UNIVERZA V MARIBORU
FAKULTETA ZA ELEKTROTEHNIKO,
RAČUNALNIŠTVO IN INFORMATIKO

Mario Casar

GONILNIK ZA REALNO-ČASOVNI ZAJEM PAKETOV IZ MREŽE ETHERCAT V OKOLJU LINUX

Diplomsko delo

Maribor, avgust 2016

UNIVERZA V MARIBORU
FAKULTETA ZA ELEKTROTEHNIKO,
RAČUNALNIŠTVO IN INFORMATIKO

Mario Casar

GONILNIK ZA REALNO-ČASOVNI ZAJEM PAKETOV IZ MREŽE ETHERCAT V OKOLJU LINUX

Diplomsko delo

Maribor, avgust 2016

GONILNIK ZA REALNO-ČASOVNI ZAJEM PAKETOV IZ MREŽE ETHERCAT V OKOLJU LINUX

Diplomsko delo

Študent:	Mario Casar
Študijski program:	Univerzitetni študijski program Računalništvo in informatika
Smer:	Programska oprema
Mentor:	doc. dr. Tomaž Kosar
Lektorica:	Helena Skarlovnik Casar, prof. ped., teol. in slov.

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko
Smetanova ulica 17
2000 Maribor, Slovenija

FERI

Številka: 93439032
Datum in kraj: 21. 04. 2016, Maribor

Na osnovi 330. člena Statuta Univerze v Mariboru (Ur. l. RS, št. 44/2015)
izdajam

SKLEP O DIPLOMSKEM DELU

1. **Mariu Casarju,** študentu univerzitetnega študijskega programa RAČUNALNIŠTVO IN INFORMATIKA, smer Programska oprema, se dovoljuje izdelati diplomsko delo.
2. **MENTOR:** **doc. dr. Tomaž Kosar**
3. **Naslov diplomskega dela:**
   GONILNIK ZA REALNO-ČASOVNI ZAJEM PAKETOV IZ MREŽE ETHERCAT V OKOLJU LINUX
4. **Naslov diplomskega dela v angleškem jeziku:**
   DRIVER FOR REAL-TIME PACKET ACQUISITION FROM ETHERCAT NETWORK
5. Diplomsko delo je potrebno izdelati skladno z "Navodili za izdelavo diplomskega dela" in ga oddati v treh izvodih (dva trdo vezana izvoda in en v spiralo vezan izvod) ter en izvod elektronske verzije do 31. 08. 2016 v referatu za študentske zadeve.

Pravni pouk: Zoper ta sklep je možna pritožba na senat članice v roku 3 delovnih dni.

Dekan:
red. prof. dr. Borut Žalik

Obvestiti:
- kandidata,
- mentorja,
- odložiti v arhiv.

ZAHVALA

Za nasvete pri izbiri teme, spodbude, ažurne strokovne usmeritve ne glede na čas dneva ter veliko mero razumevanja se zahvaljujem mentorju,
doc. dr. Tomažu Kosarju.
Za veliko potrpežljivosti in razumevanja se zahvaljujem tudi ženi in otrokom.
Vse skupaj pa ne bi bilo mogoče brez staršev, ki so mi študij sploh omogočili – tudi njim iskrena hvala.

GONILNIK ZA REALNO-ČASOVNI ZAJEM PAKETOV IZ MREŽE ETHERCAT V OKOLJU LINUX

Ključne besede: Linux, realni-čas, EtherCAT, RT-Preempt, RTAI, IgH EtherCAT Master, odprta koda.

UDK: 004.451.8(043.2)

Povzetek:

V diplomskem delu je opisana izdelava gonilnika za zajem podatkov v realnem-času pri visokih frekvencah iz omrežja EtherCAT, in sicer s pomočjo odprtokodnih rešitev za okolje Linux.

Na začetku je primerjava dveh razširjenih rešitev za strogi realni-čas v operacijskem sistemu Linux z namenom ugotoviti, ali sta rešitvi enakovredni. Primerjani rešitvi sta v drugem delu implementirani in opravljene so primerjalne meritve. Izbrana rešitev Linux RT-Preempt je nato uporabljena kot osnova za razvoj jedrnega gonilnika. Gonilnik je razvit kot jedrni modul za operacijski sistem Linux, zasnovan na odprtokodni rešitvi IgH EtherCAT Master. Gonilnik je testiran, rezultati opravljenih meritev pa primerjani s pričakovanimi.

DRIVER FOR REAL-TIME PACKET ACQUSITION FROM ETHERCAT NETWORK

Key words: Linux, real-time, EtherCAT, RT-Preempt, RTAI, IgH EtherCAT Master, open source

UDK: 004.451.8(043.2)

Abstract:

In this diploma thesis we develop a module for data acquisition from EtherCAT network using an open-source solution for real-time under Linux platform. In the beginning of the thesis, we compared two widely recognized solutions for hard real-time under the operating system Linux: RT-Preempt and RTAI. Next, we implemented both solutions and used benchmarks to check jitter of the developed systems. Further, Linux RT-Preempt was used as a base for the development of kernel module for data acquisition. Development of Linux kernel module was based on an open-source solution IgH EtherCAT Master. We also benchmark the developed module and compare the results with the expected ones.

KAZALO

KAZALO SLIK

KAZALO TABEL IN GRAFOV

UPORABLJENE KRATICE

ACPI – napredno upravljanje porabe (angl. Advanced Configuration and Power Interface)
API – aplikacijski programski vmesnik (angl. Application Programming Interface)
ASIC – aplikacijsko-specifična integrirana vezja (angl. Application-Specific Integrated Circuit)
BIOS – temeljni vhodno-izhodni sistem (angl. Basic Input/Output System)
C – programski jezik
CPU – centralna procesna enota (angl. Central Processing Unit)
CRC – ciklično preverjanje redundance (angl. Cyclic Redundancy Check)
DMA – neposredni dostop do pomnilnika (angl. Direct Memory Access)
ESC – nadzornik EtherCAT suženjske enote (angl. EtherCAT Slave Controller)
FMMU – enota za upravljanje s pomnilnikom (angl. Fieldbus Memory Management Unit)
FPGA – programabilni čip (angl. Field-Programmable Gate Array)
GNU – računalniški operacijski sistem (angl. GNU's Not Unix)
GRUB – GNU zagonski nalagalnik (angl. GNU GRand Unified Bootloader)
HPET – časovnik dogodkov visoke ločljivosti (angl. High Precision Event Timer)
HRT – časovnik visoke ločljivosti (angl. High Resolution Timer)
IEC – mednarodna komisija za elektrotehniko (angl. International Electrotechnical Commission)
ISR – upravljalnik prekinitev (angl. Interrupt Service Routine)
LED – svetleča dioda (angl. Light-Emitting Diode)
NASA – vesoljska agencija (angl. National Aeronautics and Space Administration)
PID – identifikator procesa (angl. Process Identifier)
PDO – podatkovni objekti za procesiranje (angl. Process Data Objects)
PM – upravljanje s porabo (angl. Power Management)
ROM – bralni pomnilnik (angl. Read-Only Memory)
RT – realno-časovno (angl. Real-Time)
RTAI – realno-časovni programski vmesnik (angl. Real Time Application Interface)
SMI – sistemsko upravljana prekinitev (angl. System Management Interrupt)
SoC – računalnik v čipu (angl. System on a Chip)
SSD – disk z bliskovnim pomnilnikom (angl. Solid State Drive)
TDP – maksimalna količina toplote čipa (angl. Thermal Design Point)

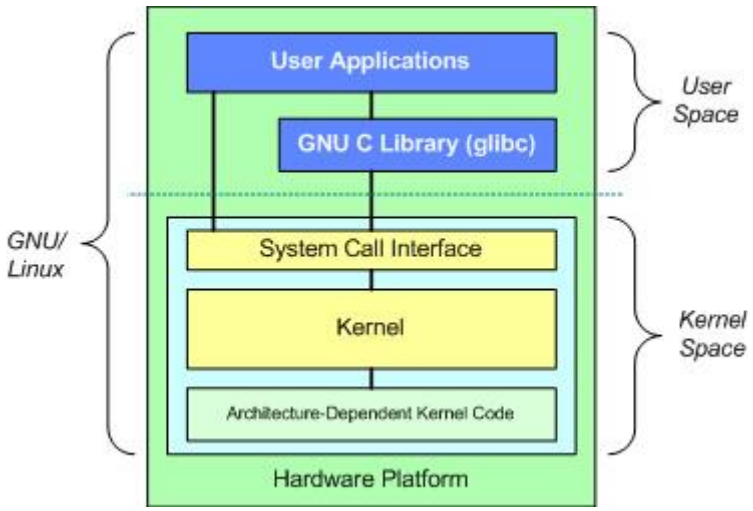
1 UVOD

Na področju industrijske avtomatizirane tehnologije narašča trend zamenjave področnih vodil (angl. fieldbus) z industrijskim Ethernetom [1]. Zaradi mnogo hitrejše in lažje integracije je razumljivo, zakaj se je industrijski Ethernet veliko hitreje razširil v dosti industrijskih proizvodnih okoljih, kjer so bili prej uveljavljeni sistemi s konvencionalnim vodilom. Ena izmed takih na Ethernet baziranih tehnologij je EtherCAT [2], katere protokol je standardiziran v IEC 61158 in je primeren tako za stroge kot ohlapno realno-časovne zahteve v avtomatizirani tehnologiji. Bistvena prednost EtherCAT pred drugimi Ethernet tehnologijami je v načinu procesiranja Ethernet paketa (angl. frame); ta se namreč izvede v načinu sprotnega procesiranja (angl. on the fly), kar bomo podrobno razložili v poglavju o tehnologiji EtherCAT. Ob želji, da bi čim bolje izkoristili zmožnosti tehnologije EtherCAT, predvsem zajema podatkov pri višjih frekvencah od 10 kHz in obdelave teh znotraj sprejemljivih časovnih intervalov, je neizogibna implementacija rešitve, ki temelji na realno-časovnem operacijskem sistemu. Ker omejitve standardnega namiznega Windows operacijskega sistema preprečujejo realizacijo rešitve, je potreben drugačen pristop.

Operacijski sistem GNU/Linux [3] – imenovali ga bomo kar Linux – je od svojega nastanka do danes postal zelo razširjen. Priljubljen je brez dvoma ravno zaradi tega, ker je zasnovan na odprti kodi, kar omogoča marsikatero prilagoditev, katere drugi namizni operacijski sistemi, kot je npr. Microsoft Windows, ne omogočajo. V tem diplomskem delu bomo najprej ugotovili, ali je stopnja razvoja oziroma zrelosti operacijskega sistema Linux postala dovolj zanesljiva, da ga lahko uporabimo za sisteme s strogim realnim časom. Primerjavo bomo izvedli z modificiranim Linux RT-Preempt jedrom in Linuxom z RTAI razširitvijo. Ob prvotni predpostavki, da je Linux jedro z RT-Preempt ustrezno za strogi realni čas, bomo v drugem delu naloge zanj razvili ustrezni gonilnik (angl. kernel modul), s katerim bomo zajemali podatke iz mreže EtherCAT. Pričakujemo, da bomo dosegli zastavljeni cilj: gonilnik bo omogočil zajeme paketov iz omrežja EtherCAT s frekvencami, višjimi od 10 kHz.

2 OPERACIJSKI SISTEM LINUX

Zgodovina Linuxa se začenja pred četrt stoletja, ko je Linus Torvalds z namenom, da bi bolje izkoristil funkcionalnosti svojega novega namiznega računalnika, spisal namenski program v MINIX-u z uporabo GNU C prevajalnika. Prevajalnik GNU C ostaja tako še danes orodje za prevajanje Linux jedra. Poimenovanje samega sistem pa je tako, če imamo skupaj z jedrom v mislih še GNU programsko opremo, GNU/Linux (glej Sliko 2.1 [4]). Skupaj z ostalimi dodatki se potem združuje v distribucije Linux, katerih je več kot 200 in med katerimi je najbolj priljubljena Mint[1]. Navsezadnje je tudi operacijski sistem Android zasnovan na Linux jedru.



Slika 2.1: Temeljna arhitekturna zasnova GNU/Linux operacijskega sistema

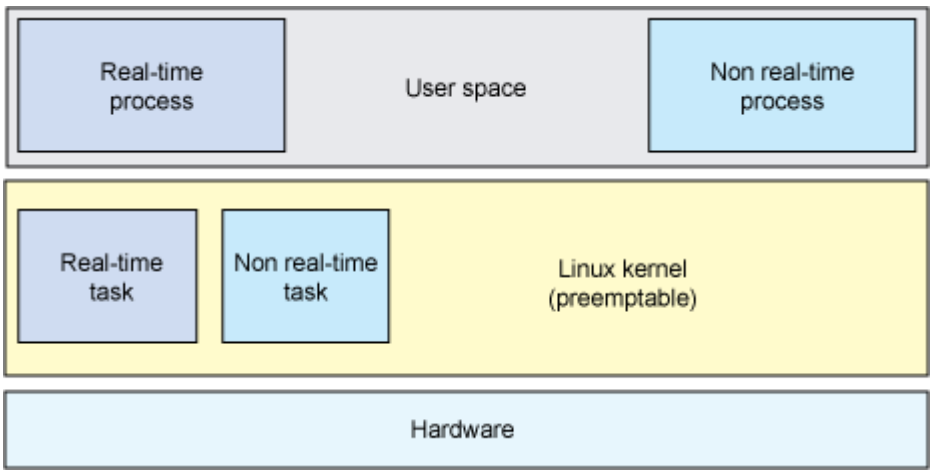
1 Podatki leta 2016 https://distrowatch.com/

2.1 Prekinitve opravil v operacijskem sistemu Linux

Da bi lahko razumeli, kako funkcionira jedro, si najprej oglejmo prekinitve procesov (angl. preemption). To je sposobnost operacijskega sistema, da zaustavi ali prekine trenutno opravilo v korist opravila z višjo prioriteto.

V operacijskem sistemu Linux so bili procesi na uporabniškem nivoju (angl. user-space) vedno takšni, da jih je možno prekiniti (angl. preemptible). Jedro tako lahko prekine procese oz. programe iz uporabniškega nivoja, da preklopi na izvajanje drugega programa z uporabo navadnega urinega signala. Jedro ne čaka, da bi ti procesi sprostili procesor, kar posledično pomeni, da neskončna zanka v takem procesu ne more blokirati sistema.

Do jedra različice 2.6[2] (glej sliko 2.2) [5] jedra niso bila prekinitvena (angl. preemtible) – kakor hitro je eden izmed procesov vstopil v jedro, ga ni bilo možno prekiniti in izvajati drugega procesa. Procesor je tako lahko začel izvajati drugi proces, ko je bil sistemski klic zaključen ali ko je trenutni proces eksplicitno zahteval od razvrščevalnika (angl. scheduler), da začne izvajati drugi proces.



Slika 2.2: Realni-čas v jedru 2.6

Z jedrom 2.6 se pojavi njegova dodatna zmožnost, da prekinja procese. To zmožnost jedra se omogoči oziroma onemogoči z opcijo CONFIG_PREEMPT. Če je opcija

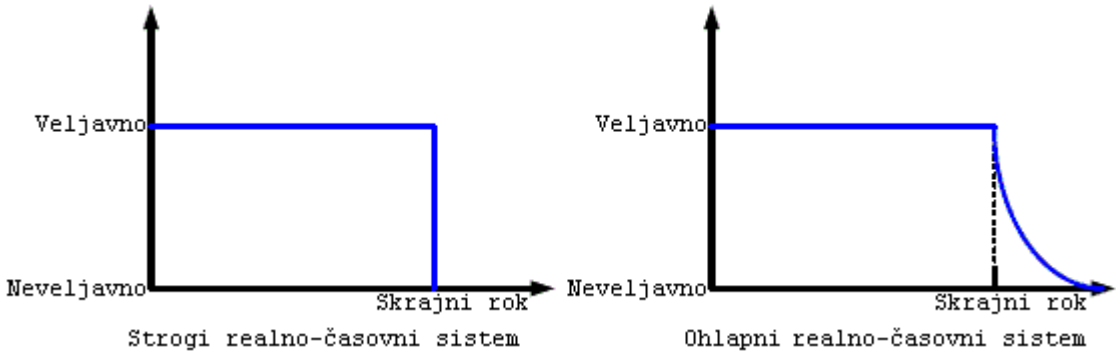
[2] Linux jedro različice 2.6 je izšel leta 2004.

CONFIG_PREEMPT omogočena, se lahko programska koda jedra prekinja vsepovsod, razen če programska koda onemogoči lokalne prekinitve. Neskončna zanka v kodi tako ne more več blokirati celotnega sistema. Če CONFIG_PREEMPT onemogočimo, se jedro obnaša kot predhodna različica 2.4.

Med prednosti jedra z omogočenimi prekinitvami vsekakor lahko štejemo izboljšano latenco in skalabilnost sistema in zmožnost, da se opravila z visoko prioriteto izvedejo in odzivajo pravočasno.

Domneva, da ima tak sistem povečano prepustnost in zmogljivost je žal napačna – celotna zmogljivost takega sistema se zmanjša ravno na račun odzivnosti [6]. Prav tako moramo, ko govorimo o zmanjšanju latence, dodati, da prekinitve v jedru operacijskega sistema zmanjšajo maksimalno latenco. Lahko povemo, da prekinitve v operacijskem sistemu naredijo sistem napovedljiv in deterministične.

Ob vpeljavi koncepta realno-časovnega sistema vpeljemo tudi koncept skrajnega roka (angl. deadline), kar v časovni domeni pomeni točko, do katere pričakujemo, da sistem opravi svojo nalogo. Razlikujemo strogi realno-časovni pristop (angl. hard real time) in ohlapni realno-časovni pristop (angl. soft real time). Pri ohlapnem realno-časovnem pristopu nam prekoračitev skrajnega roka ne pomeni težav, reducira pa veljavnost opravljene naloge povečano s časom prekoračitve. Na drugi strani strogi realno-časovni pristop ne dopušča prekoračitve in tak dogodek takoj povzroči neveljavno stanje. Največkrat gre v realnem svetu za sisteme, kjer bi taka prekoračitev pomenila katastrofo, na primer izgubo človeških življenj ali pa nepopravljivo materialno škodo. Slika 2.3 [7] nam to primerjavo med pristopoma nazorno pokaže.



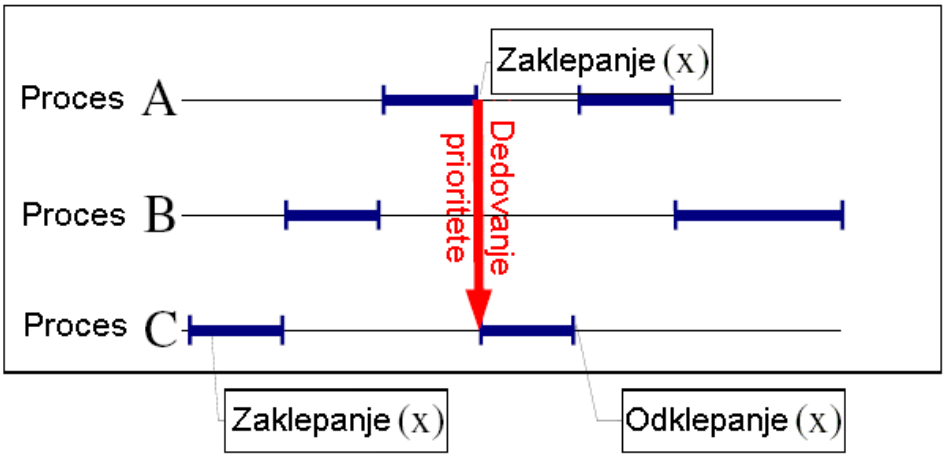
Slika 2.3: Primerjava strogega in ohlapnega realno-časovnega sistema

2.2 Programski popravek RT-Preempt

Standardno Linux jedro zadosti osnovnim zahtevam ohlapnega realnega-časa, nikakor pa ne zagotavlja izpolnjevanja skrajnih časovnih rokov v konceptu strogega realnega-časa. Z realno-časovnimi popravki (angl. RT-Preempt patch) avtorja Inga Molnarja in s slojem za časovne dogodke visoke ločljivosti (angl. High Resolution Timer, HRT) – tudi do 1 µs – jedro pridobi lastnosti, ki ustrezajo pogojem strogega realnega-časa. Aplikativne uporabe takega modificiranega jedra so uporabne na primer pri profesionalni obdelavi zvoka in nadzoru v industriji. RT-Preempt popravki spremenijo Linux jedro v sistem, kjer je možno prekiniti vsak proces (angl. fully preemptible).

Popravek RT-Preempt vpelje v sistem kar nekaj novih konceptov; eden izmed njih je implementacija mehanizmov zaklepanja v jedru (angl. spinlocks) s prekinjanjem. Implementacija je izvršena s pomočjo mehanizmov medsebojnega izključevanja (angl. rtmutexes). Kritične sekcije jedra, ki so prej bile zaščitene z mehanizmi, kot so spinlock_t in rwlock_t, je možno sedaj prekinjati. Ustvarjanje brezprekinitvenih odsekov v jedru je še zmeraj možno, le mehanizem je dobil novo ime: raw_spinlock_t. Vmesniki (angl. APIs) mehanizma so enaki kot pri spinlock_t. Dedovanje prioritete (angl. priority inheritance) je v povezavi z omenjenim zanimiv koncept, zato si ga pobliže oglejmo. Za razlago nam bo v pomoč slika 2.4 [8], iz katere so razvidni trije procesi z različnimi prioritetami: A, B in C. Proces C ima najnižjo prioriteto in je zasedel vir (x), vmes je v izvajanje prišel proces B z višjo prioriteto, kar mu zdaj omogoča prekinjanje kritičnih sekcij v jedru. Sledi prihod procesa A z najvišjo prioriteto, ki zahteva vir (x), vendar ga ne more dobiti, saj ga še

zmeraj zaseda proces C, proces z najnižjo prioriteto. Nastali situaciji pravimo inverzija prioritet. Rešitev s pomočjo dedovanja prioritet je v dodelitvi trenutne prioritete najvišjega procesa A najnižjemu procesu C; ta se lahko sedaj v miru, brez da bi ga proces B prekinil, zaključi in sprosti vir(x), na katerega čaka A. Proces z najvišjo prioriteto bo sedaj lahko pridobil vir(x) in šele nato bo na vrsto prišel proces B. Omenjena situacija in njena rešitev ima v realno-časovnih sistemih verjetno daleč najbolj odmevni dogodek, povezan z misijo agencije NASA[3] na Mars[4] iz leta 1997. Celotna misija bi lahko propadla zaradi hrošča v programu, saj je povzročil inverzijo prioritet v realno-časovnem operacijskem sistemu VxWorks[5], kateri je poganjal računalnik vozila. Ta se je zaradi inverzije prioritet ponovno zaganjal in ogrozil celotno misijo, a k sreči je strokovnjakom agencije NASA uspelo na daljavo implementirati dedovanje prioritet v sistemu in rešiti misijo.



Slika 2.4: Inverzija prioritet in dedovanje prioritete procesa

RT-Preempt vpelje tudi koncept ISR (angl. Interrupt Service Routine) kot jedrne niti, katere je možno prekinjati. RT-Preempt popravki za posamezno različico jedra so dostopni na spletnih straneh Linux Kernel Archives, zbrani pod projektom RT [9].

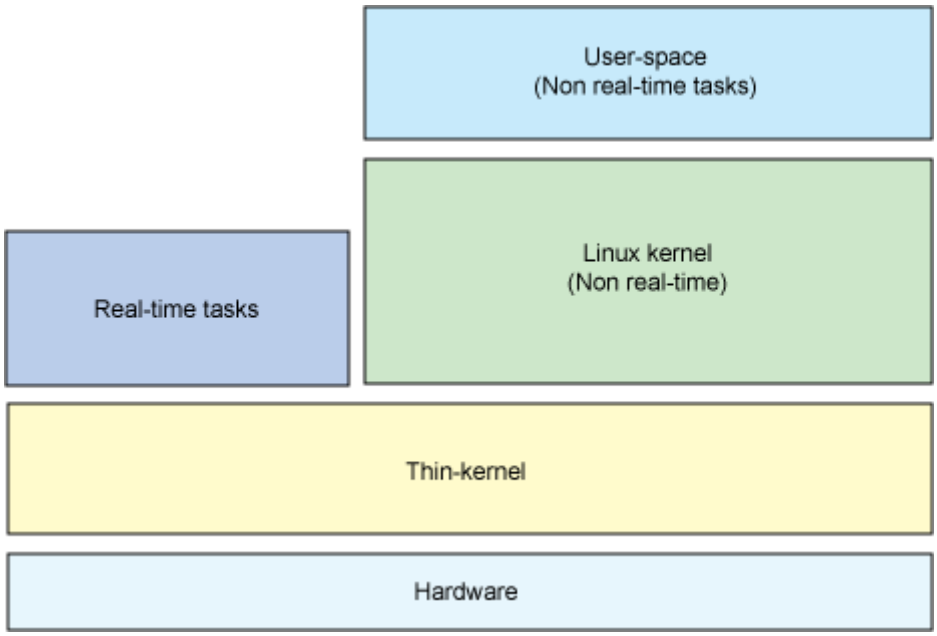
Dostopamo lahko tako do popravkov za najstarejše različice jedra verzije 2.6.x kot do najnovejših za verzijo jedra 4.6.x, pozorni pa moramo biti, da so popravki ustrezni, oziroma da je njihova verzija identična verziji jedra, katerega bomo prevajali. Prevajanju sistema se bomo posvetili v naslednjem poglavju.

[3] https://en.wikipedia.org/wiki/NASA
[4] https://en.wikipedia.org/wiki/Mars_Pathfinder
[5] https://en.wikipedia.org/wiki/VxWorks

2.3 RTAI

Še veliko pred pojavom popravkov RT-Preempt je obstajal drugačen pristop za doseganje realno-časovnih zahtev v operacijskem sistemu Linux. Ogledali si bomo predstavnika iz družine s pristopom tankega jedra (angl. thin-kernel approach), katerega začetki segajo v konec 90-ih let prejšnjega stoletja, in sicer RTAI (angl. Real Time Application Interface). Zgodovinsko gledano je bil sicer prvi predstavnik iz te družne RTLinux, ki zdaj ni več prosto dostopen in je v lasti Wind River Systems-a.

Pristop s tankim jedrom uporablja drugo jedro kot vmesnik med strojno opremo in Linux jedrom (glej sliko 2.5) [5]. Linux jedro, ki ni realno-časovno, se izvaja v ozadju kot opravilo z nizko prioriteto nadrejenega tankega jedra in izvaja vsa opravila, ki niso realno-časovna. Na tankem jedru pa se izvajajo opravila, ki so realno-časovna. Pomembna vloga, katero opravlja tanko jedro poleg izvajanja realno-časovnih opravil, je upravljanje s prekinitvami. S prestrezanjem prekinitev zagotavlja, da podrejeno Linux jedro ne more prekiniti izvajanja tankega jedra, kar temu omogoča, da zagotavlja podporo za strogi realni čas. Tak pristop pa ima tudi slabe lastnosti, izmed katerih je pri razvoju programske opreme pomemben zlasti otežen pristop razhroščevanja, saj so realno-časovna opravila in opravila, ki niso realno-časovna popolnoma neodvisna. Poleg tega ne gre spregledati, da opravila, ki se ne izvajajo v realnem času, nimajo polne podpore Linux platforme. Ker se RTAI – kot že rečeno – veliko dlje časa razvija in je tudi v uporabi v industriji, kjer so zahteve po Linux platformi in strogem realnem-času, nam bo služil kot dobra izhodiščna referenca pri primerjavi z RT-Preempt. RTAI je prosto dostopen na spletni strani projekta [10]. Zadnja testna različica je 5.0, mi pa bomo za testiranje uporabili stabilno različico 4.0.



Slika 2.5: Pristop s tankim jedrom v RTAI za strogi realni-čas

3 TESTNO OKOLJE

Da bi lahko izvedli tudi teste v praksi, bomo zgradili testno okolje, katerega izgradnjo bomo razdelili v več sklopov. Na začetku bomo na standardni namizni računalnik srednjega razreda dvakrat ločeno namestili ustrezno Linux distribucijo. V naslednjih korakih bomo ta nameščena sistema ustrezno dopolnili s popravkom RT-Preempt in uporabo vmesnika RTAI, na koncu pa bomo na tako zgrajenima sistemoma še izvedli teste s sintetičnim orodjem za testiranje.

3.1 Nastavitve BIOS realno-časovnega sistema

Preden smo se lotili namestitve operacijskega sistema, so bile potrebne določene nastavitve v BIOS-u našega računalnika, ki največ doprinesejo k znižanju latence našega sistema. Lahko jih razdelimo v štiri sklope.

Prvi tak sklop so SMI (angl. System Management Interrupts), uporabljeni pri strojni opremi, ki upravlja s porabo energije na matični plošči. Pri realno-časovnih sistemih si SMI ne želimo iz več razlogov: eden pomembnejših med njimi je trajanje teh prekinitev, ki lahko znaša več 100 µs in že samo po sebi lahko pomeni nesprejemljivo trepetanje (angl. jitter) v realno-časovnih aplikacijah. Drugi razlog je, da so to prekinitve najvišje prioritete v sistemu, tretji pa, da jih ne moremo prestreči, ker ko procesor dobi SMI, preklopi v poseben način in skoči na določeno (angl. hard-wired) lokacijo v posebnem SMM (angl. System Management Mode) naslovnem prostoru, ki se nahaja v BIOS ROM-u. Bistveno je tudi, da so SMI prekinitve z vidika operacijskega sistema »nevidne«. Čeprav SMI prekinitve v danem trenutku obravnava en procesor, vplivajo tudi na realno-časovno odzivnost SMP (angl. Symmetric multiprocessing) sistema oziroma sistema z več procesorskimi jedri. Princip ravnanja s SMI prekinitvami v sistemih z več jedri dopušča, da jedro, ki obravnava SMI prekinitev in je zaklenilo dostop do mehanizma medsebojnega izključevanja (ang. mutex) nekega vira, katerega pa potrebuje drugo jedro, prisili, da slednje čaka tako dolgo, da upravljalnik prekinitev (ang. interrupt handler) SMI, prekinitev

zaključi in sprosti mehanizem medsebojnega izključevanja. Tu obstaja problem pri vmesniku RTAI in tudi drugih operacijskih sistemih.
Osnovni postopki, s katerimi se skušamo znebiti SMI v arhitekturi x86, so:

- uporaba PS/2 miške in tipkovnice,
- onemogočanje podpore za USB miško in tipkovnico v BIOS-u,
- prevajanje jedra z omogočenim ACPI (angl. Advanced Configuration and Power Interface).

Kot posebno opozorilo velja omeniti, da SMI prekinitev ne onemogočimo globalno, ker lahko pride do resnih poškodb sistema. Na starejših Intel P4 procesorjih lahko procesor dobesedno pregori. Prav tako so SMI prekinitve velikokrat uporabljene kot popravki za hrošče v čipih, kar lahko posledično pomeni, da potem določene komponente ne delujejo več, kot je pričakovano. Zato moramo dobro premisliti preden onemogočimo katerikoli SMI.

V drugi sklop spadajo dogodki na vodilu DMA (angl. Direct Memory Access). Ti lahko povzročijo blokiranje procesorja za več mikrosekund. Generirani so lahko s strani vsake naprave, ki uporablja DMA. To so najpogosteje SATA/PATA/SCSI naprave ter mrežne in grafične kartice. V nekaterih primerih je možno vpliv teh naprav nadzirati skozi gonilnik na ta način, da žrtvujemo prepustnost za nižjo latenco.

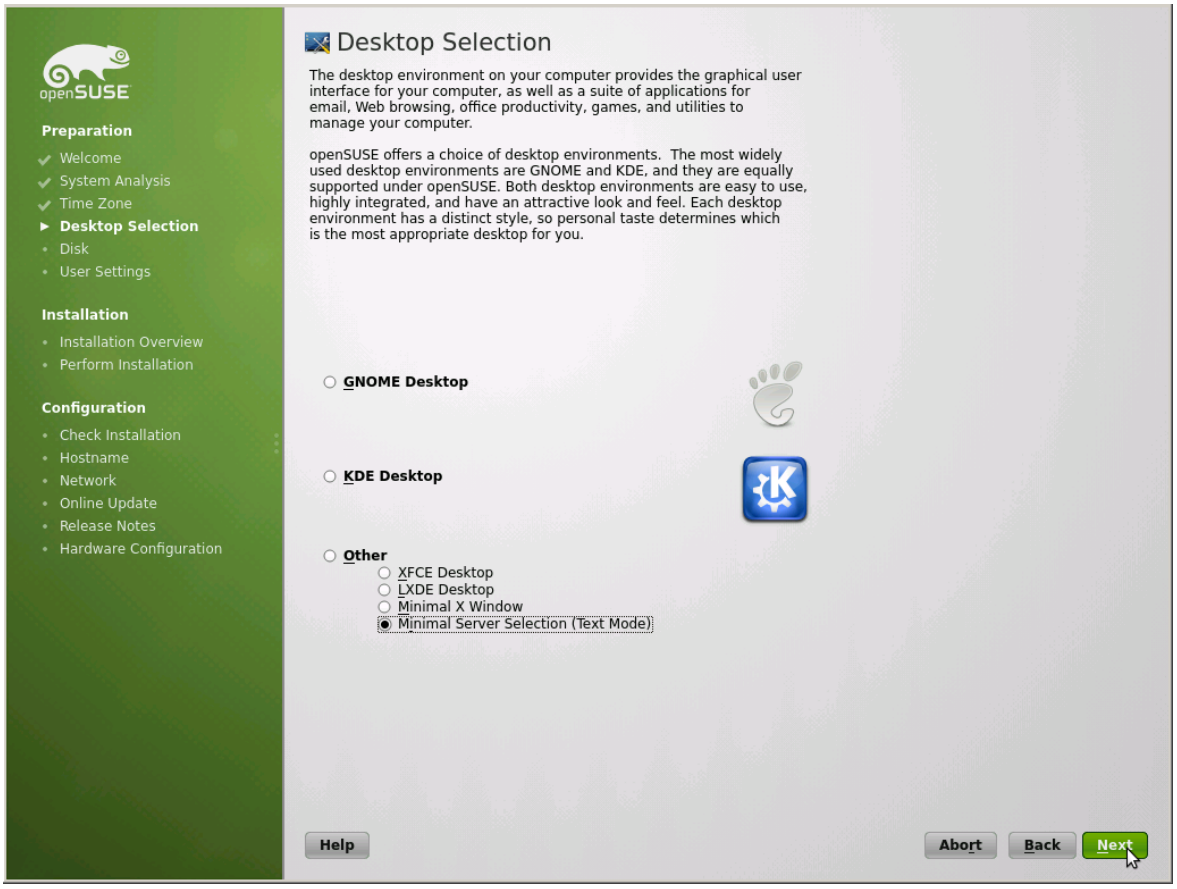
Tretji sklop je upravljanje s porabo (angl. Power Management, PM). Upravljanje s porabo je omogočeno skozi BIOS za različno strojno opremo. Na račun majhnih prihrankov energije se žrtvuje latenca sistema, zato je priporočljivo, da se ga izklopi v celoti ali pa se z dodatnimi meritvami izmeri vpliv posamezne PM opcije na latenco celotnega sistema.

Sistem za večnitenje je četrti sklop. Sistem za paralelizacijo oz. večnitenje (angl. Hyper-threading) in upravljanje opravil v računalniškem sistemu precej doprinese k naključnosti latence. Zato ga je prav tako dobro onemogočiti ali pa pazljivo izmeriti njegov vpliv na zmogljivost sistema.

V BIOS-u nekaterih računalnikov lahko naletimo na opcijo HPET (angl. High Precision Event Timer) in jo obvezno omogočimo. Sistemi, ki opcije za vklop HPET nimajo, jo »skrivajo« znotraj ACPI nastavitve, zato je pomembno, da je na takih sistemih ACPI omogočen. Koncept HPET se uporablja v naborih čipov (angl. chipsetes) že od leta 2005. Zasnovan je kot 64-bitni glavni števec, ki šteje s frekvenco vsaj 10MHz in je dopolnjen s setom vsaj treh primerjalnikov (angl. comparator) (lahko jih je do 256); primerjalniki lahko delujejo v periodičnem ali dogodkovnem (angl. one-shot) načinu. Linux jedro uporablja HPET kot vir ure (angl. clock source).

3.2 Namestitev osnovnega operacijskega sistema

Za namestitev Linuxa smo izbrali distribucijo openSUSE [11], in sicer predvsem iz razloga, ker jo privzeto podpira rešitev za gospodarja EtherCAT, katero bomo obravnavali v naslednjem poglavju. Po izboru distribucije se je treba odločiti še za različico distribucije, kjer nas zanima predvsem verzija jedra. Izberemo openSUSE 12.2 z različico jedra 3.4.6. To jedro izberemo, ker nas pri namestitvi RTAI omejuje bolj kot RT-Preempt in nam ne dopušča veliko izbire.



Slika 3.1: OpenSUSE 12.2 namestitev

Pri namestitvi izberemo minimalno verzijo brez grafičnega vmesnika, saj ta zadostuje za naš namen testiranja, kot je razvidno na sliki 3.1. Trdi disk razdelimo na dva enakovredna dela, kar bomo pozneje uporabili za namestitev obeh želenih verzij sistemov. Za zagonski nalagalnik (angl. Boot Loader) izberemo GRUB. Po zaključeni osnovni namestitvi in prvem zagonu namestimo še dodatne pakete, za kar se v distribuciji openSUSE uporablja orodje YaST (angl. Yet another Setup Tool) [12]. Nujno potrebujemo še pakete za prevajanje jedra in poznejšo namestitev RT-Preempt in RTAI. Ti paketi so:

- make
- gcc
- gcc-c++
- patch
- libnuma-devel
- ncurses-devel
- yast2-online-update-frontend

3.2.1 Namestitev in konfiguracija RT-Preempt

Za prevajanje ustreznega jedra najprej prenesemo njegovo izvorno kodo in jo razširimo.

```
wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.4.7.tar.gz
tar -xzf linux-3.4.7.tar.gz
```

Nato prenesemo še ustrezne RT-Preempt popravke in jih prav tako razširimo.

```
wget https://www.kernel.org/pub/linux/kernel/projects/rt/3.4/older/patch-
3.4.7-rt15.patch.gz
gunzip patch-3.4.7-rt15.patch.gz
```

Sledi zelo pomemben korak, saj uveljavi realno-časovne popravke na izvorni kodi jedra.

```
cat ../patch-3.4.7-rt15.patch | patch -p1
```

Izvorna koda jedra je tako pripravljena. Iz zagonskega direktorija kopiramo še obstoječo nastavitveno datoteko, na podlagi katere bomo ustvarili novo RT-Preempt optimizirano nastavitveno datoteko.

```
cp /boot/config-3.4.6-2.10-desktop .config
make oldconfig
make menuconfig
```

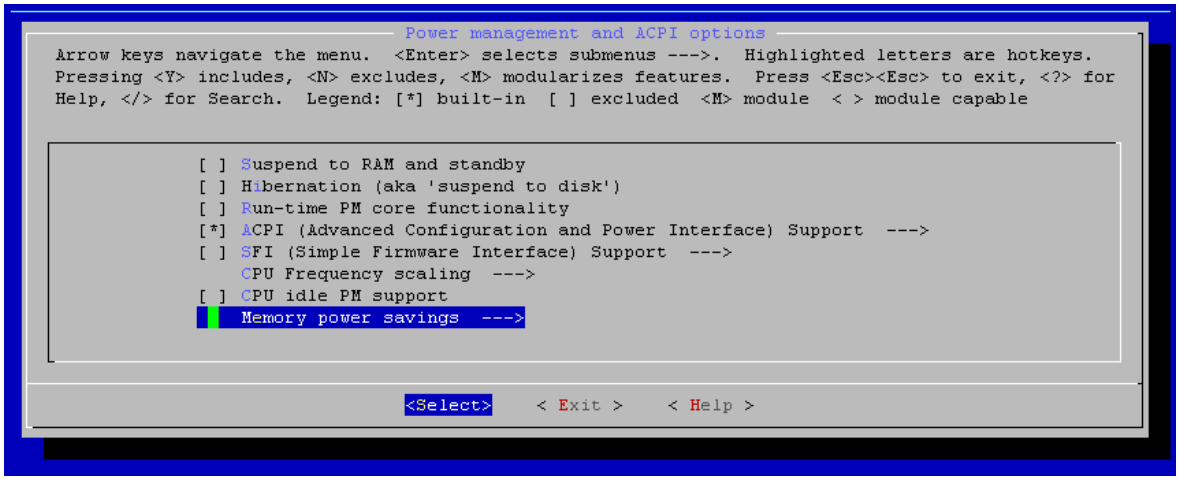
Zažene se nam konfigurator jedra. Vmesnik je zelo podoben YaST konfiguratorju. V konfiguratorju bomo dejansko določili nastavitve, s katerimi se bo novo jedro prevedlo. Konfigurator jedra nam v osnovnem meniju ponuja več sklopov nastavitev. Nastavitve, katere bomo spreminjali, se nahajajo v »Processor type and features« (Tabela 3.1) ter »Power management and ACPI options« (Tabela 3.2).

Tabela 3.1: Sklop nastavitev za procesor

»Processor type and features«
High Resolution Timer Support (omogočimo)
Maximum number of CPUs (privzeto opcijo zmanjšamo iz 512 na 4)
SMT (Hyperthreading) scheduler support (onemogočimo)
Preemption Model (izberemo Fully Preemptible Kernel (RT)) – zelo pomemben korak
Timer frequency (izberemo 1000 HZ)

Tabela 3.2: Sklop upravljanja porabe

»Power management and ACPI options«
V podmeniju »ACPI (Advanced Configuration and Power Interface) Support«:
AC Adapter (onemogočimo)
Battery (onemogočimo)
Dock (onemogočimo)
Processor (onemogočimo)
V podmeniju »CPU Frequency scaling«
CPU Frequency scaling (onemogočimo)
V podmeniju »Memory power savings«:
Intel chipset idle memory power saving driver (onemogočimo)
V nadrejenem meniju onemogočimo vse preostale opcije, kot je prikazano na sliki 3.2.



Slika 3.2: Konfiguriranje datoteke za prevajanje jedra

Na koncu še shranimo spremembe v konfiguracijsko datoteko in zapustimo konfigurator.

Sledi prevajanje jedra, modulov in namestitev.

```
make -j3   # stikalo -j služi hitrejšemu prevajanju na več jedrnih procesorjih, število jeder+1
make modules # prevedemo module
make modules_install # jih namestimo
make install # namestimo prevedeno jedro
```

Pred ponovnim zagonom sistema lahko preko YaST spremenimo privzeto jedro in to se bo naložilo na naše novo prevedeno jedro »3.4.7-rt15«.

Pri osnovni namestitvi smo razdelili trdi disk na dva enaka dela. V prvem imamo tako sedaj nameščen delujoč sistem Linux RT-Preempt, na drugi del diska pa v naslednjem koraku namestimo Linux sistem za razširitev RTAI.

3.3 Namestitev in konfiguracija RTAI

Znova za prevajanje ustreznega jedra najprej prenesemo njegovo izvorno kodo in jo razširimo, podrazličica (angl. minor version) je za malenkost nižja kot prej (3.4.6), da ustreza RTAI popravku.

```
wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.4.6.tar.gz
tar -xzf linux-3.4.7.tar.gz
```

Nato prenesemo še paket RTAI verzije 4.0 in ga razširimo.

```
wget https://www.rtai.org/userfiles/downloads/RTAI/rtai-4.0.tar.bz2
bzip2 -d rtai-4.0.tar.bz2
```

Po razširitvi paketa znotraj direktorijske strukture poiščemo ustrezen popravek, v našem primeru se ta datoteka nahaja v:
`rtai_dir/base/arch/x86/patches/ hal-linux-3.4.6-x86-4.patch`

Podobno kot pri RT-Preempt uveljavimo popravke na izvorni kodi jedra.
`patch -p1 -b < rtai_dir/base/arch/x86/patches/ hal-linux-3.4.6-x86-4.patch`

Nadaljnji postopki pri izgradnji konfiguracijske datoteke in prevajanju jedra so enaki kot pri RT-Preempt zato jih ne bomo še enkrat ponavljali. Dodatno je treba urediti samo nastavitve, podane v tabeli 3.3.

Tabela 3.3: Dodatna konfiguracija za RTAI

V meniju »Processor type and features«:
»Use register arguments« (onemogočimo)
V meniju »Loadable module support«:
»Module versioning support« (onemogočimo)

Po ponovnem zagonu sistema nam zagonski nalagalnik sedaj poleg obeh osnovnih jeder in jedra RT-Preempt ponudi še jedro z RTAI razširitvijo.

3.4 Meritve

Za primerjavo naših testnih sistemov smo izbrali orodje Cyclictest [13] za primerjalno meritev (angl. benchmark), ki se tudi najpogosteje omenja pri citiranju realno-časovnih Linux metrik. Cyclictest je razvil Thomas Gleixner, vzdržujeta pa ga Clark Williams in John Kacur. Orodje je prosto dostopno, zadnja verzija je 1.0, poleg te verzije pa lahko prenesemo tudi arhivske iz spletne strani Linux Kernel Archives, zbrane pod rt-test [14].

Prenesemo izvorno kodo zadnje verzije 1.0, jo razširimo in prevedemo. Orodje je takoj pripravljeno za uporabo, celoten nabor možnosti oziroma ukazov se nam izpiše, če zaženemo orodje s stikalom »--help«.

Podrobneje si oglejmo, kaj pomenijo stikala, uporabljena pri testiranju z ukazom ./cyclictest:

-t stikalo določi eno nit na vsak procesor, ki je na voljo,
-n stikalo določi uporabo clock_nanosleep,
-p stikalo in število, ki mu sledi, določi prioriteto niti z najvišjo prioriteto,
-i stikalo in število, ki mu sledi, določa bazni interval niti v (µs), privzeta vrednost je enaka 1000 oziroma 1 ms,
-l stikalo in število, ki mu sledi, določa število ponovitev, privzeta vrednost je enaka 0, kar pomeni neskončno ponovitev.

Tabela 3.4: Primerjava meritev latence v testnih sistemih

Sestava testnega računalnika Matična plošča: ASUS B85M-E Procesor: Intel Pentium G3220 3.0GHz Pomnilnik: DDR3 4GB Trdi disk: Segate SATA3 250GB			
./cyclictest -t -n -p 80 -i 10000 -l 0			
Oznaka jedra	Rezultati meritev latence sistema [µs]		
	Minimum	Povprečje	Maksimum
Jedro 3.4.6 brez prekinitev	723	1277	1744
Jedro 3.4.6 z omogočenim prekinitvami, brez RT-Preempt popravkov	1	8	36
Jedro 3.4.7 z omogočenimi prekinitvami, z RT-Preempt popravki	1	2	17
Jedro 3.4.6 z razširitvijo RTAI 4.0	1	2	16

Iz tabele 3.4 je razvidno, da ima nespremenjeno jedro Linux brez razširitve RT-Preempt največjo latenco, saj se tako jedro, kot smo že zapisali, obnaša v osnovi kot jedro verzije 2.4. Situacija se kar precej izboljša, ko v jedru omogočimo prekinjanje. Iz območja milisekund preidemo v območje mikrosekund. Latenca sistema se še dodatno izboljša pri jedru z RT-Preempt modifikacijo in uporabo vmesnika RTAI, v primerjavi z jedrom, ki ima omogočeno samo prekinjanje in nima dodatnih modifikacij. Razlika se zdi na prvi pogled majhna, le 20 µs, vendar pa je odzivnost slabša za več kot 100 %. Pri zadnji primerjavi rezultatov za RT-Preempt in RTAI takoj opazimo, da so skoraj identični, ker pa nas zanima obnašanje sistema v strogem realnem-času, moramo preveriti, če slednje drži tudi, ko sistem maksimalno obremenimo in šele nato izvedemo meritev z orodjem cyclictest.

Obremenitev izvedemo z več opravili, tako da vsa skupaj maksimalno obremenijo procesor, delovni pomnilnik, disk in dodatno še mrežno kartico. Nabor ukazov je videti takole:

```
dd if=/dev/zero bs=100M | gzip | gzip -d | gzip | gzip -d | gzip | gzip -
d > /dev/null
du /
ping -l 100000 -q -s 10 -f v
```

Podobno kot v prejšnjem koraku izmerimo latenco, le da se tokrat osredotočimo samo na sistema RT-Preempt in RTAI.

Tabela 3.5: Primerjava meritev latence v obremenjenih testnih sistemih

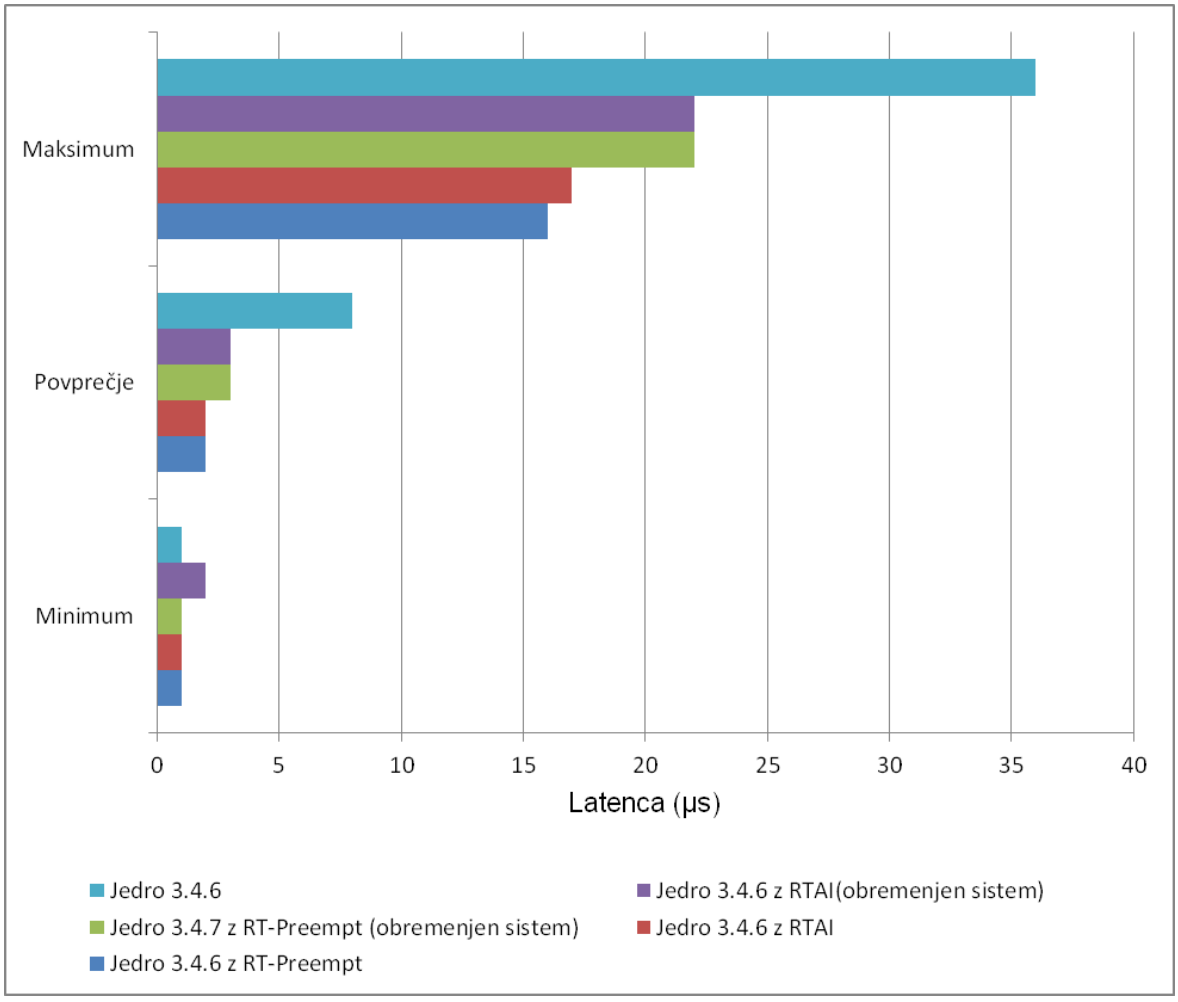
Sestava testnega računalnika Matična plošča: ASUS B85M-E Procesor: Intel Pentium G3220 3.0GHz Pomnilnik: DDR3 4GB Trdi disk: Segate SATA3 250GB			
./cyclictest -t -n -p 80 -i 10000 -l 0 Maksimalna obremenitev CPU, RAM, I/O			
Oznaka jedra	Rezultati meritev latence sistema v [µs]		
	Minimum	Povprečje	Maksimum
Kernel 3.4.7 z omogočenim preempt, z RT-Preempt patch	1	3	22
Kernel 3.4.6 z razširitvijo RTAI 4.0	2	3	22

Iz tabele 3.5 vidimo, da so tudi tokrat rezultati obeh sistemov skoraj identični. Zgornji meji, pomembni za deterministično realno-časovni sistem, pri obeh izmerimo 22 µs. Teoretično nam torej naš sistem omogoča zajem podatkov s frekvenco okoli 45 kHz ($f_z=1/T_{lat.max}$) tako z RT-Preempt jedrom kot z RTAI razširitvijo. Iz dobljenih rezultatov lahko z gotovostjo trdimo, da odzivnost sistema z jedrom z RTAI razširitvijo ni nič boljša kot odzivnost sistema z jedrom s popravki RT-Preempt. Strogi realni čas v okolju Linux je

z razvojem jedra in popravkov za RT postal dovolj zanesljiv v primerjavi z RTAI, za katerega je programske rešitve veliko težje implementirati in predvsem vzdrževati (za skoraj nično izboljšavo zmogljivosti).

Dobljeno ugotovitev bomo uporabili kot osnovo v naslednjem koraku, kjer bom izdelali gonilnik za zajem podatkov, zasnovan za jedro RT-Preempt. Graf 3.6 nam jasno ponazori, da se tudi pod obremenitvijo sistem RT-Preempt enakovredno kosa z RTAI sistemom.

Graf 3.6: Primerjava meritev latence v obremenjenih testnih sistemih

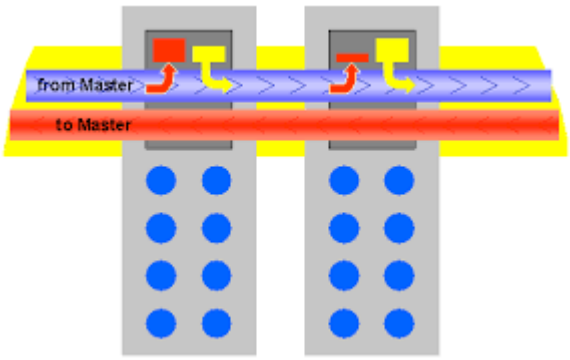


4 GONILNIK ZA ZAJEM PODATKOV IZ MREŽE ETHERCAT

4.1 EtherCAT

Tehnologija EtherCAT je na mreži Ethernet baziran sistem, s področnim vodilom, patentiran s strani Beckhoff Automation [13]. Protokol za EtherCAT je standardiziran v IEC 61158 in je primeren tako za stroge kot mehke realno-časovne (angl. hard and soft real-time) zahteve v avtomatizirani tehnologiji. Implementacija omrežja EtherCAT je izvedena po principu gospodar-suženj (angl. master-slave).

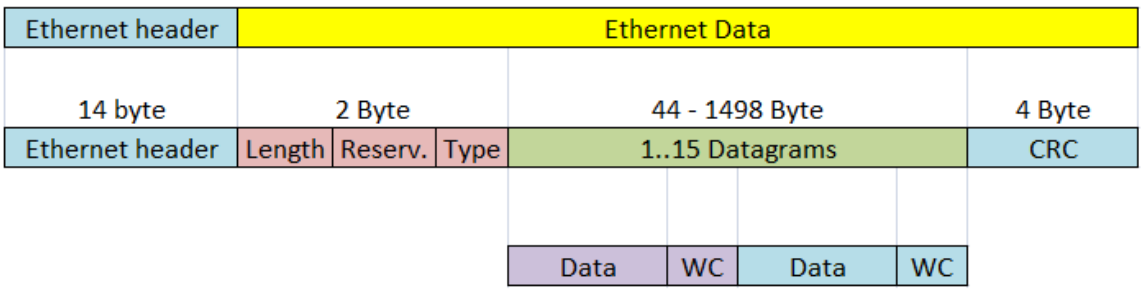
EtherCAT vpelje pomemben koncept sprotnega procesiranja (angl. on the fly). Ta pristop omogoča, da se zmanjša zakasnitve signala, zato ker se lahko podatki sprotno berejo in pišejo v Ethernet okvir (glej sliko 4.1) [13]. Omrežje samo je gledano z vidika gospodarja kot velik porazdeljen pomnilnik, v katerega se lahko zapisuje in bere brez omejitev.



Slika 4.1: Paketni tok skozi sužnja

EtherCAT okvir (angl. frame) je po strukturi (glej sliko 4.2) [2] in definiciji dolg vsaj 64 zlogov (angl. byte). Glava (angl. header) vsebuje pomembno informacijo o sami dolžini okvirja. Glavi sledi PDO (angl. Process Data Objects), ki ga sestavlja minimalno 1. in maksimalno 15 EtherCAT podatkovnih struktur (angl. datagrams). Vsaka od teh podatkovnih struktur je sestavljena iz samih podatkov in delovnega števca (angl. working cunter). Delovni števec nam skupaj s pomočjo CRC (angl. Cyclic Redundancy Check)

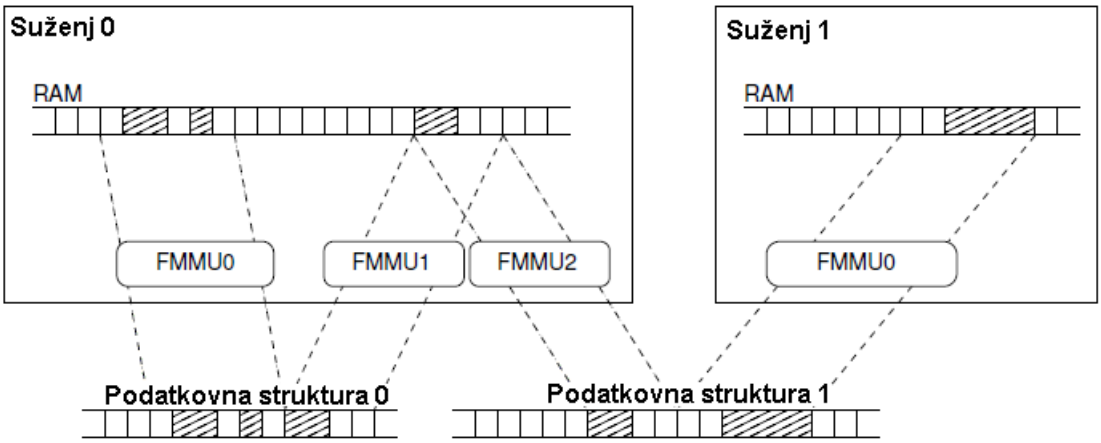
pove, če so bili podatki pravilno posredovani in obdelani na naslovljenih suženjskih napravah. Ker lahko vsaka posamezna suženjska naprava spreminja delovni števec, se kontrolna vsota (angl. checksum) preračuna za vsako napravo; če je ugotovljena napaka kontrolne vsote, se postavi statusni bit in pošlje prekinitev gospodarju, da se lahko napaka natančno določi.



Slika 4.2 Zgradba EtherCAT okvirja

Na napravah sužnjev imamo mehanizem, namenjen preslikavi podatkov iz logične podatkovne strukture (angl. image) na EtherCAT področnem vodilu v fizične pomnilniške naslove lokalnih naprav (angl. Fieldbus Memory Management Unit, FMMU). Obdelovani podatki se preslikajo celo po bitih, kar pomeni, da je posamičen bit vstavljen kjerkoli v logičnem naslovnem prostoru posameznega paketa. Vsak suženj ima implementirano lastno FMMU preslikovalno tabelo, ki skrbi za preslikavo vsakega logičnega naslova, prispelega znotraj posameznega paketa v fizični pomnilniški naslov znotraj EtherCAT sužnja (glej sliko 4.3) [14]. Tabela se nastavi ob fazi inicializacije omrežja in samo s pregledom preslikovalne tabele lahko vsak suženj pridobi podatke iz paketa, kot jih tudi vstavi v paket – glede na navodila gospodarja. Proces se izvede ob transportu paketa skozi sužnja z minimalno zakasnitvijo nekaj nanosekund. FMMU je implementiran znotraj ESC (angl. EtherCAT Slave Controller) posamezne naprave sužnja. ESC so lahko realizirani kot ASIC (angl. Application-Specific Integrated Circuit), FPGA (angl. Field-Programmable Gate Array) in nedavno tudi standardni mikroprocesorji.

Gospodar na drugi strani je lahko implementiran kot programska rešitev na poljubni napravi z Ethernet MAC naslovom. Proizvajalci nudijo kodo za različne operacijske sisteme. Mi bomo uporabili odprtokodno programsko rešitev IGH EtherCAT Master [14].



Slika 4.3: FMMU in proces preslikav

4.2 IGH EtherCAT Master

Izbrana odprtokodna rešitev za realizacijo gospodarja je distribuirana pod GPLv2 [15] licenco kot 'IgH EtherCAT Master for Linux'. Rešitev je zasnovana kot jedrni modul tako za jedra 2.6 kot za novejšo generacijo 3.x. Podpira veliko razširjenih naborov čipov mrežnih kartic, za katere nam ponuja gonilnike, s katerimi lahko upravljamo podprte mrežne kartice brez prekinitev. Za vse ostale mrežne kartice pa nudi podporo skozi generični gonilnik (angl. generic driver), ki posledično uporablja nižje sloje omrežnega sklada (angl. stack) operacijskega sistema Linux. Modul tudi omogoča sočasno izvajanje več gospodarjev EtherCAT. Realno-časovna podpora je realizirana za vse Linux rešitve, poleg RTAI in RT-Preempt, ki smo ju spoznali v poglavju 3, je med bolj znanimi še Xenomai [16]. Modul deluje tudi na Linux jedrih brez realno-časovne podpore.

4.2.1 Namestitev in konfiguracija

S spleta prenesemo zadnjo verzijo in sicer IgH EtherCAT Master 1.5.2.

```
wget http://etherlab.org/download/ethercat/ethercat-1.5.2.tar.bz2
```

Razširimo izvorno kodo in nastavimo konfiguracijsko datoteko pred prevajanjem.

```
./configure --enable-e1000e --enable-cycles --enable-hrtimer
```

Stikala, uporabljena z ukazom configure pomenijo:

--enable-e1000e	omogočimo gonilnik z mrežne kartice kompatibilne z intel e1000e
--enable-cycles	omogočimo uporabo CPU timestamp števca za Intel arhitekturo
--enable-hrtimer	omogočimo uporabo HRT (high-resolution timer); na ta način lahko gospodar preide v stanje spanja (sleep) med pošiljanjem paketov

Privzeto se prevedeta in sta omogočena še gonilnik za mrežne kartice, kompatibilne z Realtek 8139 naborom in generični gonilnik za vse ostale mrežne kartice.

Z naslednjimi ukazi prevedemo IgH EtherCAT Master-ja in gonilnike:

```
make
make modules
```

Namestitev pa dosežemo z:

```
make install
make modules_install
```

Nato je potrebno še kopirati konfiguracijsko datoteko na ustrezno lokacijo, kar naredimo z:

```
cp /opt/etherlab/etc/sysconfig/ethercat     /etc/sysconfig/
```

Prav tako je potrebno ustvariti povezavo (angl. symbolic link) in namestiti še sistemsko zagonsko skripto:

```
ln -s /opt/etherlab/etc/init.d/ethercat /etc/init.d/
insserv ethercat
```

Uredimo še konfiguracijsko datoteko »ethercat«, katero smo prej skopirali v `/etc/sysconfig/`.

Popravimo razdelek, kjer je definirana spremenljivka »MASTER0_DEVICE«, vnesti moramo MAC naslov mrežne kartice, ki jo bomo uporabili. V našem primeru je to: MASTER0_DEVICE="68:05:CA:2E:2F:3F".

Ker je v našem primeru uporabljena mrežna kartica Intel 82574L PCIe, popravimo še spremenljivko »DEVICE_MODULES«. Če tega ne storimo, bo uporabljen generični gonilnik:

```
DEVICE_MODULES="e1000e"
```

Po tako zaključnih osnovnih nastavitvah se lahko gospodar prvič zažene z ukazom:

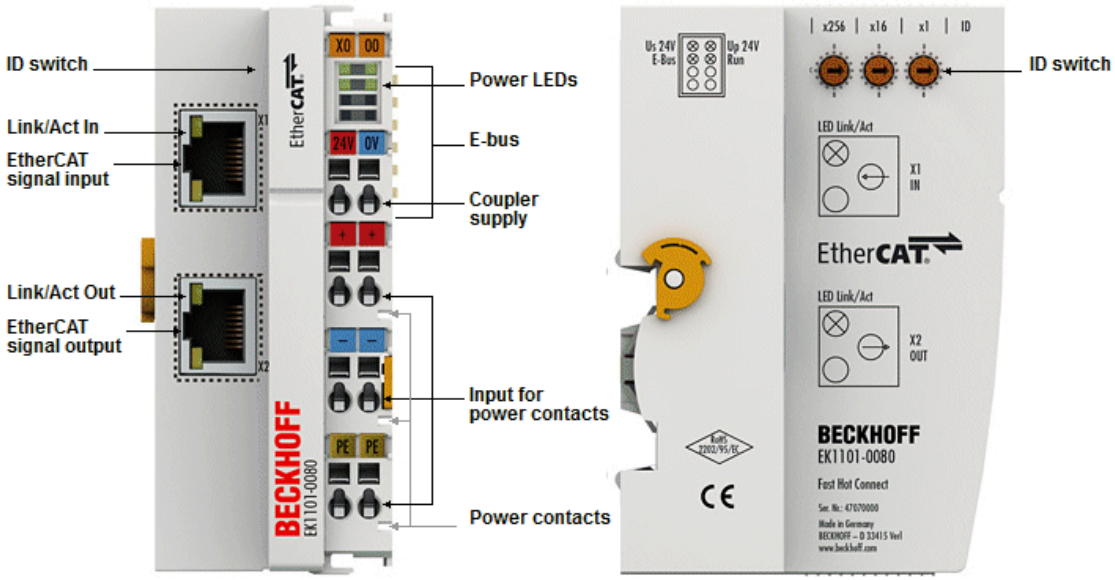
```
/etc/init.d/ethercat start
```

Če imamo priklopljeno suženjsko napravo, začnejo LED indikatorji na njej utripati. Sistemski dnevniški zapis (angl. syslog) pa vsebuje sledeče oziroma podobne zapise:

[36582.200559] EtherCAT: Master driver 1.5.2 2eff7c993a63
[36582.201275] EtherCAT: 1 master waiting for devices.
[36582.418195] ec_e1000e: EtherCAT-capable Intel(R) PRO/1000 Network Driver - 1.9.5-k-EtherCAT
[36582.418201] ec_e1000e: Copyright(c) 1999 - 2012 Intel Corporation.
[36582.517177] EtherCAT: Accepting 68:05:CA:2E:2F:3F as main device for master 0.
[36582.601420] EtherCAT 0: Starting EtherCAT-IDLE thread.
[36584.518879] e1000e: ecm0 NIC Link is Up 100 Mbps Full Duplex, Flow Control: None
[36584.518896] EtherCAT 0: Link state of ecm0 changed to UP.
[36584.520135] EtherCAT 0: 1 slave(s) responding on main device.
[36584.520141] EtherCAT 0: Slave states on main device: PREOP.
[36584.520260] EtherCAT 0: Scanning bus.
[36584.620816] EtherCAT 0: Bus scanning completed in 100 ms.

4.2.2 Testni suženj EtherCAT

Omenili smo suženjsko napravo, ki smo jo priključili na naše testno omrežje EtherCAT in katero je ob koncu prejšnjega koraka naloženi jedrni modul IgH EtherCAT Master tudi zaznal. Za testiranje smo uporabili modularno sestavljeno napravo proizvajalca Beckhoff, sestavljeno iz EL3312[6] in EK1101[7] modulov. Za povezavo z EtherCAT mrežo skrbi modul EK1101, modul EL3312 pa služi zajemu podatkov iz okolice, in sicer temperature s pomočjo navadnega termočlena (angl. thermocuple).



Slika 4.4: Beckhoff EK1101

Sužnja povežemo na mrežno kartico našega testnega sistem preko RJ45 vrat, označenih z »EtherCAT signal input« (glej sliko 4.4) [13]. Če bi imeli dodatne sužnje, bi te priklopili na vrata »EtherCAT signal output«. Posamezna razdalja med napravami ne sme preseči 100 m po standardu (100BASE-TX).

[6] http://www.beckhoff.com/EL3312/
[7] http://www.beckhoff.com/EK1101/

4.3 Gonilnik za zajem signalov

Poleg izvorne kode, dokumentacije in testnih orodij, nam IgH EtherCAT Master da na voljo tudi testne primere in lahko jih uporabimo za lastne rešitve. Ker želimo, da je naša rešitev izdelana kot jedrni modul, nam je kot osnova služil primer, ki se nahaja po razširitvi izvorne kode v:

```
\ethercat-1.5.2\examples\mini
```

Testni modul prevedemo z ukazom:

```
make modules
```

Naložimo ga z ukazom:

```
insmod ec_mini.ko
```

4.3.1 Izvorna koda

Izvorno kodo vzorčnega modula dopolnimo z našimi napravami. To naredimo s pomočjo orodja za ukazno vrstico »ethercat«, katera je del IgH EtherCAT Masterja.

Zaporedna ukaza:

```
ethercat cstruct -p 0
ethercat cstruct -p 1
```

Ukaza nam ustvarita kodo v programskem jeziku C, ki vsebuje PDO opis obeh naprav in jo lahko prenesemo v nastajajoči gonilnik. Stikalo -p in številka za njim nam povesta položaj oz. za katerega sužnja želimo kodo. Če je sužnjev več oziroma nismo prepričani, na kateri poziciji se nahajajo, lahko predhodno izvedemo ukaz:

```
ethercat slaves
```

ki po vrsticah ločeno izpiše topologijo vseh sužnjev v mreži EtherCAT z njihovo pozicijo (ki jo lahko uporabimo v predhodnem ukazu) in njihovim opisom, s pomočjo katerega lahko potem napravo tudi fizično identificiramo.

Generirano kodo v programskem jeziku C (od obeh sužnjev) vnesemo v nastajajoči gonilnik:

```
#ifdef CONFIGURE_PDOS
/* Master 0, Slave 0, "EK1101"
 * Vendor ID:       0x00000002
 * Product code:    0x044d2c52
 * Revision number: 0x00110000
 */
ec_pdo_entry_info_t slave_0_pdo_entries[] = {
    {0x6000, 0x01, 16}, /* ID */
};
ec_pdo_info_t slave_0_pdos[] = {
    {0x1a00, 1, slave_0_pdo_entries + 0}, /* ID */
};
ec_sync_info_t slave_0_syncs[] = {
    {0, EC_DIR_INPUT, 1, slave_0_pdos + 0, EC_WD_DISABLE},
    {0xff}
};
/* Master 0, Slave 1, "EL3312"
 * Vendor ID:       0x00000002
 * Product code:    0x0cf03052
 * Revision number: 0x00120000
 */
ec_pdo_entry_info_t slave_1_pdo_entries[] = {
    {0x6000, 0x01, 1}, /* Underrange */
    {0x6000, 0x02, 1}, /* Overrange */
    {0x6000, 0x03, 2}, /* Limit 1 */
    {0x6000, 0x05, 2}, /* Limit 2 */
    {0x6000, 0x07, 1}, /* Error */
    {0x0000, 0x00, 7}, /* Gap */
    {0x6000, 0x0f, 1}, /* TxPDO State */
    {0x1800, 0x09, 1},
    {0x6000, 0x11, 16}, /* Value */
    {0x6010, 0x01, 1}, /* Underrange */
    {0x6010, 0x02, 1}, /* Overrange */
    {0x6010, 0x03, 2}, /* Limit 1 */
    {0x6010, 0x05, 2}, /* Limit 2 */
    {0x6010, 0x07, 1}, /* Error */
    {0x0000, 0x00, 7}, /* Gap */
    {0x6010, 0x0f, 1}, /* TxPDO State */
    {0x1801, 0x09, 1},
    {0x6010, 0x11, 160}, /* Value */
};
ec_pdo_info_t slave_1_pdos[] = {
    {0x1a00, 9, slave_1_pdo_entries + 0}, /* TC TxPDO-MapCh.1 */
    {0x1a01, 9, slave_1_pdo_entries + 9}, /* TC TxPDO-MapCh.2 */
```

```
};
ec_sync_info_t slave_1_syncs[] = {
    {0, EC_DIR_OUTPUT, 0, NULL, EC_WD_DISABLE},
    {1, EC_DIR_INPUT, 0, NULL, EC_WD_DISABLE},
    {2, EC_DIR_OUTPUT, 0, NULL, EC_WD_DISABLE},
    {3, EC_DIR_INPUT, 2, slave_1_pdos + 0, EC_WD_DISABLE},
    {0xff}
};
#endif
```

Pomemben del gonilnika je funkcija *cyclic_task*, saj skrbi, da se paketi periodično pošiljajo oziroma sprejemajo iz omrežja. Ker želimo zajemati z višjo frekvenco, kot je v osnovnem primeru (100 Hz), je potrebno funkcijo preurediti. Uporabili bomo strukturo »hr_timer« tipa HRT, katero smo predstavili v poglavju 2.2 in je ena izmed glavnih prednosti RT-Preempt.

V modul dodamo:

```
#include <linux/hrtimer.h>
#include <linux/ktime.h>
static struct hrtimer hr_timer;
static ktime_t ktime;
```

Preurejena funkcija *cyclic_task* dobi sledečo obliko:

```
static enum hrtimer_restart cyclic_task(struct hrtimer* timer)
{
    // receive process data
    down(&master_sem);
    ecrt_master_receive(master);
    ecrt_domain_process(domain1);
    up(&master_sem);

    trafficGenerator();

    // send process data
    down(&master_sem);
    ecrt_domain_queue(domain1);
    ecrt_master_send(master);
    up(&master_sem);

    hrtimer_forward_now(&hr_timer,ktime);
    return HRTIMER_RESTART;
}
```

Takoj opazimo, da je funkcija ostala nespremenjena v delu, kjer se podatki sprejemajo in oddajajo. Dodali smo klic na funkcijo *trafficGenerator*.

Razvidno je tudi, da se je spremenil tip funkcije, zato bo treba spremeniti tudi nadrejene klice funkcije pri inicializaciji samega gonilnika.

```
printk(KERN_INFO PFX "Starting cyclic sample thread.\n");

hrtimer_init(&hr_timer,CLOCK_MONOTONIC,HRTIMER_MODE_REL);
hr_timer.function = &cyclic_task;
ktime = ktime_set( 0, FREQ_NSEC );
hrtimer_start(&hr_timer,ktime,HRTIMER_MODE_REL);

printk(KERN_INFO PFX "Started.\n");
```

Zgoraj je spremenjeni del izseka oziroma inicializacijske funkcije, kateri se izvede takoj ko ga naložimo. Ta del kode je dejansko tisti, ki omogoča, da naš gonilnik zajema podatke z visokimi frekvencami v realnem času.

Struktura *hrtimer*, ki smo jo vključili s knjižico *linux/hrtimer.h* [16]:

```
struct hrtimer {
         struct timerqueue_node          node;
         ktime_t                         _softexpires;
         enum hrtimer_restart            (*function)(struct hrtimer *);
         struct hrtimer_clock_base       *base;
         unsigned long                   state;
 #ifdef CONFIG_TIMER_STATS
         int                             start_pid;
         void                            *start_site;
         char                            start_comm[16];
 #endif
 };
```

Struktura *ktime_t,* ki smo jo vključili s knjižico *linux/ktime.h*[17]:

```
union ktime {
        s64     tv64;
#if BITS_PER_LONG != 64 && !defined(CONFIG_KTIME_SCALAR)
        struct {
# ifdef __BIG_ENDIAN
        s32     sec, nsec;
# else
         s32     nsec, sec;
 # endif
         } tv;
 #endif
 };

 typedef union ktime ktime_t;

 #define KTIME_MAX                       ((s64)~((u64)1 << 63))
 #if (BITS_PER_LONG == 64)
 # define KTIME_SEC_MAX                  (KTIME_MAX / NSEC_PER_SEC)
 #else
 # define KTIME_SEC_MAX                  LONG_MAX
 #endif
```

Pri strukturi *ktime* moramo biti pozorni, če jo uporabimo na 32-bitni arhitekturi, kar v našem primeru ni bila težava. Na 64-bitni arhitekturi je tako v eni sami 64-bitni spremenljivki shranjena interna predstavitev vrednosti časa (v nanosekundah) strukture *hrtimer*. Spremenljivka nam tako določa iz frekvence pridobljeni interval v nanosekundah, ta interval pa nam potem spet posledično določa, kako pogosto bomo zajemali podatke iz mreže. Našo ciklično funkcijo za zajem časa tako sprožimo z določenim intervalom *ktime*.

5 MERITVE

Meritve smo razdelili na tri dele. Izvedene so bile na dveh različnih sistemih, od katerih je drugi sistem, katerega bomo v nadaljevanju imenovali »produkcijski sistem«, prinesel preobrat pri izvajanju meritev. Zaradi tega smo meritve za testni sistem ponovili, ohranili pa smo oba nabora testnih rezultatov, saj smo tako dobili še boljši vpogled v rešitev, s katero smo potem optimizirali oba sistema. Rezultati so predstavljeni v treh ločenih tabelah in primerjalnima grafoma.

5.1 Testni sistem

Rezultate meritev, katere smo opravili po izdelanem gonilniku, je z besedami najkrajše možno opisati kot iskanje najvišje veljavne frekvence našega cikla zajema. Po dokumentaciji, ki je na voljo za IgH EtherCAT Master, je frekvenca zajema previsoka takrat, kadar se v sistemskih dnevniški zapisih pojavijo »Datagram SKIPPED« zapisi. Do težave pride, ker se EtherCAT okvirji (angl. frames), ki so bili odposlani ob koncu cikla, še niso vrnili ob začetku naslednjega cikla. Gospodar to zazna po delovnem števcu, ki se ni povečal in izpiše opozorilo v sistemski dnevnik. Začetno meritev smo opravili pri 20 kHz in jo potem po korakih povečevali do 30 kHz. Višje od te frekvence ni šlo, saj so se že po povečanju za samo nekaj 100 Hz pojavila opozorila.

Tabela 5.1: Meritve na testnem sistemu

Frekvenca [Hz]	20000	25000	26500	28000	30000
Čas cikla [µs]	50	40	≈37.7	≈35.7	≈33.3
Mininimalni odklon [ns]	523	520	497	520	510
Maksimalni odklon [ns]	22181	14011	21508	21888	23402
Povprečni odklon [ns]	1007	1092	969	1021	1083
[N] ciklov odklon > 5 [µs]	2592	2113	21396	44207	104131
[N] ciklov odklon > 10 [µs]	89	12	10662	20555	30544
[N] ciklov odklon > 20 [µs]	2	0	6	31	76
Število ciklov	171614270	178212639	777420344	969274663	1227691735
Trajanje meritve ≈	2h 23min	1h 59min	8h 9min	9h 37min	11h 22min

5.2 Produkcijski sistem

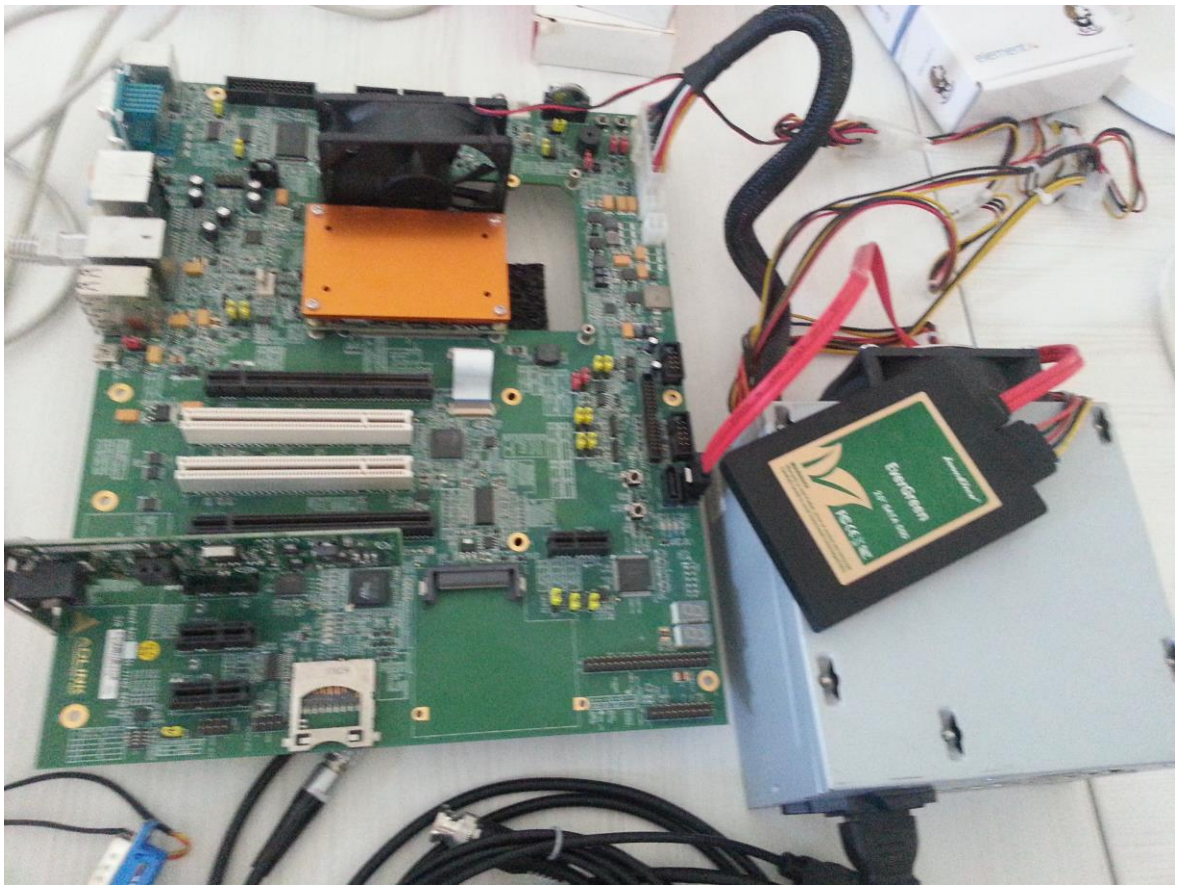
Ob izvajanju meritev na našem testnem sistemu je zunanja gospodarska družba izrazila željo, da testiramo njihov produkcijski sistem, zasnovan okoli SoC (angl. System on a Chip). SoC so največkrat v uporabi na področju vgrajenih sistemov (angl. Embedded Systems). Posebnost testiranega sistema (na sliki 5.1) je v delovanju pri ekstremnih temperaturnih razmerah (od -40 C do +85 C).

Slika 5.1: ADLINK nanoX-BT COM Express modul

SoC nam je bil posredovan z referenčno matično ploščo istega proizvajalca, dodali smo še SSD disk in že prej uporabljeno mrežno kartico. Sistem na sliki 5.2 je vseboval torej sledeče komponente:

- ADLINK nanoX-BT COM Express modul[8] z Intel Celeron J1900 CPU (SoC)
- Intel i210LM integriran mrežni vmesnik
- Intel 82574L PCIe mrežna kartica
- InnoDisk EverGreen SATA SSD 60GB

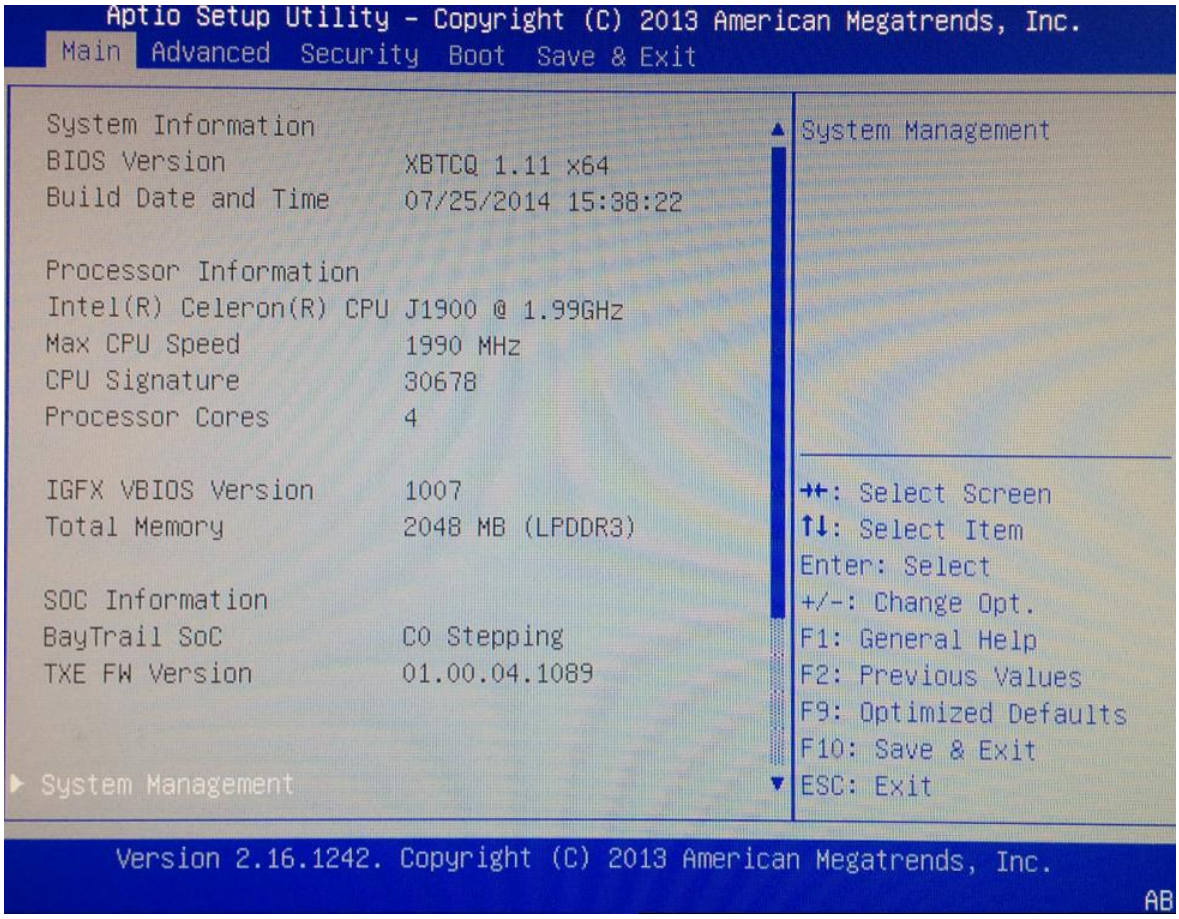
[8] http://www.adlinktech.com/PD/web/PD_detail.php?pid=1330



Slika 5.2: Sestavljeni produkcijski sistem za testiranje

Sistem smo zgradili kot je zapisano v poglavju 3.3. Ker se postopki bistveno ne razlikujejo od postopkov pri gradnji testnega sistema, jih v tem koraku ne bomo podrobneje opisovali. Posvetili se bomo samo dodatnim optimizacijam BIOS, ki so kljub temu da gre za arhitekturo Intel, zaradi zasnove samega SoC nekoliko drugačne. Sistem uporablja Aptio[9] BIOS proizvajalca AMI, kar je razvidno iz slike 5.3.

[9] https://ami.com/products/bios-uefi-firmware/



Slika 5.3: ADLINK nanoX-BT COM Express modul – BIOS nastavitve

BIOS nastavimo preko treh glavnih sklopov; vsakega bomo podrobneje predstavili.
V prvem sklopu nastavitev najprej izberemo:

- ZAVIHEK MAIN

Privzeto se na začetku prikažejo osnovne informacije o sistemu. Čisto na dnu, kot je razvidno iz slike 4.6, izberemo:

- SYSTEM MANAGEMENT

Na novem odprtem podmeniju se osredotočimo na sklop:

- HARDWARE CONTROLS

Izberemo:

- POWER UP
- ECO MODE [Disabled]

Sedaj se znova vrnemo na višji nivo in izberemo:

- SMART FAN
- CPU FAN MODE [Fan Off]

Zadnja nastavitev bi se morda naključnemu opazovalcu zdela ekstremna, saj prav gotovo ne želimo pregrevanja procesorja. Z izbranim procesorjem v sistemu teh težav nimamo, saj ima izredno nizek TDP (angl. Thermal Design Power) 10 W in zato zadostuje boljše pasivno hlajenje. Pozorni opazovalec bo sicer na sliki 5.2 opazil ventilator blizu SoC. Tega smo dodali zato, ker nismo imeli dovolj dobro realiziranega pasivnega hlajenja. Ker pa je ventilator priklopljen s standardnim (angl. molex) priključkom direktno na napajalnik, nima vpliva na sistem.
V drugem sklopu, v katerem je tudi največ nastavitev, najprej izberemo:

- ZAVIHEK ADVANCED

Na novem prikazanem podmeniju izberemo:

- CPU
- INTEL VIRTUALIZATION [Disabled]

Sledi podsklop:

- CPU PROCESSOR POWER MANAGEMENT
- EIST [Disabled]
- CPU C STATE REPORT [Disabled]
- CPU DTS [Disabled]

Nato izberemo možnost:

- GRAPHICS
- GT – POWER MANAGEMENT CONTROL RC6(RENDER STANDBY) [Disabled]

Po vrnitvi v nadrejeni meni izberemo možnost:

- SATA
- SATA MODE SELECTION [AHCI Mode]

Naslednji izbor je možnost:

- ACPI AND POWER MANAGEMENT
- ENABLE ACPI AUTO CONF [Disabled]
- ENABLE HIBERNATION [Disabled]
- ACPI SLEEP STATE [Suspend Disabled]
- EMULATION AT/ATX [ATX]

Po vrnitvi v nadrejeni meni izberemo:

- SOUND AZAILA [Disabled]

Po vrnitvi v nadrejeni meni izberemo:

- THERMAL [Disabled]

Kot zadnjo v tem sklopu nastavimo še možnost:

- MISCELLANEOUS

- HIGH PRECISION TIMER [Enabled]
- SCC CONFIGURATION
- OS SELECTION [Android]

V zadnjem kratkem sklopu nastavimo samo še sledeče:

- QUIET BOOT [Disabled]
- FAST BOOT [Disabled]
- WIN8 SUPPORT [Disabled]

Nastavitev z direktnim vplivom na odzivnost sistema je kar veliko, v večini primerov je tudi na drugih podobnih sistemih glavna ideja izklopiti vse, kar pripomore k varčevanju porabe. Včasih nam lahko na prvi pogled majhna in obrobna nastavitev vzame nekaj ur (ali celo dni) dragocenega časa pri testiranju. Drugih metod od poizkušanja tukaj ni, razen če že imamo oz. pridobimo s strani proizvajalca matične plošče nabor testiranih nastavitev za realno-časovni sistem, kar pa je žal prej izjema kot pravilo tudi v nadstandardnih rešitvah.

Zahteve pri testiranju produkcijskega sistema so bile predvsem ugotovitev maksimalnih odklonov od periode zajema podatkov iz mreže EtherCAT. Da bi bili testi res zanesljivi, smo jih izvajali dalj časa kot v primeru z našim testnim sistemom. Izvedli smo štiri ponovitve meritev, pri čemer je bila vsaka daljša od prejšnje. Vse pri frekvenci zajema 5000 Hz. Rezultati so zbrani v tabeli 5.2.

Tabela 5.2: Meritve na produkcijskem sistemu

Izolacija jedra	NE	DA	DA	DA
Frekvenca [Hz]	5000	5000	5000	5000
Čas cikla [µs]	200	200	200	200
Max. neg. odklon [ns]	-79724	-9836	-13352	-33908
Max. poz. odklon [ns]	79468	10156	13828	39364
Povpreč. neg [ns]	-748	-675	-221	-465
Povpreč. poz. [ns]	747	370	119	243
[N] ciklov odklon > 5 [µs]	5144896	4386	7787	358773
[N] ciklov odklon > 10 [µs]	64669	0	322	22574
[N] ciklov odklon > 20 [µs]	20479	0	0	46
[N] ciklov odklon < -5 [µs]	4953944	4105	7786	378829
[N] ciklov odklon < -10 [µs]	70109	0	318	46516
[N] ciklov odklon < -20 [µs]	ni podatka	0	0	85
Število ciklov	200963115	792920927	1365398030	1933655074
Trajanje meritve ≈	11h 10min	44h 3min	75h 51min	107h 25min

Iz tabele 5.2 za produkcijski sistem najbolj izstopa prva meritev; trajala je dobrih 11 ur. Ta meritev je pomembna, saj je spremenila naš pristop pri razvoju gonilnika, kar si bomo ogledali v naslednjem poglavju.

5.3 Ponovljene meritve na testnem in produkcijskem sistemu z uporabo izolacije jeder

Pred to meritvijo smo imeli izhodišče v meritvah, pridobljenih na našem testnem sistemu. Sistema se na prvi pogled veliko ne razlikujeta, vendar pa je vseeno bistvena razlika v procesorju. Procesor Intel Celeron J1900[10] produkcijskega sistema ima zelo nizek TDP, ki je za faktor 5x nižji od procesorja v našem testnem sistemu Intel Pentium G3220[11]. Vendar pa poraba energije ni edina razlika. Bistvena razlika, ki ima vpliv na realno-časovno obnašanje procesorja, je število jeder. Celeron ima štiri jedra, Pentium pa le dve. Teoretično ozadje za ta vpliv smo pojasnili v 3. poglavju, pri meritvah pa mu nismo posvečali dovolj pozornosti, saj so bile prvotne meritve na testnem sistemu v območju pričakovanih. Ker pa se je situacija pri produkcijskem okolju spremenila (glej prvo meritev iz tabele 5.2), je bilo potrebno produkcijski sistem dodatno optimizirati, oziroma poiskati rešitev za procesorje z več jedri.

Če bi eno jedro namenili zgolj za realno-časovno opravilo, bi se izognili težavam, ki prinašajo razvrščevalnik in izenačevalnik obremenitev (angl. load balancer). V operacijskem sistemu Linux obstaja ukaz *isolcpus* [18]. Kot že samo ime namiguje, ukaz izolira jedro oziroma lahko tudi več jeder. Ker gre za operacijo, ki se zgodi ob zagonu, vpišemo ukaz v zagonski nalagalnik. V našem primeru se zagonska datoteka nahaja v:

```
/boot/grub/menu.lst
```

Primer takega modficiranga zapisa zagonske datoteke na produkcijskem sistemu:

```
kernel /boot/vmlinuz-3.4.106-rt131-2.44-desktop root=/dev/disk/by-id/ata-
EverGreen_2.5_SATA_SSD_20110318AAD0IDC00002-part1         video=1280x1024
resume=/dev/disk/by-id/ata-EverGreen_2.5_SATA_SSD_20110318AAD0IDC00002-
part2 splash=silent quiet showopts isolcpus=3 vga=0x31a
```

[10] http://ark.intel.com/products/78867/
[11] http://ark.intel.com/products/77773

Ker jedra označujemo od 0 naprej, smo z gornjim ukazom izolirali četrto jedro (z indeksom 3). Če bi želeli izolirati dve jedri, ločimo indekse jeder v ukazu z vejico (isolcpus=2,3 itd.). Sistem izoliranega jedra tako ne bo uporabljal, lahko pa seveda storimo to sami, in sicer na dva načina.

Prvi način je z ukazom *taskset*, s katerim vežemo že zagnan proces na jedro.

```
taskset -cp 3 9030
```

Gornji ukaz veže proces s PID 9030 na jedro z indeksom 3. Drugi pristop pa je dodelitev jedra procesu že ob njegovem zagonu:

```
taskset 0x8 vlc
```

Izraz 0x8 pomeni jedro z indeksom 3 v šestnajstiškem zapisu oziroma njegovo masko (angl. mask). Jedro 0 ima oznako 0x1, jedro 1 0x2, jedro 2 0x4, jedro 3 0x8 itd. Maska 0x9 bi tako pomenila jedri 0 in 3.

S to ugotovitvijo in modificiranim zagonskim nalagalnikom se nato vrnemo znova k meritvam v našem produkcijskem sistemu. Rezultati so sedaj popolnoma drugačni, kar pokažejo preostale tri meritve iz tabele 5.2, čeprav je najdaljša meritev trajala več kot 107 ur in je vsebovala skoraj 10 x več ciklov zajema od prvotne meritev, noben izmed ciklov ni presegel niti polovice odstopanja prvotne meritve.

Vprašali smo se, ali je lahko kaj podobnega možno tudi na testnem sistemu? Vprašanje, podkrepljeno z meritvami na produkcijskem sistemu nas vrne na začetek k tabeli 5.1, kjer smo frekvenco zviševali do meje, kjer sistem ni več stabilno zajemal paketov iz omrežja in je prihajalo do izgubljenih (angl. SKIPPED/UNMATCHED) datagramov EtherCAT našega testnega sistema.

Če je imela izolacija jedra vpliv na 4-jedrnem procesorju, ga mora imeti tudi pri 2-jedrnem. Vsaj v obsegu, katerega bi morali zaznati z meritvami.

Podobno kot prej za produkcijski sistem spremenimo zagonski nalagalnik testnega sistema in izvedemo meritve, katerih rezultati so zbrani v tabeli 5.3.

Tabela 5.3: Meritve na testnem sistemu z izolacijo jedra

Frekvenca [Hz]	35000	38000	41000	415000	42000
Čas cikla [µs]	≈ 28.6	≈ 26.3	≈ 24.4	≈ 24.1	≈ 23.8
Mininimalni odklon [ns]	11	11	9	9	10
Maksimalni odklon [ns]	11305	8984	8924	10244	9543
Povprečni odklon [ns]	100	101	195	263	357
[N] ciklov odklon > 5 [µs]	212	216	596	1842	1634
[N] ciklov odklon > 10 [µs]	1	0	0	1	0
[N] ciklov odklon > 20 [µs]	0	0	0	0	0
Število ciklov	399626260	1459957885	1802152528	3300179343	1852320871
Trajanje meritve ≈	3h 10min	10h 40min	12h 12min	22h 5min	12h 15min

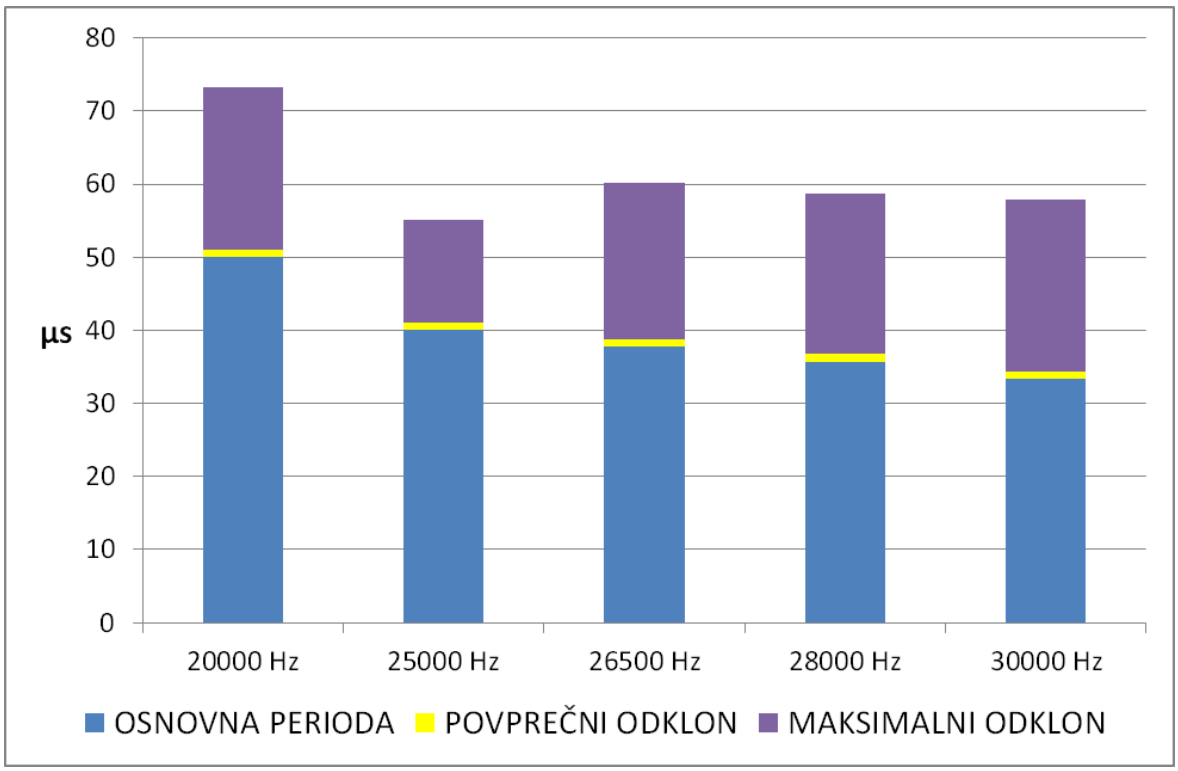
Tudi na našem testnem sistemu z izolacijo jedra dobimo precej boljše rezultate. Podobno kot pri prvotnih meritvah, zbranih v tabeli 5.1, tudi tu izberemo začetno frekvenco in jo po korakih povečujemo. Začnemo pri 35 kHz in nadaljujemo vse do 42 kHz. Višje od te frekvence ni šlo, ker so se v sistemskem dnevniku spet pojavila opozorila. Dobljena frekvenca je torej najvišja še veljavna brez izgubljenih EtherCAT datagramov. Dobljena frekvenca je zelo podobna frekvenci, določeni s predpostavko v poglavju 3, kjer smo meritve izvajali s sintetičnim orodjem. Če upoštevamo, da je natančnost cyclictesta 1 µs, lahko sklepamo, da smo z našim modulom iz testnega sistema dobili maksimalen rezultat.

Zanimivih je še nekaj interpretacij, pridobljenih rezultatov meritev v tabeli 5.3. Minimalni odkloni od periode vzorčenja so bili na vseh frekvencah, tudi pri najvišji še veljavni, okoli10 ns. Pri strogo realno-časovnemu sistemu ta karakteristika nima posebnega pomena,

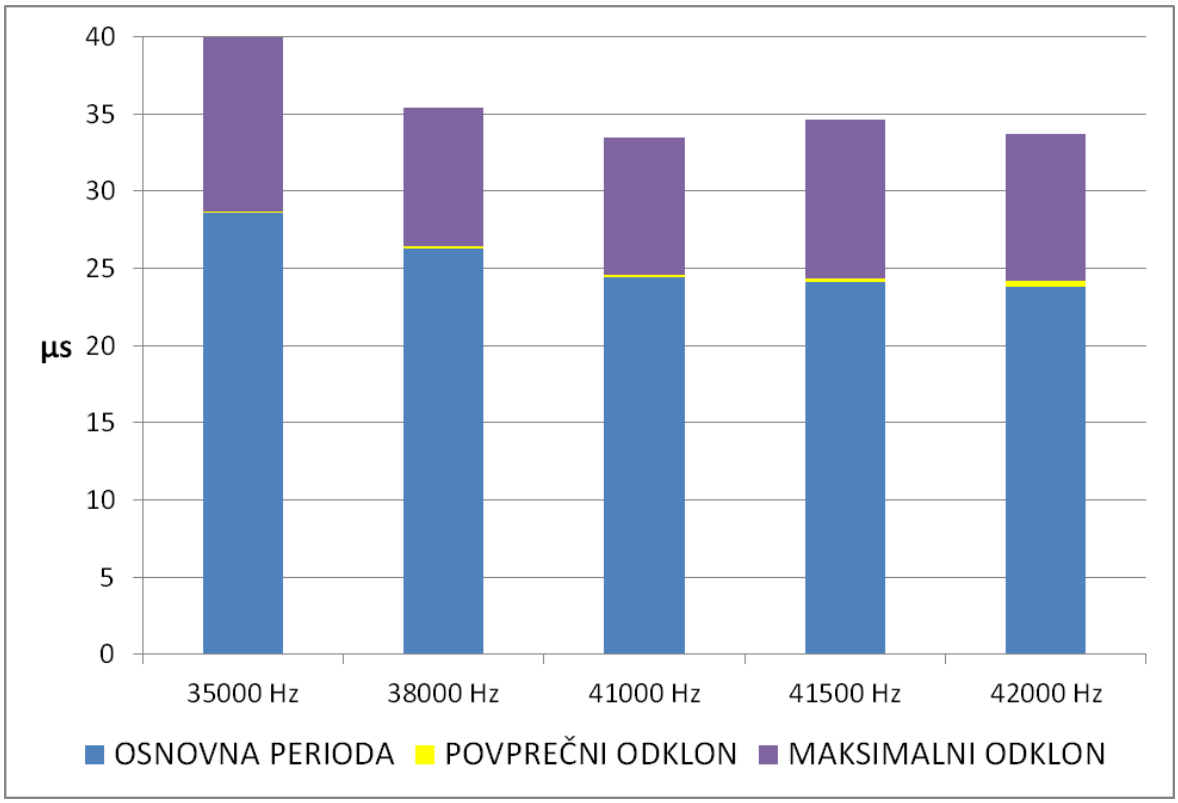
nam pa pove minimalno zakasnitev oziroma odziv našega sistema. Če primerjamo to z rezultati iz tabele 5.1, kjer jedra nismo izolirali in so bile minimalne zakasnitve okoli 0.5 µs gre za 50x izboljšanje. Karakteristika, ki je bolj zanimiva, je povprečni odklon od periode. V ohlapnih realno-časovnih sistemih se, kot že vemo, zadovoljimo s povprečno odzivnostjo sistema. Z naraščanjem frekvence se je povečevala tudi povprečna zakasnitev, izražena v nanosekundah od 100 do 357 pri najvišji možni frekvenci, kar pomeni, da se je povečala več kot 250 %. Na drugi strani pa je bil povprečni odklon od periode vzorčenja pri sistemu z neizoliranim jedrom pri vseh frekvencah okoli 1 µs, kar bi lahko pripisali vplivu razvrščevalnika poslov in izenačevalnika obremenitev procesorja, kar pa seveda ni imelo vpliva pri izoliranem jedru. Pri meritvah, zbranih v tabeli 5.3 še posebej izstopa število skrajnih, torej kritičnih odstopanj od periode, ki niso v nobenem primeru presegla 11,3 µs. V četrti, najdaljši, meritvi, ki je trajala več kot 22 ur pri frekvenci 41.5 kHz pa smo zabeležili en sam odklon, večji od 10 µs. Tudi število odklonov, večjih od 5 µs pri omenjeni frekvenci je bilo majhno in pri številu vseh ciklov, ki jih je bilo malo več kot 3,3 milijarde, predstavlja malo več kot polovico milijoninke ciklov oz. en tak cikel na vsakih 43 sekund.

Direktna primerjava zbranih rezultatov iz tabel 4.1 in 4.3 je dodatno prikazana v grafih 5.4 in 5.5.

Graf 5.4 Periode testnega sistema brez izolacije jedra



Graf 5.5 Periode testnega sistema z izolacijo jedra



6 ZAKLJUČEK

Operacijski sistem Linux je s popravki jedra RT-Preempt postal zrela rešitev v svetu, kjer so še pred nekaj leti bile kot zanesljive znane le rešitve z vmesnim oziroma dodatnim tankim jedrom. Čeprav se tudi ostale rešitve še vedno razvijajo, je podpora skupnosti in stopnjevanje razvoja na strani RT-Preempt. Kot rešitev ga je zelo preprosto uporabiti, kar smo pokazali z vzpostavitvijo sistema za uporabo v strogem realnem-času tako rekoč iz ničle. Začetni cilj, da bi podatke zajemali s frekvencami reda okoli 10 kHz, smo presegli za faktor štiri. Na začetku eksperimentalnega dela se je sicer to zdelo teoretično možno, v praksi pa takih rezultatov vsekakor nismo pričakovali.

Zelo visoka zagotovljena frekvenca zajema podatkov je z vidika njihove realno-časovne obdelave samo ena plat medalje in to smo obravnavali v tem diplomskem delu. Druga plat je seveda obdelava teh podatkov. Frekvenca nam s svojim višanjem pušča v obratnem sorazmerju seveda zelo kratke časovne intervale za obdelavo, kar posledično pri razvoju programske opreme pomeni dodatne optimizacijske probleme. Ker z gonilnikom nismo na uporabniškem nivoju, ampak na nivoju jedra operacijskega sistema, je dobro načrtovanje pri implementaciji rešitev za obdelavo še toliko pomembnejše.

S povečevanjem sočasnosti, tudi na nivoju vgrajenih sistemov, je večjedrni procesor postal vsakdanjik, srečujemo rešitve tudi z 8 in več jedri. Pokazali smo, da je možno zelo preprosto določenemu jedru v sistemu namensko določiti, za kaj ga bomo uporabili in s tem izboljšali odzivnost sistema. Na tem področju se z razvojem odpira prav gotovo še več možnosti in rešitev.

Sistem sam bi lahko dodatno optimizirali še na najnižjem nivoju z dodatnimi nastavitvami BIOS, kar pa je seveda precej odvisno od proizvajalcev, ki nam lahko izdelajo posebne (angl. custom) BIOS rešitve.

Na koncu naj omenimo še, da pred kratkim razvita najnovejša Linux jedra od verzije 3.14 naprej podpirajo novo disciplino za razvrščanje SCHED_DEADLINE[19], zasnovano na

algoritmih EDF (angl. Earliest Deadline First) in CBS (angl. Constant Bandwidth Server), razvito prav posebej z namenom realno-časovne uporabe. Če disciplino povzamemo zelo na kratko, lahko izpostavimo, da popolnoma opusti notacijo prioritet procesov in uvede tri nove parametre (angl. runtime, period, deadline). Z uporabo teh informacij razvrščevalnik potem najprej izvede procese s skrajnim rokom. V naših testih tega razvrščanja žal še nismo mogli uporabiti, saj jedro 3.14 ob času testiranja še ni bilo dovolj stabilno, niti še ni bilo ustreznih modificiranih gonilnikov za mrežno kartico. Ta izziv ostaja tako še odprt, njegova uporaba pa prav gotovo pomeni še en velik korak naprej v evoluciji razvoja realno-časovnih sistemov v okolju Linux.

7 VIRI

[1] *Industiral Ethernet*, dostopno na:
https://en.wikipedia.org/wiki/Industrial_Ethernet [29. 8. 2106]
[2] Ethercat Techonlogy Group, dostopno na:
https://www.ethercat.org/ [29. 8. 2016]
[3] *GNU/Linux naming controversy*, dostopno na:
https://en.wikipedia.org/wiki/GNU/Linux_naming_controversy [29. 8. 2016]
[4] Jones M. (2007) Anatomy of the Linux kernel, dostopno na:
https://www.ibm.com/developerworks/library/l-linux-kernel/ [29. 8. 2016]
[5] Jones M. (2007) Anatomy of real-time Linux architectures, dostopno na:
http://www.ibm.com/developerworks/linux/library/l-real-time-linux/ [29. 8. 2016]
[6] Make it Faster: More Throughput or Less Latency?, dostopno na:
http://www.ni.com/white-paper/14990/en/ [29. 8. 2016]
[7] Embedded System, dostopno na:
http://yebig.tistory.com/105 [29. 8. 2016]
[8] Scheduling and Synchronization, dostopno na:
http://www.aicas.com/jamaica/3.4/doc/html/x4057.html [29. 8. 2016]
[9] The Linux Kernel Archives, rt-projects, dostopno na:
https://www.kernel.org/pub/linux/kernel/projects/rt/ [29. 8. 2016]
[10] RTAI - the RealTime Application Interface for Linux, dostopno na:
https://www.rtai.org/userfiles/downloads/RTAI/ [29. 8. 2016]
[11] openSUSE, dostopno na:
https://www.opensuse.org/ [29. 8. 2016]
[12] YaST, dostopno na:
http://yast.github.io/ [29. 8. 2016]
[13] Cyclictest, dostopno na:
https://rt.wiki.kernel.org/index.php/Cyclictest [29. 8. 2016]
[14] The Linux Kernel Archives, rt-test, dostopno na:
https://www.kernel.org/pub/linux/utils/rt-tests/ [29. 8. 2016]

[13] Beckhoff Automation GmbH & Co. KG, dostopno na:
http://www.beckhoff.de [29. 8. 2016]
[14] IgH EtherCAT Master for Linux, dostopno na:
http://etherlab.org/en/ethercat/ [29. 8. 2016]
[15] GNU General Public License, version 2, dostopno na:
https://www.gnu.org/licenses/old-licenses/gpl-2.0.html [29. 8. 2016]
[16] Knjižica linux/hrtimer.h za jedro 3.4, dostopno na:
http://lxr.free-electrons.com/source/include/linux/hrtimer.h?v=3.4 [29. 8. 2016]
[17] Knjižica linux/ktime.h za jedro 3.4 dostopno na:
http://lxr.free-electrons.com/source/include/linux/ktime.h?v=3.4 [29. 8. 2016]
[18] Kroah-Hartman G. (2006) *Linux Kernel in a Nutshell*, dostopno na:
http://www.linuxtopia.org/online_books/linux_kernel/kernel_configuration/ [29. 8. 2016]
[19] SCHED_DEADLINE, dostopno na:
https://en.wikipedia.org/wiki/SCHED_DEADLINE [29. 8. 2016]

Univerza v Mariboru

Fakulteta za elektrotehniko, računalništvo in informatiko

FERI

IZJAVA O AVTORSTVU

Spodaj podpisani/-a Mario Casar

z vpisno številko 93439032

sem avtor/-ica diplomskega dela z naslovom:

GONILNIK ZA REALNO-ČASOVNI ZAJEM PAKETOV IZ MREŽE ETHERCAT V OKOLJU LINUX

(naslov diplomskega dela)

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom (naziv, ime in priimek)

doc. dr. Tomaža Kosarja

in somentorstvom (naziv, ime in priimek)

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela.
- soglašam z javno objavo elektronske oblike diplomskega dela v DKUM.

V Mariboru, dne 31. 8. 2016

Podpis avtorja/-ice:

Univerza v Mariboru

Fakulteta za elektrotehniko, računalništvo in informatiko

FERI

IZJAVA O USTREZNOSTI ZAKLJUČNEGA DELA

Podpisani mentor :

doc. dr. Tomaž Kosar

Izjavljam (-va), da je študent

Ime in priimek: *Mario Casar*

Vpisna številka: *93439032*

Na programu: RAČUNALNIŠTVO IN INFORMATIKA - UNI

izdelal zaključno delo z naslovom:

GONILNIK ZA REALNO-ČASOVNI ZAJEM PAKETOV IZ MREŽE ETHERCAT V OKOLJU LINUX

(naslov zaključnega dela v slovenskem in angleškem jeziku)

DRIVER FOR REAL-TIME PACKET ACQUSITION FROM ETHERCAT NETWORK

v skladu z odobreno temo zaključnega dela, Navodilih o pripravi zaključnih del in mojimi (najinimi oziroma našimi) navodili.

Preveril (-a, -i) in pregledal (-a, -i) sem (sva, smo) poročilo o plagiatorstvu.

Datum in kraj:
31. 8. 2016

Podpis mentorja:

Datum in kraj:

Podpis somentorja (če obstaja):

Priloga:

- Poročilo o preverjanju podobnosti z drugimi deli.

Univerza v Mariboru

Fakulteta za elektrotehniko, računalništvo in informatiko

FERI

IZJAVA O ISTOVETNOSTI TISKANE IN ELEKTRONSKE VERZIJE ZAKLJUČNEGA DELA IN OBJAVI OSEBNIH PODATKOV DIPLOMANTOV

Ime in priimek avtorja-ice: Mario Casar

Vpisna številka: 93439032

Študijski program: RAČUNALNITVO IN INFORMATIKA - UNI

Naslov zaključnega dela: *GONILNIK ZA REALNO-ČASOVNI ZAJEM PAKETOV IZ MREŽE ETHERCAT V OKOLJU LINUX*

Mentor: doc. dr. Tomaž Kosar

Podpisani-a Mario Casar izjavljam, da sem za potrebe arhiviranja oddal elektronsko verzijo zaključnega dela v Digitalno knjižnico Univerze v Mariboru. Zaključno delo sem izdelal-a sam-a ob pomoči mentorja. V skladu s 1. odstavkom 21. člena Zakona o avtorskih in sorodnih pravicah dovoljujem, da se zgoraj navedeno zaključno delo objavi na portalu Digitalne knjižnice Univerze v Mariboru.

Tiskana verzija zaključnega dela je istovetna z elektronsko verzijo elektronski verziji, ki sem jo oddal za objavo v Digitalno knjižnico Univerze v Mariboru.

Zaključno delo zaradi zagotavljanja konkurenčne prednosti, varstva industrijske lastnine ali tajnosti podatkov naročnika: _____ ne sme biti javno dostopno do _____(datum odloga javne objave ne sme biti daljši kot 3 leta od zagovora dela).

Podpisani izjavljam, da dovoljujem objavo osebnih podatkov, vezanih na zaključek študija (ime, priimek, leto in kraj rojstva, datum zaključka študija, naslov zaključnega dela), na spletnih straneh in v publikacijah UM.

Datum in kraj: 31.8.2016 Podpis avtorja-ice: _____

Podpis mentorja: _____
(*samo v primeru, če delo ne sme biti javno dostopno*)

Podpis odgovorne osebe naročnika in žig:
(*samo v primeru, če delo ne sme biti javno dostopno*) _____