EANeM: Energy-Aware Network Stack Management for Mobile Devices

Chungseop Lee
Sogang University
chung1204@sogang.ac.kr

Keonhyuk Lee
Sogang University
khlee23@sogang.ac.kr

Mingoo Kang
Sogang University
kangtop729@sogang.ac.kr

Hyukjun Lee*
Sogang University
hyukjunl@sogang.ac.kr

***Abstract*—In mobile computing, various energy-efficient thread scheduling schemes for heterogeneous multiprocessing architectures are proposed. For network applications, however, inaccurate prediction of the CPU load and high-priority network packet processing overdrive CPU cores, leading to large energy consumption. We present a framework including a network stack monitor, a bandwidth controller, and an energy-efficient thread scheduling scheme, which accurately estimates the CPU load for packet processing and optimally schedules CPU resource. It improves performance/watt by 4.79 times over the baseline Linux scheduler for FTP applications. In multi-threaded environments, it improves performance/watt by 2.35–3.11 times for PARSEC benchmark and network applications.**

***Index Terms*—Network stack, heterogeneous multiprocessing, low power design, packet processing**

I. Introduction

Dynamic voltage frequency scaling (DVFS) and dynamic power management (DPM) are widely employed in mobile computing devices to achieve low power/energy consumption [1] [2]. Meanwhile, heterogeneous multiprocessing (HMP) architectures containing both high-performance and low-performance cores are commercialized to allocate the energy-optimal processor core to a given application. Numerous thread scheduling schemes have been proposed to exploit HMP systems for optimal performance/watt [3] [4] [5] [6] [7].

To achieve optimal performance/watt, Muck *et al.* [3] accurately models power and performance of applications mapped on different cores in HMP systems. Their work considers predicted instructions per cycle (IPC), thread load contribution, and the scheduling policy (e.g. CFS) to find an optimal mapping for threads. Similarly, Pricopi *et al.* [4] estimates performance and power consumption of applications on asymmetric multi-cores and achieves the optimal mapping. In the commercial domain, Linux introduces its first energy-aware scheduler (EAS) for HMP systems in Kernel version 5.0. It heuristically schedules a thread to an energy-optimal processor core using the energy model (EM) for the CPU cores, the PELT load estimation algorithm, and *schedutil*—CPU frequency governor [6].

In the self-adaptive computing domain, on the other hand, application-level performance metrics such as heartbeats [7] [8] are used because system-level performance metrics such as IPC cannot represent application-level performance requirement (e.g. frame rate). Yun *et al.* [7] achieves optimal performance/watt by estimating the best core type and operating frequency for multi-threaded applications.

As the network bandwidth requirement for network applications increases and high-bandwidth network protocols (5G or Gigabit WiFi) enable hundreds-of-Mbps network interface in mobile devices [9], network packet processing in mobile devices consumes significant computing resources and energy.

However, prior works cannot achieve optimal performance/watt for applications using the network stack (a part of the operating system supporting network functions) for several reasons. First, system-level and application-level performance metric (e.g. IPC and heartbeat) cannot represent the performance metric of network applications — bandwidth (e.g. transferred bytes per second). The achieved bandwidth is a function of the number of processed packets and the average packet size of network connections (or flows). The number of processed packets is proportional to CPU usage. As shown in Fig. 1, two connections with the same bandwidth requirement (50 Mbps) could require different CPU loads (50% and 30%) as they have different average packet sizes. Without knowing the average packet size of a network connection, we cannot estimate the CPU load to guarantee its bandwidth requirement. Typically, a shared interrupt handler, *net_tx* or *net_rx softirq*, performs packet processing for multiple network connections. Energy-saving techniques and misprediction of CPU load could allocate insufficient resource and incur serious inter-flow interference among different network connections. These problems cannot be resolved without per-flow monitoring of network connections and accurate load estimation. Second, the time-consuming network packet processing (e.g. TCP packet processing) is performed by interrupt contexts (e.g. *softirq*). Interrupt contexts have higher priority than (or preempt) user threads and are asynchronous to thread scheduling. Network packet processing exhibits often bursty behavior as network packets arrive in bursts from the network interface. High-priority interrupt handling of network packet processing forces the HMP schedulers to keep using the higher clock frequency or high-performance cores to process bursty packets.

Providing solutions to the issues, we propose a framework for energy-aware network stack management. Major contribution of our work can be summarized as follows.

- We identify the issues with prior thread scheduling schemes for HMP systems in the applications using the network stack. First, they do not have means to control high priority network packet processing and bursty net-

*Corresponding author

978-1-7281-1085-1/20/$31.00 ©2020 IEEE

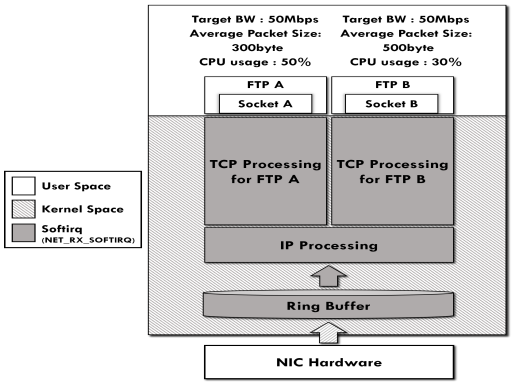

Fig. 1. Multiple TCP Connections and OS Network Stack.

work traffic, which causes frequent use of big cores at high frequency and wastes energy for processing network packets. Second, inaccurate load estimation of network packet processing due to lack of network stack monitoring leads to poor performance/watt in prior scheduling methods.

- We develop a framework that accurately monitors internals of the network stack for CPU load estimation and regulates the bandwidth of network applications to meet the requirement of network packet processing.
- We propose an Energy-Aware Network stack Management (EANeM) scheme. First, we rate-limit the network bandwidth of network-applications, indirectly controlling bursty high priority execution of packet processing, to simultaneously meet the QoS requirement of network and non-network applications, leading to optimal use of cores and less jitters for non-network applications. Second, accurate load estimation and scheduling for the interrupt handler for network packet processing enable optimal core/frequency mapping and lead to significant improvement in performance/watt over the baseline scheme.

To the best of our knowledge, this is the first work to rate-limit/schedule high priority interrupt contexts or kernel threads for HMP systems to achieve energy reduction. Our network processing thread scheduling scheme can be integrated with prior works [3] [5] [7] to resolve issues with network applications.

The paper is organized as follows. In section II, we review the TCP processing of Linux network stack and various energy-efficient thread scheduling schemes for the HMP systems. In section III, we discuss the proposed scheme. Then, experimental results are presented in section IV, followed by related works and conclusions in section V and VI.

II. Background

A. Network stack and TCP processing

The network stack in operating system includes kernel codes processing network packets in physical, data link, network, and transport layers. The most time consuming part of a network stack in Linux is TCP packet processing (a part of transport layer). Physical, data link, and network layer processing account for only a small portion of total network processing time. In current network, TCP is a dominant transport protocol. TCP packet processing is performed by the interrupt handler. Interrupt handling in general is broken into top and bottom half code [10]. Packet transmission between a network interface card (NIC) and a kernel ring buffer is processed by the top half code (NIC's hardware interrupt). Packet processing in network and transport layer is processed by the bottom half code (*net_tx* or *net_rx softirq* [10]). Processing a large amount of TCP packets takes considerable CPU resource. When too many TCP packets are processed, a kernel thread, *ksoftirqd*, is launched and *net_tx* or *net_rx softirq* is processed by *ksoftirqd*. On the other hand, when a small-to-medium number of TCP packets are processed, *softirq* is run as a high priority interrupt context. Bursty TCP processing performed in high priority context creates serious problems for the conventional energy-aware schedulers as the network bandwidth requirement increases rapidly.

B. Energy Efficient Thread Scheduling on Heterogeneous Architectures

From Linux kernel version 5.0 onward, CPU frequency governor (*schedutil*) and energy-aware scheduler (EAS) are used together to estimate loads for threads and find the energy optimal core in HMP systems. EAS has an energy model (EM) which is based on a power cost table. The power cost table stores power consumption for each CPU capacity (performance level) sorted by CPU frequency and CPU types (e.g. little, big). CPU capacity represents the maximum amount of CPU work that can be processed at given CPU frequency and CPU type. EAS predicts CPU utilization with the PELT algorithm and allocates a thread to a CPU core that provides optimal energy using the *schedutil* and EM [6]. In academia, several schemes [3] [4] provide an optimal mapping of threads in HMP systems using IPC.

In the self-adaptive HMP computing domain, on the other hand, application-level performance metrics such as heartbeats or frame rates [7] [8] are used. Hoffman *et al.* [8] proposes a software framework to guarantee application-level performance by inserting a heartbeat API into the major loop of applications. HARS [7] extended this work to achieve optimal performance/watt by estimating the best core type and operating frequency for multi-threaded applications in HMP systems.

However, prior works neglect the unique characteristics of network applications and their estimation for system-level or application-level performance is often very inaccurate due to the lack of monitoring the internals of network stack. In addition, all prior scheduling methods cannot control the bursty high priority packet processing that can cause abuse of high performance cores and incur starvation or large jitters to low priority user threads due to the lack of packet processing control.

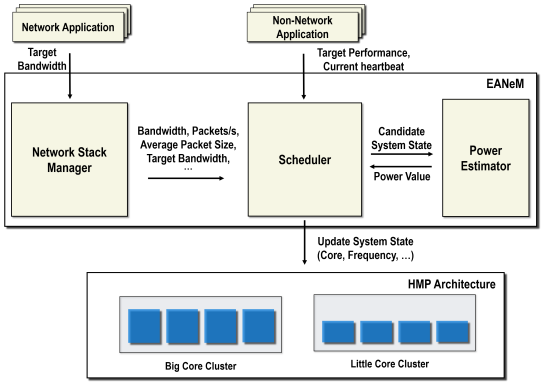

Fig. 2. Architecture Overview.

Algorithm 1 Bandwidth control using TCP receive window

```
1:  procedure TCP_SELECT_WINDOW(current_time)
2:      Initialization: // Initialization is executed once per process
3:          BW_target[i] = BW_target_prev[i] = 0
4:          previous_time = current_time
5:          // BW_target[i] can be changed externally
6:          // i is the index of the process that owns the socket
7:      elapsed_time = current_time - previous_time
8:      if elapsed_time > monitoring_interval then
9:          if BW_target[i] > 0 then
10:             if BW_target[i] ≠ BW_target_prev[i] then
11:                 RWIN[i] *= (BW_target[i]/BW_current[i])
12:                 BW_target_prev[i] = BW_target[i]
13:             else if BW_target[i] ≠ BW_current[i] then
14:                 RWIN[i]+=(BW_target[i] − BW_current[i]) ∗ δ
15:     previous_time = current_time
```

III. Energy-Aware Network Stack Management

A. Overview of System Architecture

Fig. 2 presents the overall architecture of the proposed scheme. We implement a network stack manager (a network stack monitor and a network bandwidth controller), which monitors packets through various network layers and limits the network bandwidth of flows respectively. Using statistics collected by the monitor and the power consumption values given by the power estimator, the scheduler determines the optimal core and frequency for network and non-network applications. At the same time, the bandwidth controller limits the network bandwidth to the specified network bandwidth for controlling high priority network processing. The scheduler part of EANeM uses a two-level scheduling. First, it schedules network packet processing threads and network interrupt handler (*net_tx*, *net_rx softirq* and NIC's hardware interrupt) to the optimal core or cores. Then, it schedules non-network threads using HARS [7]. However, this second level of scheduling can be performed with any prior energy efficient thread scheduling schemes.

B. Network Stack Monitor

We build a monitor inside Linux kernel to get information which is not easily accessible from the user level. It is implemented as a kernel module and its overhead is small. The proposed monitor looks into internals of the network stack, which are not visible using existing monitoring tools. It monitors the packet latency through layers (physical, data link, network, transport layer in the network stack), packet size, and the number of packets per unit time so that average latency or bandwidth can be measured. To measure the average packet size of a network connection, we collect packet length from packets before Linux kernel copies data to a user buffer and store the packet length in the storage which is located in the socket structure. Every TCP/UDP connection has its own socket instance which is managed by Linux kernel. Therefore allocating and de-allocating resource is automatically performed when a socket is created and destroyed.

At every logging interval, we read statistics storage, calculate average and variance of collected data, log results using RSYSLOG (Rocket-fast System for Log Processing), and re-initialize statistics storage for the next result. Logging interval is 0.1 second by default and its overhead is minimal.

C. Network Bandwidth Control

Limiting network bandwidth has several benefits. It could be used to reduce the power/energy consumption. We can limit network bandwidth to the target value to optimize the clock frequency and core type. In addition, it reduces burstiness of packet transmission. Bursty packet processing causes CPU to overreact and raise clock frequency frequently.

Algorithm 1 shows the algorithm for network bandwidth controlling. We modulate the TCP receive window ($RWIN$) to control network bandwidth (BW). To modulate the TCP receive window, bandwidth controlling is implemented in the Linux tcp_select_window function. The tcp_select_window function is called to set the TCP receive window size of a socket before sending a TCP acknowledge packet. When tcp_select_window function is called and the elapsed time is larger than the sampling interval, it checks whether target bandwidth (BW_{target}) is set to a non-zero value (line 8-9). When target bandwidth is set, Algorithm 1 updates the size of a receive window. If the target bandwidth of a TCP connection has changed, the receive window changes in proportion to the bandwidth difference so as to catch up the difference rapidly (line 10-12). Otherwise, the algorithm compares the target and current achieved bandwidth and slowly increments/decrements the window size to avoid fluctuation (line 13-14). Although bandwidth controlling is designed for TCP, it can be generalized to any emerging protocols that have a bandwidth control mechanism.

D. Performance Estimation for Network Stack TCP Packet Processing

In applications using the network stack, the most time-consuming part of the network stack is TCP processing. It is performed by *net_tx* or *net_rx softirq*. *softirq* processes packets and the number of packets processed per unit time is proportional to the CPU load for network packet processing. To allocate proper CPU load for packet processing and guarantee required network bandwidth, we should be able to estimate the performance of TCP processing in the network stack. For this,

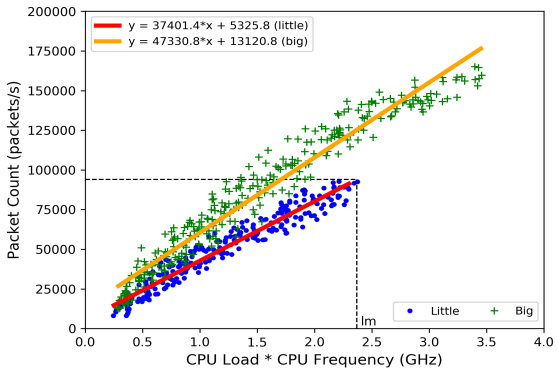

Fig. 3. Packet processing vs CPU resource with respect to different average packet sizes and bandwidths of TCP flows

one should know (1) average packet size of the application and (2) relationship between CPU load at given clock frequency and the number of processed TCP packets.

Fig. 3 shows the number of TCP packets processed in FTP with respect to the product of clock frequency and CPU load for big cores (Cortex-A15) and little cores (Cortex-A7). We map network threads and network interrupt handler to the same cores (big or little). The value 1 in x axis stands for 100 % load of the core at 1 GHz. The number of packets is proportional to the product of clock frequency and CPU load as modeled in (1).

$$N_k = \alpha_k * L_k * F_k + \beta_k \quad (1)$$

where N_k is the number of packets processed per second and L_k and F_k are the CPU load (in fraction) and frequency (in GHz) of CPU core type k. α_k and β_k are coefficients obtained using linear regression as show in Fig. 3. They differ for big and little cores. Obtaining coefficients can be performed only once by running a script[1].

Multiplying the measured average packet size with the number of processed packets estimates the achieved network bandwidth. For network applications, we can model the network bandwidth, B_i, for the application i.

$$B_i = N_k * S_i \quad (2)$$

where N_k is the number of packets processed per second for core k and S_i is the average packet size for the application i.

E. Power Model

Power model for a single core cluster is given by (3). L_{total} is a sum of CPU loads for applications mapped on the core cluster, which acts as an activity factor. V_F is the voltage for guarantee the frequency F and λ is a constant. $\theta(V_F)$ is a static power consumption. We estimate parameters, λ and $\theta(V_F)$, using linear regression in the commercial board (e.g. Odroid XU3 board with a power sensor).

$$P = \lambda * L_{total} * V_F^2 * F + \theta(V_F) \quad (3)$$

[1]We designed a script to automatically extract coefficients in Eqn. 1. This technique can be extended to other protocols such as UDP.

Algorithm 2 Network Application Mapping Algorithm

```
1:  procedure MapNetworkApp(nc_L,nc_B)
2:      N_total = 0, P_min = FLOAT_MAX
3:      for i in NetworkAppList do
4:          N_total += BW_target[i]/S[i]
5:      R_L = (N_total-β_L)/α_L, R_B = (N_total-β_B)/α_B
6:      for i in [1, nc_B] do
7:          n_L = 0, n_B = i, f_L = f_L.min, f_B = R_B/i
8:          P_cur = EstimatePower(f_L,n_L,f_B,n_B)
9:          if P_min > P_cur then
10:             state = UpdateState(f_L,n_L,f_B,n_B)
11:             P_min = P_cur
12:     if R_L ⩽ lm then
13:         for i in [1, nc_L] do
14:             n_L = i, n_B = 0, f_L = R_L/i, f_B = f_B.min
15:             P_cur = EstimatePower(f_L,n_L,f_B,n_B)
16:             if P_min > P_cur then
17:                 state = UpdateState(f_L,n_L,f_B,n_B)
18:                 P_min = P_cur
19:     MapNetworkAppToState(state,NetworkAppList)
```

F. Energy-Aware Scheduling for mixed network and non-network applications

EANeM performs a two-level scheduling. The first-level schedule maps network applications as the network packet processing has priority over non-network applications. The second-level schedule maps non-network applications.

First of all, we map network threads and the network interrupt handler to appropriate cores as shown in Algorithm 2. The algorithm determines the best configuration (core type, core number, frequency) to simultaneously satisfy the load requirement for packet processing and incur the least power consumption. The algorithm takes the number of available big or little cores (nc_B or nc_L) to be used for packet processing as inputs. In line 3-4, the total number of packets, N_{total}, to be processed to guarantee the bandwidth of all network connections is determined. Then, the product of CPU load and frequency (x axis in Fig. 3), R_L (little core) or R_B (big core), to guarantee N_{total} is determined (line 5), using the model in (1) and Fig. 3.

Assume that packet processing can be mapped to 1 to nc_L or nc_B cores in either little core or big core cluster. From line 6 to 11, we choose the best configuration for the big core cluster. We sweep the number of big cores in line 6 and estimate the power consumption in line 8 using the model in (3). In line 7, n_L, n_B, f_L, f_B are the number of little and big cores and the frequency of little and big cores in the current configuration. f_B is divided by the number of cores assuming cores are 100% utilized. If the current configuration shows less power consumption, $state$ is set to the current configuration (line 9-11). From line 12 to 18, we choose the best configuration for the little core cluster. In this case, however, we explore configurations only when the load requirement is less than lm as shown in Fig. 3 (line 12). The chosen configuration among little and big core clusters provides the best performance/watt for network applications.

The selected cores and core frequencies become the starting points for the 2nd-level scheduling which maps non-network applications. In EANeM, we map non-network applications

TABLE I
ODROID-XU4 SPECIFICATION

CPU	Samsung Exynos 5422 (Cortex-A15 Quad 0.2 - 2 GHz, Cortex-A7 Quad 0.2 - 1.4 GHz)
Memory	2 Gbyte LPDDR3 RAM at 750 MHz
Network	1 Gbps Ethernet
OS	XUbuntu 18.04 Linux kernel 5.0.3

using HARS [7] because it uses application-level performance metric such as heartbeats as a performance metric. It explores the number of big and little cores and their frequencies around the best configuration (core mapping and frequency) chosen by Algorithm 2 and finds an optimal mapping that satisfies the performance requirement and gives the least power consumption. The 2nd-level scheduling explores the core frequencies equal to and larger than ones chosen in Algorithm 2. If the network packets can be processed with fewer cores as the frequency increases by the 2nd-level scheduling, the number of cores for processing the network packet may be reduced and the number of cores for non-network applications may be increased. Although we use HARS for the 2nd-level scheduling, however, EANeM can be combined with any prior energy efficient thread scheduling works [3] [5] [7].

IV. EVALUATION

A. Methodology

We use the Odroid-XU4 board (XUbuntu 18.04 and Linux Kernel 5.0) with 1 Gbps LAN NIC to emulate a mobile computing environment. Its detail specification is shown in Table I. It employs the ARM's big.LITTLE processor architecture consisting of four little cores (Cortex-A7) and four big cores (Cortex-A15). Power consumption is estimated from the model shown in (3) [2].

We use Linux Kernel 5.0, which implements CPU frequency governor *schedutil* and energy-aware scheduler (EAS). We use EAS as a baseline scheduler. For experiments with FTP application threads, all experiments are conducted between an FTP server and an Odroid-XU4 board. For multi-thread applications, we choose five PARSEC benchmark applications (bodytrack (BO), facesim (FA), ferret (FE), fluidanimate (FL), and swaptions (SW))[3] [11]. Codes are modified to report their dynamic performance — application heartbeats [8].

We use performance/watt as a metric for comparisons. The performance is measured as min(g,h)/g as in [7] where g is a performance goal and h is the achieved performance.

B. Experimental Results

We evaluate the proposed scheme in three aspects: (1) energy consumption in network applications and mixed applications (network and non-network), (2) accuracy of bandwidth control, and (3) framework overheads.

[2]We build power model using Odroid-XU3 board where power sensing for cores can be obtained using a TI INA231 chip because the Odroid-XU4 board uses the same HMP architecture but does not support the power sensor.

[3]Each application spawns eight threads.

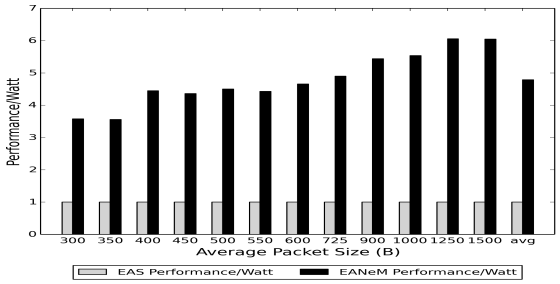

Fig. 4. Comparing EAS and EANeM in an FTP application (averaged over 100 Mbps - 400 Mbps)

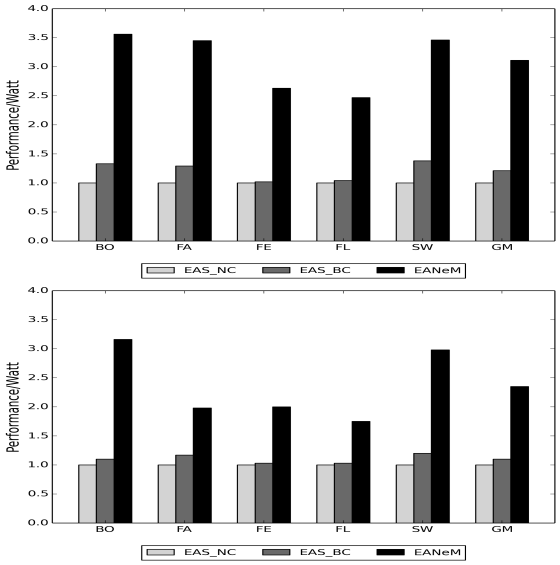

Fig. 5. Comparing different scheduling methods in terms of Performance/Watt for PARSEC and FTP applications.

1) Network applications: The energy consumption of Linux's energy-aware scheduler (EAS) and our power management scheme (EANeM) for a single FTP application are shown in Fig. 4. Note that in both schemes, the FTP connection bandwidth is controlled to the target bandwidth which is swept from 100 Mbps to 400 Mbps with interval of 100 Mbps. High-priority bursty TCP packet processing forces Linux EAS to frequently change to high clock frequencies and a big core. On the other hand, EANeM accurately estimates the CPU load and clock frequency to support the target bandwidth, chooses an optimal core, forces the clock frequency down to the optimal value, resulting in performance/watt improvement by a factor of 4.79 over EAS.

2) Mixed applications: In this experiment, we compare EANeM with two versions of EAS: one with and without network bandwidth control represented as EAS_BC and EAS_NC respectively. We use EAS_NC as a baseline. All results are normalized to the baseline. The results show performance/watt for different methods when we run 5 PARSEC benchmark

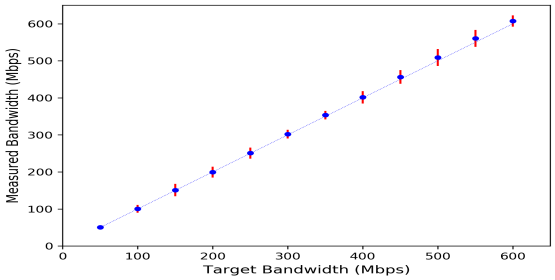

Fig. 6. Controlling network bandwidth by modulating the TCP receive window of a FTP application

applications with two FTP applications (150 Mbps each). The heartbeat rate of PARSEC benchmark applications are set to 50% and 75% of the maximum rate (unconstrained). EAS_BC is the case where EAS uses a bandwidth control method proposed by EANeM. Network applications/interrupt handling routines are mapped on two little cores as chosen by Algorithm 2. EANeM has 3.11 and 2.35 times larger performance/watt for 50% and 75% performance target over the baseline. EAS_BC (bandwidth controlled version of EAS) shows slightly larger performance/watt than EAS_NC. The improvement is larger for PARSEC applications (BO, FA, and SW) that settle with little frequency increase as their performance requirements are lower.

3) Accuracy of bandwidth control: In this experiment, we periodically change the target network bandwidth (intervals of 50 Mbps) using the proposed controlling scheme and measure the achieved bandwidth using our proposed monitor. Fig. 6 shows the mean and standard deviation of achieved network bandwidth with respect to the target bandwidth.

4) Framework Overheads: In the proposed framework, we provide a monitor and a bandwidth controller. They are run every sampling interval (0.1 seconds). On average, the monitor uses 0.74% of CPU usage as it collects statistics. Meanwhile, the overhead for the bandwidth controller is negligible as it readjusts the receiving window every sampling intervals.

5) Analysis of NIC Power Consumption: EANeM does not increase the power consumption of NIC with respect to prior schemes (e.g. EAS) as it slows down the processor cores not NIC. EANeM can work in both bandwidth-controlled and uncontrolled mode. The energy saving of EANeM mainly comes from forcing the optimal core and frequency only to provide the specified bandwidth. In bandwidth-uncontrolled mode, it can choose and force the optimal core and frequency to provide the maximum bandwidth that NIC provides while NIC operates at its maximum speed.

V. Related works

Low power thread scheduling techniques for HMP architectures have been proposed in many studies [3] [4] [5] [7]. However, these schemes neglect interaction between user threads and network packet processing (or interrupt contexts) and cannot be applied to network applications. Researches on power/energy-efficient network client design have focused on shutting off the network interface card (NIC) to reduce energy consumption due to transmitting/receiving network packets [12]. Our work is complementary to this approach. Finally, prior researches on the network stack have mainly focused on improving the performance of packet processing by reducing the overhead of TCP processing [13] or converting the network stack to a user thread [14]. They do not address the issue of low power consumption design.

VI. Conclusion

Achieving optimal performance/watt for mixed network and non-network applications has not been provided by prior energy efficient thread schedulers. In this study, we present a framework that achieves significant improvement in performance/watt for network and non-network applications while provisioning their performance requirements.

We envision that this work will have an impact on related research areas where bursty high priority jobs competing with various jobs with or without QoS requirements under strict power/energy constraints.

References

[1] L. Benini *et al*., "A Survey of Design Techniques for System-Level Dynamic Power Management," *IEEE Transactions on VLSI Systems*, vol. 8, no. 3, pp. 299-316, 2000.

[2] J. Flinn *et al*., "Energy-aware adaptation for mobile applications," in *Proc. the seventeenth ACM symposium on Operating systems principles*, 1999, pp. 48-63.

[3] T. *Muck et al*., "Run-DMC: Runtime Dynamic Heterogeneous Multicore Performance and Power Estimation for Energy Efficiency," in *Proc. International Conference on Hardware/Software Codesign and System Synthesis*, 2015, pp. 173-182.

[4] M. Pricopi *et al*., "Power-Performance Modeling on Asymmetric Multi-Cores," in *Proc. International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2013.

[5] K. Yu *et al*., "Power-aware task scheduling for big.LITTLE mobile processor," in *Proc. IEEE International SoC Design Conference (ISOCC)*, 2013, pp. 208-212.

[6] The Linux Kernel, "Energy Aware Scheduling," [Online]. Available: https://kernel.org/doc/html/latest/scheduler/sched-energy.html

[7] J. Yun *et al*., "Hars: A heterogeneity-aware runtime system for self-adaptive multithreaded applications," in *Proc. ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015.

[8] H. Hoffmann *et al*., "Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments," in *Proc. the 7th International Conference on Autonomic Computing,* 2010, pp. 79-88.

[9] S.Hossain, "5G wireless communication systems," *American journal of Engineering Research (AJER)*, vol. 2, no. 10, pp. 344-353, 2013.

[10] C. Benvenuti, *Understanding Linux Network Internals, 1st Edition*, sebastopol, CA: OŔeilly Media, Inc., 2005.

[11] C. Bienia *et al*., "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp.72-81.

[12] M. Gupta *et al*., "Dynamic Ethernet Link Shutdown for Energy Conservation on Ethernet Links," in *Proc. IEEE International Conference on Communications*, 2007, pp. 6156-6161.

[13] K. Yasukata *et al*., "Stackmap: Low-latency networking with the os stack and dedicated nics," in *Proc. USENIX Annual Technical Conference (USENIX ATC)*, 2016, pp. 43-56.

[14] E. Jeong *et al*., "mtcp: a highly scalable user-level tcp stack for multicore systems," in *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 489-502.