delay from changes of an amount of workload.

In this paper, we propose a 2-step power scheduling algorithm with the adaptive control interval to optimize power consumption as well as maintain the performance of packet processing for NIDSs. The first step is to estimate the total processing time of packets in the buffer at the very beginning of the control interval, then setting the reasonable control interval until the next estimation time after that assigning a feasible power budget according to the estimated control interval. The second step is to adjust the clock frequency at the packet detection module of Suricata regarding to an amount of remaining packets in the buffer and the current processing time within the control interval decided at the first step. Furthermore, we also propose a core controlling algorithm to switch cores in case of multiplexing applications and the NIDS executed on a home server or an embedded system to maintain the performance capability for them.

In order to evaluate our techniques, we implemented our 2-step scheduling with the adaptive control interval into Suricata. We also implemented other approaches such as 1-step scheduling with the adaptive interval and 2-step with the fixed interval into Suricata for comparison. The performance capability and power consumption of the proposed approaches are compared with the Ondemand governor and the Performance governor in Linux on ODROID-XU3 board, which has four big ARM Cortex-A15 cores and four small Cortex-A7 cores, at different levels of network traffic congestion. We also evaluate power consumption as well as the packet processing capacity when an application and Suricata are simultaneously executed on the board.

The rest of this paper is organized as follows. Section II presents related works of power optimization especially for network servers. Section III shows the proposed algorithms. Section IV reports experimental results. Finally, Section V summarizes this paper.

II. RELATED WORK

DVFS control has been a widely used technique to manage power consumption for computer systems. One of the main issues of using DVFS is how to select the appropriate clock frequency without forcing the major performance impacts on a system. There has been, of course, a large amount of literature dealing with DVFS for various applications and systems. In this section, we mainly focus on the previous works dealing with the network-oriented applications employing DVFS technique for optimizing energy consumption.

There have been several approaches to control DVFS for network devices [2], [3]. AFCMBOT tries to adjust the clock frequency to reduce power consumption of network equipment by calculating the amount of network traffic [2]. A main point of this technique is introducing dual threshold values corresponding with the different frequency states to ensure the stability of clock frequency control. PBD applied DVFS to a network controller by checking the packet buffer at each specific point [3]. This introduces two states such as low and high power mode. If the packet pool has more than a threshold value, then the system enables the high power state for computing performance, otherwise switching to the low power state.

Other papers have focused mainly on power management techniques for server computers in data centers [4], [5], [6], [7], [11], [12], [13], [14]. They have tried to ensure the performance of systems while keeping their power consumption low. In [4], the authors proposed a hybrid resource allocation strategy for cloud computing environments to make balance between resource consumption and quality of service especially considering oscillatory peaks of workloads. Cheng et al. proposed a request batching mechanism with DVFS [5]. It employs two-layer control systems; the one layer is for a batching control loop including a fuzzy model predictor and the other one is for a power control loop. APPLEware is an autonomic middleware for co-located Web applications on virtualized computing systems [6]. It utilizes machine learning based self-adaptive modeling for resource allocation. One of the problems targeted by APPLEware is the inter application performance interference. Instead of solving the global performance and power control problem, it decomposes a global problem into localized subtasks. The local distributed processors handle those divided subtasks. Ghandhi et.al. proposed a queuing theoretic model consisting of power-to-frequency relationship, peak workloads, and others to predict the optimal power allocation among servers on a server farm so as to minimize mean response time [7]. Blink provides an energy abstraction on serve clusters under the assumption they are equipped with intermittent power supply [12]. It schedules power modes on computation nodes between the high power active state and the low power inactive state along with a predefined policy. PowerNap was proposed as an approach eliminating the idle power waste and minimize transition time into and out of low power nap state for server computers [14]. When a server exhausts all pending workloads, it rapidly transitions to the nap state. In the nap state, all components in a server such as disk, memory, and others are switched to the inactive mode so as to eliminate the waste of idle power. A network device informs the arrival of workloads to wake up the server from the nap state to the active state.

There are several frequency switching algorithms targeting to task processing for CPUs have been also proposed [1], [15]. Time reservation DVI (TRD) was proposed to reduce heat dissipation of CPU [1]. It estimates an expected clock frequency based on the available time in that task when a new task arrives. Then, it adjusts the frequency of the task by comparing the now status of the current task and the task timeouts. Mizotani et al. proposed the integration of a computational model and RT-VFS to assign the optimal power budget to each part of the task and simultaneously improve computation quality within a real-time constraint [15]. They employed Mandatory-First with Earliest Deadline (MFED) as an imprecise computational model to schedule tasks to processors and used RT-SVFS as well as Cycle-Conserving RT-DVFS for controlling processors frequency.

III. POWER SCHEDULING TECHNIQUE

The proposed power scheduling technique consists of two steps. At the first step, the average number of processing cycles of each packet is estimated to assign a reasonable frequency to meet an adaptive control interval. The adaptive control interval is determined based on statistics of historical traffic workloads. At the second step, the clock frequency is adjusted to keep the performance capability during the control interval decided at the first step and achieve more optimal power consumption. This step is done by checking the correlation between the current amount of processed packets and the current processing time. The adaptive control interval is also defined as a deadline constraint in this paper. In addition, when both Suricata and other programs are simultaneously executed on the same machine, the CPU-core controlling algorithm is used to switch cores in order to maintain the performance for all applications as well as power optimization for the NIDS on multicores. 2-step power scheduling technique is implemented in Suricata as shown in Fig. 1. Suricata consists of four modules such as the stream phase, the decode phase, the detection phase, and the output packet phase. The first step is implemented at the packet receiver in front of the stream phase, and the second step is implemented in the detection phase.

A. *Scheduling the feasible frequency to meet the adaptive control interval*

Giving x is the number of received packets available in the pool and T_i is the processing time of its own packet. The processing time of all ones can be described as the expression (1).

$$\sum_{i=1}^{i=x} T_i = T_1 + T_2 + \ldots + T_i \qquad (1)$$

From expression (1), it can be seen that adjusting the clock frequency depends on each packet processing cost and the quantity of captured packets from the network. In practice, we can estimate the average processing time of each packet, T_{mean} by profiling technique. Additionally, x can be determined from the packet statistic of the NIDS after each schedule interval update.

We define the frequency control interval as a deadline constraint that the algorithm must ensure to maintain the performance of the system. To meet the deadline constraint denoted by $T_{deadline}$, the processing time of x received packets must be finished as soon as the deadline constraints arriving. The feasible frequency, $f_{\min} \le f_{feasible} \le f_{\max}$, is selected in responding with the given deadline control interval, *5us* $\le T_{deadline} \le$ *30ms*. The value *5us* is originally taken from

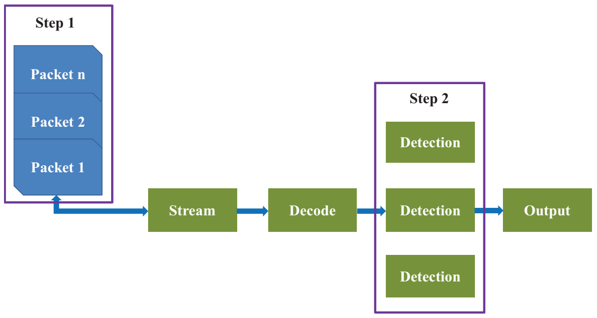


Fig. 1. 2-step scheduling with the adaptive control interval implemented into Suricata.

Suricata while *30ms* can be estimated depending on the hardware processing capacity. We can derive the expression (2) below from the expression (1).

$$\sum_{i=1}^{i=x} \frac{1}{f_{feasible}} * T_i = x * \frac{1}{f_{feasible}} * T_{mean} \le T_{deadline} \qquad (2)$$

In many real-time systems like an MPEG movie player, there is an explicitly specified deadline interval to process each workload, and if this deadline requirement is not fulfilled, then it can violate the desired execution time. For network tracking applications, network workloads change unpredictably from low to big traffic and there often have unstable and stable points. According to the expression (2), if we choose too small values for the fixed interval, then the system cannot minimize power consumption. For the fixed large interval, there may exist in major degradation in the performance capability in the context of very heavy network congestion level. Therefore, the control interval should be variable according to network traffic status.

In the algorithm 1, the packet counter is used to record how many received packets processed and it is reset to be zero after each control interval update. The number of received packets of previous updates is denoted by y. If there are some existing packets at the line 2 which have not been yet inspected previously, it is counted at the line 3 otherwise just considering the current number of received packets at the line 5. At the line 7 of the algorithm 1, when received packets x is more than or equal to a dominant value *LB* so called the limited buffer, then $T_{deadline}$ should be assigned to *5us* coming originally from Suricata source code as the default value at the line 8. *LB* is picked up depending on the hardware's processing ability by the profiling technique. For the else cases, $T_{deadline}$ should be determined based on some past buffers collected to ensure the performance capability.

Giving an available amount of cores for the NIDS is denoted by *No. cores* and a set of the amount of collected packets into the buffer at historical update times denoted by $x_{history} = \{x_{history1}, x_{history2}, x_{historyi}\}$ then taking the average value $x_{average} = (x_{history1} + x_{history2} + \ldots + x_{historyi})/(i+1)$.

The number of elements in the set should be limited by 10 to ensure the calculation of control interval correctly. If this value is set to a very small value like 3 elements, then the estimation of the control interval can get incorrect. This is due to, for instance, 2 elements of 3 may indicate unstable points and they may introduce an incorrect deadline. This value, of course, is much better with using the bigger value. However, it becomes convergent at a point like 10 elements and if we set to more than 10, the probability of correctness is the same with 10. In practice, if we take only the current number of received packets *x* for calculating the deadline constraint, that is, $x_{average} = x$, then the estimation of the deadline interval may get incorrect. This is due to the collected workload x sometimes become unstable causing by network bottleneck. For example, the stable network traffic is about 500 packets/seconds, but this rate can increase to 1,000 or 10,000 packets/seconds at unstable points.

Network bottleneck situations happening during the control intervals can affect the overall performance capability when the estimated deadline interval is very long.

Algorithm 1: Select the feasible clock frequency

```
1: while (true) {
2:     if (packet counter < $y$) {
3:         $x = x + (y -$ packet counter$)$}
4:     else {
5:         $x = x$}
6:     packet counter = 0
7:     if ($x \geq LB$) {
8:         $T_{deadline} = 5us$}
9:     else {
10:        if (all elements in $x_{history} \leq LB$) {
11:            $T_{deadline} = 5us + T_{mean} * \frac{(LB - x_{average})}{No.cores}$ }
12:        else if (at most 1/5 elements in $x_{history} \geq LB$) {
13:            $T_{deadline} = 5us + T_{mean} * \frac{(LB - \frac{4}{5} x_{average})}{No.cores}$ }
14:        else if (at least 2/5 elements in $x_{history} > LB$) {
15:            $T_{deadline} = 5us$}
16:        else {
17:            $T_{deadline} = 15ms$}
18:    }
19:    if ($T_{deadline} \geq 30ms$) {
20:        $T_{deadline} = 30ms$}
21:    scheduling the frequency from the calculated
           $T_{deadline}$ according to equation (2)
22:    sleep($T_{deadline}$)}
```

We collect some historical buffers, then giving several possible cases to estimate the control interval. As the first situation, if all past results in $x_{history}$ are under *LB* value, then the prediction for the next future workload is possible to less than *LB*, so we can extend the frequency control interval by adding $T_{mean} * (LB - x_{average}) / (No.cores)$ with *5us* as shown at the line 11. The second case of congestion is that at most 1/5 elements in $x_{history}$ is more than *LB* value as written at the line 13. For instance, $x_{history}$ includes 10 past buffers and at most 10*1/5= 2 historical buffers or 2 unstable points in out of 10 are over *LB,* so it should be considered at least 8 remained buffers which are less than *LB* to estimate the interval. In the scenario that at least 2/5 elements in $x_{history}$ over *LB* or if at least 4 previous receive buffers in out of 10 collected past buffers are greater than *LB,* it should be used *5us* in order to ensure the performance capability at the line 15 since the next network status is possible to become very big traffic. In case of historical received workloads unpredicted, the deadline update should be assigned to be less than *15ms* or an arbitrary interval at line 17. The fraction of historical buffers such 2/5 and 1/5 can be changed according to the traffic characteristic of target network. After the estimation of the deadline interval is done, the feasible frequency is determined according to the formula (2) at the line 21.

Finally, the calculated deadline by the algorithm 1 is taken as the control interval for the frequency scheduling.

B. *Adjusting the clock frequency within the packet detection phase*

Because of the more sophistication of the attacks from the Internet, the known signatures are often being extended to detect new attacks. From this extension of the rule set, the cost of malicious messages can become much more expensive than that of benign packets as a result of consuming more the inspection time for suspicious ones. In a real system, if the collected packets are benign packets, then the actual processing cost is much less than the average estimated value. As a result, there exists in a long interval of the idle time because the selected clock frequency from the first step regarding to the average estimated cost is too fast. On the other hand, when the buffer receives the majority of malicious packets, the actual processing cost consumes much more than the estimated cost. Since the selected clock frequency from the first step is too slow, so the system is unable to keep the data processing capability well. Therefore, there is a need for increasing or decreasing frequency so as to achieve minimal power consumption and avoid violating the performance capability.

The algorithm 2 of the second step is used to avoid degradation in the performance capacity as well as eliminate the idle time after selecting the reasonable clock frequency at the first step. We compare the current processing time and the calculated control interval based on the packet counter variable. The packet counter should be checked at appropriate points such as at 50% the amount of processed packets, or at 75% to respond the estimated real-time interval at the first step. During the estimated control interval, when the packet counter reaches to 50% the number of processed packets at the line 2, if the current processing time of collected packets is shorter than 50% of the estimated control interval as shown in Fig. 2 (a), then there is nothing to do. Otherwise, it should be adjusted to the higher frequency as shown in Fig. 2 (b) to meet the estimated control interval constraint at the first step. Similar to the case of 75% processed packets or we can take more points to ensure the performance capability.

When the packet counter reaches to the last processed packet at the line 8 before the calculated deadline arrives by an acceptable interval, the clock frequency should be switched to the lowest frequency or be off as shown in Fig. 2 (c).

Algorithm 2: Adjusting the frequency during the packet detection phase

```
1: packet counter++
2: if (packet counter = 50% of $x$) {
3:         if (current processing time > 50% $T_{deadline}$ ) {
4:                 switch to the higher frequency}}
5: else if (packet counter = 75% of $x$) {
6:         if (current processing time > 75% $T_{deadline}$ ) {
7:                 switch to the higher frequency}}
8: else if (packet counter = 100% of $x$) {
9:         if (current processing time < $T_{deadline}$ ) {
10:                switch to the lowest frequency}}
```

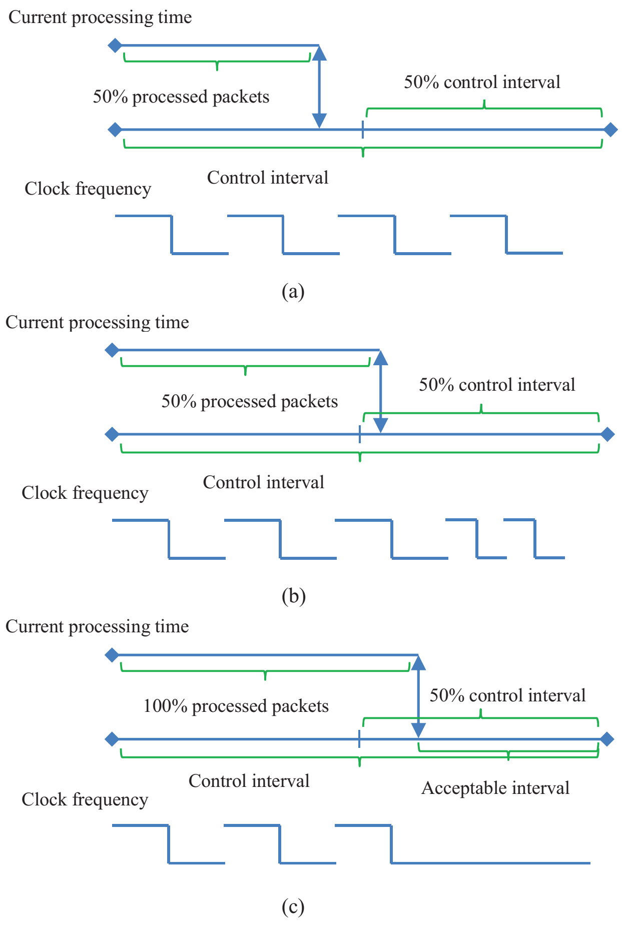


Fig. 2. At (a) 50% packets processed, do nothing if the current processing time is less than 50% control interval (b) 50% packets processed, increasing the clock frequency if that is over 50% control interval (c) 100% packets processed, if that is shorter than the control interval by an acceptable interval, then minimizing the clock frequency or switching to be off.

C. Multiplexing NIDS and other programs executed concurrently

The algorithm 3 controls the number of cores assigned to NIDS depending on the case of with or without other program executed on the same machine. At the very beginning, Suricata is run on all cores in the system. All thread IDs are recorded at the line 2. The system explores whether the multiplexed programs are posed. Then, the required number of cores used for NIDS are switched so that the NIDS and other programs can be run on different cores or group of cores at the line 4 and 5. In case of only the NIDS used, all NIDS threads should be scheduled to all cores on the device. A program is implemented from the algorithm 3 run concurrently to schedule the number of cores for multiple programs.

. **Algorithm 3**: Controlling the number cores for NIDS

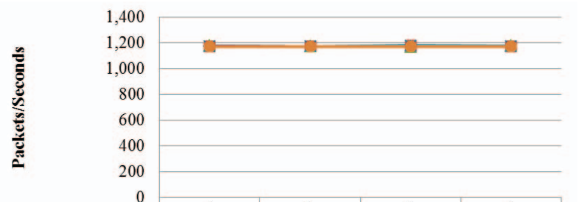
```
1: run NIDS on all cores of machine
2: get all thread IDs created by NIDS
3: while (true) {
4:         if (other program requested) {
5:                 switch all threads of NIDS and other
                   program to different cores}
6:         else if (only NIDS run) {
7:                 switch all thread IDs of NIDS to all cores}
8:         sleep()}
```

IV. EXPERIMENTAL RESULT

We implement proposed algorithms into Suricata version 2.0.8 [16], then measure and compare power consumption as well as the performance capability of Suricata among CPU frequency scaling governors of Linux and various optimization methods by using network traffic traces on ODROID-XU3 (2.0 GHz 4 big ARM Cortex-A15, 1.4 GHz 4 small ARM Cortex-A7, 2 GB RAM) with Ubuntu 15.04 mate ODROID-XU3 operating system [17].

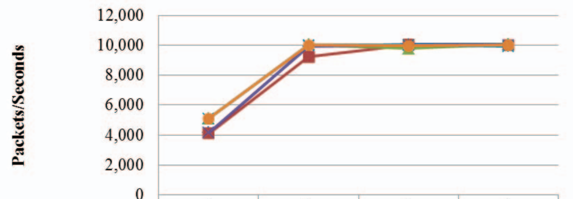
There are totally 6 different scheduling methods integrated into Suricata for experimental evaluations including the Performance and the Ondemand governor in Linux kernel, 2-step scheduling with the fixed 15 and 30 milliseconds control interval, 1-step scheduling with the adaptive control interval using only the algorithm 1, and 2-step scheduling with the adaptive interval. We use tcpreplay tool [18] as a packet generator to cause the different level of network traffic congestion at the router and run Suricata on ODROID-XU3 to detect packets at the router for measurement.

At 1,000 packets/seconds as shown in Fig. 4 (a), the frequency scheduling algorithm with the fixed interval as well as with the adaptive interval gives the best power optimization compared with Linux scaling governors. When the network traffic is increased to 10,000 packets/seconds as shown in Fig. 4 (b), the power dissipation becomes significant by using the Performance and the Ondemand scaling governor in Linux kernel. Typically, when four cores are used, the Performance and Ondemand scaling governor account for considerable power consumption, about 3.8 and 3.2 watts, respectively. In contrast, applying 2-step scheduling with both the adaptive and the fixed 30ms interval, the power consumption in Suricata is reduced to



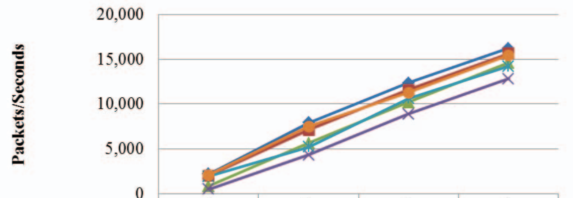
	1	2	3	4
Performance	1,173	1,172	1,171	1,172
Ondemand	1,170	1,171	1,175	1,171
fixed 15ms	1,175	1,171	1,168	1,177
fixed 30ms	1,173	1,170	1,171	1,172
1-step adaptive	1,172	1,170	1,180	1,173
2-step adaptive	1,170	1,171	1,171	1,172

(a)



	1	2	3	4
Performance	4,166	9,928	10,053	10,052
Ondemand	4,122	9,216	10,042	9,980
fixed 15ms	5,105	10,023	9,789	10,052
fixed 30ms	4,160	9,950	10,020	10,012
1-step adaptive	5,101	10,010	10,012	9,918
2-step adaptive	5,110	10,002	9,956	10,005

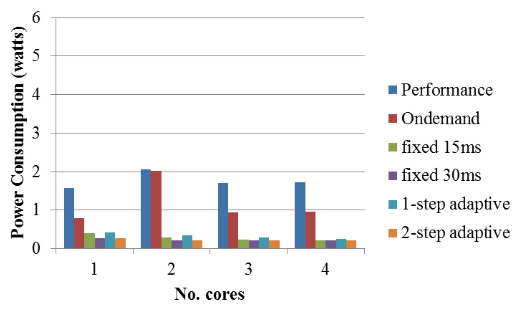
(b)



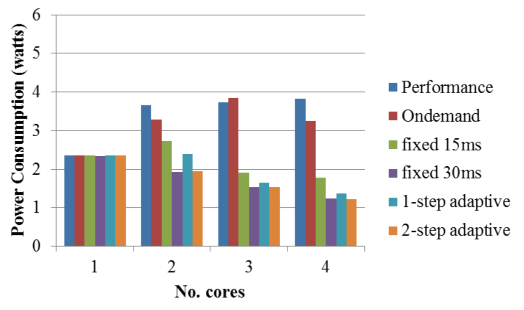
	1	2	3	4
Performance	2,224	7,918	12,351	16,205
Ondemand	2,035	7,120	11,611	15,631
fixed 15ms	883	5,655	10,202	14,598
fixed 30ms	525	4,335	8,902	12,822
1-step adaptive	1,961	5,221	10,625	14,225
2-step adaptive	2,100	7,515	11,290	15,445

(c)

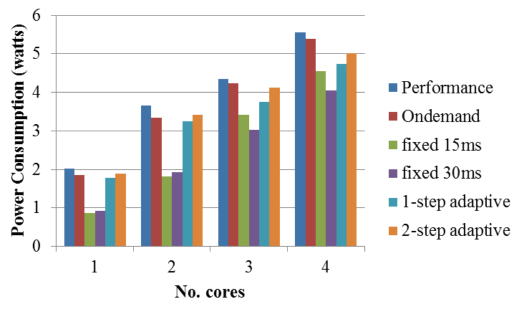
Fig. 3. Suricata performance among scheduling approaches over various big ARM Cortex-A15 at traffic (a) 1,000 (b) 10,000 (c) 17,000 packets/seconds.



(a)



(b)



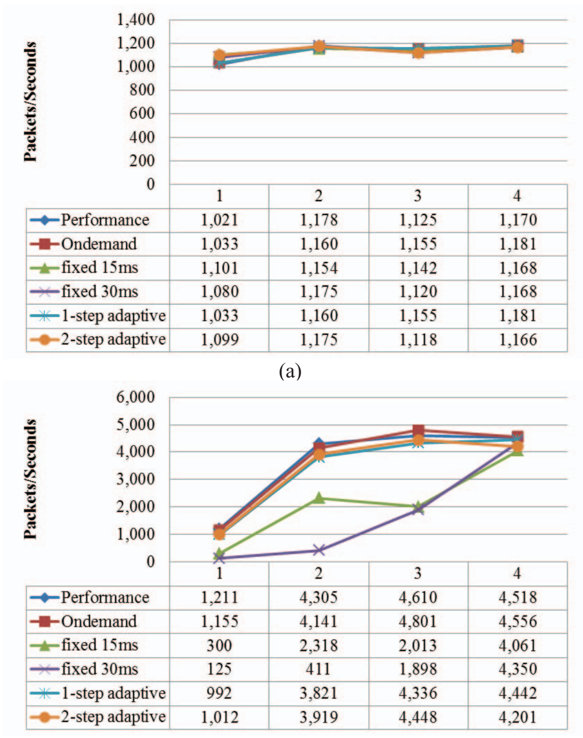
(c)

Fig. 4. The average power consumption of Suricata among power scheduling approaches over various big ARM Cortex-A15 at traffic (a) 1,000 (b) 10,000 (c) 17,000 packets/seconds.

1.5 watts and 1.2 watts in respect with on 3 and 4 cores. Fig. 4 (c) illustrates the power usage of Suricata at the big network traffic, 17,000 packets/seconds. Clearly, 2-step scheduling with the adaptive control interval can provide about 10% lower power consumption than the Ondemand governor on 4 cores. In addition, it still maintains the data processing capacity very well at 2,100, 7,515, 11,290, and 15,445 packets/seconds on 1, 2, 3 and 4 cores respectively compared with other techniques as illustrated in Fig. 3 (c). For 2-step with the fixed 30ms interval, it attains much lower power consumption at the network traffic rate of 17,000 packets/seconds as Fig. 4 (c) shown, but this method causes the performance violation considerably compared with the adaptive mechanisms. For instance, it just processes 525, 4,335, 8,902, and 12,822 packets/seconds on 1, 2, 3 and 4 cores.

Fig. 6 (a) shows that power consumption can be reduced about 65% on four cores by using both 2-step adaptive and fixed 30 milliseconds control interval in comparison with the Ondemand governor at the network traffic 1,000 packets/seconds. Regarding to the performance capability, at the network traffic 5,000 packets per seconds, 2-step controlling with both the fixed 15 and 30 milliseconds interval causes a major degradation of performance on 1, 2, and 3 cores as shown in Fig. 5 (b). In contrast, the adaptive control interval can maintain the performance capability over the different number of cores comparable with the Performance and Ondemand governor. Typically, the Ondemand governor scaling and 2-step scheduling algorithm with the adaptive interval process 1,155 and 1,012 packets/seconds respectively while the fixed 15ms and 30ms just process only 300 and 125 packets/seconds on one core.

We combine 2-step scheduling with the adaptive control interval with the algorithm 3 presented in Section III to measure power consumption as well as performance in the case of Suricata and another program executed at the network traffic 10,000 packets/seconds on ODROID-XU3 board. Combining the core controlling algorithm and 2-step power scheduling achieves lower power consumption than the Ondemand governor as in Fig. 8. It attains an average of 1.25 watts on big cores and an average of 0.362 watts on small cores as shown in Fig. 7. In contrast, the usage of the Ondemand governor consumes an average of 3.42 watts on big ARM Cortex-A15 cores and an average of 0.251 watts on small cores. It is clear that the data processing capacity of two these approaches are almost the same as in TABLE I.



	1	2	3	4
Performance	1,021	1,178	1,125	1,170
Ondemand	1,033	1,160	1,155	1,181
fixed 15ms	1,101	1,154	1,142	1,168
fixed 30ms	1,080	1,175	1,120	1,168
1-step adaptive	1,033	1,160	1,155	1,181
2-step adaptive	1,099	1,175	1,118	1,166

(a)

	1	2	3	4
Performance	1,211	4,305	4,610	4,518
Ondemand	1,155	4,141	4,801	4,556
fixed 15ms	300	2,318	2,013	4,061
fixed 30ms	125	411	1,898	4,350
1-step adaptive	992	3,821	4,336	4,442
2-step adaptive	1,012	3,919	4,448	4,201

(b)

Fig. 5. Suricata performance among scheduling approaches over various small ARM Cortex-A7 at traffic (a) 1,000 (b) 5,000 packets/seconds.

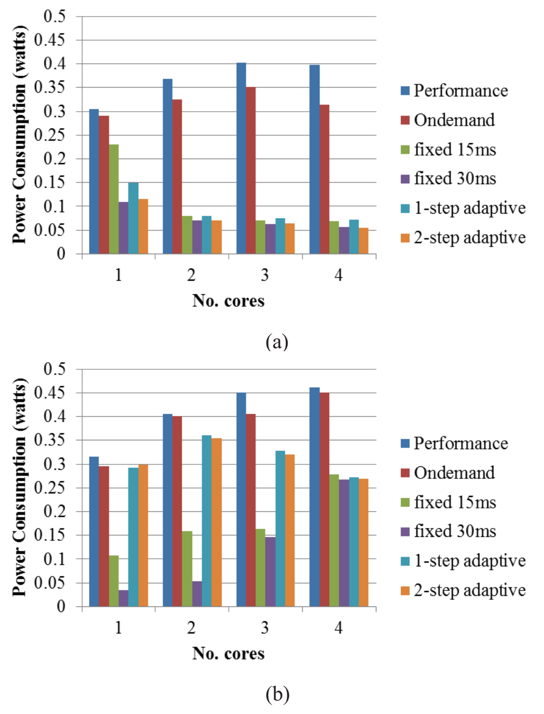


Fig. 6. The average power consumption of Suricata among power scheduling approaches over various small ARM Cortex-A7 at traffic (a) 1,000 (b) 5,000 packets/seconds.

TABLE I. PERFORMANCE WHEN SURICATA AND ANOTHER PROGRAM EXECUTED ON ODROID-XU3 AT NETWORK TRAFFIC 10,000 PACKETS/SECONDS.

Governor scaling	Performance (packets/seconds)
Ondemand	10,122
2-step adaptive	10,067

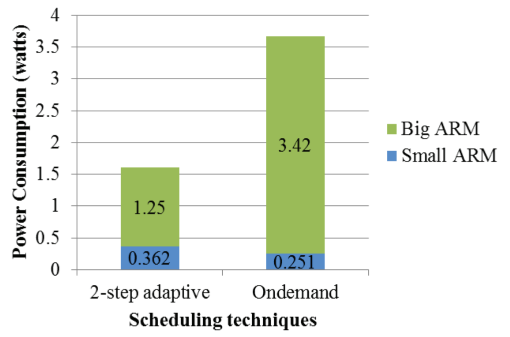


Fig. 7. The average power consumption between Ondemand and 2-step scheduling with adaptive interval combining with core controlling algorithm when Suricata and another program executed on big ARM Cortex-A15 and small ARM Cortex-A7 at traffic 10,000 packets/seconds.

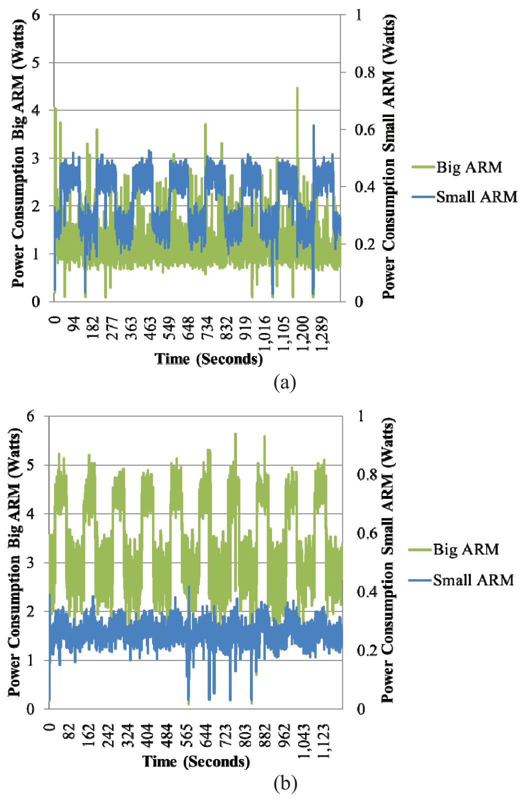


Fig. 8. Changes in the power supply between (a) 2-step scheduling with the adaptive interval combining with the core controlling algorithm (b) Ondemand when Suricata and another program executed on big ARM Cortex-A15 and small ARM Cortex-A7 at traffic 10,000 packets/seconds.

V. CONCLUSION

In this paper, we proposed the 2-step power scheduling algorithm with the adaptive control interval to reduce the power consumption in Suricata. Additionally, we also presented an algorithm to adjust the number of cores in order to maintain power optimization and the performance capacity in case of Suricata and another program executed on the same multicore system as in a home server, or a mobile device. The evaluations show that the 2-step power scheduling with the adaptive control interval achieves the best power saving while maintains the data processing capability for both low and big network traffic on ODROID-XU3 board. Typically, the 2-step scheduling with the adaptive control interval and the fixed 30-millisecond interval can save an average of 87% power usage without affecting their performance considerably compared with the Performance governor in Linux over various big cores at traffic 1,000 packets/seconds. Furthermore, at 17,000 packets/seconds, the 2-step power scheduling with the adaptive interval can ensure the performance very well, while the fixed 30-milliseconds interval violates the performance over various cores.

REFERNCES

[1] J. Xiao, C. Xu, and L. Zeng, "A Time Reservation Dynamic Algorithm based on the Variable Frequency Interval DVFS Technology" International Conference on Computer Science and Service System (CSSS 2014).

[2] L. Zhou, L. Li, X. Li, "A Low Power Consumption Frequency Adaptation Mechanism based on the Traffic and Implementation on NetFPGA", International Journal of Future Generation Conmmunication and Networking Vol. 7, No. 6, 2014, pp-141-154.

[3] E. Nave, R. Ginosar, "PBD: Packet Buffer DVFS", Proc. of the 23rd AMC International Conference on Great Lakes Symposium on VLSI (GLSVLS'13), pp.319-320, 2013.

[4] G. A. Geronimo, J. Werner, R. Weingartner, C. B. Westphall, C. M. Westphall, "Provisioning, Resource Allocation, and DVFS in Green Clouds" International Journal on Advances in Networks and Services, Vol. 7, No 1 & 2, 2014.

[5] D. Cheng, Y. Guo, X. Zhou, "Self-tuning Batching with DVFS for Improving Performance and Energy Efficiency in Servers", Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2013 IEEE 21st International Symposium, August 2013, pp 40-49.

[6] P. Lama, Y. Guo, C. Jiang, X. Zhou, "Autonomic Peformance and Power Control for Co-located Web Applications on Virtualized Servers Quality of Service (IWQoS)", 2013 IEEE/ACM 21st International Symposium, June 2013, pp 1-10.

[7] A. Gandhi, M. Harchol-Balter, R. Das, C. Lefurgy "Optimal Power Allocation in Server Farms", Proceedings of 11 International Joint Conference on Measurement and Modeling of Computer Systems, June 2009.

[8] H. Jiang, G. Xie, K. Salamatian, "Load Balancing by Ruleset Partition for Parallel NIDS on Multi-Core Processors", IEEE International Conference on Computer Communications and Networks, ICCCN, 2013.

[9] D. J. Day, B. M. Burns, "A Performance Analysis of Snort and Suricata Network Intrusion Detection and Prevention Engines", ICDS 2011: The Fifth International Conference on Digital Society.

[10] E. Albin, "A Comparative analysis of the Snort and Suricata Instrution Detection System", (Doctoral dissertation, Monterey, California. Naval Postgraduate School), 2011.

[11] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. Culler, R. Katz, "NapSAC: Design and Implementation of a Power-Propotional Web Cluster", ACM SIGCOMM computer communication review 41.1 (2011): 102-108.

[12] N. Sharma, S. Barker, D. Irwin, P. Shenoy, "Blink: Managing Server Clusters on Intermittent Power", ACM SIGPLAN Notices. Vol. 46. No. 3. ACM, 2011.

[13] D. Meisner, C. M. Sadler, L. A. Barroso, W. Weber, T. F. Wenisch, "Power Management of Online Data-Intensive Services", Computer Architecture (ISCA), 2011 38th Annual International Symposium on IEEE, 2011.

[14] D. Meisner, B. T. Gold, T. F. Wenisch. "PowerNap: eliminating server idle power" ACM Sigplan Notices. Vol. 44. No. 3. ACM, 2009.

[15] K. Mizotani, Y. Hatori, Y. Kumura, M. Takasu, H. Chishiro, N. Yamasaki, "An integration of imprecise computation model and real-time voltage and frequency scaling", In Proceedings of the 30th International Conference on Computers and Their Applications, CATA 2015. (pp. 63-70). The International Society for Computers and Their Applications (ISCA).

[16] http://suricata-ids.org/download/

[17] http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127

[18] http://tcpreplay.synfin.net/