Jurnal Ilmiah Sinus (JIS) Vol : 18, No. 2, Juli 2020
ISSN (Print) : 1693-1173, ISSN (Online): 2548-4028

High Througput Multicore Server Dengan Kernel *Bypass*

Dimas Febriyan Priambodo
Jurusan Ilmu Komputer dan Elektronika, FMIPA UGM, Yogyakarta
e-mail: dimas.febriyan@poltekssn.ac.id

ABSTRACT

Kernel performs one part of the overall operating system utility as a bridge between the user and the hardware. In multi core hardware, the kernel was also created with a general approach so it needs to be adjusted for specific purposes. Kernels for servers using common Kernels have many limitations, such as single socket descriptor, single IRQ, and lack of pooling so that they require some modification to run optimally. Kernel bypass is a method by eliminating the automation of a server used to realize a server, namely high throughput. Kernel bypass is a combination of techniques, modification at the driver level with hashing rx signal and modification of multiple receivers with multiple ip receivers, multiple thread receivers, and multiple port listeners. This combined modification makes the server more reliable with an average throughput increase of 250.44%.

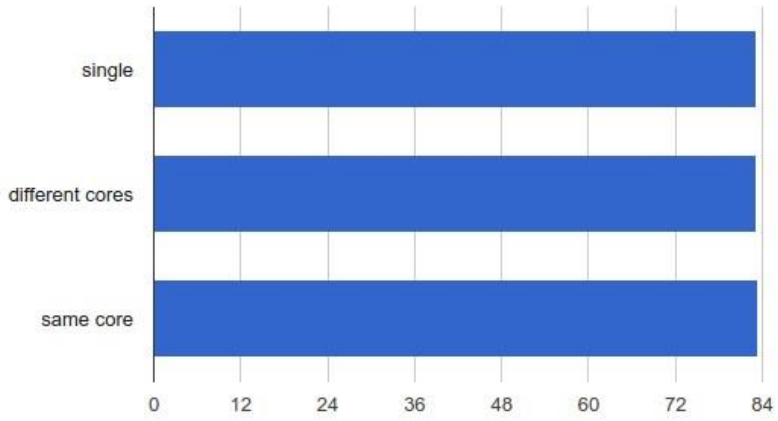
Keyword: *kernel bypass, hashing rx, multiple ip receiver, multiple thread receiver, multiple port listener*

I. PENDAHULUAN

Sebuah sistem operasi dibangun dengan dua komponen utama yaitu dengan GUI (*Graphical User Interface*) dan *kernel* sebagai penghubung antara user dan mesin. Kernel didesain untuk melakukan keseluruhan dari proses yang berlangsung pada komputer. Pendekatan keumuman dijadikan aspek utama perancangan dari sebuah kernel karena untuk melayani kebutuhan user secara umum.

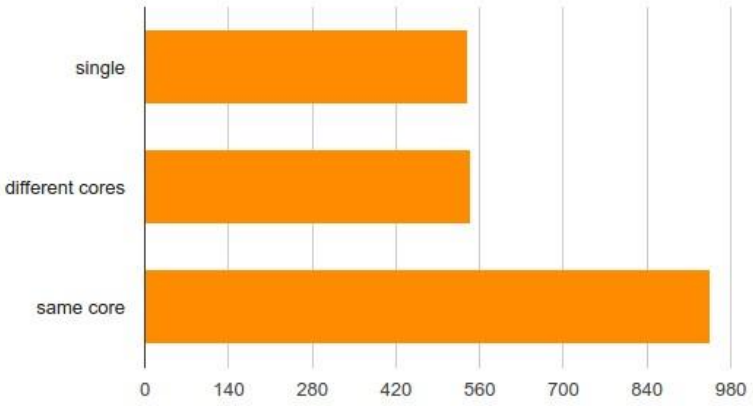
Peladen merupakan seperangkat komputer super atau biasanya berupa *High Performance Computer* (HPC). Pada komputer peladen biasanya akan ditanamkan sistem operasi khusus peladen yang mempunyai utilitas khusus. Namun tidak jarang server menggunakan sistem operasi yang sama dengan sistem operasi *personal computer*. Linux yang banyak digunakan pada peladen mempunyai *kernel* yang sama dengan yang digunakan oleh *personal computer*. Sehingga penelitian ini dilakukan untuk dapat mengoptimalisasi sistem operasi linux pada khususnya untuk mendapatkan peladen yang optimal khususnya dari sisi *throughput*.

Program client server dibuat sebagai salah satu uji pendahuluan yang menunjukkan kelemahan *default kernel* dalam menjalankan fungsinya sebagai peladen terlebih dalam sistem *multicore*. Gambar 1 menunjukkan tidak ada perbedaan signifikan terhadap waktu pemrosesan baik dalam *core* atau inti yang sama maupun dalam inti yang berbeda dan bisa disimpulkan *HyperThreading* bekerja dengan baik karena menjalankan program dalam satu inti yang sama dan tidak mempengaruhi kinerja CPU lainnya. Hasil tersebut juga memberi kesimpulan bahwa kinerja *singgle CPU* tidak terlalu bagus karena bisa berarti tidak effisien.



Gambar 1. Hasil *Core Performance test 1*

Gambar 2 menunjukkan perbedaan yang cukup signifikan pada pemrosesan program yang kompleks dimana menunjukkan waktu proses yang sama pada core yang berbeda namun akan cenderung menurunkan *performance* sebesar 75% (tujuh puluh lima persen) pada core yang sama (*HT*). Analisis juga menunjukkan, dengan menggunakan 2 (dua) *thread* akan melakukan pemrosesan yang lebih cepat. Menggunakan 2 thread menghasilkan catatan waktu 945 ms dan untuk single thread memerlukan waktu (540 ms x 2) 1080ms.



Gambar 2. Hasil *Core Performance test 2*

Gambar 2 juga memberikan kesimpulan perlu adanya sistem untuk menghindari migrasi interupsi yang sering terjadi pada *default kernel multicore processor*. Pada *default kernel* menggunakan metode *round robin* dalam pengelolaan interupsinya, sehingga ada suatu waktu dimana beban proses dibabankan pada *sharing core* sehingga menggunakan *share memory* dan mengakibatkan beban salah satu *core processor* menjadi berlebih, di sisi *core* yang lain mengalami *free resource*.

Server peladen yang berkembang ke arah *multicore* membutuhkan modifikasi berdasar konsep *multicore aware*. Membuat sistem yang *multicore aware* tidak hanya berhubungan dengan *affinity* aplikasi untuk membaginya pada CPU saja, namun juga berkaitan dengan hardware lainnya khususnya yang memberikan *interupt* kepada CPU pada peladen. Modifikasi untuk pengaturan signal dengan memodifikasi algoritma *hashing* menjadi *multi queue* untuk mengatasi kekurangan *single IRQ* pada *kernel*. *Multi queue* juga berarti mengaktifkan *multi thread* interupsi dan memaksanya secara manual menuju *core* yang berlainan.

 DOI : https://doi.org/10.30646/sinus.v18i2.472

Pemaparan permasalahan diatas juga membuka urgensi untuk melakukan modifikasi pada *port listener* sehingga tidak terjadi *lock* pada *default kernel*, dengan *binding socket* maka setiap *port* dapat digunakan secara bersamaan oleh semua CPU tanpa terjadi *lock* dan proses *waiting* yang meningkatkan *load* dari *processor*. Penggabungan modifikasi ini diharapkan menjadi jawaban kebutuhan komputer peladen yang baik ditunjukkan dengan throughput yang tinggi.

II. TINJAUAN PUSTAKA

Penelitian ini mengacu pada beberapa referensi sumber pustaka yang berasal dari penelitian yang sudah dilakukan sebelumnya. Adapun beberapa penelitiannya yaitu simulasi pengaruh *core affinity* pada kenaikan *throughput* (Hanford et al., 2015). Penelitian ini melengkapi kedua penelitian diatas yang berkesimpulan peningkatan *performance* proses menggunakan *thread* pada *core* yang berbeda namun harus pada *socket* yang sama menjadi nyata dengan metode *kernel bypass* .

Penerapan teknologi *socket* mengacu pada NUMA (*Non Uniform Memory Access)* dikerjakan pada simulasi yang dilakukan oleh (Angelo et al., 2016) yang berkonsentrasi pada *thread sceduling*. Sejalan hasil simulasi tersebut Penelitian (Bo et al., 2016), (Diener *et al*.,2016) juga menyatakan peningkatan *performance* dengan NUMA. NUMA mampu meningkatkan *performance* dengan menghalangi perpindahan *thread* melalui *partitioning memory*. Bertentangan dengan beberapa penelitian tersebut, penelitian ini menjawab apakah NUMA memang solusi yang tepat untuk meningkatkan kemampuan peladen atau ada kombinasi yang lebih baik. Jika diperhatikan lebih detil beberapa penelitian tentang NUMA diatas belum menggunakan program yang *cache sensitive* sehingga tersamarkan peningkatannya. Peningkatan optimal terjadi jika proses diperoleh dan diolah dalam 1 (satu) *interface socket* yang sama. Proses pengolahan *message* dengan *cache sensitive* memungkinkan terjadi perpindahan *socket* karena program akan melakukan pencarian *latency* terbaik.

Pertukaran data atau *large scale data replication*, data harus dilewatkan melalui *Wide Area Network* (WAN) yang merupakan inter sistem dan mempunyai banyak batasan. Penelitian ini memberikan rekomendasi pentingnya sebuah *hardware* mampu mengatur *affinity* baik *core* maupun *socket* dalam melakukan pemrosesan penerimaan data sehingga mengoptimalkan sistem secara keseluruhan. Penelitian ini semakin mengukuhkan pentingnya mengatur sebuah *affinity* khususnya dalam sebuah *network* karena didalam sebuah *hardware* meskipun dalam satu kesatuan *motherboard* juga merupakan jaringan *network* atau disebut *inter system*.

Perkembangan *hardware* sayangnya tidak seiring dengan perkembangan penggunaan *software*. Kesalahan programmer dalam membuat program dikarenakan memang beberapa *developer* tidak memasukkan fitur untuk menangani perkembangan *multicore*, hal ini membuat programmer terbiasa untuk menghasilkan program yang tidak *core aware*. Program yang tidak *core aware* akan cenderung tidak efisien. Dalam gambaran sederhana, ada beberapa program yang tidak memunculkan perbedaan signifikan jika dikerjakan di *core i3* dan *core i5* padahal biaya yang dikeluarkan untuk membeli *hardware* tersebut tidaklah murah. Beberapa kasus pada program juga terlihat dari aplikasi yang masih menggunakan database *access*. *Access* tidak dirancang untuk dikerjakan dalam beberapa akses secara simultan sehingga tidak akan memproses *request* sebelum pendahulunya selesai. Hal ini hanya akan antrian sehingga dapat disimpulkan dengan menggunakan basis data *Access* di *multicore* tidak memberi keuntungan lebih. Namun menggunakan *multicore* akan membantu karena memberi *overhead* tambahan untuk mengelola *concurrency* pada akses database. Pada penelitian oleh (Gu et al., 2014) *Operating System* sendiri hingga saat penelitian berlangsung memberikan kendala dalam menganalisa termasuk juga dalam *CPU*

affinity flow, *socket access*, *lock contention* sehingga menggunakan aplikasi pihak ketiga yang menanganinya. Berkaitan dengan minimnya penelitian dan aplikasi yang khusus untuk *affinity* maka penelitian ini bertujuan untuk memberikan gambaran sehingga memberi perhatian lebih kepada peneliti lainnya.

Perkembangan kernel *Operating System* (OS) yang menangani *network* untuk *multicore* baru mulai berkembang sejak analisis *signal processing* (Fusco & Deri, 2010) yang memberikan rekomendasi tentang pentingnya pengelolaan *signal* khususnya *rx* atau *receive signal* dan membaginya kedalam beberapa *queue*. Meskipun belum memberikan hasil namun penelitian tersebut membuka *open issue* berkenaan pengelolaan signal secara parallel. Berdasar penelitian diatas diciptakanlah teknologi RSS (*Receive Side Scalling*) yang mencoba memberikan solusi pada *Operating System*. RSS membagi *load* ke beberapa *queue* dan meneruskannya ke beberapa *core.* Pemrosesan secara parallel akan menaikan kemampuan proses namun tidak berbanding lurus pada *latency* terutama jika *rx queue* dikerjakan oleh CPU yang sedang sibuk. Penelitian ini memberikan peluang penelitian berikutnya dikarenakan keterbatasan RSS sebab pada dasarnya komputer tidak mengetahui *processor* mana yang baik untuk melakukan proses.

Penelitian ini juga memberikan dukungan terhadap penelitian yang dilakukan oleh (Huang et al., 2010) dengan menggunakan *pooling* didukung *hashing table* untuk memodifikasi *signal rx*. Dalam usulan ini juga mencoba untuk menjawab analisis dari (Rivera et al., 2014) yang mengungkapkan penurunan *performance* terhadap *UDP socket* dalam penanganan *multicore* karena masing - masing *core* melakukan *lock* pada *port* yang sama. *Multi receiver* diterapkan dalam penelitian ini bertujuan untuk menaikkan *througput* meskipun menggunakan *port* yang sama, yaitu dengan menggunakan skema *bind port* sebelum melakukan *accept()*.

Sistem ini disuguhkan melalui multi NIC yang diharapkan mampu meningkatkan *throughput* sehingga mempunyai *scalability* yang lebih, sejalan dengan penelitian oleh (Galagan et al., 2013). Selain itu memodifikasi *signal transmit* dan *receive* yang selama ini merupakan 1 (satu) kesatuan diubah menjadi beberapa bagian dalam sebuah *hash* sehingga diharapkan *signal* dapat dikerjakan secara *multicore*.

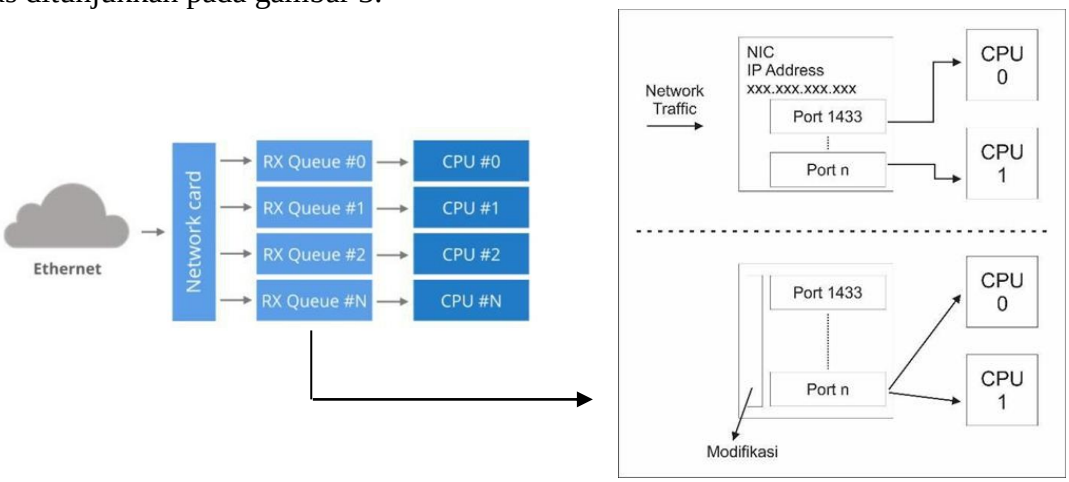
Selain itu usulan ini juga akan memberikan solusi lain dari penelitian dari (Jie et al., 2016) yang mengaktifkan *receive offload* untuk melakukan modifikasi dengan meng-*generate* ulang ACK pada Large Packet untuk meningkatkan *throughput*. Penelitian ini mencoba memberikan beberapa modifikasi yang lebih mengacu pada protocol UDP tanpa *receive offload* namun dengan tujuan yang sama yaitu peningkatan *throughput*. Hal ini dilakukan karena penelitian ini kedepannya diharapkan untuk dapat diimplementasikan pada *real time aplication* yang dimana kebanyakan menggunakan *protocol* UDP.

III. METODE PENELITIAN

Penelitian dimulai dengan meneliti perbedaan antara *shared* core dan *dedicated core* melalui membuat program *looping workload* sehingga diperoleh grafik perbedaan waktu dalam *mili second* (*ms*). Langkah berikutnya menentukan besaran *throughput* sistem *server* default dengan menggunakan program *sender* dan *receiver* . Angka standar *default* digunakan sebagai parameter keberhasilan modifikasi pada setiap langkah modifikasi yang disebut juga dengan metode *kernel bypass* . Modifikasi ini terdiri dari *hashing signal receive* yang ditujukan untuk membagi *signal* menjadi beberapa *queue* yang secara *default* merupakan satu *queue* pada *ethernet card*. Modifikasi dilakukan menggunakan *multiple IP (Internet Protocol) receiver* yang ditujukan untuk membagi *load* dan membagi *interupt* menjadi beberapa bagian sehingga *throughput* meningkat serta melakukan *affinity* pada *thread* dan lebih lanjut disebut sebagai *multi thread receiver* . Modifikasi yang dilakukan

 DOI : https://doi.org/10.30646/sinus.v18i2.472

tersebut juga mengatasi limitasi dari batasan dalam menerima *interupt*, yang bisa dipastikan hampir mustahil untuk menerima paket data lebih besar dari pemrosesan *singgle CPU*. Satu *interupt* biasanya hanya untuk satu *ethernet* atau *peripheral*. Modifikasi berikutnya dilakukan dengan menggunakan *bind port* yang ditujukan untuk membuat *multi socket listener* menyesuaikan analisis (Rivera et al., 2014). Secara garis besar modifikasi diatas ditunjukkan pada gambar 3.



Gambar 3. Gambaran Modifikasi kernel *bypass*

3.1. *Core Performance*

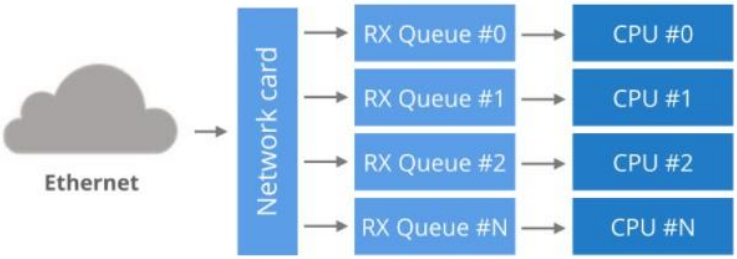
Pembuatan program untuk *looping thread* menjadi bagian pertama dari pengujian pertama yang digunakan untuk mengetahui apakah benar OS (*Operating System*) bisa memindah *thread* berdasar *looping* yang dibuat. Pada bagian ini akan memunculkan *capture* gambar yang bisa terlihat seberapa sering *thread* akan berpindah.

Bagian kedua akan dibuat *coding* untuk melakukan *pinning* secara pasti ke CPU(*Central Proccessing Unit*) sehingga mengunci *thread* agar tidak berpindah. Proses *affinity* dengan menempatkan *interupt* ada spesifik CPU akan meningkatkan *performance* proses sesuai dengan pembahasan dari (Shambharkar, 2015).

Pada bagian inti atau *performance demo* antara *core sharing* dan *separate core* diuji dan disajikan dalam time series. Program akan menciptakan *workload* dan dibebankan pada *logical CPU* dalam pararel *thread* serta membandingkannya dengan *single core*. Pararel *thread* diperjelas dengan membaginya menjadi 2 bagian yaitu yang menggunakan *thread* pada *core* yang sama dan yang mengunakan *thread* pada *core* yang berbeda.

3.2. Modifikasi Untuk Memaksimalkan Throughput

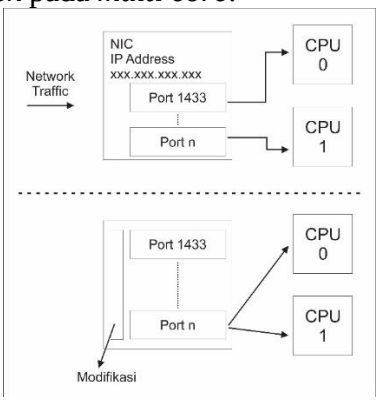
Kernel Bypass mengambil alih fungsi utama *kernel* pada bagian *network* menggunakan metode baru dengan mencoba teknik *hashing* pada *signal* yang bertujuan membagi *queue* untuk diarahkan pada spesifik CPU seperti ditunjukkan dalam Gambar 4. Modifikasi ini bergantung pada kompabilitas dari *ethernet card* yang dipakai. Modifikasi ini dilakukan dengan perintah *rx-flow-hash* dikuti *tupple* pembaginya yaitu *ip address* dengan perintah *udp4* dan *tupple port* dengan perintah *sdfn.*



Gambar 4. Proses Hashing Signal

Modifikasi berikutnya menggunakan multi IP (*Internet Protocol*) yang ditujukan untuk membagi beban CPU sehingga mengatasi limitasi CPU dalam mengolah atau memproses IRQ (*Interupt Re Quest*). Modifikasi ini umum untuk dilakukan dan bukan merupakan modifikasi yang baru. Dalam penelitian ini modifikasi dengan menambahkan *ethernet card* baru dengan IP yang berbeda dan *sender* diarahkan kedalam 2 IP tersebut.

Konsep *multi threading sender* diadopsi untuk melakukan modifikasi tahap selanjutnya yaitu *multi threading receiver* . Modifiksi ini dilakukan dengan *taskset -c 1,2 ./udpreceiver 1 0.0.0.0:4321 2* yang menunjukkan mengaktifkan CPU 1 dan 2 untuk menjalankan program udp *receiver* . Penerapan *multi thread receiver* merupakan penerapan dari penelitian (Rivera et al., 2014). Penerapan modifikasi ini memungkinkan terjadi penurunan *throughput* sehingga mengharuskan adanya modifikasi pada *socket discriptor* untuk menghindari *lock* pada *multi core*.



Gambar 5. Proses Modifikasi Port Listener

Penelitian ini menggunakan program untuk mengatasi perebutan *lock* dengan menggunakan *multi listnener* sehingga masing-masing thread menggunakan *listenner* yang berbeda sehingga meskipun menggunakan *port* yang sama dan tidak terjadi *bottleneck.* Modifikasi *multi listenner* ini dilakukan sebelum fungsi *accept().* Penerimaan koneksi pada umumnya tidak melakukan *fork* atau *binding socket* sehingga semua *core* menggunakan mode *waiting*. Pada penelitian ini juga diterapkan *affinity* pada NIC IRQ sebagai pendamping sehingga meningkatkan *performance* proses seperti yang dilakukan oleh (Tsai et al., 2012).

IV. HASIL DAN PEMBAHASAN

Percobaan ini dilakukan dengan 2 (dua) komputer yang dianggap sebagai *sender* atau *client* dan *receiver* atau dalam kenyataannya nanti disebut sebagai peladen. dengan melakukan standarisasi terlebih dahulu perangkat yang akan digunakan dengan melakukan *raw prerouting* dengan *no track* fitur pada *port* 4321 protokol UDP yang digunakan dalam keseluruhan percobaan. Penggunaan ip secara static juga dilakukan pada komputer *sender* dan komputer *receiver* sehingga menghilangkan hambatan konflik dengan *iptables*.

Kernel linux sudah menyediakan perintah *send syscall* namun untuk menghindari *context switches* namun memerlukan *cost* tersendiri sehingga dibentuk program baru dengan nama *sender* yang bertujuan mengirimkan *generating packet* ke peladen dengan protokol UDP dengan interval konstan seperti terlihat dalam Gambar 6 dan *receiver* dalam cuplikan Gambar 7.

 DOI : https://doi.org/10.30646/sinus.v18i2.472

udpsender.h

```
10
11
12  struct state {
13      struct net_addr *target_addr;
14      int packets_in_buf;
15      const char *payload;
16      int payload_sz;
17      int src_port;
18  };
19
20  void thread_loop(void *userdata) {
21      struct state *state = userdata;
22
23      struct mmsghdr *messages = calloc(state->packets_in_buf, sizeof(struct mmsghdr));
24      struct iovec *iovecs = calloc(state->packets_in_buf, sizeof(struct iovec));
25
26      int fd = net_connect_udp(state->target_addr, state->src_port);
27
28      int i;
29      for (i = 0; i < state->packets_in_buf; i++) {
30          struct iovec *iovec = &iovecs[i];
31          struct mmsghdr *msg = &messages[i];
32
33          msg->msg_hdr.msg_iov = iovec;
34          msg->msg_hdr.msg_iovlen = 1;
35
36          iovec->iov_base = (void*)state->payload;
37          iovec->iov_len = state->payload_sz;
38      }
39
40      while (1) {
41          int r = sendmmsg(fd, messages, state->packets_in_buf, 0);
42          if (r <= 0) {
43              if (errno == EAGAIN || errno == EWOULDBLOCK || errno == EINTR) {
44                  continue;
```

Gambar 6. Cuplikan kode *sender*

```
19  struct state {
20      int fd;
21      volatile uint64_t bps;
22      volatile uint64_t pps;
23      struct mmsghdr messages[MAX_MSG];
24      char buffers[MAX_MSG][MTU_SIZE];
25      struct iovec iovecs[MAX_MSG];
26  } __attribute__ ((aligned (64)));
27
28  struct state *state_init(struct state *s) {
29      int i;
30      for (i = 0; i < MAX_MSG; i++) {
31          char *buf = &s->buffers[i][0];
32          struct iovec *iovec = &s->iovecs[i];
33          struct mmsghdr *msg = &s->messages[i];
34
35          msg->msg_hdr.msg_iov = iovec;
36          msg->msg_hdr.msg_iovlen = 1;
37
38          iovec->iov_base = buf;
39          iovec->iov_len = MTU_SIZE;
40      }
41      return s;
42  }
43
44  static void thread_loop(void *userdata)
45  {
46      struct state *state = userdata;
47
48      while (1) {
49          /* Blocking recv. */
50          int r = recvmmsg(state->fd, &state->messages[0], MAX_MSG, MSG_WAITFORONE, NULL);
51          if (r <= 0) {
52              if (errno == EAGAIN || errno == EWOULDBLOCK || errno == EINTR) {
53                  continue;
```

Gambar 7. Cuplikan kode *receiver*

Untuk mendapatkan default throughput di multicore server maka harus digunakan paramater perhitungan multithread yaitu menggunakan multithread sender seperti terlihat dalam gambar 8 dan penambahan pining CPU pada gambar 9.

```
sender$ taskset -c 1 ./udpsender 192.168.254.1:4321
receiver$ taskset -c 1 ./udpreceiver1 0.0.0.0:4321
  0.362M pps  11.058MiB /  92.760Mb
  0.374M pps  11.411MiB /  95.723Mb
  0.369M pps  11.252MiB /  94.389Mb
  0.370M pps  11.289MiB /  94.696Mb
  0.365M pps  11.152MiB /  93.552Mb
  0.360M pps  10.971MiB /  92.033Mb
```

Gambar 8. Cuplikan perintah client server

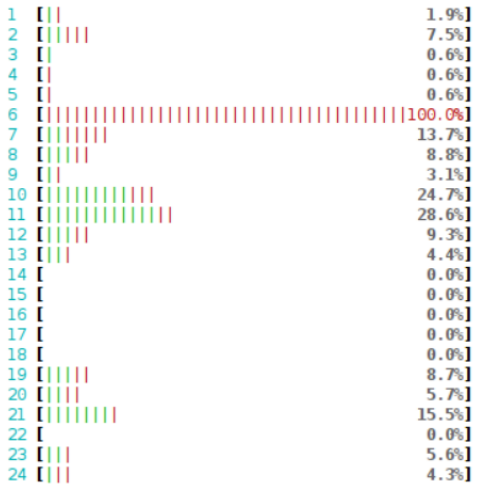
```
sender$ taskset -c 1,2 ./udpsender \
            192.168.254.1:4321 192.168.254.1:4321
receiver$ taskset -c 1 ./udpreceiver1 0.0.0.0:4321
  0.349M pps  10.651MiB /  89.343Mb
  0.354M pps  10.815MiB /  90.724Mb
  0.354M pps  10.806MiB /  90.646Mb
  0.354M pps  10.811MiB /  90.690Mb
```

Gambar 9**.** Cuplikan perintah multitread dan pinning CPU

Gambar 8 dan 9 tidak terlihat perubahan namun dengan perintah ethtool -S akan memperlihatkan berapa paket maksimal dari sebuah multicore server default seperti terlihat dalam gambar 10. Dan angka pada gambar 8 dan 9 tidak menyentuh angka optimal disebabkan karena terdapat busy processor yaitu pada processor ke 6 seperti terlihat dalam gambar 11.

```
receiver$ watch 'sudo ethtool -S eth2 |grep rx'
     rx_nodesc_drop_cnt:    451.3k/s
     rx-0.rx_packets:       8.0/s
     rx-1.rx_packets:       0.0/s
     rx-2.rx_packets:       0.0/s
     rx-3.rx_packets:       0.5/s
     rx-4.rx_packets:     355.2k/s
     rx-5.rx_packets:       0.0/s
     rx-6.rx_packets:       0.0/s
     rx-7.rx_packets:       0.5/s
     rx-8.rx_packets:       0.0/s
     rx-9.rx_packets:       0.0/s
     rx-10.rx_packets:      0.0/s
```

Gambar 10**.** *Default througput multicore server*



Gambar 11. Gambar htop *busy processor*

Percobaan menggunakan NUMA didapatkan beberapa hal sebagai berikut, *receiver* menggunakan CPU yang berbeda namun dengan node sama dengan *rx queue* didapatkan *throughput* diantara 360kpps, *receiver* di CPU yang sama dengan node *rx queue* didapatkan sekitar 430kpps, *receiver* dalam shared core di teknologi HT akan mendapatkan sekitar 200kpps, *receiver* di numa node berbeda akan mendapatkan sekitar 330kpps.

 DOI : https://doi.org/10.30646/sinus.v18i2.472

Modifikasi *kernel bypass* dimulai dengan mengoptimalkan *receiver* . Konsep tersebut mengambil persamaan dari *multiple thread sender* maka dimungkinkan juga *multiple thread receiver* namun terlebih dahulu dengan menambah *ip address* dengan menggunakan multi *NIC card*. Selanjutnya dieksekusi perintah dalam gambar 12 dan didapatkan hasil dalam gambar 13. Gambar 13 juga diperjelas dalam gambar 14 tentang sebaran paketnya menuju ke masing masing CPU.

```
sender$ taskset -c 1,2 ./udpsender 192.168.254.1:4321 192.168.254.2:4321
```

Gambar 12**.** Perintah *multiple* ip *receiver*

```
receiver$ taskset -c 1 ./udpreceiver1 0.0.0.0:4321
  0.609M pps   18.599MiB / 156.019Mb
  0.657M pps   20.039MiB / 168.102Mb
  0.649M pps   19.803MiB / 166.120Mb
```

Gambar 13. *throughput multiple* ip *receiver*

```
receiver$ watch 'sudo ethtool -S eth2 |grep rx'
     rx-0.rx_packets:     8.0/s
     rx-1.rx_packets:     0.0/s
     rx-2.rx_packets:     0.0/s
     rx-3.rx_packets:  355.2k/s
     rx-4.rx_packets:     0.5/s
     rx-5.rx_packets:  297.0k/s
     rx-6.rx_packets:     0.0/s
     rx-7.rx_packets:     0.5/s
     rx-8.rx_packets:     0.0/s
     rx-9.rx_packets:     0.0/s
     rx-10.rx_packets:    0.0/s
```

Gambar 14. Sebaran paket dalam *multiple ip receiver*

Modifikasi kernel *bypass* berlanjut dengan mengaplikasikan penerapan *multiple thread receiver* dengan melakukan affinity secara manual dengan perintah dan hasil terlihat dalam gambar 16. Gambar 16 juga menunjukkan penurunan yang dari sebelumnya berkisar di 650kpps menjadi berkisar 480pps. Modifikasi ini terlihat kurang memberikan peningkatan yang signifikan dan cenderung menurun dikarenakan adanya *lock contection* seperti ditunjukkan pada paper (Rivera et al., 2014). Perubahan berikutnya diletakkan pada *port binding* sehingga mampu mengatasi masalah *lock contection*. *Port binding* digunakan untuk meletakkan proses *binding port* sebelum *fungsi accept ()* sehingga port yang sama dapat digunakan lebih dari sekali. Performance dari hasil modifikasi akan meningkat karena setiap port tidak dalam mode wait yang terlalu lama sehingga menurunkan utilisasi dari *processor*. Modifikasi tersebut dapat dilihat dari cuplikan kode mulport listener dalam Gambar 15.

```
8
9   void* do_work(void *arg)
10  {
11      int *port = (int *) arg;
12
13      int listen_socket = socket(AF_INET, SOCK_STREAM, 0);
14      int one = 1;
15      setsockopt(listen_socket, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &one, sizeof(one));
16
17      struct sockaddr_in serv_addr;
18      memset(&serv_addr, 0, sizeof(serv_addr));
19      serv_addr.sin_family = AF_INET;
20      serv_addr.sin_addr.s_addr = INADDR_ANY;
21      serv_addr.sin_port = htons(*port);
22
23      int ret = bind(listen_socket, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
24      if (ret == -1)
25      {
26          printf("Failed to bind to port\n");
27          return NULL;
28      }
29
30      listen(listen_socket, 5);
31
32      struct sockaddr_in cli_addr;
33      memset(&cli_addr, 0, sizeof(cli_addr));
34      int addr_length = sizeof(cli_addr);
35
36      do
37      {
38          int cli_sock = accept(listen_socket, (struct sockaddr *) &cli_addr, (socklen_t *) &addr_length);
39          close(cli_sock);
```

Gambar 15. Cuplikan code program *multiport listener*

Port binding membuat port yang sama dibuka oleh masing masing *logical CPU* sehingga dalam satu *port* terlihat dapat digunakan berkali kali dalam beberapa proses dari *logical CPU* yang berbeda. Cuplikan kode program dalam Gambar 15 dapat dikutip pada bagian *listen* yang melakukan *binding* sebanyak *address* yang dibutuhkan diikuti oleh *socket* yang tersedia di dalam *hardware*. Kode program tersebut merupakan bagian utama dari *binding port* yang memberikan perubahan yang cukup signifikan dalam mengatasi *lock*.

Modifikasi *port binding* membuat *port listenner* menjadi lebih banyak memberikan peningkatan signifikan dan merupakan kesuksesan dalam memberikan peningkatan *througput* seperti ditunjukkan dalam Gambar 16.

```
receiver$ taskset -c 1,2,3,4 ./udpreceiver1 0.0.0.0:4321 4 1
  1.114M pps  34.007MiB / 285.271Mb
  1.147M pps  34.990MiB / 293.518Mb
  1.126M pps  34.374MiB / 288.354Mb
```

Gambar 15. Hasil modifikasi final kernel *bypass*

Tabel 1 menunjukkan hasil keseluruhan modifikasi yang dilakukan dalam menaikkan *throughput* sehingga mudah untuk diamati. Modifikasi dalam publikasi yang dilakukan mampu mengalahkan metode NUMA yang selama ini digunakan untuk menaikkan *throughput*. Penggunaan *multiple thread receiver* mengalami hambatan seperti yang dicontohkan oleh (Rivera et al., 2014) dipakai menjadi acuan untuk melakukan modifikasi dengan *binding port* sehingga memberikan hasil yang signifikan dalam prosesnya. Hasil modifikasi secara keseluruhan yang tertuang dalam tabel 1, jika dihitung dalam rata-rata akan menghasilkan kenaikan sebesar 250,44%.

Tabel 1. Hasil Modifikasi *Througput*

no	***Default***	**NUMA**		***Multiple IP***		***Multiple thread***		***Binding port***	
		Throughput (pps)	Gap (%)	*Throughput* (pps)	Gap (%)	*Througput* (pps)	Gap (%)	*Throughput* (pps)	Gap (%)
1	451,3k	430k	4,72	609k	34,94	495k	9,68	1114k	246,84
2		428k	5,16	657k	45,58	480k	6,36	1147k	254,16
3		432k	4,28	649k	43,81	461k	2,15	1126k	249,50
4		425k	5,83	653k	44,69	486k	7,69	1138k	252,16
5		427k	5,38	640k	41,81	490k	8,58	1142k	253,05
6		431k	4,50	651k	44,25	477k	5,69	1129k	250,16
7		428k	5,16	655k	45,14	485k	7,47	1131k	250,60
8		431k	4,50	623k	38,05	491k	8,80	1136k	251,72
9		433k	4,05	641k	42,03	481k	6,58	1121k	248,39

V. KESIMPULAN DAN SARAN

5.1 Kesimpulan

Berdasarkan hasil penelitian dan pembahasan yang telah dilakukan, maka dapat disimpulkan beberapa hal sebagai berikut :

1. Sistem yang dikembangkan dengan menggunakan *kernel bypass* mampu untuk menaikkan kemampuan server ditunjukkan dengan kenaikan *througput* dibanding *default kernel*.
2. Dari hasil percobaan, kenaikan *througput* rata-rata adalah sebesar 250,44%.
3. Hasil penelitian menunjukan HT (*Hyper Threading*) mampu meningkatkan kemampuan proses menjadi lebih cepat namun dalam penanganan program yang kompleks akan terjadi penurunan *performance* proses jika ditempatkan pada *shared core.*

5.2 Saran

Perlu adanya kombinasi yang lebih kompleks dan penelitian korelasi antar kombinasi serta urutannya sehingga dapat mewujudkan peladen yang lebih optimal dan menjadi standar secara umum.

DAFTAR PUSTAKA

Angelo, G. D., Marchetti-spaccamela, A., & Cnr, I. (2016). Multiprocessor Real-Time Scheduling with Hierarchical Processor Affinities. *2016 28th Euromicro Conference on Real-Time Systems*, 237–247. https://doi.org/10.1109/ECRTS.2016.24

Bo, Z. (2016). Analysis of the Resource Affinity in NUMA Architecture for High Performance Network. *2016 5th International Conference on Measurement, Instrumentation and Automation*, 547–550.

Diener, M., Cruz, E. H. M., Alves, M. A. Z., Navaux, P. O. A., Busse, A., & Heiss, H. U. (2016). Kernel-Based Thread and Data Mapping for Improved Memory Affinity. *IEEE Transactions on Parallel and Distributed Systems*, *27*(9), 2653–2666. https://doi.org/10.1109/TPDS.2015.2504985

Fusco, F., & Deri, L. (2010). High Speed Network Traffic Analysis with Commodity Multi-core Systems. *proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, 218–224.

Galagan, V., Yurchenko, O., Preobrazhensky, E., Zhuravkov, P., & Dombrougov, M. (2013). Multi-gigabit intel-based software routers. *Proceedings - RoEduNet IEEE International Conference*. https://doi.org/10.1109/RoEduNet.2013.6714193

Gu, Q., Wen, L., Dai, F., Gong, H., Yang, Y., Xu, X., & Feng, Z. (2014). StackPool: A high-performance scalable network architecture on multi-core servers. *Proceedings - 2013 IEEE International Conference on High Performance Computing and Communications, HPCC 2013 and 2013 IEEE International Conference on Embedded and Ubiquitous Computing, EUC 2013*, 17–28. https://doi.org/10.1109/HPCC.and.EUC.2013.13

Hanford, N., Ahuja, V., Farrens, M., Ghosal, D., Balman, M., Pouyoul, E., & Tierney, B. (2015). Improving network performance on multicore systems : Impact of core affinities on high throughput flows. *Future Generation Computer Systems*. https://doi.org/10.1016/j.future.2015.09.012

Huang, C., Yu, X., & Luo, H. (2010). Research on high-speed network data stream capture based on multi-queue NIC and multi-core processor. *ICIME 2010 - 2010 2nd IEEE International Conference on Information Management and Engineering*, *2*, 248–251. https://doi.org/10.1109/ICIME.2010.5477440

Jie, L., Shuhui, C., & Jinshu, S. (2016). Implementation of TCP large receive offload on multi-core NPU platform. *2016 International Conference on Information and Communication Technology Convergence, ICTC 2016*, 258–263. https://doi.org/10.1109/ICTC.2016.7763481

Rivera, D., Ach, E., & Bustos-jim, J. (2014). Analysis of Linux UDP Sockets Concurrent Performance. *2014 33rd International Conference of the Chilean Computer Science Society*, 65–69.

Shambharkar, S. A. (2015). A Study on Setting Processor or CPU Affinity in Multi-Core Architecture for Parallel Computing. *International Journal of Science and Research*, *4*(5), 2013–2016.

Tsai, W. Y., Huang, N. F., & Hung, H. W. (2012). A port-configuration assisted NIC IRQ affinitization scheme for multi-core packet forwarding applications. *GLOBECOM - IEEE Global Telecommunications Conference*, 2547–2552. https://doi.org/10.1109/GLOCOM.2012.6503500