UNIVERSITETET I OSLO
Institutt for informatikk

Evaluering av data-stiimplementasjoner for nedlasting og streaming-applikasjoner

Masteroppgave

Tom Anders Dalseng

27. oktober 2005

Forord

Dette dokumentet utgjør min masteroppgave ved Institutt for informatikk, Universitetet i Oslo.

Jeg vil takke mine veiledere Pål Halvorsen og Carsten Griwodz for god veiledning, og mange gode forslag til forbedringer gjennom arbeidet med denne oppgaven. Jeg vil også takke mine medstudenter Andreas Petlund og Øystein Yri Sunde for gode innspill og gjennomlesing av deler av oppgaven. Til slutt vil jeg takke min samboer Mona for støtte, og for gjennomlesing og retting av skrivefeil.

Tom Anders Dalseng
Blindern, oktober 2005

Sammendrag

Denne oppgaven ser på dataflyttings- og kontekstbytteoverhead i Linux ved nedlasting og streaming. Dataflytting er en kostbar operasjon, og mange mekanismer har blitt foreslått og implementert for å redusere dataflyttingsoverhead. Vi har derfor sett på om nyere hardware og vanlige operativsystemer som Linux har klart å overkomme problemene med dataflyttingsoverhead, og hvor nært en mer optimalisert datasti de eksisterende løsningene er. I den sammenheng har vi evaluert forskjellige mekanismer som bruker forskjellige datastier i Linux 2.6 kjernen.

Vi undersøkte også i hvilken utstrekning eksisterende løsninger er tilstrekkelig i streamingsammenheng, og om dedikert støtte for streamingapplikasjoner kan gi forbedringer i ytelsen. Ved oppbygging av applikasjonslagsheadere, er det for noen typer media nødvendig med databerøringsoperasjoner for å hente ut informasjon om dataene som overføres, mens for andre er det tilstrekkelig med informasjon som gjelder for hele mediainnholdet og som vanligvis ikke er en del av filen som overføres. Det er derfor skilt mellom implementasjoner som støtter databerøringsoperasjoner, og de som ikke støtter dette. For å teste om dedikert støtte kan gi forbedringer i ytelsen er det implementert nye systemkall med støtte for streaming, og Real-time Transport Protocol-motor (RTP-motor) i kjernen.

Eksperimentene viste at Linux har en optimal implementasjon for nedlastingsoperasjoner med tanke på kopioperasjoner og kontekstbytter, men de eksisterende løsningene er ikke optimale for streaming. For media som krever databerøringsoperasjoner har vi klart å redusere CPU-bruken med 24 prosent ved fjerning av kopioperasjoner, mens for media som ikke krever databerøringsoperasjoner har vi klart å redusere CPU-bruken med 10 prosent ved redusering av kontekstbytter. RTP-motorene ga en ytterligere forbedring på 16 prosent for media som krever databerøringsoperasjoner og 19 prosent for media uten databerøringsoperasjoner. Sammenlignet med beste eksisterende løsninger gir RTP-motorene forbedringer på henholdsvis 36 prosent med databerøringsoperasjoner og 27 prosent uten databerøringsoperasjoner.

Innhold

Figurer

Tabeller

Kapittel 1

Introduksjon

Distribuerte multimediasystemer og applikasjoner som Media-on-Demand (MoD) begynner å få stor utbredelse og vil bli øke i antall i årene som kommer. MoD-servere lagrer multimediadata som video og audio og tilbyr avspillingstjenester til klienter. Den mest ressurskrevende operasjonen i en slik server er overføring av data fra disk til nettverk. I denne oppgaven ser vi på kostnadene ved overføring av data fra disk til nettverk og prøver å optimalisere denne datastien for å øke ytelsen for streamingapplikasjoner.

1.1 Bakgrunn og motivasjon

Eksperimenter med audio og video over pakkenettverk begynte tidlig på 1970-tallet, men først de siste årene har det blitt vanlig med streaming av audio og video over Internett. Tidligere var den mest vanlige Internettilkoblingen oppringt forbindelse over POTS (plain old telephone service) med hastigheter opp til 56Kbps. De siste årene har det kommet nyere teknologi, som Digital Subscriber Line (DSL) som muliggjør høyere hastigheter over kobbertelefonkablene og kabelmodem for dataoverføring over koaksial-kabeltv kablene. De høyere hastighetene, samt muligheten til å være tilkoblet til en fast pris, gjør det mulig å overføre store datamengder og har gjort nedlasting av mediafiler som iTunes populært. De høyere hastighetene åpner også for distribuerte multimedia streamingapplikasjoner, og det finnes i dag et stort antall streamingtjenester som Video-on-Demand (Filmarkivet.no), Audio-on-Demand (Lycos Rhapsody), Internettradio (NRK) og Internetttelefoni (Skype).

Multimedia på Internett kan leveres ved standard nedlasting eller ved streaming. Ved nedlasting overføres multimediadataene til klienten før avspilling, hvor over-

føringshastigheten har liten betydning, mens ved streaming overføres de ved samme hastighet som de blir avspilt.

I et nedlastingsscenario kan for eksempel en File Transfer Protocol-server (FTP-server) eller HyperText Transfer Protocol-server (HTTP-server) benyttes for overføring av multimediadataene. Ved bruk av HTTP refereres multimediafilene av en Uniform Resource Locator (URL) og ved hjelp av denne kan klienten utføre en HTTP-forespørsel til serveren, som starter overføringen av dataene. Overføringen skjer så raskt som mulig, hvor begrensningen vanligvis er båndbredden til klienten. Det er ingen spesielle krav til servere eller nettverk og overføringen kan ta lengre eller kortere tid enn avspillingen av multimediadataene. Klienten kan så velge å lagre dataene til disk for senere avspilling, eller de kan spilles av under overføring. Avspilling samtidig som dataene overføres kalles ofte HTTP-streaming, men fungerer som standard nedlasting, og hvis bitraten til multimediadataene er høyere enn båndbredden til klienten, vil HTTP-streaming føre til pauser i avspillingen. For å unngå pauser ved avspilling overføres en del av dataene før avspillingen starter, men det er likevel ikke garantert at dataene overføres raskt nok.

Et alternativ til nedlasting er streaming, hvor multimediadataene overføres i avspillingshastighet. Streamingservere kan brukes til live-streaming hvor brukerne ser innholdet på samme tid, eller til streaming av lagret materiale som audio og video når brukeren etterspør det. Streamingservere tilbyr vanligvis lagret innhold, samt at de tilbyr brukerne en utvidet mulighet til å pause eller hoppe frem og tilbake i videoen, likt det man kan gjøre med en dvd- eller videospiller.

Den mest basale egenskapen til en streamingserver er at brukeren mottar media i sanntid. Det betyr at video eller audio blir spilt i normal hastighet, og at den kan spilles rent uten pauser eller avbrudd fordi den blir lastet ned i en akseptabel rask hastighet. En video på to timer tar to timer å overføre fra serveren. Streaming har fordeler fremfor nedlasting, ettersom det ikke krever store buffere hos klienten og det er mulig å se innholdet med en gang. Man slipper å vente på at hele filen skal lastes ned. Ulempen er at innholdet må overføres hver gang det skal spilles av, noe som øker belastningen på server og nettverk. Dette kan være aktuelt for Audio-on-Demand-tjenester, men er ikke like aktuelt for Video-on-Demand-tjenester. I tillegg stiller streaming høyere krav til server og nettverk når det gjelder tidsforsinkelse. Ved bruk av Realtime Transport Protocol, som er den mest brukte protokollen for streaming av tidsavhengig data som video og audio fra en streamingserver til klienter, kreveres det vanligvis at det legges til headere med timing- og sekvensnummerinformasjon til hver pakke. Ved streaming av media fra en streamingserver til en klient opprettes det også en kontrollforbindelse. Denne brukes for utveksling av statistikk mellom server og klient for å overvåke streamingen. Hvis det for eksempel er stort pakketap, kan klient og server bli

enige om å redusere bitraten.

Multimediadata har høye krav til båndbredde og tidsforsinkelse. På servere er det derfor en stor skaleringsutfordring å støtte tusenvis av samtidige klienter som etterspør tidsavhengig data som audio og video. Den mest kostbare operasjonen er overføring av data fra disk til nettverk gjennom serverens I/O datasti. Datastien omfatter flere forskjellige subsystemer som befinner seg i forskjellige adresserom. Dataene må derfor kopieres mellom de forskjellige subsystemene for at de skal kunne leses eller manipuleres. Dataene overføres fra disk til filsystemets buffercache, fra buffercache til serverens applikasjonsminneområde for oppbygging av applikasjonlagsheadere, fra applikasjonen til kommunikasjonssystemets buffer for oppbygging av nettverksheadere og videre fra kommunikasjonssystemet til nettverkskortet. Ettersom subsystemene befinner seg i forskjellige adresserom innebærer overføring av data fra disk til nettverk også kontekstbytter. På grunn av at minnekopiering og kontekstbytter er ressurs- og tidskrevende, kan servere som flytter store datamengder gjennom flere subsystemer oppleve høy belastning. Ved å benytte teknikker som delte minneområder, sideremapping eller en kombinasjon av disse, er det mulig å fjerne kopiering av data mellom subsystemene ved overføring fra disk til nettverk, og dermed redusere systembelastningen. En mulighet for reduksjon av kontekstbytter er å flytte oppbygging av applikasjonlagsheadere til samme adresserom som filsystem og kommunikasjonssystem. Ved å kombinere dette vil det dermed være mulig å overføre multimediadata fra disk til nettverk uten minnekopiering og kontekstbytter.

1.2 Problemstilling

I de siste 15 årene har dataflyttingsoverhead vært et viktig område i operativsystemforskning. Mange mekanismer har blitt foreslått og implementert for å redusere dataflyttingsoverheaden med virtuell minneremapping og delt minne som basisteknikker. Dagens operativsystemer inkluderer datastioptimaliseringer for vanlige applikasjoner, som for eksempel webserverfunksjoner. Men de legger ikke til eksplisitt støtte for mediastreaming, og som en konsekvens lager mange streamingtilbydere egne implementasjoner. En annen kostbar operasjon, ved siden av dataflytting, er kontekstbytter fra applikasjonsmodus til kjernemodus og tilbake.

I denne oppgaven har vi sett på Linux 2.6 kjernen, for å avgjøre om nyere hardware og vanlige operativsystemer som Linux har klart å overkomme problemene med dataoverføringsoverhead, og hvor nært en mer optimalisert datasti de eksisterende løsningene er. I den sammenheng har vi sett på dataflyttingsproblemet og evaluert forskjellige mekanismer som bruker forskjellige datastier i Linux 2.6 kjernen.

Vi undersøkte også i hvilken utstrekning eksisterende løsninger er tilstrekkelig i streamingsammenheng, og om dedikert støtte for streamingapplikasjoner kan gi forbedringer i ytelsen. For testing av dette har vi implementert nye systemkall med støtte for streaming og implementert en Real-time Transport Protocol-motor (RTP-motor) i kjernen, for å se om dette kan gi forbedringer i ytelsen.

Ved oppbygging av applikasjonslagsheadere, er det for noen typer media nødvendig med databerøringsoperasjoner for å hente ut informasjon om dataene som overføres, mens for andre er det tilstrekkelig med informasjon som gjelder for hele mediainnholdet og som vanligvis ikke er en del av filen som overføres. I eksperimentene skiller vi derfor mellom implementasjoner for media som krever databerøringsoperasjoner og de som ikke krever det. Mer komplekse berøringsoperasjoner som kryptering og vannmerking er ikke behandlet i oppgaven.

For media som krever databerøringsoperasjoner har vi klart å redusere CPU-bruken med omtrent 24 prosent ved fjerning av kopioperasjoner, mens for media som ikke krever databerøringsoperasjoner har vi klart å redusere CPU-bruken med 10 prosent ved redusering av kontekstbytter. RTP-motorene ga en ytterligere forbedring på 16 prosent for media som krever databerøringsoperasjoner og 19 prosent for media uten databerøringsoperasjoner. Sammenlignet med beste eksisterende løsninger gir RTP-motorene forbedringer på henholdsvis 36 prosent med databerøringsoperasjoner og 27 prosent uten databerøringsoperasjoner.

1.3 Struktur

Resten av oppgaven beskriver design, implementasjon og evaluering av nye og eksisterende løsninger for multimediastreaming og er organisert som følgende:

Kapittel 2 beskriver mediadata sine krav til båndbredde og forsinkelse, forskjellige protokoller for streamingservere, og mediakomprimeringsformater som er aktuelle for dette prosjektet. Til slutt i kapitlet beskrives en tradisjonell disk-nettverksdatasti og forskjellige dataoverføringsmekanismer.

Kapittel 3 beskriver foreslåtte mekanismer for fjerning av kopioperasjoner og redusering av kontekstbytter samt implementasjoner i operativsystemer som er mye brukt i dag.

Kapittel 4 gir en oversikt over eksisterende mekanismer for nedlasting og streaming i Linux. Vi ser på bufferhåndtering og systemkall for overføring av data fra disk til nettverk og presenterer testresultater for disse systemkallene. Det presenteres også testresultater med sammenligning av ulike filsystemer og en sammenligning av Linux-kjerne 2.4 og 2.6.

Kapittel 5 beskriver streamingutvidelser for Linux. Først presenteres implementasjonene av nye systemkall optimalisert for streaming og to prototyper for RTP-motorer i kjernen. Videre presenteres testresultater for de nye systemkallene og prototypene og en sammenligning med de eksisterende systemkallene. Vi ser deretter på implementasjonen av RTP-motorene og resultatene for disse.

Kapittel 6 konkluderer denne oppgaven.

Kapittel 2

Multimediasystemer

I dette kapitlet skal vi se på bakgrunnsinformasjon for multimediasystemer. Dette er et veldig stort område, men vi har fokusert på informasjon som er viktig for vårt videre arbeid i oppgaven. Vi starter med en beskrivelse av krav til mediadata i seksjon 2.1, seksjon 2.2 beskriver protokoller som benyttes ved mediastreaming, seksjon 2.3 gir en liten oversikt over MPEG, og i seksjon 2.4 ser vi på en tradisjonell disk-nettverksdatasti og mekanismer for dataoverføring.

2.1 Mediadata

Mediadata stiller store krav til servere og nettverk på grunn av høye bitrater og sanntids leveringskrav. Som vi ser i tabell 2.1 er bitratene veldig høye og komprimering er nødvendig for å oppnå håndterlige størrelser. En time ukomprimert High-definition television (HDTV) krever 460GB lagringsplass, men ved bruk av en komprimeringskodek som Moving Pictures Experts Group (MPEG) kan dette komprimeres til 2.5-7GB. Komprimering reduserer krav til lagringsplass og båndbreddekravene i server og kommunikasjonsnettverket, men øker kostnadene hos server og klient ved komprimering og dekomprimering, men på grunn av stor økning i prosessorytelse de siste årene er dekomprimering ikke lenger et problem for klienter. Båndbreddekravet til en HDTV video komprimert med MPEG2 er omtrent 15 Mbps og dette betyr at alle komponenter i datastien fra server til klienten må støtte dataraten til dataene som overføres.

I tillegg til store krav til båndbredde stiller mediadata krav til sanntidslevering. For eksempel har en video et visst antall bilder i sekundet som må leveres i presise intervaller for at videoen ikke skal bli hakkete. For HDTV med 50 bilder i sekundet må det leveres et bilde hvert 20 millisekund. Dette stiller store krav til servere

Audio	**bit per sample**	**KHZ**	**Ukomprimert**	**MPEG1/2 Layer 3**	**MPEG2 AAC**
CD	16	44.1	1.4Mbps	32-320 bit/s	70% av layer 3
Video	**pixels**	**fps**	**Ukomprimert**	**MPEG2**	**MPEG4**
TV	640x480	25	176 Mbps	3-6 Mbit/s	0.7-1.5 Mbit/s
DVD	720x576	25	237 Mbps	4-9 Mbit/s	1-2 Mbit/s
HDTV 720p	1280x720	50	1055 Mbps	12-16 Mbit/s	6-8 Mbit/s
HDTV 1080p	1920x1080	50	2373 Mbps	24-30 Mbit/s	12-15 Mbit/s

Tabell 2.1: Båndbreddekrav for audio og video [2][4][3][29][7]

og kommunikasjonsnettverk for at forsinkelse og variasjonen i forsinkelse (jitter) ikke skal bli for stor. Stor variasjon i forsinkelse fører til stort pakketap fordi dataene ankommer etter at de skal presenteres og dermed må kastes, mens en stor forsinkelse uten stor variasjon kan være akseptabel for en del applikasjoner, som for eksempel Video-on-Demand. For å fjerne effekter av jitter kan det benyttes et buffer hos mottaker, men dette øker forsinkelsen. For IP telefoni og videokonferanse er det akseptabelt med en forsinkelse på inntill 150ms for tale [40], mens en forskyving på inntill +/-80 ms er akseptabelt ved leppesynkronisering [41]. For videostreaming kan det benyttes et større buffer og dermed øke forsinkelsen, og noen sekunder forsinkelse kan være akseptabelt.

2.2 Protokoller for multimediastreaming

En streamingserver bruker protokoller for beskrivelse, opprettelse og kontroll av multimediasesjoner, i tillegg til protokoller for overføring av multimediadata og monitorering av dataoverføringen. I denne seksjonen beskrives protokoller som kan brukes til disse formålene. Den første protokollen er Session Description Protocol (SDP) som brukes for beskrivelse av multimediasesjoner. Videre beskrives Real Time Streaming Protocol (RTSP) som brukes ved opprettelse og kontroll av multimediasesjoner, og Realtime Transport Protocol (RTP) og Realtime Control Protocol (RTCP) som er protokoller for overføring og monitorering av dataoverføringen.

2.2.1 Session Description Protocol (SDP)

SDP er ment for beskrivelse av multimediasesjoner for bruk til sesjonsannonsering, sesjonsinvitasjoner, og andre former for multimediasesjonsinitiering [24].

Formålet med SDP er å formidle nok informasjon om mediastrømmer i multimediasesjoner til mottakere slik at de er i stand til å delta i sesjonene. En multimediasesjon, for dette formålet, er definert som et sett av mediastrømmer som eksisterer i en viss varighet i tid. Mediastrømmer kan være mange til mange, og tiden som sesjonen er aktiv trenger ikke være sammenhengende. SDP brukes blant annet sammen med RTSP for beskrivelse av presentasjoner eller sammen med Session Announcement Protocol (SAP) for annonsering av konferanser på multicastnettverk.

Brukt sammen med RTSP, er SDP-beskrivelsen lagret som en fil på multimediaserveren eller på en egen dedikert server, hvor det er en fil for hver lagret presentasjon eller strøm. SDP filen inneholder informasjonen som er nødvendig for at klienter skal være i stand til å etterspørre multimediapresentasjonene. For overføring av SDP-beskrivelser kan for eksempel HTTP GET eller RTSP DESCRIBE forespørsler sendes.

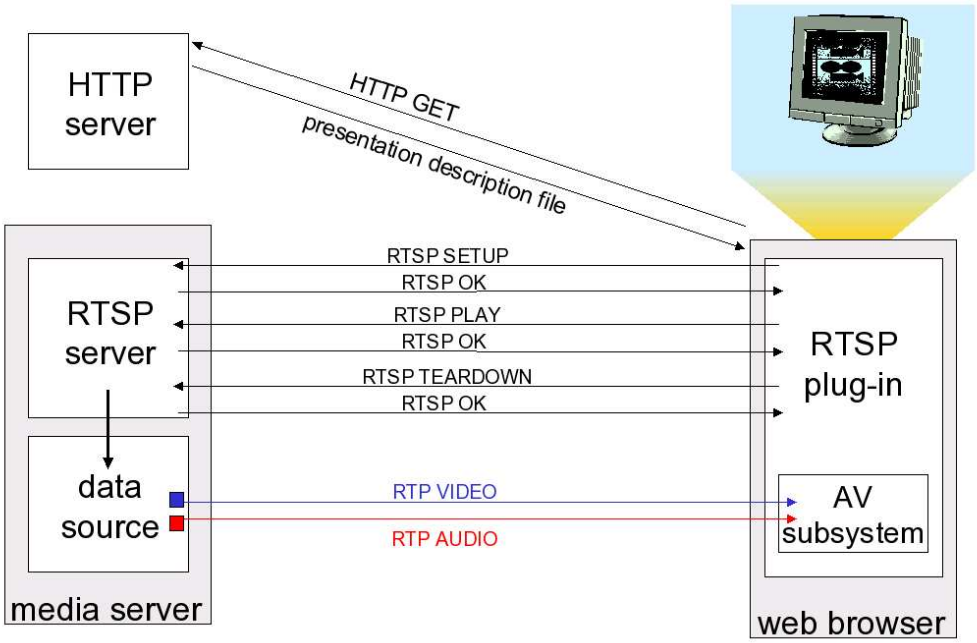
En SDP beskrivelse inkluderer sesjonsnavn og formål, tid(er) sesjonen er aktiv, media som omfattes av sesjonen, og informasjon som for eksempel adresse og portnummer for å motta media. Mediainformasjonen i SDP omfatter type media som for eksempel video eller audio, mediaformat som for eksempel MPEG video eller H.261, og protokoller for overføring som for eksempel RTP, UDP og IP.

2.2.2 Real Time Streaming Protocol (RTSP)

Real Time Streaming Protocol (RTSP) er en applikasjonslagprotokoll designet for å opprette og håndtere multimediasesjoner [27]. RTSP er en kontrollprotokoll og ikke en dataprotokoll. RTSP kan brukes over en hvilken som helst transportprotokoll, men det foretrukne vil være en pålitelig transportprotokoll som TCP, ettersom protokollen ikke håndterer resending av tapte meldinger.

RTSP kan brukes av klienten for å få beskrivelse om multimediadata på streamingservere, og for å opprette, starte, kontrollere og avslutte en multimediasesjon (se figur 2.1). Ved opprettelse av en sesjon tilbyr RTSP en måte å velge leveringskanaler som for eksempel UDP, multicast UDP eller TCP, portnummer, og leveringsmekanismer basert på RTP. Etter at en sesjon mellom klient og server er opprettet fungerer RTSP som en “fjernkontroll”, hvor klienten har mulighet til å for eksempel starte, stoppe, pause eller hoppe frem og tilbake i mediastrømmen.

Klient og server kommuniserer ved hjelp kontrollmeldinger, hvor de viktigste kontrollmeldingene er SETUP, PLAY OG TEARDOWN. SETUP-meldingen brukes for å opprette en multimediasesjon. Ved opprettelse av en ny sesjon sender klienten en SETUP-forespørsel til serveren. SETUP-forespørselen inneholder en



Figur 2.1: RTSP-multimediasesjon [18]

Uniform Resource Identifier (URI) til mediafilen på serveren, samt en en spesifikasjon av transportprotokoller og portnummer som klienten ønsker å benytte under streamingen av den aktuelle mediafilen. Serveren allokerer ressurser for sesjonen og lager en unik sesjonsid. Sesjonsid'en returneres til klienten i et SETUP svar og benyttes ved kommunikasjon mellom klient og server. Når en sesjon er opprettet kan streamingen av mediafilen starte.

Streamingen av mediafilen startes ved at klienten sender en PLAY-forespørsel. PLAY-forespørselen inneholder en URI til mediafilen og sesjonsid'en. I tillegg kan den inneholde en `range` som angir hvilken del av filen som skal streames. Hvis den ikke inneholder en `range` streames filen fra starten, og hvis den ikke blir avbrutt av for eksempel en TEARDOWN-forespørsel, streames den til slutten av filen. En PLAY-forespørsel kan også sendes etter en PAUSE-forespørsel. Det streames da fra pausepunktet eller fra angitt `range`. Hvis det sendes flere PLAY-forespørsler blir de lagt i en kø og spilt av i rekkefølge. Serveren vil sende tilbake et PLAY-svar med den aktuelle `rangen` som skal streames.

Når klienten ønsker å avslutte sesjonen sender den en TEARDOWN-forespørsel. Serveren vil da stoppe streamingen av den angitte URI'en og frigjøre ressursene assosiert med den.

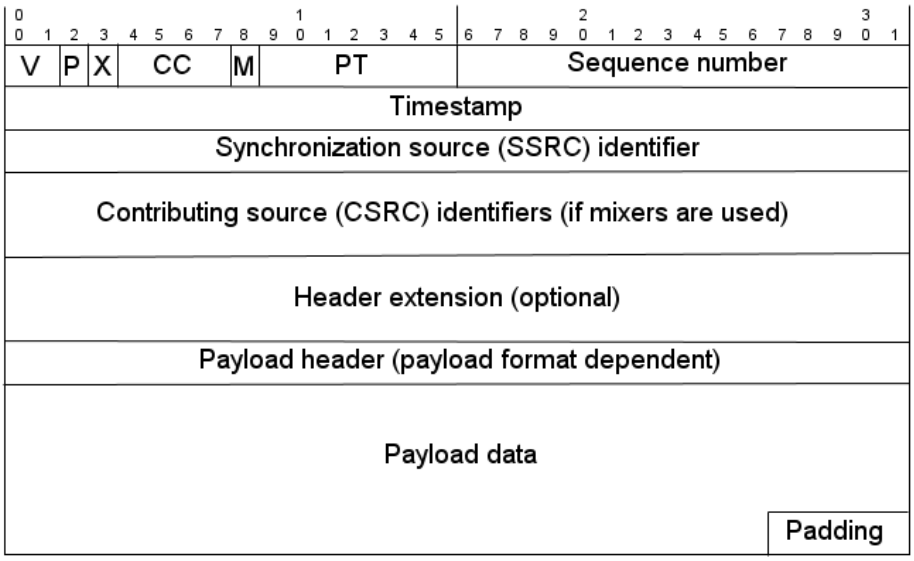
2.2.3 Realtime Transport Protocol (RTP)

Realtime Transport Protocol (RTP) tilbyr ende-til-ende nettverks transportfunksjonalitet for applikasjoner som overfører tidsavhengig data, som audio og video, over multicast eller unicast nettverkstjenester [23]. RTP er utviklet for robust levering av tidsavhengig data over et upålitelig transportlag som for eksempel UDP, og bygger på ideen med applikasjonsnivåframing [13] og ende-til-ende prinsippene. Ideen med applikasjonsnivåframing er at bare applikasjonen har tilstrekkelig kjennskap til dataene til å ta den begrunnede avgjørelsen om hvordan dataene skal transporteres. For at applikasjonen skal kunne ta disse avgjørelsene og håndtere feil, må transportlaget synliggjøre detaljer ved overføringen, og transportlaget må samarbeide med applikasjonen for å oppnå pålitelig overføring. Ende-til-ende prinsippene betyr at ansvaret for pålitelig overføring ligger hos endepunktene og pålitelig overføring kan oppnås over et upålitelig nettverk. Konsekvensen av ende-til-ende prinsippene er at intelligensen ligger i endepunktene og at nettverkene kan være enkle.

RTP består av to deler; en dataoverføringsprotokoll og en kontrollprotokoll. RTP dataoverføringsprotokollen håndterer leveringen av mediadataene mellom endesystemene, og tilbyr et ekstra nivå for innramming av mediadataene, med blant annet sekvensnummerering for å oppdage tap av pakker, tidsstempel for gjenoppretting av timing ved pakkeforsinkelse, mediatype, og avsenderidentifikator. Det er ingen flytkontroll, feilkontroll, acknowledgements, og ingen retransmissjonsmekanismer, men RTP brukes vanligvis på toppen av UDP som tilbyr multiplexing og checksumtjenester. Både RTP og UDP protokollene bidrar med deler til transportprotokollfunksjonaliteteten, men RTP kan også brukes sammen med andre egnede underliggende nettverk- eller transportprotokoller.

RTP kontrollprotokollen (RTCP) tilbyr monitorering av overføringen og gir tilbakemelding på kvaliteten. Den håndterer også deltakeridentifisering og tilbyr informasjon for synkronisering mellom mediastrømmer. RTCP sender periodiske rapporter med noen sekunders mellomrom, med blant annet informasjon om pakkene som er sendt eller mottatt siden siste rapport. Dette er mer hensiktsmessig enn tilbakemelding på hver pakke ved overføring av tidsavhengig data, ettersom resending av pakker vanligvis ikke er mulig. Rapportene brukes blant annet for å kunne justere bitraten på strømmen ved stort pakketap. Rapportene som sendes av RTCP er `Receiver Reports` (RR) med informasjon om blant annet pakketap og variasjon i forsinkelse, og `Sender Reports` (SR) med informasjon om blant annet antall sendte pakker, og tidsstempler for synkronisering av strømmene. `Receiver Reports` sendes fra mottaker til sender av dataene, mens `Sender Reports` sendes fra sender til mottaker. I tillegg til rapportene kan RTCP sende SDES, BYE og APP pakker. SDES pakker inneholder deltakeridentifikasjon og

informasjon, mens BYE pakker brukes når en deltaker har forlatt sesjonen eller har endret sin synkroniseringskildeidentifikator (SSRC). APP pakker brukes for applikasjonsdefinerte utvidelser, og er ment for eksperimentell bruk av nye utvidelser som kan resultere i nye registrerte pakketyper.



Figur 2.2: RTP-header

RTP-headeren illustrert i figur 2.2 består av 12 byte pluss eventuelle utvidelser, hvor de viktigste feltene er mediatype, sekvensnummer og tidsstempel. Disse brukes av applikasjonen for å bestemme mediaformatet, oppdage pakketap og feil i pakkerekkefølge, og for å vite når dataene skal presenteres. I tillegg kommer felter som versjon, padding, utvidelse (extension), bidragskildeteller (CC), marker, synkroniseringskilde (SSRC) og bidragskildeliste (CSRC).

Mediatype (PT) er et 7 bits felt med en identifikator som forteller applikasjonen hvilket format mediadataene er på, og hvordan den skal tolkes. Mediaformatet kan endres i løpet av en sesjon.

Sekvensnummeret er på 16 bit og økes med en for hver RTP-pakke som sendes, og kan brukes til å oppdage pakketap og gjennomrette pakkerekkefølgen. Dette er nødvendig ved bruk av RTP over en upålitelig transport protokoll, som for eksempel UDP, ettersom pakketap, forsinkelser og feil pakkerekkefølge er vanlig.

Tidsstemplet (32 bit) brukes av applikasjonen for å vite når dataene i RTP-pakken skal presenteres. Tidstemplet velges tilfeldig for første pakke og er relativ til starten på strømmen, slik at bare forskjeller mellom tidsstemplene har betydning. Inkrementeringen avhenger av mediatypen som streames. Brukes blant annet for å

tillate små buffere hos mottaker for fjerning av jittereffekt, og for at forskjellige strømmer skal kunne bli synkronisert med hverandre. For synkronisering må RTP tidsstemplene i strømmene relateres til en felles klokke, og dette gjøres ved hjelp av RTCP pakker som gir Network Time Protocol (NTP) og RTP tidsstempler.

Synkroniseringskildeidentifikatoren (SSRC) (32 bit) identifiserer deltakere i en sesjon og forteller hvilken strøm en pakke tilhører. Brukes for å multiplekse og demultiplekse flere datastrømmer i en enkelt strøm av UDP pakker.

Versjonsfeltet (V) identifiserer RTP versjonen og er på 2 bit. Nyeste versjon av RTP er 2. Paddingbittet (P) angir om pakken inneholder en eller flere ekstra byte på slutten som ikke er en del av mediadataene. Den siste byten i paddingen inneholder en teller over hvor mange byte som skal ignoreres. Utvidelsesbittet (X) settes hvis det følger en headerutvidelse etter den faste headeren. CSRC-telleren (CC) (4 bit) inneholder antall CSRC-identifikatorer som følger den faste headeren, brukes bare av RTP-proxyer (RTP translators [1] og RTP mixers [2]). Markerbittet (M) brukes for å angi spesielle hendelser i strømmen. Regler for bruk av markerbittet varierer med de forskjellige mediatypene.

2.2.4 Transportprotokoller UDP, TCP og SCTP

RTP brukes sammen med en annen transportprotokoll og aktuelle protokoller er for eksempel UDP[37], TCP[20] eller SCTP[35]. UDP er en upålitelig meldingsorientert transportprotokoll, som tilbyr multipleksing og checksumming, og er den foretrukne transportprotokollen til bruk sammen med RTP. Dataene blir levert til applikasjonen som meldinger. Meldinger kan forsvinne eller bli levert i feil rekkefølge, men RTP har støtte for å gjenopprette rekkefølgen, og tap av meldinger er ikke kritisk for multimediastreaming. UDP egner seg bra for overføring av tidsavhengig data med høye krav til forsinkelse ettersom det ikke er resending av tapte eller ødelagte pakker. UDP har også støtte for multicast, noe som kan være med på å redusere belastningen på server og overføringsnettverk. Dataene blir levert til applikasjonen som meldinger, noe som gjør det enkelt å lokalisere applikasjonsheadere.

TCP er en pålitelig byteorientert transportprotokoll hvor dataene leveres feilfrie og sekvensielt. Med TCP er dataene garantert å komme fram, men TCP er ikke veldig godt egnet til streaming av tidsavhengig data. Hovedgrunnen til å bruke TCP istedenfor UDP er at mange brannmurer slipper gjennom TCP forbindelser men blokkerer UDP. TCP leverer dataene som en bytestrøm til applikasjonen og

[1] RTP tranlaters er proxyer som brukes til bytte av protokoll for eksempel IP til ST-2 eller endring av media koding for eksempel redusering av bitrate

[2] RTP mixers er proxyer for multiplexing og resynkronisering av RTP-strømmer

applikasjonen må derfor søke gjennom dataene for å finne igjen applikasjonslagheadere. Ved pakketap blir dataene resendt og leveringen av data til applikasjonen blir forsinket, dette kan føre til at det blir pauser i avspillingen. Det er heller ikke støtte for multicast noe som kan føre til høyere belastning på server og overføringsnettverk.

Et nytt alternativ er Stream Control Transmission Protocol (SCTP), som har mange av karakteristikkene til TCP, men som tilbyr ytterligere støtte for streamingapplikasjoner. SCTP tilbyr i likhet med TCP en feilfri og sekvensiell datatransport og oppretter en ende-til-ende forbindelse med endepunktene under datatransporten. Til forskjell fra TCP tilbyr SCTP multistreaming, og leveringen av data til applikasjonen er meldingsorientert istedenfor byteorientert. Ved pakketap resendes pakkene slik at alle dataene kommer fram, men det er mulig å hoppe over den sekvensielle leveringen, slik at dataene leveres med en gang de ankommer. Dette gjør SCTP til en egnet tranportprotokoll for RTP-streaming, ettersom dataene blir levert som meldinger, og det er enkelt å lokalisere applikasjonsheadere. Det blir heller ikke forsinkelse av hele strømmen ved resending av pakker, og applikasjonen kan avgjøre om pakker som resendes ankommer for sent.

2.3 Moving Picture Expert Group (MPEG)

Moving Pictures Expert Group (MPEG) har utviklet standarder for komprimering av audio og video. I denne seksjonen skal vi se på strukturene til MPEG-1 og MPEG-2 strømmer og se hvordan de brukes i sammenheng med RTP-streaming.

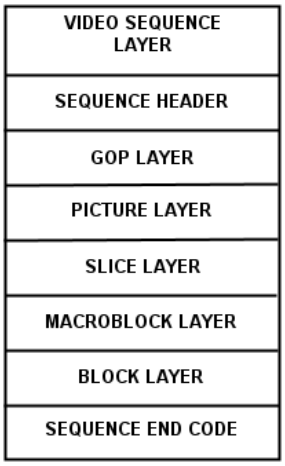
MPEG encodere brukes til komprimering av audio og video, og resultatet leveres i form av MPEG Elementary Streams (MPEG ES). Det brukes separate encodere for audio og video, og de produserer hver sin elementary stream. For at disse strømmene skal kunne synkroniseres ved avspilling, deler encoderne en felles klokke som brukes for å sette tidsstempler i strømmene.

Flere MPEG Elementary Streams kan kombineres til en MPEG Packetized Elementary Stream (MPEG PES). En MPEG PES består av pakker med størrelse opptil 64 KB, og hver pakke består av header og payload. Headeren forteller blant annet hva slags type data payloaden består av, for eksempel audio eller video, og hvilken elementary stream payloaden kommer fra. MPEG Elementary Stream'ene splittes opp for å lage PES pakker.

PES pakkene brukes også for å lage andre typer MPEG strømmer, egnet for lagring eller streaming over nettverk. For MPEG-1 er dette MPEG-1 System Stream (MPEG-1 SS), som i tillegg til PES pakkene består av ytterligere headere. For

MPEG-2 finnes det to typer, MPEG-2 Program Stream (MPEG-2 PS) og MPEG-2 Transport Stream (MPEG-2 TS). MPEG-2 PS består av en eller flere PES og brukes av applikasjoner hvor transmisjonsmediet innfører veldig få feil. Brukes blant annet på digital versatile disc (DVD). MPEG-2 TS er laget for å transportere audio og video over nettverk. Lengden på en MPEG-2 transport pakke er 188 byte, og består av en header på 4 byte og en payload på opptil 184 byte. Payloaden består av PES pakker, og ettersom de vanligvis er lengre enn 184 byte må de splittes.

For RTP-streaming av MPEG-2 PS og MPEG-2 TS benyttes standard RTP-header, og ingen mediaspesifikk header er nødvendig. RTP-tidsstemplet inkrementeres for hver pakke og bestemmes av bitraten til MPEG-strømmen. For MPEG-2 PS er det ingen regler for oppsplitting, og RTP-mediadataene vil vanligvis ha en fast lengde som er hensiktsmessig i forhold til bitrate og MTU for underliggende nettverk. En MPEG-2 TS må splittes opp mellom pakkene og RTP-mediadataene består derfor av *n* MPEG-2 transportpakker på 188 byte. MPEG-1 ES-video krever en mediaspesifikk RTP-header i tillegg til standard RTP-header. Før vi ser på oppbyggingen av denne headeren skal vi først se på strukturen til MPEG-1 ES-video.

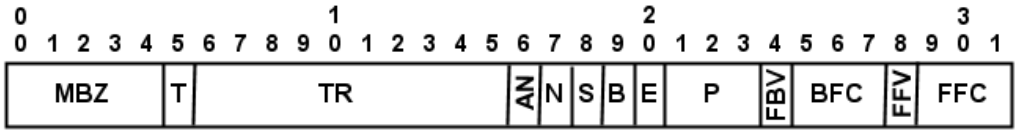


Figur 2.3: MPEG Elementary Stream

Figur 2.3 illustrerer strukturen på MPEG ES-video [22], hvor Groupe of Pictures (GOP)-, Picture- og Slice-lagene er de viktigste i en RTP-streamingsammenheng. En MPEG ES-video starter med en sequenceheader med informasjon som gjelder for hele videosekvensen, dette er blant annet startkode og videoparametere. Etter sequenceheaderen følger GOP-laget. GOP-laget inneholder en GOP-header, hvor de første 4 bytene består av GOP-startkode, og opptil 1024 etterfølgende Picture-lag.

Picture-laget starter med en Picture-header. De 4 første bytene i headeren består av en Picture-startkode, etterfulgt av en temporal referanse som viser visningsrekkefølgen for det aktuelle bildet. Videre følger kodingstypen for bildet som kan være av typen Intracoded (I), Predictive (P), Bidirectional (B) eller DC-coded (D). I bilder er kodet uten referanse til andre bilder, P krever informasjon fra det forrige I eller P bildet for encoding og decoding og B krever informasjon fra det forrige og etterfølgende I eller P bildet for encoding og decoding. D er intrakodet som I bilder, men består av bare de laveste frekvensene i bildet. D bildene brukes til "fast forward" eller "fast rewind" modus. Hvis det er P eller B bilder inneholder Picture-lag headeren "forward motion vector" og kode, og for B bilder "backward motion vector"[3] og kode.

Etter hver Picture-header følger Slice-laget med Slice-header. Slice-headeren starter i likhet med de andre headerne med 4 byte startkode. Slice-laget inneholder, i tillegg til headeren, Makroblock-lag som igjen inneholder Block-lag, men disse er uten betydning for oppbygging av RTP-headere. Informasjonen i MPEG ES som er viktig for oppbygging av RTP-headere er startkode for GOP-, Picture- og Slice-lagene, og Picture-headeren.



Figur 2.4: RTP-spesifikk MPEG ES header

For streaming av en MPEG Elementary Stream brukes standard RTP-header og i tillegg en RTP-mediaspesifikk header. De aktuelle feltene i RTP-headeren er versjonnummer (V), marker (M), mediatype (PT), sekvensnummer, tidsstempel og synkroniseringskildeidentifikator (SSRC). Markerbittet brukes når en pakke inneholder en sluttkode for et MPEG-bilde. Når det gjelder tidsstemplet inkrementeres den for hvert bilde og ikke for hver pakke. Mediatypen for MPEG ES Video er 32.

Figur 2.4 viser den RTP-spesifikke headeren som må brukes i tillegg til standard RTP-header ved streaming av MPEG ES. Feltene i denne headeren bestemmes av innholdet i de siste Sequence, GOP, Picture eller Slice headerne i strømmen. MBZ, T, AN og N brukes ikke. TR viser referansen til et bilde innenfor en GOP. Den hentes fra Picture-headeren og har verdi fra 0 til 1023. S indikerer at pakken inneholder en Sequence-header. B indikerer at starten på pakkens mediadata er en Slice-startkode, mens E indikerer at den siste byten i mediadataene er slutten på en

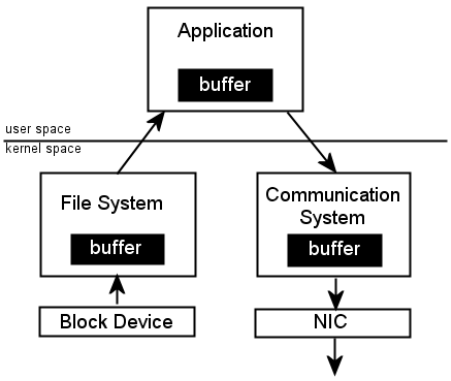
[3] "forward motion vector" og "backward motion vector" er bevegelseskompensasjonsvektorer.

Slice. P angir bildetype I , P, B eller D, mens FBV, BFC, FFV og FFC feltene gir bevegelseskompensasjonvektoren. FFV og FFC brukes for P og B bilder, mens FBV og BFC bare brukes for B-bilder. Vektoren og P-feltet hentes fra Picture-headeren. Ved fragmentering av et bilde i flere RTP-pakker er TR, P, FBV, BFC, FFV og FFC feltene de samme for hver pakke [12].

Siden MPEG-bilder kan bli store, vil de vanligvis bli fragmentert i pakker med størrelse mindre enn en typisk LAN/WAN MTU og RTP-pakker med MPEG ES-mediadata må derfor følge visse fragmenterings regler:

- Når en MPEG Video_Sequence_Header forekommer i RTP payloaden må den alltid komme i starten.
- Når en MPEG GOP_header forekommer i RTP-payloaden må den alltid komme i starten eller etterfølge en Video_Sequence_Header.
- Når en MPEG Picture_header forekommer i RTP-payloaden må den alltid komme i starten eller etterfølge en GOP_header.
- ES headere kan ikke fragmenteres.
- Begynnelsen på en slice må enten komme i starten av en pakke, etterfølge en MPEG ES-header, eller etterfølge et helt antall slicer.

2.4 Tradisjonell datasti

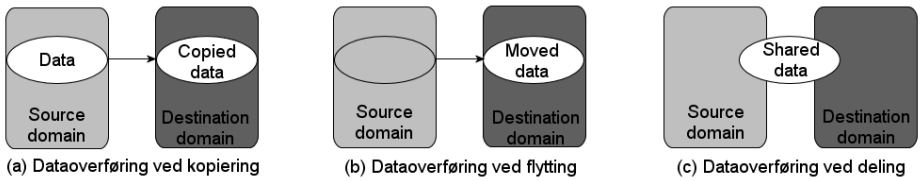


Figur 2.5: Disk-nettverksdatasti

Overføring av data mellom disk og nettverk i en multimediaserver følger vanligvis en tradisjonell datasti (figure 2.5) som inkluderer filsystem, applikasjon og kommunikasjonssystem. Hvert av disse subsystemene har sitt eget bufferhåndteringssystem som er implementert med forskjellig struktur for å oppfylle sitt subsystems behov. Disse bufferne er vanligvis ikke kompatible. Bufferstrukturen i filsystemet kan for eksempel ikke brukes av kommunikasjonssystemet, eller bufferne befinner seg i forskjellige beskyttelsesdomener som applikasjons- og kjerneadresserom. Dette innebærer at data må kopieres mellom de forskjellige bufferne for at subsystemene skal kunne manipulere dataene. Nedenfor skal vi se på overføring av data mellom subsystemer og mellom enheter og subsystemer.

2.4.1 Minne-til-minne dataoverføring

Overføringen av data mellom subsystemene kan være fysisk eller virtuell. Fysisk overføring involverer flytting av data i fysisk minne, slik at hver byte data fra kildedomenets fysiske minne flyttes til destinasjonsdomenets fysiske minne. Fysisk overføring er fleksibelt, men kostbart med hensyn på delte ressurser som prosessor, systembuss og minne. Data kan overføres fra og til enhver lokasjon, og dataoverføringsenhetene er byte hvor dataelementene kan være én byte eller større. Virtuell overføring involverer flytting av data i virtuelt minne, slik at de fysiske pagene med data mappes i destinasjonsdomenets adresserom under overføringen. Ved virtuell overføring er størrelsen på overføringsenhetene hele fysiske sider (pager). Dataene som overføres må ligge i en eller flere sider og sidene bør ikke inneholde andre data.



Figur 2.6: Modeller for dataoverføring

Ved fysisk og virtuell overføring av data er det tre overførings modeller. Disse er overføring ved kopiering, overføring ved flytting og overføring ved deling [34]. Overføring ved kopiering innebærer at data kopieres fra et domene til et annet domene. Orginalen befinner seg fortsatt i kildedomenet, mens en eksakt kopi befinner seg i destinasjonsdomenet (figur 2.6a). Dette er den mest vanlige modellen for fysisk overføring. Dataene blir kopiert fra kildedomenet til mottakerdomenet, og de har hver sin kopi av dataene, som kan manipuleres uten at det påvirker

dataene i det andre domenet. En virtuell kopiering mapper et område i destinasjonsdomenets adresserom til de fysiske sidene og påvirker ikke mappingen av disse sidene i kildedomenets adresserom. Hvis prosessene i begge domenene bare leser dataene vil de fysiske dataene være delt, men ved skriving vil det lages en egen fysisk kopi slik at endringer i et domene ikke påvirker dataene i et annet domene. Dette er det som kalles “copy-on-write”. Dette er mer kostbart enn en vanlig fysisk kopiering, ettersom dataene må kopieres fysisk og mappingen må endres.

Overføring ved flytting innebærer at dataene fjernes fra kildedomenet og plasseres i destinasjonsdomenet (figur 2.6b). Ved fysisk overføring innebærer dette at dataene blir kopiert fra kildedomenet til mottaker domenet, og deretter slettes fra kildedomenet. En virtuell flytting fjerner mappingen for dataene fra kildedomenets adresserom og mapper dataene i et område i destinasjonsdomentets adresserom. Dette innebærer i motsetning til virtuell kopiering ingen “copy-on-write”, og kan være en bedre løsning ettersom dataene ofte ikke brukes av kildedomenet.

Overføring ved deling innebærer at både kilde- og destinasjonsdomenet har tilgang til de samme dataene etter at de er overført. Etter at dataene er overført vil prosessene i både kilde- og destinasjonsdomenet ha tilgang til de samme dataene og alle endringer som gjøres av prosesser i et domene vil være synlig for prosesser i det andre domenet (figur 2.6c). Ved fysisk overføring innebærer dette at det er en kopi av datene i hvert domene, og en egen prosess synkroniserer dataene i disse bufferne. Ved virtuell deling mappes et område i både kilde og destinasjons domenets adresserom til de samme fysiske pagene. Dette er det samme som virtuell kopiering uten “copy-on-write”. Ulempen med denne modellen er at endringer av en prosess i et domene kan påvirke prosesser i et annet domene, noe som øker programmeringskompleksiteten og feil kan overføres mellom domener.

2.4.2 Enhet-til-minne dataoverføring

Dataoverføring langs disk-nettverksdatastien medfører at dataene må overføres fra disk til minne og fra minne til nettverkskortet. De mest brukte teknikkene for dette er direkte minneaksess (DMA) og programmert I/O (PIO).

Ved DMA overfører I/O adapteren data direkte fra og til minne uten å involvere prosessoren. Dataoverføringen kan skje samtidig som annen prosessoraktivitet så lenge dataene som er nødvendig for prosessoren ligger i cache. Hvis prosessoren er avhengig av minneaksess kan prosessoren måtte stoppe ved stor DMA trafikk. Dataene som overføres til og fra minnet ligger ofte i forskjellige minnelokasjoner, og for at dataene skal kunne overføres i en DMA operasjon må I/O adapteren støtte sprednings (scatter) og samle (gather) operasjoner.

2.5 Oppsummering

I dette kapittelet har vi sett på krav til mediadata og en tradisjonell datasti for overføring av data fra disk til nettverk, ved siden av protokoller for multimediastreaming og forskjellige typer MPEG strømmer. Mediadata har høye bitrater og dermed høye krav til båndbredde, noe som fører til stor belastning på blant annet systemets systembuss og prosessor ved overføring til nettverk. Ved streaming av mediadata kreves det applikasjonslagskontroll for timing og oppbygging av applikasjonslagsheadere som RTP-headere. Oppbygging av RTP-headere krever at det for noen typer media, som for eksempel MPEG ES, hentes ut informasjon fra mediafilen som overføres. Dette krever at det benyttes en datasti som gjør dataene tilgjengelig for applikasjonen slik at de kan leses. Ved streaming benyttes derfor vanligvis en tradisjonell datasti som innebærer at dataene overføres via filsystemets buffer, applikasjonens buffer, og kommunikasjonssystemets buffer. Dette fører til mange kopioperasjoner og kontekstbytter og ettersom mediadata har høye bitrater gir dette stor belasting på systemkomponenter som systembuss og prosessor.

I neste kapittel skal vi se på mekanismer som er foreslått for å optimalisere overføringen av data fra disk til nettverk, og hvilke mekanismer som er implementert på operativsystemer som er mye brukt i dag.

Kapittel 3

Relatert arbeid

I dette kapitlet skal vi se på relatert arbeid for optimalisering av disk-nettverks-datastien for nedlasting- og streamingapplikasjoner. I seksjon 3.1 ser vi på foreslåtte mekanismer og forskningsprototyper, i seksjon 3.2 ser vi på implementasjoner i operativsystemer som er mye brukt i dag, og i seksjon 3.3 kommer en oppsummering.

3.1 Foreslåtte mekanismer og forskningsprototyper

Fjerning av kopioperasjoner for å øke I/O ytelsen ved overføring av data mellom domenener har vært et tema i operativsystemforskning i mange år. Flere varianter med virtuell remapping og delt minne har blitt implementert i forskjellige operativsystemer, og allerede i 1972 ble virtuell kopiering implementert i Tenex systemet [9].

Ved bruk av virtuelt minne kan virtuelle adresser i forskjellige adresserom referere til samme fysiske side, og en interprosess dataoverføring utføres ved å fjerne mappingen til minneområdet i det virtuelle adresserommet til den ene prosessen og remappe minneområdet i den andre prosessen. Basert på ideene om sideremapping, datadeling eller en kombinasjon er det foreslått flere generelle mekanismer for støtte av “zerocopy” datastier mellom disk og nettverk. Dette inkluderer Container Shipping [34], IO-Lite [32] og UVM virtual memory system [15].

I tillegg til mekanismer for fjerning av kopioperasjoner i alle former for I/O er det designet mekanismer for å lage en rask datasti innenfor kjernen for overføring av data fra enhet til enhet, for eksempel disk-til-nettverk datastien. Disse mekanismene overfører ikke data mellom applikasjonens adresserom og kjernens adresserom, men beholder dataene innenfor kjernen og mapper dem mellom

forskjellige kjernesubsystemer. Dette er egnet for applikasjoner som ikke manipulerer data på noen måte og ingen databerøringsoperasjoner er utført av applikasjonen. Eksempler på slike mekanismer er systemkallene `sendfile` på UNIX og Linux, `TransmitFile` på Windows og `stream` i Roadrunner operativsystemet. Unix, Linux og Windows har forskjellige buffere for filsystem og kommunikasjonssystem slik at ved overføring av data fra disk til nettverk kopieres dataene til filsystemets buffer og referansen til disse dataene overføres til kommunikasjonssystemets buffer. I Roadrunner operativsystemet er det et felles buffer for filsystem og kommunikasjonssystem og det er dermed ikke nødvendig med referanseoverføring mellom forskjellige buffere ved bruk av `stream` systemkallet.

En annen kostbar operasjon, i tillegg til kopioperasjoner, er kontekstbytter i forbindelse med systemkall hvor hvert kall til kjernen medfører to kontekstbytter. Selv om datastier innenfor kjernen fjerner noe av denne overheaden, ved at overføring fra disk til nettverk bare krever et systemkall istedenfor to eller flere ved en tradisjonell datasti, krever mange applikasjoner applikasjonsnivåkode som utfører kjernekall. For eksempel vil en streamingapplikasjon utføre et systemkall for hver pakke som sendes på grunn av at streaming krever applikasjonskontroll og hele filen kan ikke overføres i et kall. Relevante teknikker for å redusere antall kontekstbytter og øke ytelsen inkluderer `batched systemkall` [14] og `event batching` [36]. En annen teknikk er å flytte applikasjonsnivåkode som utfører kjernekall ned i kjernen. Et eksempel på dette er `Stream Engine`[28] som flytter oppbyggingen av RTP-headere fra applikasjonen til kjernen for å unngå kontekstbytter for hver pakke som sendes. `Stream Engine` er en RTP-motor implementert for Linux 2.4 kjernen og brukes i Cisco's Content Engine. Den fjerner kopioperasjoner og reduserer kontekstbytter for streamingapplikasjoner. `Stream Engine` bruker en en-strøm-per-prosess programmeringsmodell, og for hver strøm forkes en ny prosess som utfører et systemkall som starter streamingen. Systemkallet returner når streamingoperasjonen er ferdig, eller når en melding fra klienten kommer på kontrollforbindelsen. RTP-motoren bygger RTP-headere og sender ut blokker med data med en fast forsinkelse. Overføringen av data fra disk til nettverk er basert på Linux sitt `sendfile` systemkall. Fjerningen av kopioperasjonene og kontekstbyttene førte til en forbedring på nesten 60 prosent sammenlignet med en implementasjon på applikasjonsnivå basert på `read` og `send`.

Et annen eksempel på kontekstbyttereduksjon er `Kstreams`[26] som er implementert på Linux 2.4 kjernen. `Kstreams` implementerer en datasti innenfor kjernen, men hvor applikasjonen har mulighet til å sette inn en streamhåndterer for applikasjonsspesifikk datamanipulering mens dataene blir overført fra disk til nettverk. Dataene overføres fra disk til nettverk uten kontekstbytter og manipuleringen som vanligvis utføres i applikasjonen utføres i kjernen. Applikasjonen har også mulighet til å fjerne eller bytte til en annen streamhåndterer under overføringen av

dataene. `Kstreams` viste en stor ytelsesforbedring sammenlignet med en implementasjon på applikasjonsnivå..

Disse eksemplene viser at det er lagt ned mye arbeid for å fjerne kopioperasjoner og kontekstbytter, men mange av de foreslåtte løsningene har forblitt prototyper. Dette skyldes blant annet at de er implementert i egne operativsystemer som aldri har fått stor utbredelse, eller at det bare har vært begrensede implementasjoner for testing som ikke er integrert i kildetreet. I neste seksjon presenteres implementasjonene i operativsystemene som er mest brukt i dag.

3.2 Eksisterende mekanismer på Solaris, Windows, FreeBSD og Linux

Vi skal nå se på eksisterende løsninger som er implementert på operativsystemer som har stor utbredelse i dag. Alle operativsystemene har implementert systemkallene `read`, `write`, og `mmap`, og forskjellige varianter av disse som for eksempel `send` og `writev`, for overføring av data fra disk til nettverk via applikasjonslaget. Alle disse bruker en eller flere kopioperasjoner ved overføring av data. I tillegg har alle operativsystemene støtte for overføring av data fra disk til nettverk innenfor kjernen uten kopioperasjoner. Implementasjonene på Solaris, FreeBSD, Windows og Linux er presentert nedenfor:

- `Solaris` har et systemkall kalt `sendfilev` [42] som bruker en datasti innenfor kjernen for overføring av data fra disk til nettverk, eller fra disk til disk uten kopioperasjoner. `sendfilev` muliggjør også overføring av applikasjonsbuffere slik at for eksempel headere generert av applikasjonen kan kombineres med data fra disk og overføres i et systemkall. Data fra applikasjonsbuffere kopieres til kjernen og kombineres med data fra disk i en DMA samleoperasjon. Det er ikke mulig med datberøringsoperasjoner på dataene som overføres fra disk til nettverk ettersom det benyttes en datasti innenfor kjernen.

- `FreeBSD` har implementert systemkallet `sendfile` [17] for overføring av data fra disk til nettverk uten kopioperasjoner. `sendfile` kan overføre hele eller deler av en fil i et systemkall, og det er mulig å legge til applikasjonsbuffere før og etter filen i det samme systemkallet. Overføringen bruker en datasti innenfor kjernen. I likhet med løsningen på `Solaris` må applikasjonsbuffere kopieres til kjernen og kombineres med data fra disk i en DMA samleoperasjon.

- `Windows` har implementert systemkallene `TransmitFile` og `TransmitPacket` [30] for overføring av data fra disk til nettverk uten kopioperasjoner. `TransmitFile` bruker en datasti innenfor kjernen, hvor en hel fil kan overføres i et systemkall. Det er mulig å angi størrelsen på dataelementene i hver pakke, og for overføring av store filer er det mulig å starte en systemtråd som overfører filen etter at systemkallet har returnert. Det er også mulig å overføre et applikasjonsbuffer før og etter fildataene i et systemkall. Dette fungerer som `sendfile` på `FreeBSD`. `TransmitPacket` er i likhet med `sendfilev` for Solaris et vektorbasert systemkall hvor data fra forskjellige buffere og filer på disk, kan kombineres i én pakke. Flere etterfølgende buffere kan kombineres i en pakke og flere pakker kan sendes i et systemkall.

- `Linux` har implementert systemkallet `sendfile` [1] for overføring av data fra disk til nettverk uten kopioperasjoner. `sendfile` bruker en datasti innenfor kjernen og en hel fil eller en del av en fil kan overføres i et systemkall. Det er ikke mulig å legge til applikasjonsbuffere i `sendfile`, og eventuelle headere til dataene må overføres med for eksempel `send`.

Disse systemkallene gir økt ytelse på for eksempel webservere. En webserver kan konstruere og sende en HTTP-respons i et systemkall, med data fra disk og applikasjonsbuffere. Men disse systemkallene kan ikke benyttes av alle streamingapplikasjoner, ettersom det ikke er mulig med databerøringsoperasjoner på dataene som overføres fra disk til nettverk. For noen av de vektorbaserte systemkallene, blant annet `TransmitFile` i windows, kan flere pakker sendes i et systemkall, men det er ikke mulig å bestemme hvor lang tid det skal være mellom hver pakke, noe som er en forutsetning for streaming av multimedia. For streamingapplikasjoner vil derfor systemkall som `read` og `write` eller `mmap` og `writev` være aktuelle, men ulempen med disse, sammenlignet med en `sendfile` løsning, er at dataene kopieres flere ganger ved overføring fra disk til nettverk.

3.3 Oppsummering

I dette kapitlet har vi sett at det er lagt ned mye arbeid i fjerning av kopioperasjoner og redusering av kontekstbytter ved overføring av data fra disk til nettverk i nedlasting- og streamingsammenheng. Til tross for dette er bare en begrenset støtte implementert i de mest brukte operativsystemene i dag. Dette gjelder spesielt for streamingapplikasjoner som i mange tilfeller krever databerøringsoperasjoner på applikasjonsnivå, og dermed ikke kan benytte seg av de optimaliserte løsningene uten kopioperasjoner. I neste kapittel skal vi se nærmere på de eksisterende

løsningene i Linux 2.6 kjernen for nedlasting og streaming, og evaluere ytelsen til I/O pipelinen.

Kapittel 4

Eksisterende mekanismer i Linux

I dette kapittelet skal vi se på eksisterende mekanismer i Linux for nedlasting og streaming. I seksjon 4.1 ser vi på bufferhåndtering i subsystemene som er involvert i disk-nettverksdatastien. Videre i seksjon 4.2 beskrives de forskjellige systemkallene som er aktuelle for overføring av data fra disk til nettverk. I seksjon 4.3 beskrives eksperiment omgivelsene og resultatene fra eksperimentene presenter og beskrives. Til slutt kommer en oppsummering av resultatene i seksjon 4.4.

4.1 Bufferhåndtering

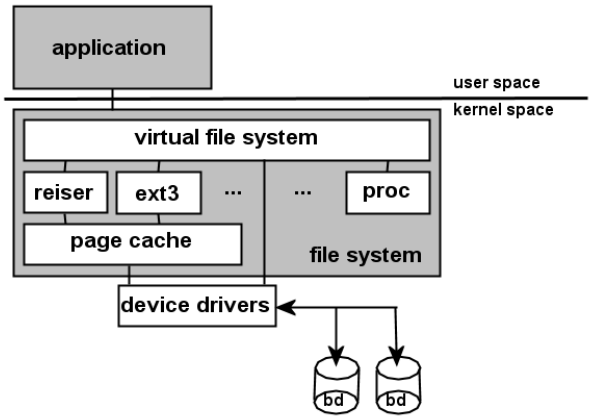
Dataoverføring fra disk til nettverk følger vanligvis en tradisjonell datasti som inkluderer filsystem, applikasjon og kommunikasjonssystem. Hvert av disse subsystemene har egne bufferhåndteringssystem tilpasset deres behov. I denne seksjonen skal vi se på hvilke bufferhåndteringssystemer som brukes i de forskjellige subsystemene.

4.1.1 Bufferhåndtering på applikasjonsnivå

Applikasjonene må tilby sine egne buffer- og cachingmekanismer, og det er ingen støtte for dette på applikasjonsnivå i Linux-systemer. I/O data blir vanligvis kopiert mellom operativsystemet og applikasjonens buffer ved en `read` eller `write` operasjon.

4.1.2 Bufferhåndtering i filsystemet

Filsystemet har flere forskjellige typer buffer, som `page cache`, `buffer cache` og `direct access buffer`. `page cache` brukes for caching av fildata, `buffer cache` for caching av superblock og inoder, og `direct access buffer` brukes ved direct I/O for applikasjoner med egen caching funksjonalitet, som for eksempel databaser.

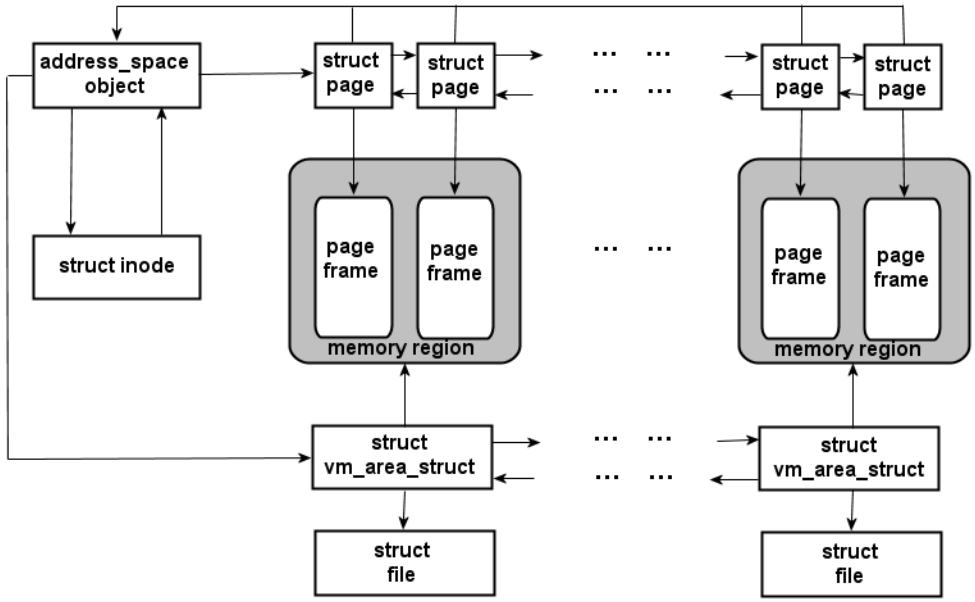


Figur 4.1: Linux filsystem

`page cache` er en diskcache bestående av sider (page frames). Den brukes for å redusere antall disk aksesser og dermed redusere tiden prosessen bruker på å vente på I/O. Hver side i `page cache` korresponderer til flere blokker i en blokkenhetsfil.

Blokkenheter, for eksempel harddisk, aksesseres via filsystemets grensesnitt, og består av, som vist på figur 4.1, det virtuelle filsystemet, lokale filsystemer som for eksempel `ext3` og `reiser` og enhetsdrivere. Ved en data forespørsel på en blokkenhet utføres først et oppslag i `page cache` for å se om dataene allerede finnes der. Bare når dataene ikke finnes i `page cache` startes en I/O operasjon som fører til diskaksess. Dataene kopieres først fra disk til `page cache` og videre til applikasjonsbufferet. Hvis dataene etterspørres på et senere tidspunkt kan de hentes direkte fra `page cache`. I et typisk UNIX system blir over 85% av disk forespørslene servert fra cachen [38].

Hvor lenge dataene blir liggende i `page cache` avhenger av tilgjengelig minne, men de blir ikke fjernet før de er aksessert. Når en side er aksessert blir den merket, slik at den kan overskrives eller skrives til disk når mengden ledig minne kommer under et visst nivå.

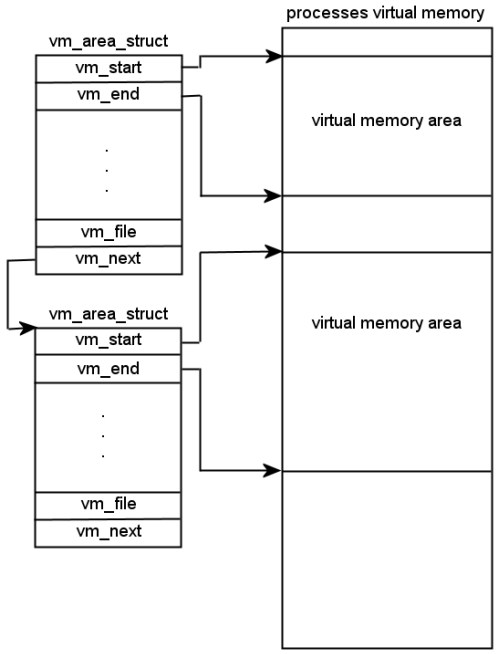


Figur 4.2: page cache

Cachingen som tilbys av filsystemets page cache redusere antall diskaksesser for de fleste applikasjoner, men mediadata som audio og video i MoD-servere drar ikke store fordeler av denne cachingen [11]. Ettersom bufferplassen er begrenset blir multimediadataene ofte overskrevet før de kan gjenbrukes. Bare prosesser som leser de samme dataene innenfor et kort tidsintervall kan bruke de cachede dataene.

I tillegg til å cache data som etterspørres og overføres fra disk tilbyr filsystemets page cache en readahead-mekanisme. read-ahead brukes når en fil leses sekvensielt, og funger slik at det overføres flere etterfølgende sider med data før de faktisk blir etterspurt. På denne måten vil dataene ligge i page cache når de etterspørres og bare enkelte ganger er det nødvendig å vente på at dataene skal overføres fra disk til page cache. Dette kan øke diskytelsen merkbart, ettersom diskkontrolleren må håndtere færre men større forespørsler, og disklesehodet flyttes færre ganger. For MoD-servere fører read-ahead til at tidsfristen for når dataene skal sendes overholdes oftere [33].

Figur 4.2 viser et utdrag av page cache hvor to områder av en fil er overført til page cache. Dataene i page cache er sider med data som blir identifisert av address_space-objekter med et objekt for hver fil. Sidene som tilhører en fil er referert av en liste med sideobjekter (instanser av struct page) som igjen

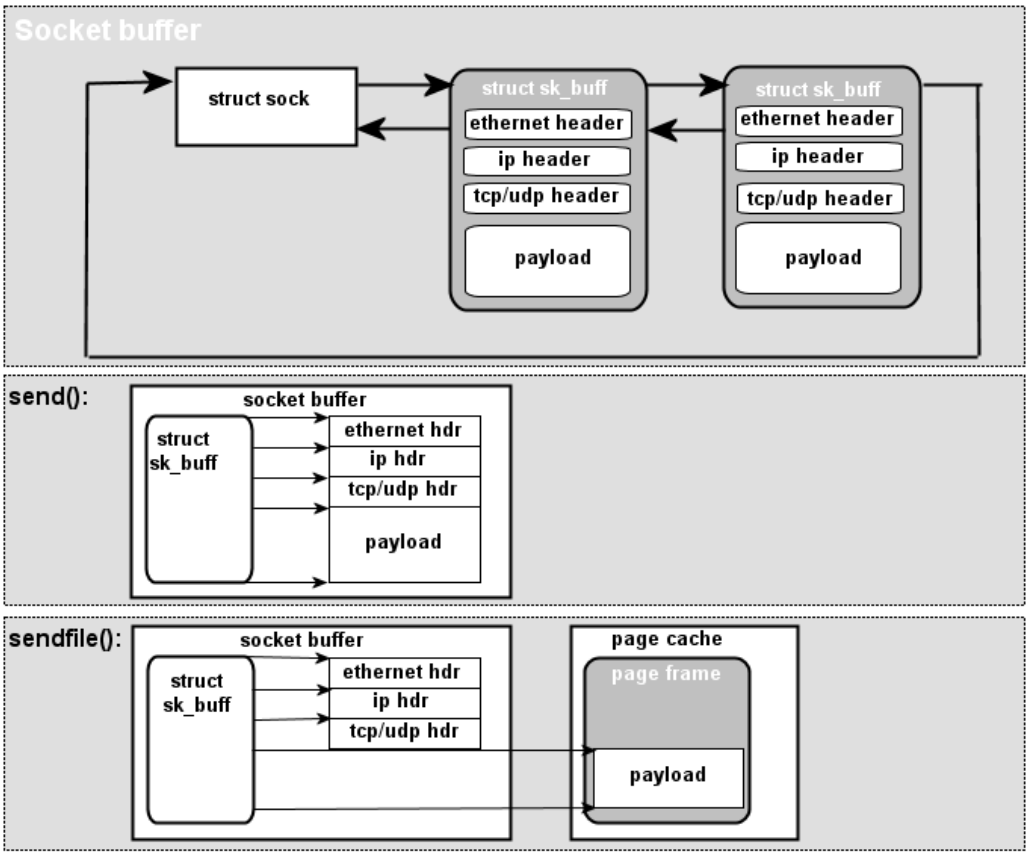


Figur 4.3: virtual memory area

er refert av et `address_space`-objekt. `address_space`-objektet referer også en liste med `vm_area_struct`-objekter som referer til et virtuelt minneområde som mapper en del av eller en hel fil. `vm_area_struct`-objektet brukes når en fil er mappet med for eksempel `mmap`. Når en applikasjon prøver å lese data i et virtuelt minneområdet som ikke finnes i `page cache` oppstår en "page fault". `vm_area_struct`-objektet brukes for å finne `file`-objektet og `address_space`-objektet. En ny side allokeres og sideobjektet som refererer til denne settes inn i `address_space` objektets liste med sideobjekter. Dataene overføres deretter fra disk til `page cache`, før sidetabellene til prosessen oppdateres. Figur 4.3 viser sammenhengen mellom `vm_area_struct`-objektetet og prosessens virtuelle minne.

Når en applikasjon starter et `read` systemkall for overføring av data fra disk til applikasjonsbufferet brukes fildeskriptoren for å finne `inode`-objektet og det tilhørende `address_space`-objektet. Deretter traverseres `address_space`-objektets sideliste for å sjekke som dataene allerede finnes i `page cache`. Hvis de allerede finnes i `page cache` kopieres de til applikasjonsbufferet. Hvis de ikke finnes i `page cache` allokeres en ny side som settes inn `address_space`-objektets sideliste. Dataene overføres deretter fra disk til `page cache`, og kopieres videre til applikasjonsbufferet.

4.1.3 Bufferhåndtering i kommunikasjonssystemet



Figur 4.4: Socket buffer

Kommunikasjonssystemet brukes for å sende data til nettverket. Det består av flere lag og bruker et bufferhåndteringssystem kalt `socket buffer`. Det øverste laget er socket laget og tilbyr et grensesnitt til applikasjonen. Videre kommer protokollagene som transportlaget (TCP/UDP), nettverkslaget (IP), og datalinklaget. Kommunikasjonssystemets `socket buffer` har en annen funksjon enn filsystemets `page cache`. Filsystemets `page cache` skal blant annet redusere antall diskaksesser ved gjenbruk av data, mens `socket buffer` er et buffer hvor dataene lagres under oppbygging og stripping av header, og oppbevares midlertidig til nettverkskortet eller applikasjonen kan overføre dataene.

Alle pakker som sendes til eller mottas fra nettverket lagres i kommunikasjonssystemets `socket buffer`, se figur 4.4. Når data sendes gjennom kommunikasjonssystemet opprettes det en instans av en `sk_buff`-struct, og det allokeres plass

til pakkeheadere og eventuelt payload i socket buffer. Når et nytt sk_buff-objekt er opprettet, plasseres det i socketens sk_buff-liste. sk_buff-objektet inneholder kontrolldata og referanser til de forskjellige pakkeheaderne og payload i det allokerte området. Referansen til sk_buff sendes med til alle protokollagene og det er dermed ingen kopiering av data mellom protokollagene, men det er en kopioperasjon av dataene mellom applikasjonsbufferet og komunikasjonssystemets socket buffer. Etter at pakken er ferdig plasseres referansen i nettverkskortets overføringskø, og headere og payload overføres til nettverkskortets buffer i en DMA-operasjon.

Hvis send brukes for å sende data til nettverket må det allokeres plass til payloaddataene i socket buffer, og de kopieres fra applikasjonsbufferet til kommunikasjonssystemets socket buffer. Når det gjelder sendfile opprettes det referanse til payloaddataene som ligger i filsystemets page cache, og bare plass for headere allokeres i kommunikasjonssystemets socket buffer.

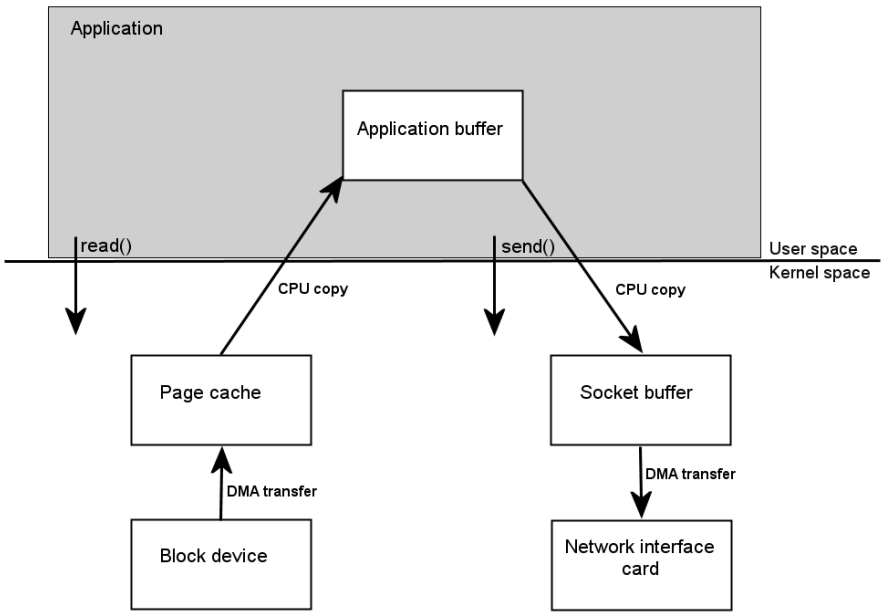
4.2 Eksisterende systemkall

Vi har nå sett på bufferhåndtering i filsystemet og kommunikasjonssystemet, og skal nå se på hvilke systemkall som eksisterer for overføring av data fra disk til nettverk. Det finnes flere muligheter for overføring av data fra disk til nettverk, og dataene kan overføres via et applikasjonsbuffer eller det kan benyttes en datasti innenfor kjernen. For overføring fra disk til et applikasjonsbuffer og videre til nettverk benyttes systemkallene read og send, og for overføring innenfor kjernen kan sendfile eller en kombinasjon av mmap og send benyttes.

4.2.1 read

ssize_t read(int fd, void *buf, size_t count);

Systemkallet read, se figur 4.5, benyttes for overføring av data fra en blokkenhetsfil, nettverkskort eller lignende til et applikasjonsbuffer [1]. Applikasjonsbufferet må allokeres før et read kall. Fildeskriptoren, peker til applikasjonsbufferet og antall byte data som skal kopieres til applikasjonsbufferet, overføres til kjernen. I kjernen opprettes en leseoperasjonsbeskivelse som inneholder statusen til leseoperasjonen. Ved overføring av data fra en blokkenhetsfil lokaliseres deretter address_space-objektet for å finne ut om dataene finnes i page cache. Hvis dataene ikke finnes i page cache, startes en DMA-overføring av data fra disk til page cache, deretter overføres dataene fra page cache til applikasjonsbufferet i en kopioperasjon. Hvis dataene etterspørres på et senere tidspunkt kan de hentes



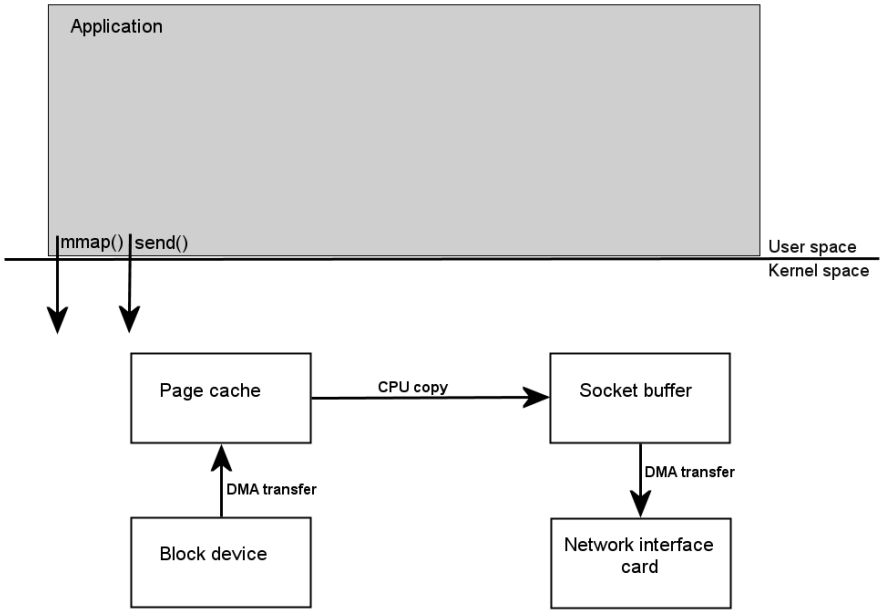
Figur 4.5: Datasti for `read` og `send`

direkte fra `page cache` istedenfor å hentes fra disk. Hvis filen leses sekvensielt vil `readahead`-logikken sørge for at dataene ligger i `page cache` når de leses og et `read` systemkall vil dermed bare medføre at dataene kopieres fra `page cache` til applikasjonsbufferet.

Det finnes også en mulighet for å overføre dataene direkte fra disk til applikasjonsbufferet. Dette gjøres ved å sette `O_DIRECT` flagget når filen åpnes. Dermed brukes `direct access bufferet` istedenfor `page cache`. Dataene overføres til `direct access buffer` i en DMA-operasjon og mappes i prosessens sidetabeller. På denne måten fjernes kopioperasjonen som ellers er nødvendig i forbindelse med `read`. Dette brukes av applikasjoner som har egen caching som for eksempel databasesystemer. Dette kan også være en bra løsning for MoD applikasjoner, ettersom de ikke drar store fordeler av `page cache`. Ved å overføre dataene direkte til applikasjonsbufferet fjernes en kopioperasjon og dataene tar mindre plass i minnet ettersom de bare lagres et sted.

Et kall på `read` fører til et kontekstbytte fra applikasjon til kjernen, og et kontekstbytte fra kjernen til applikasjonen når `read` kallet returner.

4.2.2 mmap



Figur 4.6: Datasti for mmap og send

void * mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);

Systemkallet mmap [1] er et alternativ til read for overføring av data fra disk til minne. mmap lager en virtuell minnemapping for en fil, et virtuelt sammenhengende minneområde som etter at det er opprettet ikke inneholder noen sider med data. Når en virtuell minnemapping er laget, kan applikasjonen lese dataene i filen ved å referere en linear (virtuell) adresse innenfor minneområdet. Dette vil føre til en "page fault", og som figur 4.6 viser, vil det startes en DMA-overføring av dataene fra disk til page cache. Dataene i page cache blir referert til av address_space-objektet, som beskrevet i seksjon 4.1.2, og av prosessens sidetabeller, slik at de kan aksesseres av både applikasjonen og kjernen. Ulempen med mmap er at det kreves et stort virtuelt adresserom og på en 32bits arkitektur er dette begrenset til 4GB for hver prosess, noe som blir lite ved håndtering av store mediafiler. Mappingen av dataene i sidetabellene er en ekstra overhead, men er sannsynligvis billigere med hensyn på CPU-cykler enn en kopiering av dataene til et applikasjonsbuffer.

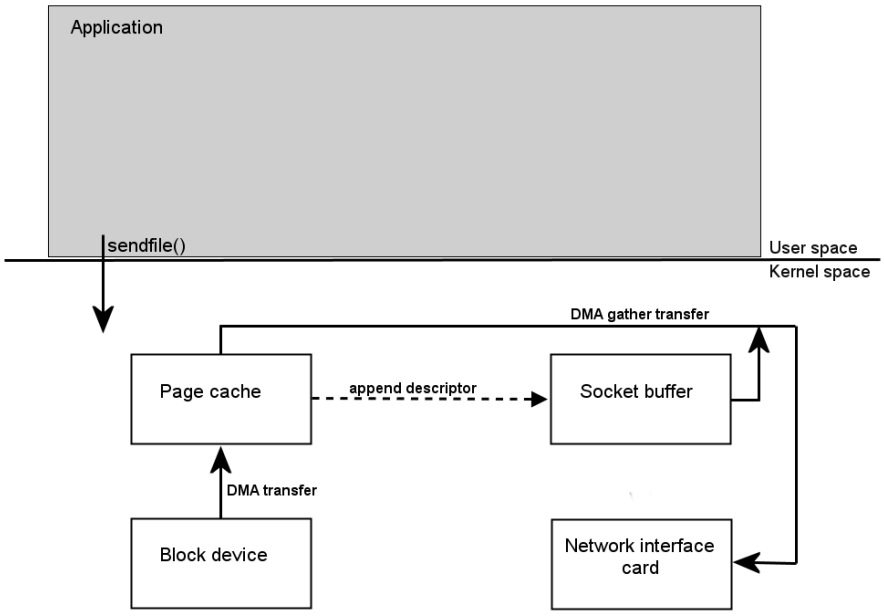
mmap fører i likhet med read til to kontekstbytter, fra applikasjon til kjernen og

tilbake til applikasjonen, men kopioperasjonen fra `page cache` til applikasjonsbufferet er fjernet. Ved bruk av `mmap` får man i tillegg kontekstbytter ved "page fault" når dataene som leses av applikasjonen ikke finnes i `page cache`.

4.2.3 `send`

ssize_t send(int s, const void *buf, size_t len, int flags);

For å overføre data fra et applikasjonsbuffer til nettverkskortet brukes systemkallet `send` [1]. Det finnes også varianter av `send` som `write` og `writev`. Ved et kall på `send` vil dataene enten bli overført fra applikasjonsbufferet, - hvis brukt sammen med `read` (figur 4.5), eller fra `page cache` hvis `mmap` er brukt (figur 4.6), - til kommunikasjonssystemets `socket buffer` i en kopioperasjon. Deretter legges det til headere før headere og dataene overføres til nettverkskortets buffer i en DMA-operasjon. Kombinasjon `read` og `send` har en datasti som omfatter applikasjonen, mens kombinasjon `mmap` og `send` har en datasti innenfor kjernen. Et kall på `send` fører til to kontekstbytter, fra applikasjonen til kjernen og tilbake til applikasjonen.



Figur 4.7: Datasti for `sendfile`

4.2.4 sendfile

ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);

`sendfile` [1] brukes for overføring av data fra disk til nettverkskortet i et kall langs en datasti innenfor kjernen og kan brukes av applikasjoner som ikke utfører noen operasjoner på dataene. Figur 4.7 viser at et kall på `sendfile` medfører at det startes en DMA-overføring av dataene fra disk til filsystemets `page cache`. Deretter overføres referansen til dataene fra `page cache` til `socket buffer`, men dataene blir ikke kopiert. Videre blir headere bygget opp og en DMA-overføring sørger for at headere blir overført fra `socket buffer`, og payloaddataene blir overført fra `page cache` til netverkskortets buffer. `sendfile` innebærer dermed ingen kopioperasjoner for å overføre dataene fra disk til nettverkskortet. Ettersom headere og payloaddata ikke er plassert i fysisk sammenhengende minne, må nettverksgrensesnittet støtte DMA-samleoperasjoner. `sendfile` fører til to kontekstbytter, fra applikasjon til kjernen og tilbake til applikasjonen.

4.3 Eksperimenter

I denne seksjonen skal vi se på eksperimenter med systemkallene `read`, `mmap`, `send`, `writev` og `sendfile`, og se på kopiytelsen og kontekstbytteytelsen på testmaskinen. Først presenteres noen innledende eksperimenter hvor vi ser på ytelsen til filsystemene `ext3`, `reiser` og `xfs` for å se om valg av filsystem har betydning ved nedlasting og streaming, og en sammenligning av Linux 2.4 og 2.6 for å se om endringene i 2.6 kjernen gir ytelsesforbedringer ved overføring av data fra disk til nettverk i nedlasting og streamingsammenheng. Etter de innledende eksperimentene ser vi på kostnadene ved kopioperasjoner og konteksbytte, før vi ser på profilen av systemkallene for å avdekke hvilke metoder i utførelsen av systemkallene som har det høyeste antall CPU-cykler. Vi ser videre på en sammmenligning av ressursforbruk for de forskjellige systemkallene i medianedlasting- og streamingsammenheng. Til slutt presenteres resultatene fra et “hyperthread”-eksperiment, hvor vi ser om kopioperasjoner og kryptering kan utføres parallelt på en CPU med “hyperthreading”.

4.3.1 Testoppsett

Eksperimentene beskrevet i dette kapitlet ble utført på Dell Optiplex GX240. Testmaskinene har et Intel 845 chipset, 1.70GHz Intel Pentium4 CPU, 8KB level 1 data cache, 256KB level 2 data cache, 400MHZ front side bus, 1GB PC133

SDRAM, 40GB 7,200RPM ATA/100 disk, og integrert 3Com Fast EtherLink 10/100 PCI nettverks interface. De to maskinene ble koblet sammen via en 100Mbps switch. For "hyperthread"-eksperimentene ble det benyttet en Intel Pentium 4 CPU 3.00GHz med Hyper-Threading.

I eksperimentene ble ressursforbruk målt ved hjelp av `rdtsc`[5] instruksjonen, `getrusage`[1] og `gettimeofday`[1]. `rdtsc` ble brukt i kopieksperimentet og brukes til å lese antall klokkecykler. `getrusage` måler konsumert tid i applikasjonen og kjernen og ble brukt i kontekstbytteeksperimentene og for testing av systemkallene. `gettimeofday` ble brukt i "hyperthread"-testen. For profiling av systemkallene, ble `Oprofile` [6] benyttet.

4.3.2 Sammenligning av filsystemene ext3, reiser og xfs

Vi skal nå se på eksperimenter med med filsystemene `ext3`, `reiser` og `xfs`. Formålet med eksperimentene er å se om det er forskjell i CPU-belastning mellom de forskjellige filsystemene. `ext3` og `reiser` versjon 3 ble valgt fordi de er standard filsystem på henholdsvis `ReadHat`- og `Suse`-distribusjonene, mens `xfs` ble valgt fordi det er konstruert for å yte bra på store filer. Eksperimentene ble utført på Linux 2.4.21. Det ble brukt en partisjon på 3GB, med to filer på 1GB hver. CPU-bruk for overføringen av filene fra disk til nettverk ble målt ved hjelp av `getrusage` og hver test ble utført ti ganger, hvor de to filene ble overført annenhver gang.

Eksperimentene er delt i fire deler. De to første viser tradisjonell disk-til-nettverks-dataoverføring ved bruk av UDP og TCP. Her er kombinasjonene `read` og `send`, `mmap` og `send`, og `sendfile` brukt. `write` er utelatt ettersom tester har vist at `write` og `send` gir samme resultater og `writev` gir ingen mening når det kun er et buffer som skal overføres. De to siste eksperimentene viser RTP-streaming over TCP og UDP. For RTP-streaming over TCP er `read`, `mmap`, `send`, `writev` og `sendfile` valgt. Ved kombinasjon av `mmap` og `send` er det bruk `setsockopt` for corking av socketen slik at RTP-header og mediadata ikke skal sendes i to forskjellige pakker. Det samme gjelder også for `sendfile` hvor RTP-headeren overføres med `send` og mediadataene med `sendfile`. For RTP-streaming over UDP ble `read`, `mmap`, `send` og `writev` benyttet. `sendfile` og kombinasjonen av `mmap` og `send` ble utelatt ettersom corking med `setsockopt` for UDP ikke er implementert i Linux 2.4.21.

I det første eksperimentet ble dataene overført fra disk til nettverk uten applikasjonsnivåkontroll og genering av applikasjonsnivåheadere. Transportprotokollen UDP ble benyttet. I tabell 4.1 ser vi resultatene for de forskjellige filsystemene og en prosentvis sammenligning av filsystemene `ext3` og `reiser`, `ext3` og `xfs`, og

		tid sekunder			forskjell i prosent		
syscall	**size byte**	**ext3**	**reiser**	**xfs**	**ext3/reiser**	**ext3/xfs**	**xfs/reiser**
`read`/	512	11.53	12.21	12.19	5.51	5.40	0.11
`send`	1024	7.35	7.67	7.82	4.17	5.99	-1.94
	2048	5.49	5.92	5.57	7.14	1.41	5.82
`mmap`/	512	11.03	13.87	12.97	20.46	14.95	6.48
`send`	1024	7.57	8.36	7.73	9.4	2.06	7.49
	2048	7.43	7.55	7.48	1.58	0.76	0.83
`sendfile`	512	9.03	9.60	8.92	6.01	-1.16	7.09
	1024	6.35	6.68	6.34	4.88	-0.2	5.06
	2048	5.33	5.67	5.28	6.14	-0.86	6.93

Tabell 4.1: Sammenligning av `ext3`, `reiser` og `xfs` for nedlastingsoperasjoner ved bruk av UDP

		tid sekunder			forskjell i prosent		
syscall	**size byte**	**ext3**	**reiser**	**xfs**	**ext3/reiser**	**ext3/xfs**	**xfs/reiser**
`read`/	10.31	10.77	11.46	512	4.34	10.06	-6.36
`send`	7.73	8.36	8.03	1024	7.5	3.76	3.88
	6.50	7.17	6.76	2048	9.34	3.76	5.8
`mmap`/	10.22	10.88	10.23	512	6.05	0.09	5.97
`send`	8.61	9.53	9.18	1024	9.64	6.24	3.63
	8.17	9.09	8.51	2048	10.07	4.01	6.32
`sendfile`	4.14	4.83	4.51	512	14.29	8.21	6.62
	2.96	3.56	2.80	1024	16.73	-5.66	21.19
	2.13	2.67	2.03	2048	20.3	-4.59	23.8
	1.30	1.83	1.30	1 GB	28.67	-0.23	28.84

Tabell 4.2: Sammenligning av `ext3`, `reiser` og `xfs` for nedlastingsoperasjoner ved bruk av TCP

`xfs` og `reiser`. Vi kan se at `ext3` og `xfs` har lavere ressursforbruk en `reiser`, med unntak av `read` og `send` med payloadstørrelse 1024 byte hvor `xfs` har høyest resursforbruk. Når det gjelder sammenligningen mellom `ext3` og `xfs` kan vi se at `xfs` er bedre enn `ext3` ved bruk av `sendfile`, som har en datasti innenfor kjernen, mens `ext3` er best for de andre systemkallene. Med unntak av `mmap` og `send` med payloadstørrelse 512 byte, er forskjellene mindre enn 6 prosent mellom filsystemene `ext3` og `xfs`, mens for `reiser` er forskjellene noe større.

I det neste eksperimentet ble de samme testene utført, men denne gangen med

transportprotokollen TCP. Ved bruk av `sendfile` sammen med TCP er det mulig å overføre en hel fil i et systemkall. Denne testen ble derfor tatt med blant annet for å se om `xfs` ville yte bra i et slikt scenario. Tabell 4.2 viser resultatene for testene utført med TCP, og vi ser tilsvarende resultat som for UDP. `ext3` og `xfs` yter bedre enn `reiser`, mens `xfs` er bedre enn `ext3` for `sendfile`. I eksperimentet med `sendfile` hvor filen overføres i et systemkall er det noe overraskende nesten ikke forskjell mellom `ext3` og `xsf`, mens `reiser` er nesten 30 prosent dårligere.

		tid sekunder			forskjell i prosent		
syscall	**size byte**	**ext3**	**reiser**	**xfs**	**ext3/reiser**	**ext3/xfs**	**xfs/reiser**
`read/`	512	10.91	11.22	11.43	2.73	4.55	-1.9
`send`	1024	7.43	7.95	7.97	6.45	6.74	-0.31
	2048	5.33	6.05	5.77	11.91	7.51	4.76
`read/`	512	11.13	11.22	11.42	0.77	2.5	-1.78
`writev`	1024	7.26	7.77	7.97	5.29	7.73	-2.64
	2048	5.55	6.24	5.92	11.02	6.33	5.01
`mmap/`	512	13.34	14.51	13.68	8.11	2.52	5.73
`writev`	1024	7.63	8.39	7.82	8.98	2.4	6.75
	2048	6.90	7.42	6.95	7.08	0.84	6.29

Tabell 4.3: Sammenligning av `ext3`, `reiser` og `xfs` for RTP-streamingoperasjoner ved bruk av UDP

Vi skal nå se på eksperimenter med filsystemene når det legges til RTP-headere generert på applikasjonsnivå, til mediadataene som overføres fra disk til nettverk. Tabell 4.3 viser resultatene hvor UDP er benyttet som transportprotokoll. I disse eksperimentene er ikke `sendfile` med ettersom UDP_CORK ikke var implementert for `setsockopt` i Linux 2.4.21. Dette er nødvendig for at RTP-header og mediadata skal sendes i samme UDP-pakke. Resultatene viser at `ext3` har lavest resursforbruk i alle testene, men forskjellene sammenlignet med `xfs` ikke er store.

Det siste eksperimentet er også med applikasjonsnivågenererte headere, men denne gangen med TCP som transportprotokoll. I dette eksperimentet er `sendfile` med, og det er også en test med `mmap` og `send` som krever bruk av `setsockopt` for corking av socketen. I tabell 4.4 ser vi at resultatene for dette eksperimentet er tilsvarende som for de andre eksperimentene. `xfs` kommer best ut for `sendfile` med størrelse 1024 og 2048, mens i alle de andre testene er `ext3` best. `reiser` er dårligst i nesten alle testene.

Disse eksperimentene viser at det i de fleste tilfellene ikke er stor forskjell mellom de forskjellige filsystemene. `ext3` er best i 81.4 prosent av testene, mens `xfs` er best i de resterende 18.8 prosent av testene. De testene `xfs` er best er med

		tid sekunder			forskjell i prosent		
syscall	**size byte**	**ext3**	**reiser**	**xfs**	**ext3/reiser**	**ext3/xfs**	**xfs/reiser**
read/	512	10.33	10.99	11.56	5.95	10.59	-5.19
send	1024	7.94	8.36	8.30	4.99	4.23	0.79
	2048	6.73	7.15	7.01	5.95	4.08	1.96
read/	512	10.42	10.88	11.63	4.21	10.39	-6.9
writev	1024	7.77	8.06	8.08	3.59	3.78	-0.19
	2048	6.53	6.97	6.73	6.3	2.92	3.49
mmap/	512	19.18	19.77	19.26	2.95	0.38	2.57
send	1024	12.77	15.39	14.08	16.99	9.29	8.48
	2048	10.70	11.71	11.13	8.65	3.93	4.91
mmap/	512	10.61	11.26	10.68	5.78	0.68	5.13
writev	1024	9.01	10.04	9.50	10.22	5.11	5.39
	2048	8.37	9.26	8.74	9.65	4.25	5.64
sendfile	512	14.39	15.02	14.66	4.13	1.82	2.36
	1024	7.68	7.93	7.35	3.13	-4.52	7.32
	2048	4.42	4.76	4.21	7.24	-4.9	11.58

Tabell 4.4: Sammenligning av `ext3`, `reiser` og `xfs` for RTP-streamingoperasjoner ved bruk av TCP

`sendfile` som har en datasti innenfor kjernen, men forskjellen til `ext3` er ikke store og varierer fra 0.2 til 6 prosent. Når det gjelder `reiser` yter den dårligst i nesten 80 prosent av testene og er dårligere enn `ext3` på alle testene. Ut fra dette kan det se ut som `ext3` er det beste alternativet for `read`, `mmap`, `send` og `writev`, mens `xfs` er det foretrukne alternativet ved bruk av `sendfile` og størrelse over 1KB. For mindre datastørrelser ser det ut som ext3 er det beste alternativet også for `sendfile`.

4.3.3 Sammenligning av Linux 2.4 og 2.6

De forrige eksperimentene ble utført på Linux 2.4.21. Vi skal nå se på om det er forbedringer i ytelsen fra Linux 2.4 til 2.6 for nedlastings- og streamingoperasjoner.

Linux 2.6 kjernen er en stor oppgradering fra kjerne 2.4, med mange forbedringer [25]. Noen av forbedringene er blant annet redusert bruk av den globale kjernelåsen `Big Kernel Lock (BKL)`, implementering av en O(1) scheduler, optimalisert blokkopi, implementering av `New API (NAPI)`, forbedret `readahead` og

forbedringer i TCP-stakken med ny opphopningskontroll (congestion) og gjenn-opprettingsalgoritmer (recovery).

Den globale kjernelåsen (BKL) er en spinlås som ble innført i kjerne 2.2 for å tillate flere prosessorer å kjøre kjernekode samtidig. Multiprosessor støtte ble innført i kjerne 2.0 men bare en oppgave kunne være i kjernen av gangen. BKL brukes for å beskytte kritiske regioner og medfører at bare kode i en kritisk region kan utføres av en oppgave av gangen. Kode i to ikke-relaterte kritiske områder kan ikke utføres samtidig av to oppgaver hvis de er beskyttet av BKL, og bruk av BKL kan dermed medføre dårlig utnyttelse av flere prosessorer. Målet når BKL ble implementert var å splitte den opp og etterhvert fjerne bruk av den helt. I kjerne 2.6 er det meste av BKL bruken fjernet. For enkeltprosessorsystemer har ikke kjernelåsen noe betydning.

Scheduleren i kjerne 2.4 er O(n), mens den nye i kjerne 2.6 er O(1). Dette betyr at alle algoritmene i 2.6 scheduleren utføres på konstant tid uavhengig av hvor mange prosesser det er i systemet, mens 2.4 scheduleren inneholder algoritmer hvor tiden det tar å utføre dem avhenger av antall prosesser. En av disse algoritmene er beregningen av tidsintervallet som hver prosess kan kjøre. Når alle prosessene har brukt sitt tidsintervall, må 2.4 scheduleren gå gjennom listen med prosesser og beregne nye tidsintervaller for alle prosessene. Denne algoritmen er dermed O(n). I 2.6 scheduleren er det to lister med prosesser, en liste med aktive prosesser og en med prosesser som har brukt opp sitt tidsintervall. Når en prosess har brukt sitt tidsintervall beregnes nytt tidsintervall og prosessen flyttes over i listen med prosesser som har brukt sitt intervall. Når alle prosessene har brukt sitt tidsintervall endres pekerne til disse listene slik at aktiv-listen blir utgått-listen og omvendt. Denne algoritmen er dermed O(1). Dette har stor betydning for servere med mange prosesser.

Det er også gjort forbedringer i nettverksstakken med implementering av NAPI [39]. De viktigste forbedringene er redusering av avbrudd (interrupts) og kasting av pakker på nettverkskortet. NAPI tillater drivere å kjøre med avbrudd avslått ved høy trafikk. Normalt vil det genereres et avbrudd for hver pakke som mottas og dette fører til stor belastning ved høy trafikk. Ved å skru av avbrudd i perioder mottas bare avbrudd ved noen pakker og systemtbelastningen reduseres. Kasting av pakker på nettverkskortet skjer når systemet ikke klarer å håndtere alle pakkene. Ved å kaste pakkene allerede på nettverkskortet reduseres belastningen på systemet.

Når dataene overføres fra disk til nettverk må, dataene kopieres mellom forskjellige minnelokasjoner. I Linux 2.6 er det gjort optimaliseringer i kopirutinene for x86-arkitekturen hvor `movsd`-instruksjonen er byttet ut med en “hand unrolled loop”-teknikk [16] med integer-registre. Dette er med på å optimaliserer looper

med `load`- og `store`-instruksjoner for instruksjonsnivå parallellisme.

`readahead` er en mekanisme for å overføre data fra disk til `page cache` før dataene blir etterspurt av applikasjonen. Dermed reduseres tiden applikasjonen må vente på I/O, og gjennomstrømmingshastigheten øker. I Linux 2.6 er `readahead`-koden skrevet om og dette gir en stor forbedring i gjennomstrømmingen ved sekvensiell lesing [31].

Alle forbedringene beskrevet ovenfor kan ha betydning for ytelsen i en multimediaserver, men for eksperimentene i denne seksjonen er det muligens bare forbedringene i kopirutinen og `readahead` som har betydning, ettersom det er et én-prosessorsystem med bare en prosess for overføring av data.

Eksperimentene ble utført på Linux 2.4.21 og 2.6.5 og det ble brukt en partisjon på 3GB med 2 filer på 1GB hver. Partisjonen var den samme for alle testene og filsystemet som ble benyttet var `ext3`. CPU-bruk ved overføringen av filene fra disk til nettverk ble målt ved hjelp av `getrusage`, hvor hver test ble utført 10 ganger og de 2 filene ble overført annenhver gang.

syscall	**size byte**	**kernel 2.6.5 sec**	**kernel 2.4.21 sec**	**forbedring %**
read/	512	8.714675	11.537200	24.46
send	1024	5.844012	7.357300	20.57
	2048	8.144362	5.499400	-48.1
mmap/	512	9.399971	11.038000	14.84
send	1024	7.139615	7.579700	5.81
	2048	6.536707	7.431000	12.03
sendfile	512	6.052180	9.030200	32.98
	1024	3.951599	6.358000	37.85
	2048	5.736328	5.331400	-7.6

Tabell 4.5: Sammenligning av Linux-kjerne 2.4.21 og 2.6.5 for nedlastingsoperasjoner med UDP-transportprotokoll

Tabell 4.5 viser resultatene for tradisjonell nedlasting med UDP. Som vi ser er forbedringene for UDP fra 6 til 38 prosent, med unntak av pakkestørrelser over MTU for kombinasjonen `read` og `send` og for `sendfile`. Dette kan skyldes at fragmentering av UDP pakker er mer kostbart på 2.6-kjernen. Forbedringene skyldes muligens forbedringer i `readahead`-koden, som fører til at dataene alltid er på plass i `page cache` når de etterspørres. Dette fører til at prosessen må scheduleres færre ganger. For `read` og `send` kan også forbedringer i blokkopirutinen ha en innvirkning. For kombinasjonen av `mmap` og `send` er antall “page fault” redusert fra 26000 i 2.4 kjernen til 16000 i 2.6-kjernen, noe som tyder på at flere

syscall	size byte	kernel 2.6.5 sec	kernel 2.4.21 sec	forbedring %
read/	512	7.105820	10.310800	31.08
send	1024	5.978091	7.733400	22.7
	2048	5.504263	6.509300	15.44
mmap/	512	7.986786	10.225300	21.89
send	1024	7.827110	8.615400	9.15
	2048	7.640939	8.175900	6.54
sendfile	512	2.776278	4.144200	33.01
	1024	1.996097	2.966100	32.7
	2048	1.526668	2.132500	28.41
	1073739776	1.044841	1.308400	20.14

Tabell 4.6: Sammenligning av Linux-kjerne 2.4.21 og 2.6.5 for nedlastingsoperasjoner med TCP-transportprotokoll

etterfølgende sider blir overført til `page cache` for hver "page fault". Dette kan forklare noe av forbedringene ved bruk av `mmap`.

For TCP er det, som vi ser i tabell 4.6, forbedringer for alle testene. Disse varierer fra 6 til 33 prosent og forbedringene er størst for de laveste størrelsene. Forbedringene for TCP skyldes sannsynligvis, i likhet med UDP, forbedringer i `readahead` for `read` og `sendfile`, og for `mmap` overføring av flere etterfølgene sider ved "page fault". Forbedringene skyldes sannsynligvis også opptimaliseringer i TCP-stakken.

syscall	size byte	kernel 2.6.5 sec	kernel 2.4.21 sec	forbedring %
read/	512	8.742471	10.918800	19.93
send	1024	6.271547	7.439200	15.7
	2048	7.825210	5.336900	-46.62
read/	512	9.351078	11.135600	16.03
writev	1024	6.622893	7.361700	10.04
	2048	7.602744	5.552400	-36.93
mmap/	512	9.064322	13.342500	32.06
writev	1024	8.005983	7.638400	-4.81
	2048	7.312389	6.900000	-5.98

Tabell 4.7: Sammenligning Linux kjerne 2.4.21 og 2.6.5 for streamingoperasjoner med UDP-transportprotokoll

I tabellene 4.7 og 4.7 ser vi at resultatene for RTP-streaming er tilsvarende re-

syscall	size byte	kernel 2.6.5 sec	kernel 2.4.21 sec	forbedring %
read/	512	7.381678	10.337000	28.59
send	1024	5.956495	7.949900	25.07
	2048	5.355886	6.733600	20.46
read/	512	7.790716	10.422900	25.25
writev	1024	6.038782	7.777100	22.35
	2048	5.441773	6.536500	16.75
mmap/	512	11.453159	19.188100	40.31
send	1024	9.551148	12.779300	25.26
	2048	8.546500	10.701500	20.14
mmap/	512	8.184156	10.611900	22.88
writev	1024	7.897200	9.019000	12.44
	2048	7.708428	8.373700	7.94
sendfile	512	7.337885	14.399500	49.04
	1024	4.230557	7.683300	44.94
	2048	2.606704	4.421200	41.04

Tabell 4.8: Sammenligning Linux kjerne 2.4.21 og 2.6.5 for streamingoperasjoner med TCP-transportprotokoll

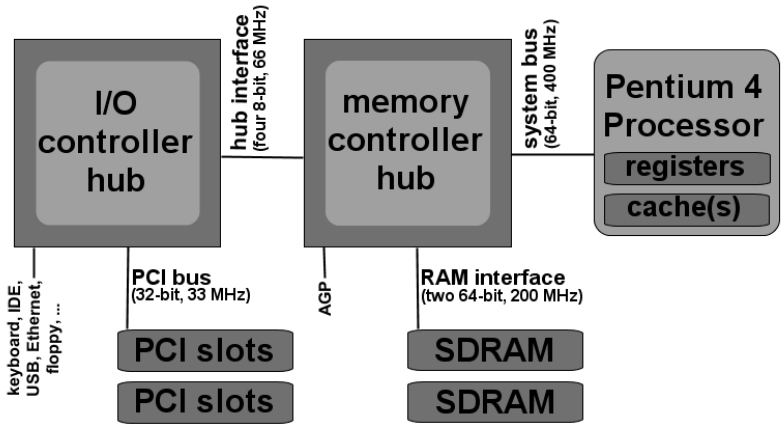
sultatene for tradisjonell nedlasting. For TCP er det forbedringer for alle testene, mens for UDP er det forbedringer for pakkestørrelser mindre enn MTU. Forbedringene for TCP er fra 8 til 49 prosent, noe som er en betydelig forbedring, med de største forbedringene for `sendfile`. Igjen er det sannsynligvis forbedringer i `readahead`-koden som er årsaken til dette.

Vi har nå sett på noen innledende eksperimenter med systemkallene som er aktuelle ved overføring av data fra disk til nettverk i nedlasting- og streamingsammenheng. I de neste eksperimentene skal vi se nærmere på de aktuelle systemkallene for å avdekke om de er optimalt implementert og se om det er mulig med forbedringer for å redusere bruk av delte resurser som for eksempel CPU, systembuss og minne.

4.3.4 Kopi- og kontekstbytteytelse

Figur 4.8 viser en oversikt over chipsetet på testmaskinen. Overføring mellom disk og minne utføres vanligvis med DMA-operasjoner som flytter dataene over PCI-bussen, I/O-kontrollerhuben, hubgrensesnittet, minnekontrollerhuben og RAM-grensesnittet. En DMA-operasjon utføres uten å involvere CPU. En minne-til-minne kopioperasjon blir utført ved å flytte data fra RAM over RAM-grensesnittet,

minnekontrollerhuben, systembussen gjennom CPU og tilbake til en annen RAM-lokasjon. Minne til minne kopioperasjoner involver CPU, og data blir overført flere ganger gjennom delte komponenter. Dette er med på å redusere ytelsen til systemet, og vi skal i dette eksperimentet se på overheaden som innføres ved en slik kopioperasjon.

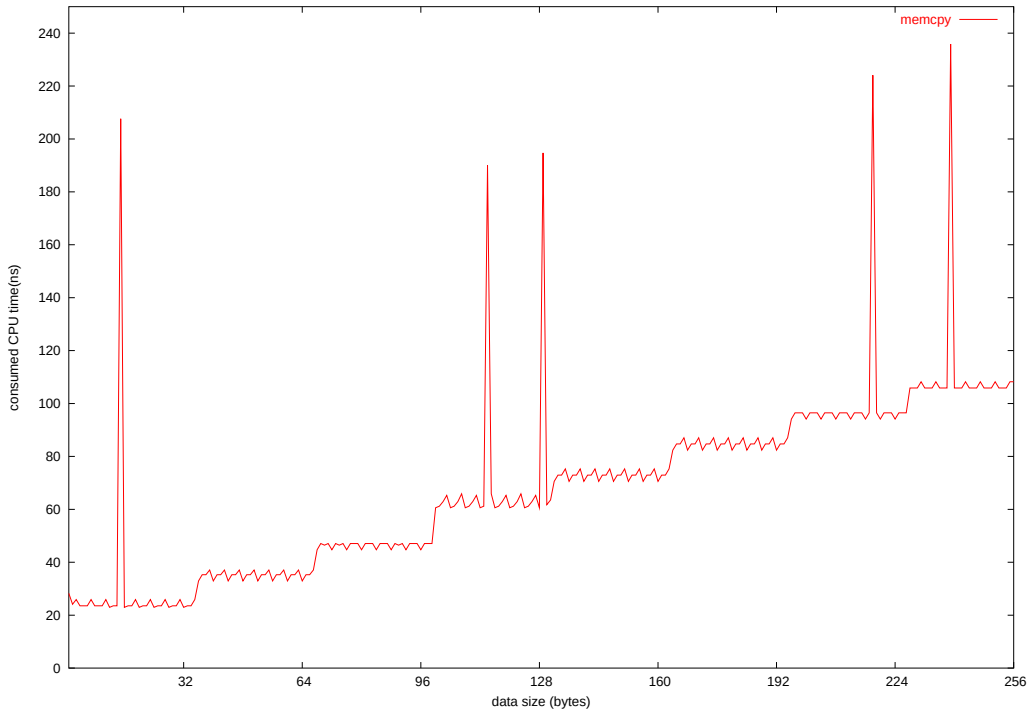


Figur 4.8: Pentium 4 prosessor og 845 chipset

For å teste dette ble `memcpy` valg. I [10] ble kopiytelsen målt på Linux (2.2 og 2.4) og Windows (2000) hvor konklusjonen var at `memcpy` gjør det bra sammenlignet med andre kopifunksjoner. Testene viser også at Linux i de fleste tilfeller er raskere enn Windows, avhengig av størrelse og kopiinstruksjonen som benyttes.

Kopitesten ble implementert i kjernen, og `memcpy` ble testet med forskjellige datastørrelser. Det ble kjørt 100 tester for hver datastørrelse, og for hver test ble det allokert et nytt fysisk sammenhengende minneområde. For måling av antall klokkecykler ble `Time Stamp Counter` lest av med `rdtsc` før og etter kopioperasjonen. Eksperimentet ble utført på Linux 2.6.5.

Figur 4.9 viser at overheaden vokser langsomt med størrelsen på dataelementet, men som vi ser i figur 4.10, vokser overheaden mer eller mindre lineært med størrelsen etter å ha nådd en bestemt størrelse. Toppene i figur 4.9 skyldes at testen ble avbrutt av en ekstern hendelse. Noen merkelige artifakter kan sees når størrelsen på minneadressene ikke er 4-byte alignet med enda større topper når eksperimentet ble utført på applikasjonsnivå. Dette er også synlig for større datastørrelser og skyldes sannsynligvis bruk av forskjellige instruksjoner. Resultatene viser at en minne til minne kopioperasjon av 1KB data bruker 238 nanosekunder. Kopiering av 1GB data vil da bruke 0.25 sekunder ved kopi av 1KB i hver kopioperasjon.

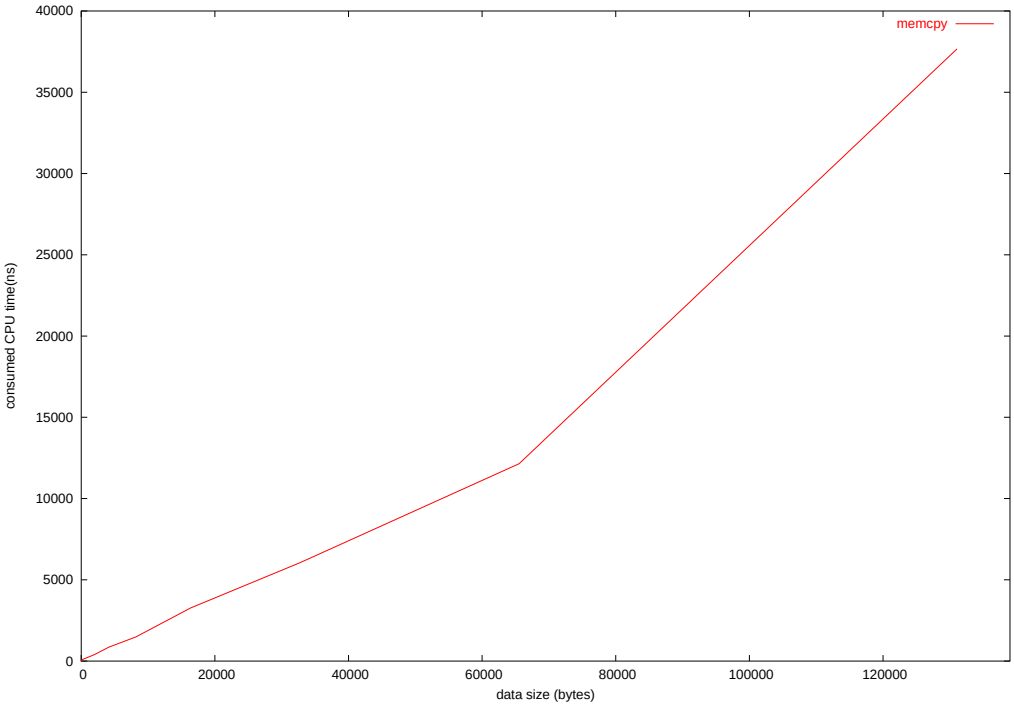


Figur 4.9: Kopieksperiment med dataelementer fra 1 til 256 byte

Kontekstbytte-eksperimentet måler tiden det tar for operativsystemet å bytte fra applikasjonsmodus til kjernemodus og tilbake til applikasjonsmodus. Disse testene ble utført ved å måle tiden det tar å utføre systemkallet `getpid`. `getpid`, som er det enkleste systemkallet tilgjengelig i operativsystemet, aksesserer kjernen og returner prosessid'en. Testen ble kjørt 10 ganger med 10.000 kall på `getpid` i hver test, og for å måle tiden ble `getrusage` brukt. Eksperimentet viser at den gjennomsnittlige tiden for å aksessere kjernen og returnere er cirka 920 nanosekunder. Dette er veldig lang tid, og tester har vist at systemkallytelsen på P4 er dårlig sammenlignet med andre arkitekturer [19].

Det ble også utført et kontekstbytte eksperiment med systemkallet `rtpsendfile` (beskrevet i seksjon 5.1.3), hvor koden var byttet ut med et `return`-kall. Denne testen viser at kontekstbytteoverheaden ved streaming av en fil på 1GB ved bruk av `rtpsendfile` er 0.7 sekunder, når payloadstørrelsen er 1456 byte (1456 payload + headere = 1500byte = MTU). Gjennomsnittlig tid for et systemkall i dette eksperimentet er 949 nanosekunder.

Kopi- og systemkallytelse har også vært et tema for hardware produsenter som Intel, som har lagt til nye instruksjoner. Dette er for eksempel MMX- og SIMD-



Figur 4.10: Kopieksperiment med dataelementer fra 2 til 131072 byte

utvidelser som kan brukes i kopioperasjoner, og `sysenter` og `sysexit` instruksjoner for systemkall. Ved bruk av SIMD-instruksjoner ble hastigheten på blokkopioperasjoner forbedret med opptil 149 prosent i Linux 2.0-kjernen, men reduksjonen i CPU-bruk var bare 2 prosent [21].

I Linuxkjernen er det implementert en løsning som kalles `vsyscall` for å øke hastigheten ved systemkall. Dette er en kjerneside (kernel page) som er mappet i applikasjonenes virtuelle minneområde med virtuell adresse 0xFFFFF000. I stedenfor `int 0x80` kan applikasjonene gjøre et kall på 0xFFFFF000 for å entre kjernen. `vsyscall`-siden inneholder kode for `int 0x80` og `sysenter`, og ved et kall 0xFFFFF000 velges den mest optimale instruksjonen som er støttet på plattformen. På Pentium4 er for eksempel `sysenter` raskere enn `int 0x80`. For enkelte systemkall, som for eksempel `getpid` og `gettimeofday`, medfører `vsyscall` at det ikke er nødvendig å entre kjernen, fordi kjernen kopierer pid og tidsstempel til `vsyscall`-siden. Når applikasjonen utfører et kall på disse systemkallene, returneres pid eller tidsstempel fra `vsyscall`-siden.

Disse eksperimentene har vist at til tross for nye instruksjoner er både kopioperasjoner og kjerneaksess ressurskrevende og fortsatt mulige flaskehalser.

4.3.5 Systemkallprofil

samples	%	image name	symbol name
1281	19.4150	vmlinux	__copy_from_user_ll
1217	18.4450	vmlinux	__copy_to_user_ll
537	8.1388	reiserfs.ko	search_by_key
230	3.4859	vmlinux	sysenter_past_esp
210	3.1828	vmlinux	ip_push_pending_frames
187	2.8342	vmlinux	ip_append_data
148	2.2431	vmlinux	udp_push_pending_frames
126	1.9097	vmlinux	buffered_rmqueue
108	1.6369	vmlinux	__kmalloc
97	1.4701	vmlinux	find_get_page
95	1.4398	vmlinux	udp_sendmsg
94	1.4247	vmlinux	sock_alloc_send_pskb
92	1.3944	vmlinux	__make_request
92	1.3944	vmlinux	do_generic_mapping_read
89	1.3489	vmlinux	__find_get_block
86	1.3034	vmlinux	sys_read
82	1.2428	vmlinux	alloc_skb
59	0.8942	vmlinux	ip_finish_output
58	0.8791	vmlinux	kmem_cache_alloc
52	0.7881	vmlinux	sys_socketcall

Tabell 4.9: Profil av `read` og `send`

For å se hvilke kjernemetoder som brukes av de forskjellige systemkallene og for å se hvilke metoder som har den høyeste CPU-bruken utførte vi et eksperiment med Oprofile [6]. Resultatene viser alle metodene som brukes av applikasjonen og prosentvis CPU-bruk, sammenlignet med de andre metodene. I tabell 4.9 ser vi resultatene for `read` og `send` som viser at kopieringsoperasjonene mellom kjernen og applikasjonen står for den største overheaden. Disse utgjør tilsammen nesten 40 prosent av den samlede CPU-bruken. `__copy_to_user_ll` brukes ved kopiering av data fra filsystemets `page cache` til applikasjonsbufferet, mens `__copy_from_user_ll` brukes ved kopiering fra applikasjonsbufferet til kommunikasjonssystemets `socket buffer`. For `mmap` og `send` i tabell 4.3.5 ser vi at kopioperasjonen står for en stor del av overheaden. `mmap` og `send` har bare en kopioperasjon, `__copy_from_user_ll`, og den brukes for å kopiere data fra `page cache` til `socket buffer`. Kopioperasjonene står for nesten 40 prosent av CPU-bruken. Disse testene sier ingenting om forskjellene mellom kombinasjonen `mmap` og send og kombinasjonen `read` og `send`, men `__copy_from_user_ll`-metoden bruker sannsynligvis like mye prosessortid i begge tilfellene. Resultatene viser også at "page fault"-metoden `do_page_fault` i `mmap` og `send` benyttes mye. `sendfile` har ingen kopieringsoperasjoner mellom subsystemer, men kopieringsoperasjonen `skb_copy_bits` står for den høyeste CPU-bruken. `skb-`

samples	%	image name	symbol name
2595	38.7082	vmlinux	__copy_from_user_ll
423	6.3097	reiserfs.ko	search_by_key
377	5.6235	vmlinux	do_page_fault
175	2.6104	vmlinux	ip_append_data
172	2.5656	vmlinux	ide_outb
171	2.5507	vmlinux	ip_push_pending_frames
140	2.0883	vmlinux	ide_do_request
136	2.0286	vmlinux	sysenter_past_esp
113	1.6856	vmlinux	__kmalloc
107	1.5961	vmlinux	__make_request
100	1.4916	vmlinux	buffered_rmqueue
97	1.4469	vmlinux	udp_sendmsg
96	1.4320	vmlinux	sock_alloc_send_pskb
84	1.2530	vmlinux	alloc_skb
79	1.1784	vmlinux	kmem_cache_alloc
77	1.1486	vmlinux	udp_push_pending_frames
71	1.0591	3c59x.ko	boomerang_start_xmit
60	0.8950	vmlinux	ide_inb
55	0.8204	reiserfs.ko	reiserfs_get_block
54	0.8055	vmlinux	__find_get_block

Tabell 4.10: Profil av `mmap` og `send`

samples	%	image name	symbol name
412	13.5482	vmlinux	skb_copy_bits
178	5.8533	reiserfs.ko	search_by_key
170	5.5903	vmlinux	ip_append_data
165	5.4258	vmlinux	ip_push_pending_frames
124	4.0776	vmlinux	__kmalloc
122	4.0118	vmlinux	do_sendfile
117	3.8474	vmlinux	sysenter_past_esp
102	3.3542	vmlinux	find_get_page
97	3.1897	vmlinux	do_generic_mapping_read
86	2.8280	vmlinux	ip_append_page
79	2.5978	vmlinux	udp_sendpage
75	2.4663	vmlinux	sock_alloc_send_pskb
65	2.1375	vmlinux	kmap_atomic
63	2.0717	vmlinux	alloc_skb
62	2.0388	vmlinux	local_bh_enable
62	2.0388	vmlinux	udp_sendmsg
61	2.0059	vmlinux	kmem_cache_alloc
53	1.7428	vmlinux	udp_push_pending_frames
42	1.3811	vmlinux	buffered_rmqueue
40	1.3154	vmlinux	ip_finish_output

Tabell 4.11: Profil av `sendfile`

`_copy_bits` brukes ved linearisering av ip-headeren ettersom den ikke nødvendigvis ligger i sammenhengende fysisk minne og brukes når ip-adressen skal sam-

menlignes med ip-tabellen.

4.3.6 Medianedlasting

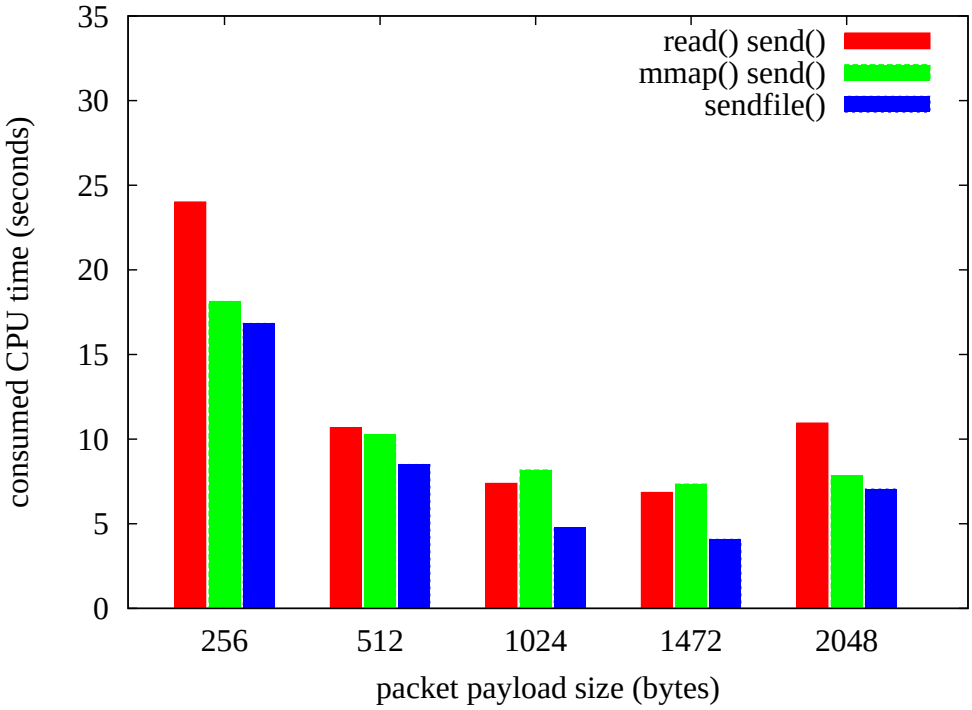
systemkall	ko	kb	kall til kjernen
`read send`	2*n*	4*n*	*n* **`read`** og *n* **`send`** kall (TCP vil samle flere mindre elementer til en større pakke med MTU størrelse.)
`mmap send`	*n*	2+2*n*	1 **`mmap`** og *n* **`send`** kall (TCP vil samle flere mindre elementer til en større pakke med MTU størrelse)
`sendfile` **(UDP)**	0	2*n*	*n* **`sendfile`** kall
`sendfile` **(TCP)**	0	2	1 **`sendfile`** kall

ko = antall `kopioperasjoner`, kb = antall `kontekstbytter`, *n* = antall pakker

Tabell 4.12: Oversikt over nedlastingsytelsestestene

I dette eksperimentet skal vi se på hele disk-nettverksdatastien i et medianedlastingsscenario. Ved medianedlasting er det ikke nødvendig med applikasjonlagskontroll, og dataene overføres fra disk og sendes så raskt som mulig. For å evaluere ytelsen utførte vi eksperimenter med systemkallene `read`, `mmap`, `send` og `sendfile`, med kombinasjonene `read` og `send`, `mmap` og `send`, og `sendfile`. `read` og `send` har en datasti som går via applikasjonsnivå, og som tabell 4.12 viser, innebærer det to kopioperasjoner. `mmap` og `send` har en datasti som går innenfor kjernen med hvor dataene kan leses av applikasjonen. Denne datastien har har en kopioperasjon. Datastien til `sendfile` går innenfor kjernen og har ingen kopioperasjoner. Eksperimentene ble utført for både UDP og TCP selv om UDP vanligvis ikke er aktuelt i et medianedlastingsscenario. Intensjonen med dette var å sammenligne UDP og TCP. Forventningene til disse eksperimentene, sett ut fra antall kopioperasjoner og kontekstbytter, er at `read` og `send` skal ha det høyeste resursforbruket, mens `sendfile` skal ha det laveste. CPU-bruken ble målt med `getrusage`.

Figur 4.11 viser resultatene for UDP-eksperimentet, og som vi ser er resultatene som forventet, bortsett fra for kombinasjonen `mmap` og `send` med datastørrelse 1KB og MTU. Resultatene viser at `mmap` og `send` har høyere CPU-bruk enn `read` og `send` ved disse størrelsene, noe som kan skyldes at `mmap` ikke bruker `readahead`. Resultatene fra `getrusage` viser at `read` og `send`, og `sendfile` har mindre enn 10 “page faults”, mens `mmap` og `send` har over 16.000. Ved bruk av `read` vil `readahead`-mekanismen sørge for å overføre dataene fra disk til minne før de blir etterspurt, og dataene kan dermed hentes fra `page cache`. Ved bruk av `mmap` vil det oppstå en “page fault” for nesten hver side som leses. En “page fault” medfører overhead ved oppdatering av prosessens sidetabeller, i tillegg til at prosessen må scheduleres mens den venter på at dataene overføres til minnet. For

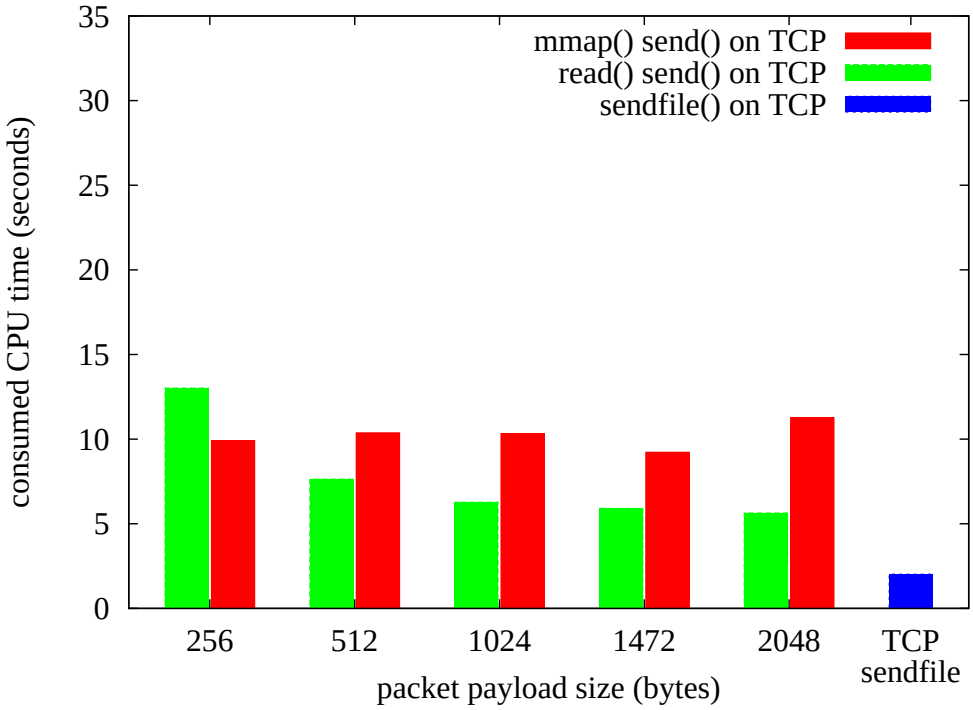


Figur 4.11: Nedlastingsoperasjoner med UDP

å starte `readahead` i forbindelse med mappede filer kan systemkallet `madvise` benyttes. Det er da mulig å gi kjernen informasjon om at et minneområde skal leses sekvensielt slik at sidene overføres til `page cache` før de blir etterspurt. Ved bruk av `madvise` sammen med `mmap` og `send` viste tester vi utførte at antall "page faults" ble redusert fra 16.000 til mindre enn 1.000, og CPU-bruken ble redusert med omtrent 13 prosent. Med MTU-pakkestørrelse utgjør dette 1 sekund.

Av de tre alternativene kan vi se at `sendfile` har det laveste ressursforbruket, og med MTU-størrelse på pakkene er sendfile 40 prosent bedre enn `read` og `send`. Ressursforbruket synker når datastørrelsene øker, på grunn av færre kontekstbytter, færre kopioperasjoner og generering av færre pakker. Kopitestene i seksjon 4.3.4 viste at kopiering av et dataelement på 1KB er mindre ressurskrevende enn kopiering av to dataelementer på 512B, noe som fører til lavere ressursforbruk når dataelementene øker. Når datastørrelsen overstiger MTU-størrelse, øker ressursforbruket på grunn av overhead i forbindelse med fragmentering.

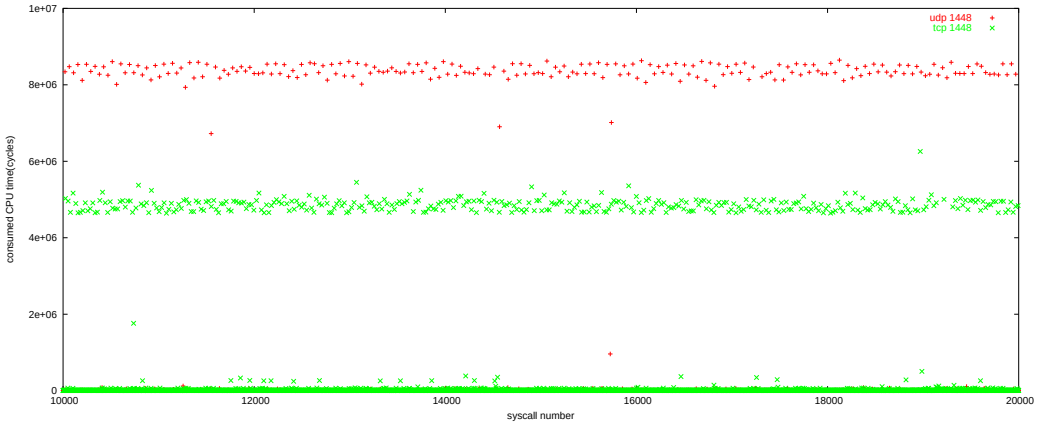
Resultatene for TCP, figur 4.12, viser i likhet med UDP-resultatene at `read` og `send` er bedre enn `mmap` og `send` med hensyn på CPU-bruk. Resultatene viser også at `sendfile` gir en dramatisk ytelsesforbedring sammenlignet med de tra-



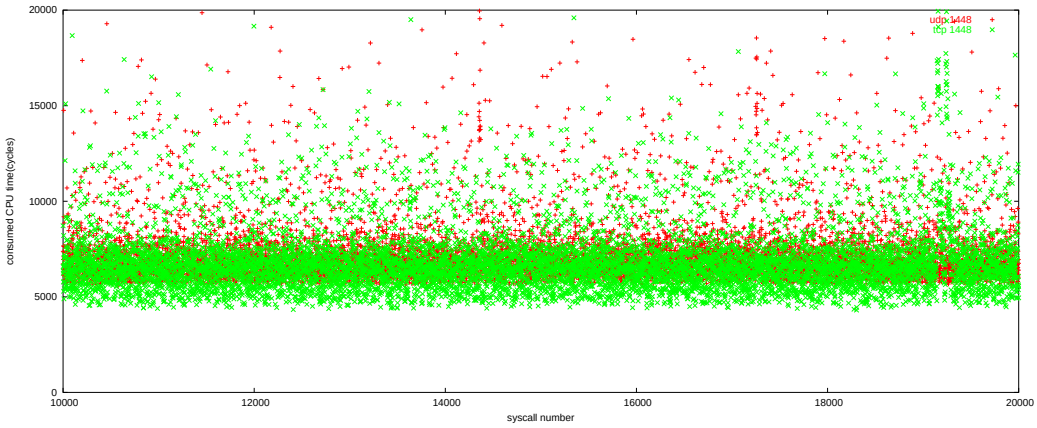
Figur 4.12: Nedlastingsoperasjoner med TCP

disjonelle systemkallene `read` og `send`. `sendfile` er minst 65 prosent bedre enn `read` og `send` og 78 prosent bedre enn `mmap` og `send`. Noe av grunnen til dette er at det er mulig å overføre en hel fil i et systemkall ved bruk av `sendfile`. Disse resultatene viser også at datastørrelsen ikke har så stor betydning ettersom TCP samler mindre dataelementer til en større pakke med MTU-størrelse.

Noe av hensikten med å kjøre tester med både UDP og TCP, var å sammenligne ressursforbruk ved bruk av disse to protokollene. På grunn av at UDP er en enklere protokoll enn TCP, var forventningene at UDP skulle ha lavere ressursforbruk enn TCP, men noe overraskende viser resultatene at dette ikke stemmer. Dette skyldes at TCP bruker `Nagel`-algoritmen som samler opp flere mindre dataelemeter til en større pakke. Med TCP-MTU-pakkestørrelse (1448 byte payload) ved bruk av `read` og `send` bruker UDP 6.9 sekunder CPU-tid ved overføring av en fil på 1 GB, mens TCP uten `Nagel`-algoritmen bruker 6.8 sekunder CPU-tid og med `Nagel`-algoritmen bruker 6.6 sekunder CPU-tid. Figurene 4.13 og 4.14 viser et utsnitt av antall CPU-cykler for pakkene 10.000 til 20.000 ved bruk av UDP og TCP med `Nagel`-algoritmen og payloadstørrelse 1448 byte. Som vi ser er antall cykler noe høyere for UDP enn for TCP. Med payload på 1400 byte bruker UDP 6.9 sekunder



Figur 4.13: Antall CPU-cykler for `send` ved bruk av TCP og UDP



Figur 4.14: Utsnitt av lave verdier i figur 4.13

CPU-tid, mens TCP bruker 8.6 sekunder CPU-tid uten `Nagel`-algoritmen, og 6.4 sekunder CPU-tid med `Nagel`-algoritmen. Vi ser dermed at grunnen til at TCP har lavere verdier enn UDP, er at TCP bruker `Nagel`-algoritmen som samler data til en større MTU-pakke. TCP buffrer data på transportlaget til det er nok data til en MTU-pakke, mens for UDP vil pakken bygges opp med UDP- og IP-headere og plasseres i enhetsdriverens kø. Med payload på 1448 byte vil dataene sendes med en gang også for TCP, og vi ser at prosesseringstiden blir nesten den samme

for TCP med og uten `Nagel`-algoritmen og for UDP.

Disse eksperimentene viser at i et medianedlastingsscenario, hvor ingen databerøringsoperasjoner, ingen applikasjonlagsheadere eller timing er nødvendig, er `sendfile` det systemkallet med lavest resursforbruk. `sendfile` gir en stor ytelsesforbedring sammenlignet med de tradisjonelle `read` og `send` systemkallene, spesielt ved bruk av TCP hvor bare et systemkall er nødvendig for å overføre en hel fil. For TCP ser det ut til at `sendfile` er optimalt implementert, og ingen ytterligere forbedringer er mulig i forhold til redusering av antall kopioperasjoner og kontekstbytter. For UDP er maks datastørrelse som kan sendes for hvert systemkall begrenset av maks pakkestørrelse for UDP pakker, som er 64KB. Det er dermed mulig å lage en mer effektiv implementasjon hvor antall systemkall reduseres til ett i likhet med TCP. Dette kan gjøres ved at `sendfile` utvides med et parameter som angir datastørrelsen for pakkene, slik at oppsplittingen dermed kan utføres i kjernen. Men på grunn av at UDP vanligvis ikke er aktuell i et medianedlastingsscenario er dette en unødvendig utvidelse.

I eksperimentene er det bare målt CPU-tid, men på grunn av at det ikke er noen kopioperasjoner vil `sendfile` også legge mindre beslag på systembussen og bruke mindre minne enn `read` og `send`, noe som øker systemytelsen. På grunn av at det ikke er flere kopier av dataene i minnet, vil dataene bli liggende lengre tid i `page cache`. Dette vil øke muligheten for at andre prosesser kan gjenbruke dataene, som igjen vil redusere antall diskaksesser og dermed øke ytelsen for systemet.

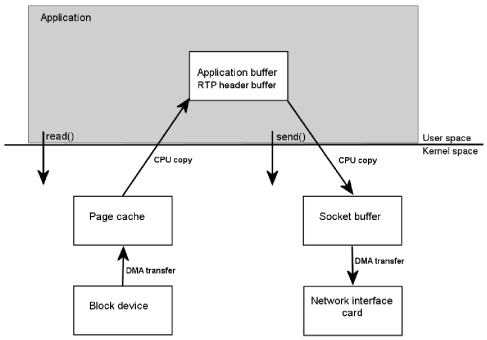
4.3.7 RTP-streaming

systemkall	ko	kb	kall til kjernen
`read send`	2*n*	4*n*	*n* **`read`** og *n* **`send`** kall (RTP headeren plasseres i applikasjonsnivåbufferet foran payloaden)
`read writev`	3*n*	4*n*	*n* **`read`** og *n* **`writev`** kall (RTP headeren blir generert i eget buffer, *writev* skriver data fra to buffere)
`mmap send`	2*n*	2+8*n*	1 **`mmap`**, *n* **`cork`**, *n* **`send`**, *n* **`send`** og *n* **`uncork`** kall (trenger et `send` kall for både data og RTP header)
mmap writev	2*n*	2+2*n*	1 **`mmap`** og *n* **`writev`** kall
`sendfile`	*n*	8*n*	*n* **`cork`**, *n* **`send`**, *n* **`sendfile`** og *n* **`uncork`** kall

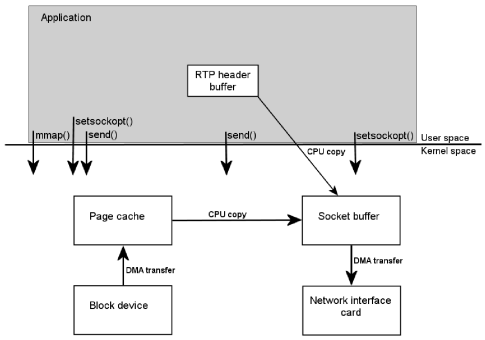
ko = antall `kopioperasjoner`, kb = antall `kontekstbytter`, *n* = antall pakker

Tabell 4.13: Oversikt over streamingytelsestestene

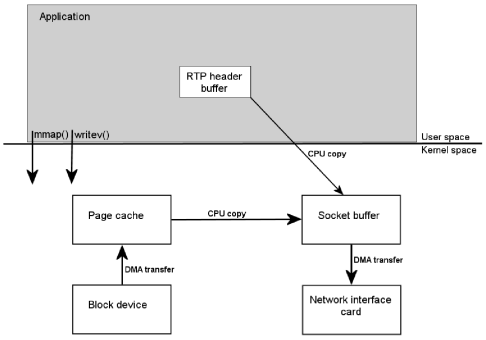
Vi skal nå se på disk-nettverksdatastiene i et mediastreamingscenario. Streaming av tidsavhengige data som audio og video krever vanligvis applikasjonlagskontroll, og ren filoverføring er ikke tilstrekkelig. Applikasjonslaget må generere hea-



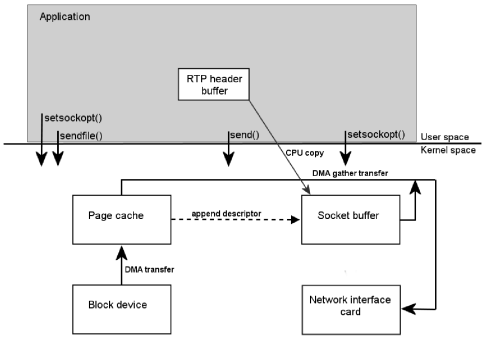
Figur 4.15: RTP-streaming med `read` og `send`



Figur 4.16: RTP-streaming med `mmap` og `send`



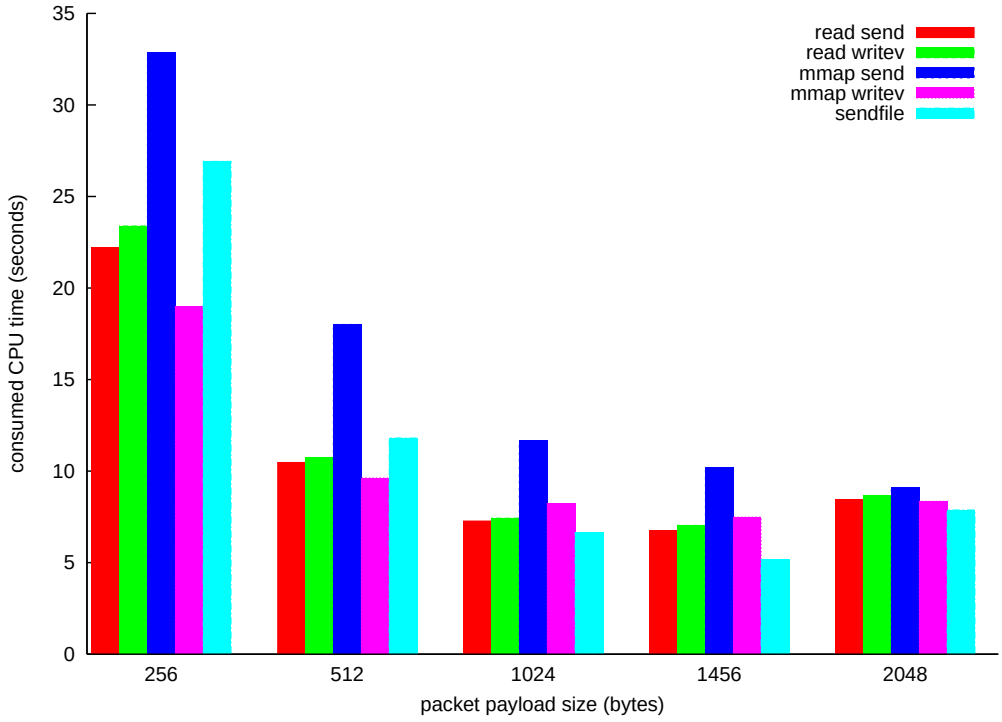
Figur 4.17: RTP-streaming med `mmap` og `writev`



Figur 4.18: RTP-streaming med `sendfile`

dere med sekvensnummer og timinginformasjon til hver pakke, og regulere overføringshastigheten. I en del tilfeller er det også behov for databerøringsoperasjoner. For å evaluere ytelsen utførte vi eksperimenter med systemkallene `read`, `mmap`, `send`, `writev` og `sendfile`. Kombinasjonen `read` og `send` og kombinasjonen `mmap` og `send` har en datasti som muliggjør databerøringsoperasjoner, men med `sendfile` er dette ikke mulig. Mediadataene kan overføres fra disk til nettverk både med og uten bruk av applikasjonsbuffere, avhengig av hvilke systemkall som benyttes, men RTP-headerne må genereres i et applikasjonsbuffer og kopieres til kjernen. Systemkall og datasti ved streaming med `read` og `send` er illustrert i figur 4.3.7, `mmap` og `send` i figur 4.3.7, `mmap` og `writev` i figur 4.3.7 og `sendfile` i figur 4.3.7. For kopiering av RTP-headere til kjernen brukes `send`, og for at RTP-headere og mediadata skal sendes i samme pakke må socketen stenges og åpnes av applikasjonen, noe som krever to ekstra systemkall. Til dette benyttes

systemkallet `setsockopt` med cork-verdiene 0 eller 1. For `read` og `send` kan samme buffer brukes for RTP-header og mediadata, og begge deler kan overføres i et kall. For `mmap` er `writev` et alternativ til `send`. `writev` gjør det mulig å sende data fra flere applikasjonsbuffere i et systemkall og bruk av `setsockopt` for å stenge socketen er ikke nødvendig. Tabell 4.13 viser antall kopioperasjoner og kontekstbytter for de forskjellige streamingeksperimentene. Alle eksperimentene er utført med transportprotokollen UDP.



Figur 4.19: RTP-streamingoperasjoner med UDP

Figur 4.19 og tabell 4.14 viser resultatene for streamingeksperimentet. Resultatene viser at `sendfile` ikke gir like store ytelsesfordeler i et streamingscenario som i et medianedlastingscenario, sammenlignet med de tradisjonelle systemkallene `read` og `send`. Dette skyldes at det er nødvendig med fire systemkall for å sende en pakke, mens det i nedlastingssammenheng bare er nødvendig med to for overføring av en hel fil. `sendfile` er på det beste 23 prosent bedre enn `read` og `send`, men for små datastørrelser medfører alle de ekstra kontekstbyttene at `sendfile` er et dårlig alternativ. For streaming som krever databerøringsoperasjoner er `sendfile` ikke mulig å bruke. For denne type streaming er det kombinasjonen `read` og `send`, og kombinasjonen `mmap` og `writev` som er aktuelle.

syscall	size byte	average sec	median sec	min sec	max sec	stdev
read/	256	22.200825	22.223121	21.484733	22.601564	0.305954
send	512	10.472308	10.512402	10.105463	10.637383	0.151605
	1024	7.275694	7.308390	6.851958	7.538854	0.189112
	1456	6.754773	6.796467	6.429022	6.858958	0.126900
	2048	8.425419	8.503208	7.868803	8.721675	0.230035
read/	256	23.390644	23.387944	23.172477	23.547421	0.101501
writev	512	10.735568	10.721371	10.659379	10.853350	0.069610
	1024	7.430270	7.396376	7.206904	7.603843	0.123475
	1456	7.032631	7.045430	6.894953	7.184907	0.094613
	2048	8.672181	8.601192	8.434718	9.015629	0.195633
mmap/	256	32.856605	32.914997	32.438068	33.331933	0.265602
send	512	18.014761	17.937773	17.796293	18.430199	0.223935
	1024	11.655828	11.682724	11.459258	11.799206	0.101963
	1456	10.199949	10.201450	10.107464	10.381422	0.079048
	2048	9.076920	9.081619	8.982634	9.163607	0.056192
mmap/	256	18.980315	18.995113	18.755150	19.196081	0.177901
writev	512	9.622337	9.649033	9.399571	9.815508	0.163922
	1024	8.213951	8.245247	7.801813	8.635687	0.265761
	1456	7.480563	7.485363	7.339884	7.588845	0.078949
	2048	8.346631	8.356729	8.146762	8.516705	0.111168
sendfile	256	26.915708	27.000396	26.135026	27.230860	0.334880
	512	11.807905	11.836200	11.380269	11.990177	0.173514
	1024	6.625093	6.681984	6.300041	6.746974	0.141983
	1456	5.171014	5.166715	4.893256	5.400179	0.130651
	2048	7.874503	7.926796	7.460867	8.005783	0.159748

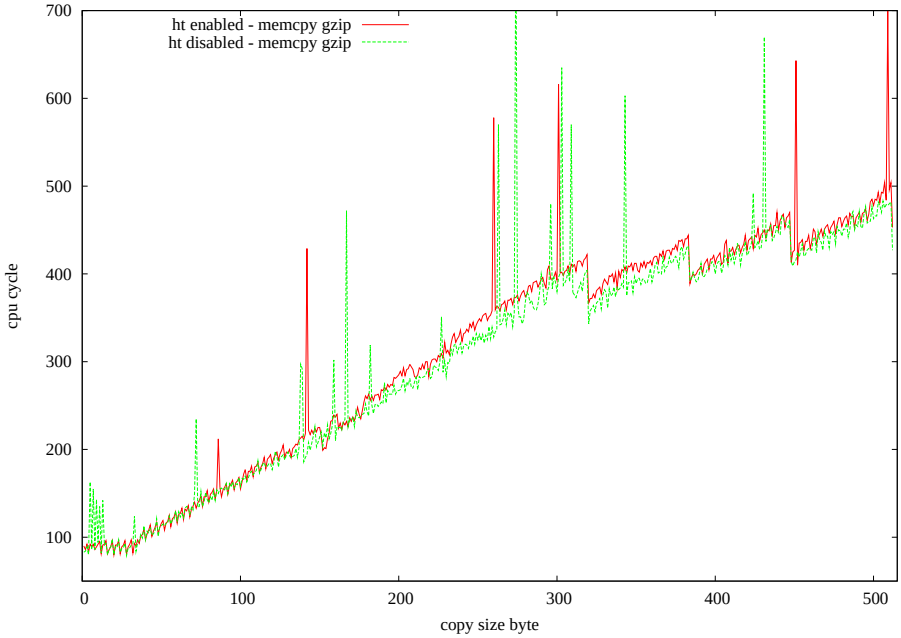
Tabell 4.14: RTP-streamingoperasjoner med UDP

Forskjellen på disse med hensyn på CPU-bruk er ikke så stor, og med MTU-størrelse er forskjellen 9 prosent, hvor kombinasjonen `read` og `send` er det beste alternativet. Forskjellen mellom kombinasjonen `read` og `send` og kombinasjonen `read` og `writev` er små, og forskjellene skyldes at `send` har én kopieringsoperasjon, mens `writev` har to, én for RTP-header og én for mediadata. Resultatene viser at antall kontekstbytter har stor betydning. Sammenligning av kombinasjonen `mmap` og `send` og kombinasjonen `mmap` og `writev`, hvor antall systemkall er 4 i det første tilfellet og 1 i det andre er forskjellene ved MTU-størrelse 26 prosent selv om antall kopieringsoperasjoner er det samme. Det er derfor rom for store forbedringer av `sendfile` ved å lage en `sendfile`-variant hvor det er mulig å overføre headere i samme systemkall som mediadataene. Ser vi på `mmap` og `send`

sammenlignet med `sendfile`, som har det samme antall kontekstbytter, ser vi at også fjerning av kopioperasjoner gir en stor ytelsesforbedring.

4.3.8 Hyperthreadeksperiment

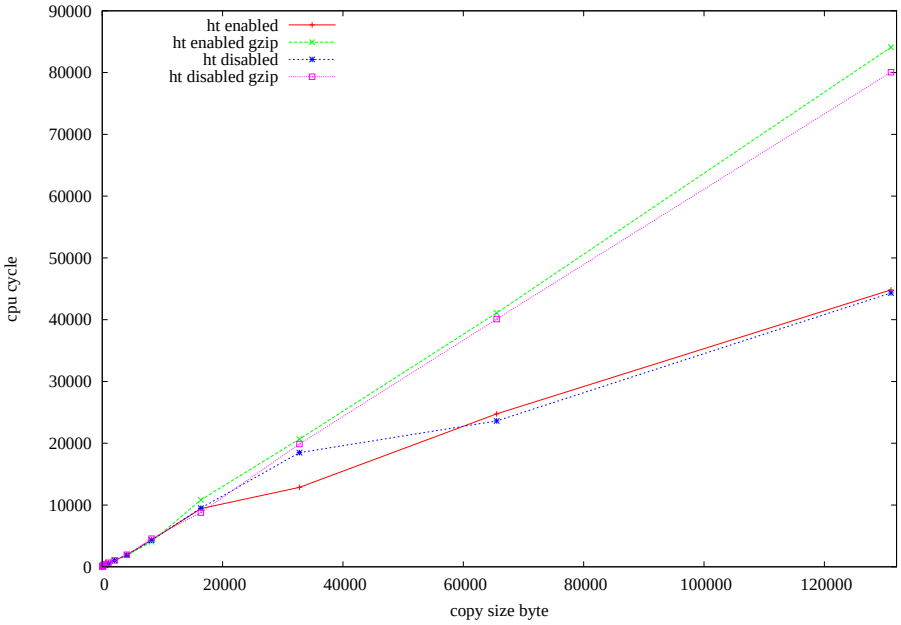
Eksperimenter tidligere i kapittelet har vist at kopioperasjoner er kostbare med hensyn på CPU-bruk, og fjerning av kopioperasjoner ved overføring av data fra disk til nettverk gir økt ytelse for systemet. Den frigjorte CPU-tiden kan for eksempel benyttes til å overføre flere samtidige mediastrømmer, eller til kryptering av datapakkene som overføres. Nyere prosessorer benytter en teknikk som kalles "hyperthreading", som i enkelte tilfeller tillater to tråder å kjøre samtidig på en enkeltkjerne prosessor. For at dette skal være mulig må de to trådene benytte forskjellige deler av prosessoren, de kan ikke utføre de samme instruksjonene samtidig.



Figur 4.20: Hyperthreadeksperimenter med memcpy og gzip

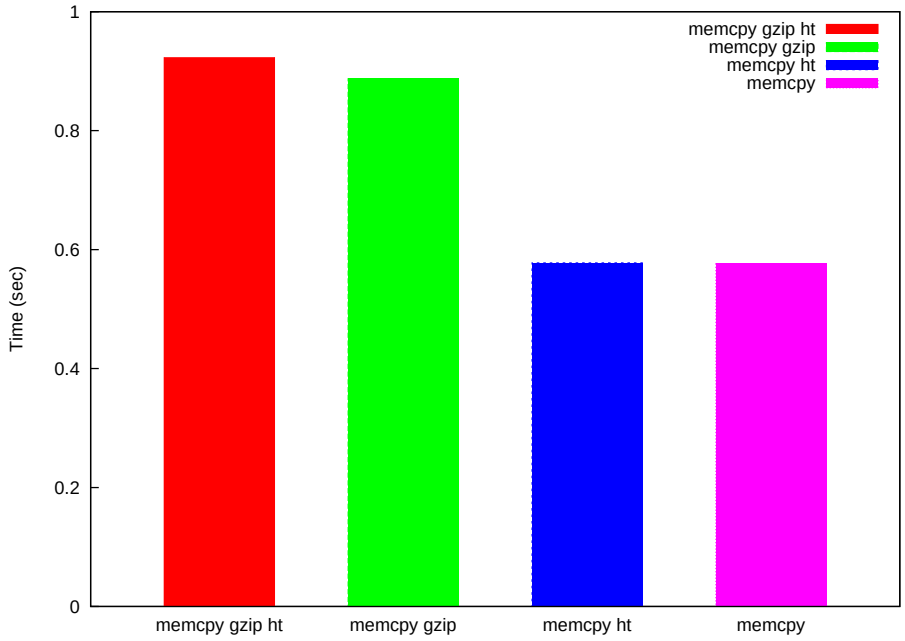
I denne seksjonen ønsker vi å undersøke om kopioperasjoner og kryptering kan utføres samtidig på en "hyperthreaded"-prosessor. For å teste dette er kopitesten beskrevet i seksjon 4.3.4 utført samtidig som `gzip` komprimerer en fil. `gzip` er valgt fordi den bruker de samme instruksjonene som brukes ved kryptering. I

dette eksperimentet er det benyttet en Intel Pentium 4 CPU 3.00GHz med Hyper-Threading. Kopitesten ble utført med og uten gzip og med og uten "hyperthreading". For å utføre testene uten "hyperthreading" ble "hyperthread" støtten i scheduleren fjernet ettersom dette ikke var mulig i BIOS. Dette krever rekompilering av kjernen. For å finne ut om kopioperasjoner og kryptering kan kjøres parallelt målte vi antall CPU-cykler som benyttes for hver kopioperasjon, og hvis antall cykler er høyere når "hyperthreading" er slått, av tyder det på at kopioperasjoner og kryptering kan kjøres parallelt. For å måle antall cykler ble instruksjonen `rdtsc` benyttet. Tiden det tar å utføre testene er også målt ved å benytte systemkallet `gettimeofday` før og etter testene.



Figur 4.21: Hyperthreadeksperimenter - CPU-cykler

Figur 4.20 viser resultatene for kopieksperimentene med dataelementer fra 1 til 512 byte. Begge eksperimentene er utført sammtidig med `gzip`. Det første eksperimentet er utført uten "hyperthread"-støtte i scheduleren, mens det andre er utført med "hyperthread"-støtte. Som vi kan se er resultene for begge eksperimentene nesten identiske, noe som tyder på at kopiering og kryptering ikke kan utføres parallelt. Hvis de kunne utføres parallelt ville antall cykler vært halvvert ved bruk av "hyperthread". For noen av kopioperasjonene er antall cykler litt høyere med "hyperthread"-støtte enn uten, som kan skyldes at "hyperthreading" gir noe høyere svitsjingoverhead.



Figur 4.22: Hyperthreadeksperimenter - klokketid

I figur 4.21 og tabell 4.15 ser vi resultatene for kopieksperimentene med dataelementer fra 1 til 131.072 byte. Vi ser at antall cykler er identiske eller litt høyere også for større dataelementer. Det er også utført to kopieksperimenter uten `gzip`, med og uten "hyperthread"-støtte. Som vi ser er antall cycler for testene med `gzip` nesten dobbelt så høyt som for de uten `gzip`, dette fordi komprimering og kopiering ikke kan kjøres parallelt.

Figur 4.22 viser tiden målt med `gettimeofday` for de fire eksperimentene. Som vi ser er disse resultatene i samsvar med det vi så ved måling av antall cykler. For kopieksperimentene med `gzip` bruker testen hvor "hyperthread" er støttet 4 prosent lenger tid enn med "hyperthread" skrudd av. Dette skyldes som nevnt tidligere at "hyperthreading" gir noe høyere svitsjingoverhead. Forskjellen i tid for eksperimentene med og uten `gzip` er 35 og 40 prosent.

"Hyperthread"-eksperimentene tyder på at kopioperasjoner og kryptering ikke kan utføres parallelt på en prosessor med "hyperthreading". Det kan derfor være nyttig å fjerne kopioperasjoner ved overføring av data fra disk til nettverk og for eksempel bruke de frigjorte ressursene til for eksempel kryptering.

4.4 Oppsummering

I dette kapitlet har vi sett på eksisterende mekanismer i Linux for nedlasting og streaming. Vi startet med noen innledende eksperimenter hvor vi så på ytelsesforskjeller mellom filsystemene `ext3`, `xfs` og `reiser` versjon 3 i nedlasting- og streamingsammenheng, og så videre på ytelsesforbedringer fra Linux 2.4 til 2.6.

Filsystemeksperimentene viste at det ikke er så stor forskjell i CPU-bruk mellom `xfs` og `ext3`, hvor `xfs` er det beste alternativet for `sendfile`, og `ext3` er best for de andre systemkallene for overføring av data fra disk til nettverk. Når det gjelder `reiser` er det dårligst for nesten alle systemkallene, og forskjellene til `xfs` og `ext3` var i noen tilfeller store. Vi så videre at forbedringene i Linux 2.6 fra Linux 2.4 ga store reduksjoner i CPU-bruk ved både nedlasting og streaming. Endringene som har betydning for eksperimentene vi utførte, er forbedringer i kopirutinen, `readahead`-koden og TCP-stacken.

Etter de innledende eksperimentene så vi på en ytelsessammenligning av systemkallene `read`, `mmap`, `send` og `sendfile` i et medianedlastingsscenario. Eksperimentene ble utført med både TCP og UDP, hvor hensikten var å sammenligne disse transportprotokollene, i tillegg til en sammenligning av systemkallene. Noe overraskende viste resultatene at TCP har lavere CPU-bruk enn UDP, og dette skyldes at TCP samler flere mindre dataelementer til en større pakke med MTU-størrelse. Når det gjelder sammenligningen av systemkallene så vi at `sendfile` er det beste alternativet både ved bruk av TCP og UDP. Med MTU-pakkestørrelse ved bruk av TCP er forskjellen i CPU-bruk mellom `sendfile` og kombinasjonen `read` og `send` 65 prosent, og ved bruk av UDP er forskjellen 40 prosent. `mmap` og `send` har noe høyere CPU-bruk enn `read` og `send`, noe som skyldes mange "page faults". `sendfile` har ingen kopioperasjoner, og er dermed det beste alternativet også med tanke på andre delte resurser som systembuss og minne. Det kan derfor se ut til at `sendfile` ved bruk av TCP er optimalt implementert for nedlastingsoperasjoner, og det vil derfor sannsynligvis ikke lønne seg å bruke tid på ytterligere optimaliseringer.

Vi så videre på en sammenligning av systemkallene `read`, `mmap`, `send`, `writev` og `sendfile` i et streamingscenario. Med MTU-pakkestørrelse viste resultatene at `sendfile` er det beste alternativet, og forskjellen i CPU-bruk mellom `sendfile` og kombinasjonen `read` og `send` er 23 prosent. Forskjellen er ikke like stor som ved nedlasting, og skyldes at `sendfile` ikke er optimalt implementert for streaming, og derfor krever mange kontekstbytter for hver RTP-pakke som overføres. I et streamingscenario er det ofte nødvendig med databerøringsoperasjoner, noe som gjør `sendfile` til et uaktuelt alternativ, ettersom dataene overføres innenfor kjernen. I et slikt scenario så vi at `read` og `send` er det beste alternativet

med tanke på CPU-bruk, hvor forskjellen til `mmap` og `writev` er 9 prosent med MTU-pakkestørrelse. Bruk av `read` istedenfor `mmap` er et dårligere alternativ med tanke på systembussen og utnyttelse av minnet, ettersom `read` fører til en ekstra kopiering av dataene. En av grunnene til at `read` har lavere CPU-bruk enn `mmap`, er at `mmap` ikke bruker `readahead` og dermed får et stort antall "page faults". Ved bruk av `readahead` på det mappede minneområdet reduseres CPU-bruken med 13 prosent, noe som gjør `mmap` og `writev` til det beste alternativet også med tanke på CPU-bruk. I et streamingscenario er derfor `mmap` og `writev` det beste alternativet.

Kopi- og kontekstbytteeksperimentene viste at kopioperasjoner og kontekstbytter er kostbare med tanke på CPU-bruk. Kopioperasjoner medfører også økt datatrafikk på systembussen, og dårligere utnyttelse av minnet ettersom dataene lagres på flere steder. Profil resultatene viste at ved overføring av data fra disk til nettverk med kombinasjonen `read` og `send`, og kombinasjonen `mmap` og `writev` utgjør kopioperasjonene 40 prosent av CPU-bruken. Dette viser at det er rom for store reduksjoner i CPU-bruk i et streamingscenario ved å fjerne kopioperasjonene.

Ved bruk av `mmap` er det mulig å overføre mediadataene til applikasjonen uten kopioperasjoner, men overføring videre til nettverk er ikke mulig uten kopioperasjoner med eksisterende systemkall. For å få til en optimal overføring av mediadata fra disk til nettverk ved streaming, er det derfor nødvendig å optimalisere `writev` ved å fjerne kopioperasjonen. Med denne optimaliseringen vil det være mulig å overføre mediadata fra disk til nettverk uten kopioperasjoner, og med bare to kontekstbytter for hver pakke som overføres. Ved streaming av mediadata der det ikke er nødvendig med databerøringsoperasjoner, kan en optimalisert versjon av `sendfile` være et bra alternativ. Problemet med `sendfile` er at det er nødvendig med 8 kontekstbytter for hver pakke som overføres, men ved å utvide `sendfile` med parametere for applikasjonsbuffere er det mulig å redusere dette til 2 kontekstbytter for hver pakke. Med denne forbedringene vil det være mulig å overføre mediadata fra disk til nettverk uten kopioperasjoner, og med bare to kontekstbytter for hver pakke. Forbedringene foreslått for `writev` og `sendfile` krever fortsatt et kall til kjernen for hver pakke som overføres, men ved å flytte koden for timing og oppbygging av applikasjonslagsheadere til kjernen kan disse fjernes.

Videre i neste kapittel skal vi se på muligheten for å implementere de foreslåtte forbedringene, for `writev` og `sendfile`, for å bedre støtte streaming i Linux. Vi skal også se på muligheten for å flytte timing og oppbygging av applikasjonslagsheadere til kjernen.

Med hyperthreadstøtte uten gzip				Med hyperthreadstøtte med gzip			
size byte	**average cycle**	**min cycle**	**max cycle**	**size byte**	**average cycle**	**min cycle**	**max cycle**
1	69	60	210	1	90	75	151
2	71	60	278	2	86	75	203
4	66	53	248	4	83	75	263
8	76	53	833	8	86	75	436
16	67	53	173	16	81	68	210
32	67	53	173	32	82	75	128
64	109	90	293	64	120	105	166
128	174	150	323	128	191	173	248
256	343	300	555	256	347	315	915
512	413	368	1275	512	453	428	1058
1024	620	547	1043	1024	713	592	1613
2048	1032	885	1965	2048	1020	885	1695
4096	1906	1553	4957	4096	1873	1560	3225
8192	4328	3420	14475	8192	4054	3427	8648
16384	9424	8392	27862	16384	10836	9824	32745
32768	12855	10709	37747	32768	20672	18480	53588
65536	24757	21165	114922	65536	41092	33885	138442
131072	44822	41813	80566	131072	84100	72667	106748

Uten hyperthreadstøtte uten gzip				Uten hyperthreadstøtte med gzip			
size byte	**average cycle**	**min cycle**	**max cycle**	**size byte**	**average cycle**	**min cycle**	**max cycle**
1	72	59	419	1	83	74	180
2	72	52	247	2	84	74	187
4	61	52	172	4	80	67	202
8	64	44	187	8	86	67	277
16	68	52	330	16	80	67	232
32	68	44	607	32	92	67	495
64	106	89	300	64	122	104	255
128	69	67	180	128	188	164	555
256	234	232	240	256	325	292	547
512	286	277	413	512	427	382	773
1024	456	389	4972	1024	658	547	1568
2048	1047	885	1988	2048	1039	877	2407
4096	1907	1538	4080	4096	1950	1552	3180
8192	4301	3374	9907	8192	4560	3487	11992
16384	9496	8265	29895	16384	8772	8363	19583
32768	18477	16770	42510	32768	19879	17107	41887
65536	23614	20940	119370	65536	40095	34801	133478
131072	44320	41820	78240	131072	80063	69405	99248

Tabell 4.15: Hyperthreadeksperimenter - CPU-cykler

Kapittel 5

Streamingutvidelser for Linux

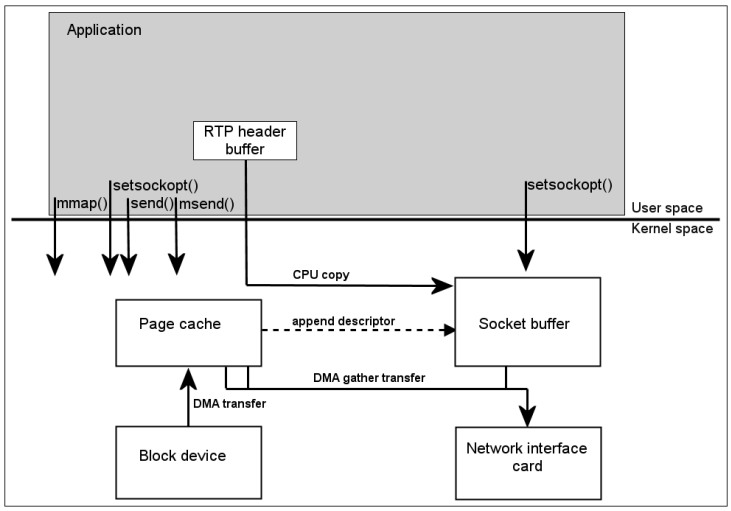
Kopi- og kontekstbytteeksperimentene i forrige kapittel viste at kopioperasjoner og kontekstbytter er kostbare, med tanke på CPU-bruk og andre delte ressurser som systembuss og minne. For tradisjonell nedlasting, som ikke krever databerøringsoperasjoner, ingen applikasjonsgenererte headere eller timing støtte, ser vi at `sendfile` er effektivt implementert. For TCP er `sendfile` optimal med hensyn på kontekstbytter, og det er heller ingen kopioperasjoner involvert ved overføring av data fra disk til nettverk. Det vil derfor sannsynligvis ikke lønne seg å bruke mye tid på å forbedre denne. I streaming sammenheng er ikke `sendfile` like effektivt implementert og krever mange kontekstbytter. `sendfile` tillater heller ikke databerøringsoperasjoner som er nødvendig i mange streamingsammenhenger. De andre aktuelle systemkallene som `read`, `mmap`, `send` og `writev` muliggjør databerøringsoperasjoner og krever færre kontekstbytter, men innebærer kopioperasjoner.

I dette kapitlet skal vi se om det er mulig å redusere antall kontekstbytter og fjerne kopioperasjonene. I seksjon 5.1 presenteres nye systemkall for streaming, hvor vi har redusert antall kontekstbytter for `sendfile`, og implementert et nytt `send` systemkall uten kopioperasjoner. Videre i seksjon 5.2 presenteres to prototyper, hvor genereringen av RTP-headere er flyttet ned i kjernen, for å ytterligere redusere antall kontekstbytter. I seksjon 5.3 presenteres resultatene for de nye systemkallene og prototypene, før implementasjonen av RTP-motorene (RTP engines) presenteres i seksjon 5.4. Resultatene fra RTP-motoreksperimentene presenteres i seksjon 5.5, før oppsummeringen i seksjon 5.6.

5.1 Systemkall optimalisert for streaming

For å bedre støtte streaming og for å kunne legge til applikasjonsinformasjon som RTP-headere har vi implementert flere nye systemkall for å se om vi kan bedre ytelsen. Disse er `msend`, `rtpmsend` og `rtpsendfile`, og er beskrevet nedenfor.

5.1.1 `msend`



Figur 5.1: Datasti for `mmap` og `msend`

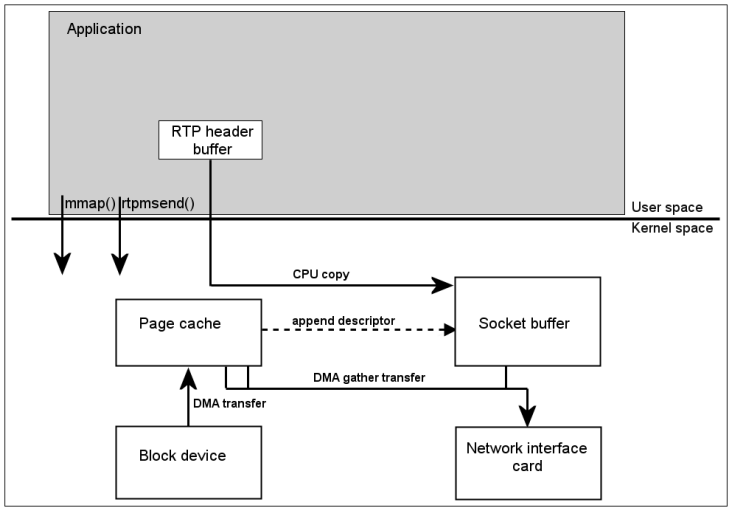
sys_msend(int fd, unsigned long buffaddr, size_t len, unsigned flags);

`msend`, illustrert i figur 5.1, er en forbedret versjon av systemkallet `send` hvor kopioperasjonen er fjernet. Parameterne til systemkallet er socket-deskriptoren, startadressen til bufferet som skal sendes og lengden på dataene. Det er også et flag-parameter som er identisk med flag-parameteret til `send`. `msend` brukes i kombinasjon med `mmap`. Når en fil mappes med `mmap` er det virtuelle minneområdet som opprettes delt mellom applikasjonen og kjernen. Dataene som tilhører dette minneområdet ligger i filsystemets `page cache` i kjernen, og minneområdet er mappet i applikasjonens sidetabeller. Ettersom sidene som inneholder dataene er tilgjengelige i hele kjernen, er det ingen grunn til å kopiere dataene fra `page cache` til kommunikasjonssystemets `socket buffer`. Den forbedrede `send` metoden bruker den lineare adressen[1] for å finne det virtuelle minneområdet, og dermed sidebeskrivelsen(e) til siden(e) i `page cache`, der datene som skal overføres

[1] lineare adresse er Intels betegnelse for virtuell adresse

til nettverkskortet ligger. Referansen til sidene kopieres til kommunikasjonssystemets `socket buffer` ved hjelp av `sendpage`-metoden i kjernen. `sendpage` sørger for, i tillegg til overføring av referanser, at det opprettes UDP-headere når det er nødvendig. Dataene overføres deretter til nettverkskortets minne fra filsystemets `page cache` i en DMA-operasjon. `mmap` og `msend` har den samme datastien for overføring av data fra disk til nettverk som `sendfile` (figure 4.7), og dermed ingen kopioperasjoner og det samme antall kontekstbytter. Men til forskjell fra `sendfile` har applikasjonen mulighet til databerøringsoperasjoner, noe som ofte er en forutsetning for å få nok informasjon om mediadatene til å kunne bygge opp for eksempel RTP-headere. `msend` er implementert med tanke på at applikasjonen bare skal lese dataene i det mappede minneområdet. Applikasjonen vet ikke når dataene er overført fra `page cache` til nettverkskortet, og en skriveoperasjon kan endre dataene etter at referansen er overført til `socket buffer`, men før de er overført til nettverkskortet. Leseoperasjoner er vanligvis tilstrekkelig i streamingsammenheng, men for å støtte for eksempel kryptering av dataene må implementasjonen av `mmap` og `page cache` endres.

5.1.2 `rtpmsend`



Figur 5.2: Datasti for `mmap` og `rtpmsend`

sys_rtpmsend(int fd, void __user * rtpbuff, unsigned long buffaddr, size_t len, size_t rtplen, unsigned flags);

`rtpmsend`, illustrert i figur 5.2, er en utvidelse av `msend` som gir muligheter til å legge til applikasjonsgenererte headere. Parameterne til `rtpmsend` er som for

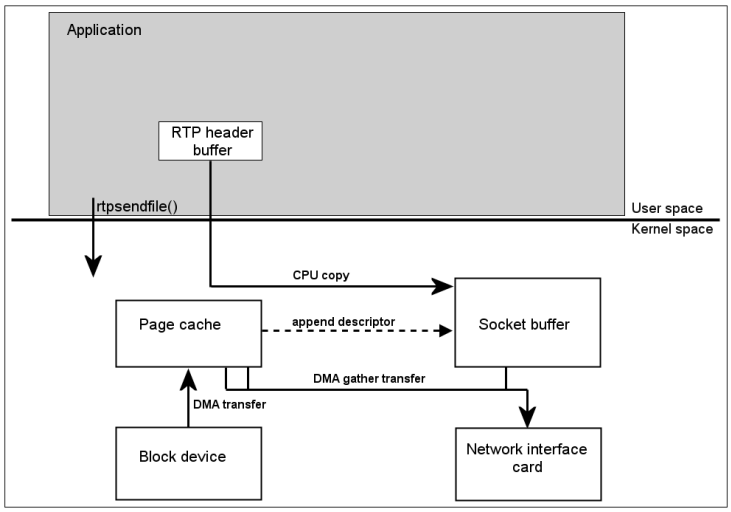
msend utvidet med peker til RTP-headerbufferet og lengden på RTP-headeren. I tillegg til msend-koden, benyttes metoden sys_sendto for å kopiere RTP-headerene fra applikasjonsbufferet til kommunikasjonssystemets socket buffer. sys_sendto er systemkallet sendto kalt fra kjernen. For at RTP-header og mediadata skal sendes i samme pakke, benyttes MSG_MORE flagget som et alternativ til setsockopt for "corking" av socketen. Deretter overføres pakken til nettverkskortet i en DMA-operasjon. For at dette skal være mulig, må nettverkskortet støtte DMA-samleoperasjoner, ettersom alle headerne overføres fra kommunikasjonssystemets socket buffer, mens mediadataene overføres fra filsystemets page cache. rtpmsend bruker fortsatt en kopioperasjon for å overføre RTP-headerne fra applikasjonsbufferet til kommunikasjonssystemets socket buffer. For fjerne denne siste kopioperasjonen kan applikasjonsbufferet opprettes som et anonymt minneområde[2] ved hjelp av mmap istedenfor å opprettes med malloc. På denne måten vil det være mulig å bruke sendpage-metoden istedenfor send_to og bare referansen til RTP-headeren blir overført til socket buffer. Men ettersom RTP-headerne vanligvis utgjør mindre enn 1 prosent av dataene som overføres vil dette ikke gi stor reduksjon i CPU-bruk. Sending av en RTP-pakke med det implementerte systemkallet rtpmsend fører til to kontekstbytter og en kopioperasjon, i motsetning til msend som ved RTP-streaming vil gi åtte kontekstbytter og det samme antall kopioperasjoner.

5.1.3 rtpsendfile

sys_rtpsendfile(int out_fd, int in_fd, off_t __user *offset, size_t count, void __user * buff, size_t len);

rtpsendfile, illustrert i figur 5.3, er en utvidelse av sendfile som gjør det mulig å legge til applikasjonsgenererte headere. Parameterne er som for sendfile utvidet med en peker til applikasjonsbufferet med RTP-headere og lengden på RTP-headeren. Dette reduserer antall kontekstbytter for sending av en pakke fra åtte til to. rtpsendfile benytter seg i likhet med rtpmsend av sys_sendto metoden for å overføre RTP-headere fra applikasjonsbufferet til kommunikasjonssystemets socket buffer. For overføring av mediadata fra disk til nettverkskortet benyttes eksisterende sendfile-kode. For fjerning av kopioperasjonen i forbindelse med RTP-headeren kan den samme forbedringen beskrevet for rtpmsend benyttes.

[2] Et anonymt minneområde er et virtuelt minneområde som ikke mapper en fil.



Figur 5.3: Datasti for `rtpsendfile`

5.2 RTP-motorprototyper

Vi har nå fjernet kopioperasjonene, men ettersom headere genereres på applikasjonsnivå får vi fortsatt kontekstbytter. For å unngå dette kan oppbyggingen av headere flyttes fra applikasjonsnivå til kjernen. Vi skal i denne seksjonen se på to prototyper for RTP-motorer i kjernen. Disse ble implementert for å se om dette gir ytelsesforbedringer. Ettersom dette er prototyper er det bare overføringen av RTP-headere og mediadata som er implementert. RTP-headerne generert av disse prototypene er bare “dummy” headere, i likhet med RTP-headerne generert i eksperimentene i kapittel 4. For å kunne bygge opp RTP-headere er det nødvendig med informasjon om blant annet type mediadata, transmisjonsinterval, inkrementeringsverdi for tidsstemplet og synkroniseringskilde. Disse er tenkt overført fra applikasjonen til kjernen før streamingen starter.

5.2.1 `krtpmsend`

sys_krtpmsend(int in_fd, int out_fd, off_t offset, size_t len, size_t count, unsigned flags);

`krtpmsend` er en prototype for en RTP-motor i kjernen. Parameterne er deskriptorene til filen som skal streames og socketen, start-offset i filen, antall byte som skal streames, mediadatastørrelsen for hver pakke og flag. Mediadatastørrelsen er lik for alle pakkene, men ved streaming av enkelte typer mediadata kan den væ-

re varierende. krtpmsend bruker samme kode som rtpmsend, men istedenfor at filen mappes av applikasjonen gjøres dette i kjernen. I tillegg er kopioperasjonen for headere fjernet, ettersom de blir generert i kjernen. Det er dermed bare nødvendig å overføre referansene til headeren til kommunikasjonssystemets socket buffer. krtpmsend muliggjør også databerøringsoperasjoner som er nødvendig ved oppbygging av mediaspesifikke headere, som for eksempel MPEG Elementary Streams, se seksjon 2.3. I denne implementasjonen utføres ingen databerøringsoperasjoner ettersom det bare genereres "dummy"-headere. krtpmsend medfører ingen kopioperasjoner og bare to kontekstbytter ved streaming av en hel fil, i motsetning til rtpmsend som medfører to kontekstbytter per pakke.

5.2.2 krtpsendfile

sys_krtpsendfile(int out_fd, int in_fd, off_t __user *offset, size_t count, size_t len);

krtpsendfile er i likheten med krtpmsend en prototype for en RTP-motor i kjernen. Parameterne er som for sendfile utvidet med mediadatastørrelsen for pakkene. krtpsendfile er for RTP-headerne implementert på tilsvarende måte som krtpmsend, men overføring av mediadata fra disk til nettverk er basert på sendfile istedenfor mmap og sendpage. Denne prototypen er beregnet på streaming av multimediadata som ikke krever databerøringsoperasjoner for oppbygging av RTP-headere. Dette gjelder blant annet MPEG Program Stream, MPEG System Stream og MPEG Transport Stream, se seksjon 2.3. krtpsendfile medfører ingen kopioperasjoner og det samme antall kontekstbytter som krtpmsend.

5.3 Eksperimenter

Vi skal nå se på eksperimenter med de nye systemkallene for RTP-streaming og RTP-motorprototypene, og sammenligne disse med eksisterende systemkall. Testoppsettet for disse eksperimentene er beskrevet i seksjon 4.3.1, og UDP er benyttet i alle ytelsestestene. Først ser vi på profilresultatene for mmap og msend, før vi ser på ytelsesresultatene for systemkallene msend og rtpmsend, og prototypen krtpmsend. Videre ser vi på systemkallene og prototypen basert på sendfile, og avslutter med en sammenligning av nye og eksisterende systemkall.

5.3.1 Streaming med `writev`, `msend`, `rtpmsend` og `krtpmsend`

I disse eksperimentene skal vi se på de nye systemkallene `msend` og `rtpmsend` og prototypen `krtpmsend` . Først ser vi på en profil av `msend`, før vi ser på ytelsesresultatene for systemkallene.

systemkall	ko	kb	kall til kjernen
mmap writev	2*n*	2+2*n*	1 **mmap** og *n* **writev** kall
mmap msend†	*n*	2+8*n*	1 **mmap**, *n* ***cork***, *n* **send**, *n* **msend** og *n* ***uncork*** kall (ingen datakopi med **msend**, men RTP-header må kopieres fra applikasjonsbuffer med **send**)
mmap rtpmsend†	*n*	2+2*n*	1 **mmap** og *n* **rtpmsend** kall (**rtpmsend**† kopierer RTP-headere fra applikasjonsnivåbuffer og legger til mediadata fra **mmap**'ede filer i kjernen)
krtpmsend†	0	2	1 **krtpmsend** kall (**krtpmsend**† legger til RTP-headerere til **mmap msend** kombinasjonen (se over) i kjernen)

ko = antall *kopioperasjoner*, kb = antall *kontekstbytter*, *n* = antall pakker, † = nytt systemkall, cork og uncork er systemkallet `setsockopt`

Tabell 5.1: Oversikt over `mmap`-streamingytelsestestene

Tabell 5.2 viser at det ikke er noen kopioperasjoner mellom subsystemene, og som for `sendfile` sin profil i tabell 4.11 er det metodene `search_by_key` og `skb_copy_bits` som står for for mye av CPU-bruken. `msend`-metoden i `mmap_send`-imaget og `system_call`-metoden i `vmlinux`-imaget er de to andre metodene med høyt CPU-bruk. Disse to metodene er del av kontekstbyttene fra applikasjonens adresserom til kjernens adresserom, og den høye CPU-bruken kan tyde på at koden som brukes her ikke er like effektivt implementert som for `sendfile`. Til tross for dette kan det se ut til at fjerningen av kopioperasjonen vil gi redusert CPU-bruk sammenlignet med `send`.

Vi skal nå se på ytelsesresultatene for de nye systemkallene `msend`, `rtpmsend` og prototypen `krtpmsend`, sammenlignet med `writev`. Som vi ser i tabell 5.1 har `writev` to kopioperasjon for hver pakke, en for RTP-header og en for mediadata. `msend` og `rtpmsend` har en kopioperasjon for RTP-header og ingen for mediadata, mens `krtpmsend` ikke har noen kopioperasjoner. Når det gjelder kontekstbytter har `writev` og `rtpmsend` det samme antall for hver pakke. For `msend` er det åtte kontekstbytter for hver pakke, på grunn av at `msend` må brukes sammen med `send` for overføring av RTP-headere, og "corking" av socketen er nødvendig for at RTP-header og mediadata skal sendes i samme pakke. `krtpmsend` har bare to kontekstbytter for streaming av en hel fil på grunn av at RTP-headerene genereres i kjernen. Forventningene til disse eksperimentene, sett ut fra antall kontekstbytter og kopioperasjoner, er at `krtpmsend` skal ha det laveste ressursforbruket, mens `writev` skal ha det høyeste.

Tabell 5.3 og figur 5.4 viser testresultatene for de nye systemkallene. Det eksisterende systemkallet `writev` er brukt som sammenligningsgrunnlag, ettersom det

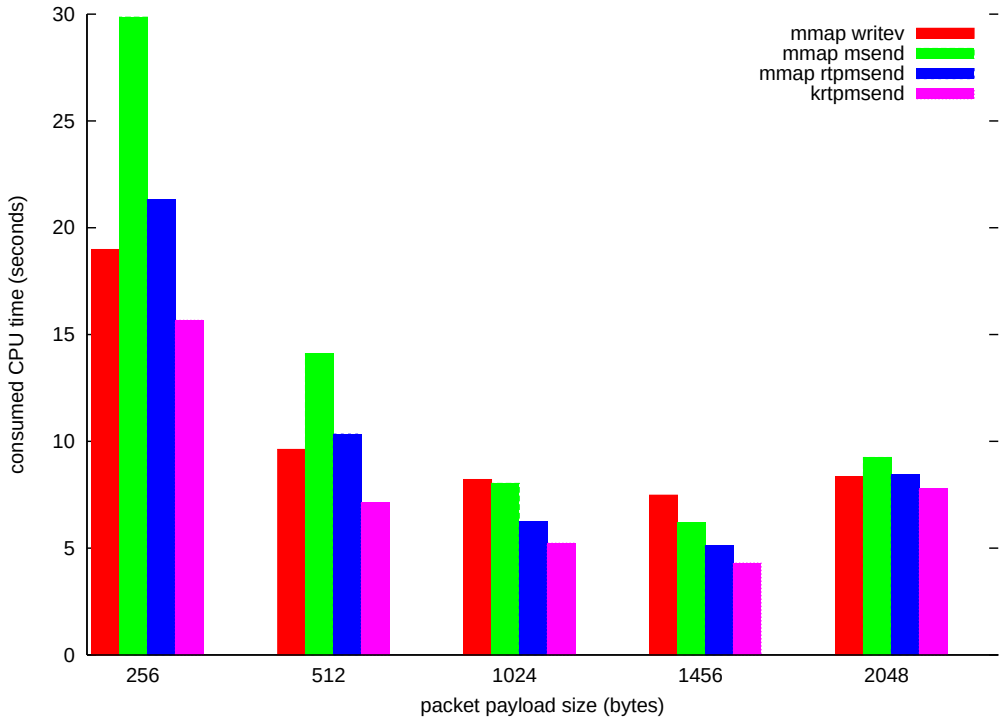
samples	%	image name	symbol name
442	8.7542	reiserfs.ko	search_by_key
392	7.7639	mmap_msend	msend
354	7.0113	vmlinux	skb_copy_bits
335	6.6350	vmlinux	system_call
196	3.8820	vmlinux	ip_push_pending_frames
194	3.8423	vmlinux	ide_outb
189	3.7433	vmlinux	kmap_atomic
157	3.1095	3c59x.ko	boomerang_start_xmit
142	2.8124	vmlinux	ip_append_data
113	2.2381	vmlinux	__kmalloc
109	2.1588	vmlinux	sock_alloc_send_pskb
102	2.0202	vmlinux	buffered_rmqueue
89	1.7627	vmlinux	ide_do_request
84	1.6637	vmlinux	follow_page
83	1.6439	vmlinux	ip_append_page
77	1.5251	vmlinux	udp_sendmsg
73	1.4458	vmlinux	__find_get_block
71	1.4062	vmlinux	__make_request
71	1.4062	vmlinux	kmem_cache_alloc
70	1.3864	vmlinux	local_bh_enable
67	1.3270	vmlinux	udp_sendpage

Tabell 5.2: Profil av `mmap` og `msend`

viste seg å være det beste av de eksisterende systemkallene i kombinasjon med `mmap`. For 1KB- og MTU-pakkestørrelse er imidlertid kombinasjonen `read` og `writev` bedre med tanke på CPU-bruk, men ved bruk av `madvise` er `mmap` og `writev` bedre også for disse pakkestørrelsene. En sammenligning av `read` og `send`, og de nye systemkallene kommer senere i dette kapitlet.

For pakkestørrelser mindre enn 1KB ser vi at `msend` og `rtpmsend` ikke gir noen ytelsesforbedringer, og at CPU-bruken er høyerer enn for `writev`. Dette er tilsvarende som for `sendfile`, som ikke har like lavt CPU-bruk sammenlignet med `mmap` og `writev` for mindre pakkestørrelser. Grunnen til at `msend` har høyere CPU-bruk enn `rtpmsend` er de ekstra kontekstbyttene.

Med 1KB- og MTU-pakkestørrelse har `msend` og `rtpmsend` lavere CPU-bruk samenlignet med `writev`. `rtpmsend` har en forbedring på over 30 prosent med MTU-pakkestørrelse, mens `msend` ikke har like store forbedringer. For pakkestørrelser over MTU ser vi tilsvarende resultater som for mindre pakkestørrelser, men



Figur 5.4: Streaming med `writev`, `msend`, `rtpmsend` og `krtpmsend`

forverringene er ikke like store.

Ser vi på `krtpmsend`, hvor alle kopioperasjonene og kontekstbytter er fjernet, har den lavest CPU-bruk uansett pakkestørrelse. For MTU-pakkestørrelse er forbedringene over 40 prosent.

Resultaene viser at fjerning av kopioperasjoner og redusering av kontekstbytter har stor betydning for ytelsen. I tillegg til å redusere prosessorbelastningen har dette stor betydning for andre delte ressurser som systembuss og minne. Grunnen til at CPU-bruken øker ved payload større enn MTU er at fragmentering av pakkene medfører ekstra overhead.

5.3.2 Streaming med `sendfile`, `rtpsendfile` og `krtpsendfile`

I dette eksperimentet skal vi se på `sendfile` systemkallene. Som vi ser i tabell 5.4 har `sendfile` og `rtpsendfile` bare kopioperasjoner i forbindelse med overføring av RTP-headere, mens prototypen `krtpsendfile` ikke har noen kopioperasjoner. De andre forskjellene mellom disse systemkallene er antall kontekstbyt-

syscall	size byte	average sec	median sec	min sec	max sec	stdev
mmap/	256	18.980315	18.995113	18.755150	19.196081	0.177901
writev	512	9.622337	9.649033	9.399571	9.815508	0.163922
	1024	8.213951	8.245247	7.801813	8.635687	0.265761
	1456	7.480563	7.485363	7.339884	7.588845	0.078949
	2048	8.346631	8.356729	8.146762	8.516705	0.111168
mmap/	256	29.857161	29.825966	29.431526	30.374382	0.290558
msend	512	14.112755	14.135852	13.825899	14.391812	0.184452
	1024	8.034578	8.013282	7.911796	8.259743	0.110406
	1456	6.178561	6.160064	5.984090	6.383029	0.128040
	2048	9.243194	9.212100	9.089618	9.510554	0.143229
mmap/	256	21.306761	21.331257	21.005807	21.689702	0.235854
rtpmsend	512	10.339728	10.341929	10.052471	10.587390	0.175895
	1024	6.240251	6.246550	6.109071	6.369032	0.089168
	1456	5.118422	5.100725	4.960246	5.328190	0.110608
	2048	8.436318	8.430218	8.278742	8.675680	0.129064
krtpmsend	256	15.667218	15.623125	15.279676	16.079556	0.283470
	512	7.138215	7.208405	6.851959	7.367880	0.182028
	1024	5.226905	5.287696	4.959246	5.429175	0.158638
	1456	4.301946	4.335341	4.141370	4.391332	0.084022
	2048	7.774818	7.741323	7.684832	7.936794	0.081234

Tabell 5.3: Streaming med `writev`, `msend`, `rtpmsend` og `krtpmsend`

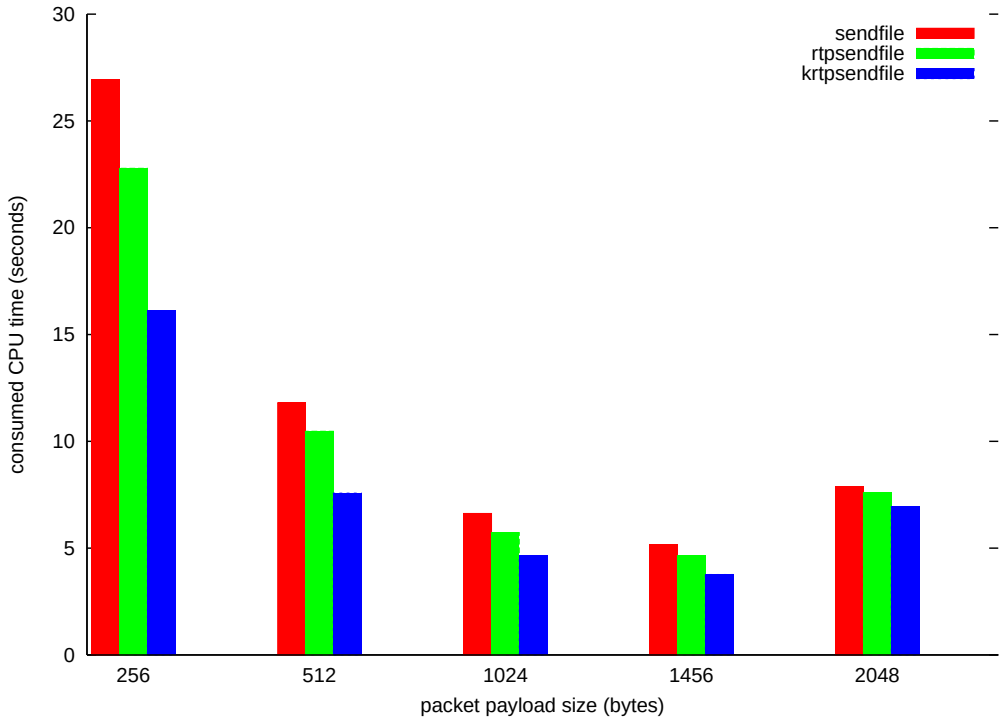
ter, hvor `sendfile` har åtte kontekstbytter per pakke, `rtpsendfile` har to, mens `krtpsendfile` har to kontekstbytter for streaming av en hel fil.

systemkall	ko	kb	kall til kjernen
sendfile	*n*	8*n*	*n* ***cork***, *n* **send**, *n* **sendfile** og *n* ***uncork*** kall
rtpsendfile†	*n*	2*n*	*n* **rtpsendfile** kall (**rtpsendfile**† legger til RTP-header kopioperasjon til **sendfile** systemkallet)
krtpsendfile†	0	2	1 **krtpsendfile** kall (**krtpsendfile**† legger RTP-headere til **sendfile** i kjernen)

ko = antall *kopioperasjoner*, s = antall *kontekstbytter*, *n* = antall pakker, † = nytt systemkall, cork og uncork er systemkallet `setsockopt`

Tabell 5.4: Oversikt over `sendfile`-streamingytelsestestene

Tabell 5.5 og figur 5.5 viser resultatene for disse systemkallene, og vi ser at reduksjonen av kontekstbytter har stor betydning for ytelsen. Forbedringene for `rtpsendfile` er 10 prosent sammenlignet med `sendfile` ved MTU pakkestørrelse, og forbedringene øker ved mindre pakkestørrelse. For `krtpsendfile` er forbedringene nesten 30 prosent ved MTU-pakkestørrelse, og ved pakkestørrelse



Figur 5.5: Streaming med `sendfile`, `rtpsendfile` og `krtpsendfile`

på 256byte er forbedringene hele 40 prosent. Dette viser at reduksjon av kontekstbytter har stor betydning for systemytelsen.

5.3.3 Sammeligning av nye og eksisterende systemkall

Vi skal nå se på en sammenligning av de nye og eksisterende systemkallene for overføring av data fra disk til nettverk. Som vi ser i figur 5.6 og tabell 5.6 yter `sendfile` og `rtpsendfile` i likhet med `msend` og `rtpmsend` dårligere, sammenlignet med `send` og `writev` for mindre pakkestørrelser. Men selv om CPU-bruken er identisk eller litt høyere vil `msend`, `rtpmsend`, `sendfile` og `rtpsendfile` være bedre med hensyn på andre delte systemressurser som blant annet systembuss og minne, ettersom kopioperasjonene er fjernet. Vi ser også at `sendfile` og `rtpsendfile` har litt lavere ressursforbruk enn `msend` og `rtpmsend` ved samme antall kopioperasjoner og kontekstbytter. Dette skyldes muligens overhead ved mapping av sidene i prosessens sidetabeller og at `readahead`-mekanismen ikke er like effektiv for mappede filer. Vi så i seksjon 4.3.6 at bruk av `madvise` for å aktivisere en mer agressiv `readahead` ga en forbedring på 13 prosent for `mmap` og

syscall	size byte	average sec	median sec	min sec	max sec	stdev
sendfile	256	26.915708	27.000396	26.135026	27.230860	0.334880
	512	11.807905	11.836200	11.380269	11.990177	0.173514
	1024	6.625093	6.681984	6.300041	6.746974	0.141983
	1456	5.171014	5.166715	4.893256	5.400179	0.130651
	2048	7.874503	7.926796	7.460867	8.005783	0.159748
rtpsendfile	256	22.784536	22.805033	22.316607	23.040497	0.206507
	512	10.468409	10.378923	9.840505	11.442262	0.429962
	1024	5.727029	5.799118	5.349187	5.901102	0.177252
	1456	4.652693	4.720783	4.221358	4.775275	0.168676
	2048	7.581947	7.627840	7.020933	7.809813	0.233051
krtpsendfile	256	16.122549	16.222034	15.511642	16.399507	0.263133
	512	7.576549	7.637839	7.034931	7.751822	0.204987
	1024	4.641295	4.597802	4.145370	5.518161	0.344770
	1456	3.769927	3.833917	3.354490	3.916405	0.172816
	2048	6.926947	6.998436	6.329038	7.202905	0.260033

Tabell 5.5: Streaming med `sendfile`, `rtpsendfile` og `krtpsendfile`

`send`. Bruk av `madvise` i forbindelse med `mmap` og `msend` vil muligens gi resultater tilsvarende for `sendfile`. Prototypene for RTP-motorene `krtpsendfile` og `krtpmsend` har det laveste ressursforbruket i alle testene og skyldes at alle kontekstbyttene og kopiering av RTP-headere og mediadata er fjernet. Sammenlignet med `read` og `send` er forskjellene ved MTU-pakkestørrelse 36 prosent for `krtpmsend` og 44 prosent for `krtpsendfile`. Vi ser dermed at det gir en ytelsesforbedring ved å flytte genereringen av RTP-headere ned i kjernen, og vi skal i neste seksjon se på implementasjonen av tre RTP-motorer i kjernen og det tilhørende systemkallet `mediastream`.

syscall	size byte	average sec	median sec	min sec	max sec	stdev
read/	256	22.200825	22.223121	21.484733	22.601564	0.305954
send	512	10.472308	10.512402	10.105463	10.637383	0.151605
	1024	7.275694	7.308390	6.851958	7.538854	0.189112
	1456	6.754773	6.796467	6.429022	6.858958	0.126900
	2048	8.425419	8.503208	7.868803	8.721675	0.230035
read/	256	23.390644	23.387944	23.172477	23.547421	0.101501
writev	512	10.735568	10.721371	10.659379	10.853350	0.069610
	1024	7.430270	7.396376	7.206904	7.603843	0.123475
	1456	7.032631	7.045430	6.894953	7.184907	0.094613

	2048	8.672181	8.601192	8.434718	9.015629	0.195633
mmap/	256	32.856605	32.914997	32.438068	33.331933	0.265602
send	512	18.014761	17.937773	17.796293	18.430199	0.223935
	1024	11.655828	11.682724	11.459258	11.799206	0.101963
	1456	10.199949	10.201450	10.107464	10.381422	0.079048
	2048	9.076920	9.081619	8.982634	9.163607	0.056192
mmap/	256	18.980315	18.995113	18.755150	19.196081	0.177901
writev	512	9.622337	9.649033	9.399571	9.815508	0.163922
	1024	8.213951	8.245247	7.801813	8.635687	0.265761
	1456	7.480563	7.485363	7.339884	7.588845	0.078949
	2048	8.346631	8.356729	8.146762	8.516705	0.111168
mmap/	256	29.857161	29.825966	29.431526	30.374382	0.290558
msend	512	14.112755	14.135852	13.825899	14.391812	0.184452
	1024	8.034578	8.013282	7.911796	8.259743	0.110406
	1456	6.178561	6.160064	5.984090	6.383029	0.128040
	2048	9.243194	9.212100	9.089618	9.510554	0.143229
sendfile	256	26.915708	27.000396	26.135026	27.230860	0.334880
	512	11.807905	11.836200	11.380269	11.990177	0.173514
	1024	6.625093	6.681984	6.300041	6.746974	0.141983
	1456	5.171014	5.166715	4.893256	5.400179	0.130651
	2048	7.874503	7.926796	7.460867	8.005783	0.159748
mmap/	256	21.306761	21.331257	21.005807	21.689702	0.235854
rtpmsend	512	10.339728	10.341929	10.052471	10.587390	0.175895
	1024	6.240251	6.246550	6.109071	6.369032	0.089168
	1456	5.118422	5.100725	4.960246	5.328190	0.110608
	2048	8.436318	8.430218	8.278742	8.675680	0.129064
rtpsendfile	256	22.784536	22.805033	22.316607	23.040497	0.206507
	512	10.468409	10.378923	9.840505	11.442262	0.429962
	1024	5.727029	5.799118	5.349187	5.901102	0.177252
	1456	4.652693	4.720783	4.221358	4.775275	0.168676
	2048	7.581947	7.627840	7.020933	7.809813	0.233051
krtpmsend	256	15.667218	15.623125	15.279676	16.079556	0.283470
	512	7.138215	7.208405	6.851959	7.367880	0.182028
	1024	5.226905	5.287696	4.959246	5.429175	0.158638
	1456	4.301946	4.335341	4.141370	4.391332	0.084022
	2048	7.774818	7.741323	7.684832	7.936794	0.081234
krtpsendfile	256	16.122549	16.222034	15.511642	16.399507	0.263133
	512	7.576549	7.637839	7.034931	7.751822	0.204987
	1024	4.641295	4.597802	4.145370	5.518161	0.344770
	1456	3.769927	3.833917	3.354490	3.916405	0.172816

2048	6.926947	6.998436	6.329038	7.202905	0.260033

Tabell 5.6: Streaming med nye og eksiterende systemkall

5.4 RTP-motorimplementasjon

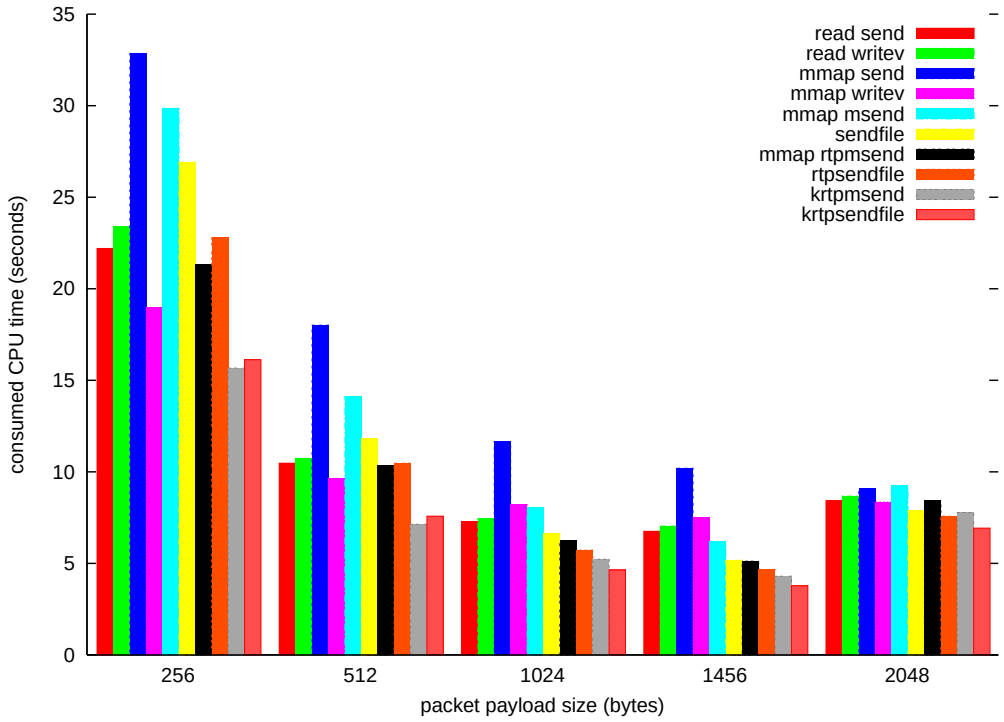
RTP-motorene er implementert som kjernetråder, med en tråd per RTP-strøm. Det er implementert tre forskjellige RTP-motorer, en basert på `sendfile` og to basert på `mmap`. Den første som er basert på `sendfile` brukes for streaming av MPEG Transport Stream, se seksjon 2.3. De to andre som er basert på `mmap` brukes for streaming av MPEG Elementary Stream, se seksjon 2.3, hvor videodataene hentes fra henholdsvis disk og applikasjonsbuffer. For kommunikasjon mellom applikasjonen og RTP-motorene brukes systemkallet `mediastream`.

5.4.1 `mediastream`

Applikasjonsnivåserverne RTSP og RTCP kommuniserer med RTP-motorene via systemkallet `mediastream`.

sys_mediastream(int8_t action, uint32_t sid, void __user * info, void __user * ret);

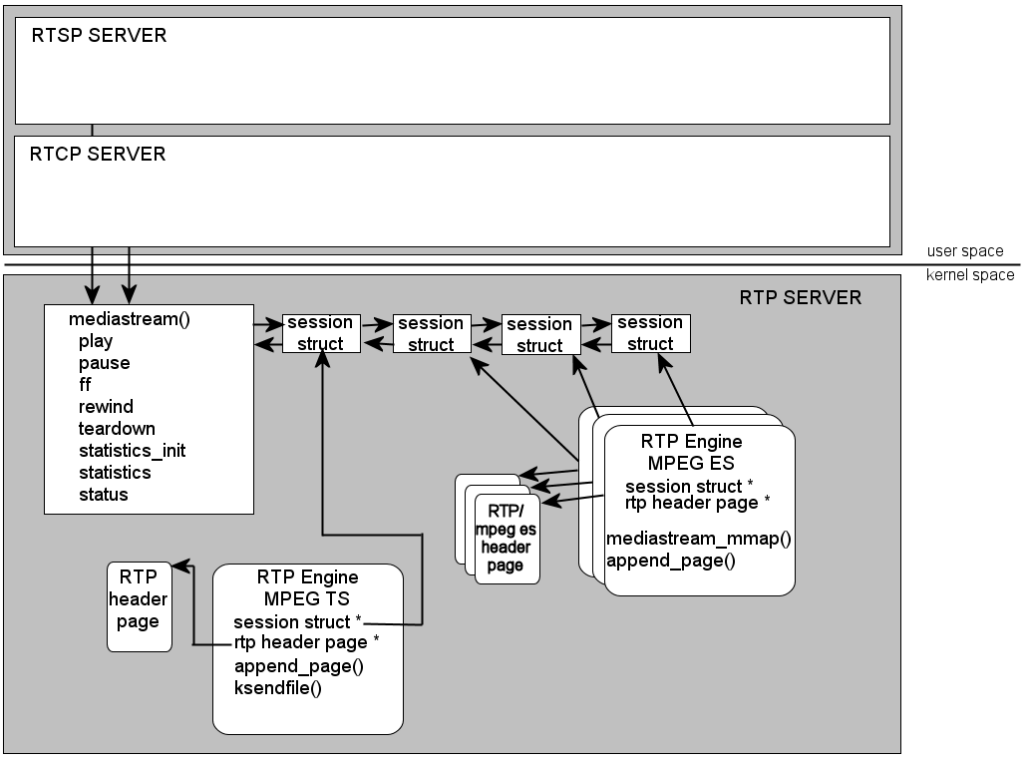
`mediastream` har fire paramentere, `action`, `sid`, `info` og `ret`. RTSP-serveren bruker `action`-verdiene PLAY, PAUSE, FF, REWIND, TEARDOWN og STATISTICS_INIT. PLAY og TEARDOWN brukes for å starte og stoppe RTP-motorene, PAUSE, FF og REWIND for at klient skal ha mulighet for å pause og hoppe frem og tilbake i strømmen. STATISTICS_INIT brukes av RTSP-serveren for å få kjennskap til de initielle NTP- og RTP-tidsstemplene og det initielle RTP-sekvensnummeret. I tillegg til de nevnte `action`-verdiene er det også definert STATISTICS og STATUS. STATISTICS brukes av RTCP-serveren for å få kjennskap til antall sendte byte og pakker, samt NTP- og RTP-tidsstempel og RTP-sekvensnummer. STATUS kan blant annet brukes for å finne ut om en strøm er avsluttet. `sid` er sesjonsid'en som opprettes av RTSP-serveren og brukes for å finne igjen den riktige RTP-motoren. For å overføre informasjonen til kjernen allokerer applikasjonen et `playinfo`-objekt, som er en instans av en struct med informasjon som er nødvendig for å starte og kontrollere en RTP-motor. Applikasjonen allokerer også plass for informasjon som overføres fra kjernen til applikasjonen. Dette er et `ses_stat`-objekt som kjernen overfører informasjon til på



Figur 5.6: Streaming med nye og eksiterende systemkall

forespørsel fra applikasjonen. Informasjonen som overføres er blant annet antall sente pakker. `info`- og `ret`-parameterne i `mediastream`-systemkallet er pekere til disse objektene og informasjonen må kopieres til og fra kjernen. Informasjonen i `playinfo`- og `ses_stat`-objektene er beskrevet nedenfor.

Som figur 5.7 illustrerer er `mediastream`-systemkallet implementert for å starte og kontrollere RTP-motorer. Systemkallet bruker en liste med `session`-objekter for å håndtere RTP-motorene. Hver RTP-motor har en referanse til sitt eget `session`-objekt og kommunikasjon mellom systemkallet og RTP-motoren foregår via dette objektet. Informasjonen som er nødvendig for RTP-motoren lagres i et `playinfo`-objekt og overføres fra applikasjonen til `session`-objektet i kjernen, mens informasjon om RTP-strømmen som er nødvendig for applikasjonen overføres fra `session`-objektet i kjernen til `ses_stat`-objektet i applikasjonen. Et alternativ til implementasjon av et nytt systemkall for kommunikasjon mellom applikasjonen og RTP-motorene er bruk av `Netlink Socket`, som er en IPC løsning for kommunikasjon mellom applikasjonsnivåprosesser og kjerneprosesser. Med en slik løsning kan applikasjonen kommunisere med en kjernetråd som starter og stopper RTP-motorer, og annen informasjonsutveksling kan skje direkte med RTP-



Figur 5.7: RTP-motorer og systemkallet `mediastream`

motorene. En slik løsning muliggjør også kommunikasjon fra RTP-motorene til applikasjonen, som dermed slipper å "polle" kjernen for å finne ut for eksempel når en RTP-motor har avsluttet streamingen.

Informasjonen som overføres fra applikasjonen til kjernen er følgende:

- Type mediadata som skal streames. Brukes for å starte riktig type RTP-motor. Det er implementert RTP-motorer for MPEG TS og MPEG ES.
- Fildeskriptor for mediafilen som skal streames og for socket.
- Start- og stoppoffset for mediadataene som skal streames.
- Maks pakkelengde for RTP-pakkene med RTP-header(e) og mediadata. For MPEG TS vil pakkelengden være lik denne verdien, men for MPEG ES vil pakkelengden variere men aldri overstige denne verdien.

- Transmisjonsintervall angir antall mikrosekunder mellom hver pakke for MPEG TS, og antall mikrosekunder mellom hver bilderamme for MPEG ES.
- Tidsstempelinkrement angir RTP-tidsstempelinkrementeringsverdien. For MPEG TS inkrementeres RTP-tidsstemplet for hver pakke, mens for MPEG ES for hver bilderamme.
- Synkroniseringskilde(SSRC) identifiserer deltakerne i en RTP-session. RTP-motoren kopierer SSRC'en til alle RTP-headerne men bruker den ikke til andre ting.
- Bufferaddresser brukes hvis mediadataene hentes fra et buffer istedenfor en fil på disk. Dette er adresser til pekere som igjen peker til hode og hale i bufferet, samt størrelsen på dataene i bufferet.
- PID er id'en til kjernetråden (RTP-motoren). Denne initieres i `playinfo`-objektet av applikasjonen etter at RTP-motoren er startet.

Informasjonen som overføres fra kjernen til applikasjonen er følgende:

- NTP-tidsstemplet angir NTP-tiden for første eller siste pakke sendt. Tiden for første pakke brukes av RTSP, mens tiden for siste pakke brukes av RTCP.
- RTP-tidsstemplet angir RTP-tiden for første eller siste pakke sent. Tiden for første pakke brukes av RTSP, mens tiden for siste pakke brukes av RTCP.
- pakketeller angir antall pakker sendt siden forrige avlesing av pakketelleren.
- octetteller angir antall byte sendt siden forrige avlesing av bytetelleren.
- RTP-sekvensnummer angir sekvensnummer for siste pakke sendt.

Når en ny RTP-strøm skal startes, utføres et `mediastream`-playkall, det opprettes et nytt `session`-objekt og informasjon som RTP-motoren trenger kopieres fra applikasjonen. Det startes deretter en RTP-motor, som er en kjernetråd, med referanse til `session`-objektet, som umiddelbart starter streaming av mediadataene. Etter at RTP-motoren er startet settes `session`-objektet inn i `session`-listen, før systemkallet returnerer pid'en til RTP-motoren til applikasjonen.

All kommunikasjon mellom applikasjonen og RTP-motoren foregår via `mediastream`-systemkallet og `session`-objektet. Når applikasjonen trenger informasjon om RTP-strømmen utfører et `mediastream`-systemkall, og informasjonen kopieres

fra `session`-objektet til applikasjonen. For å pause og senere starte en RTP-strøm brukes en semaphore som er initiert til 0. Operasjonene som kan utføres på en semaphore er `up` og `down` som inkrementerer og dekrementerer verdien til semaphoren. Når en tråd utfører en `down` på en semaphore som er 0 vil den bli schedulert og lagt i en kø til en annen tråd utfører en `up`. Ved et `mediastream`-pausekall settes statusen i `session`-objektet til PAUSED, og når RTP-motoren leser dette utfører den en `down` på semaphoren. Ettersom semaphoren er 0 vil RTP-motoren stoppe. Når den skal startes igjen utføres et `mediastream`-playkall, og statusen settes tilbake til RUNNING og det utøres en `up` på semaphoren slik at RTP-motoren startes igjen. For avslutning av en RTP-motor utføres et `mediastream`-teardownkall.

5.4.2 RTP-motor for MPEG Transport Stream

RTP-motoren for MPEG TS er basert på ideene fra prototypen `krtpsendfile`. For MPEG TS er det ikke nødvendig å utføre noen databerøringsoperasjoner på mediafilinnholdet for oppbygging av RTP-headere. Den eneste regelen som gjelder for mediadataene i RTP-pakken er at de må bestå av *n* antall MPEG-transportpakker som hver er 188 byte, se seksjon 2.3. Informasjonen som er nødvendig for oppbygging og kontroll av strømmen regnes ut og overføres fra applikasjonsnivå til kjernen. RTP-motoren for MPEG TS kan også brukes for MPEG Program Stream som ikke har noen regler for oppdeling av mediadataene i RTP-pakker.

Etter at RTP-motoren er startet initieres nødvendige datastrukturer og variabler som er nødvendig under streamingen av mediafilen. Det allokeres en side som gir plass til 340 RTP-headere. Denne siden settes som reservert for at den ikke skal "swappes" ut av minnet. Siden brukes som et ringbuffer for at RTP-headerne ikke skal bli overskrevet før de er overført til nettverkskortet. I denne implementasjonen brukes en hel side per RTP-strøm til RTP-headere. Dette er sløsing med minne, og det kunne med fordel vært implementert en egen bufferallokeringsmekanisme for RTP-headerbuffere slik at flere RTP-strømmer kan dele en side. Allokeringsmekanismen kan ha ansvar for å holde orden på sidene, allokere og deallokere sider ved behov, og returnere referanse til siden samt start- og stoppoffset for bufferet innenfor siden.

RTP-motoren må videre generere RTP-tidsstempel og sekvensnummer, initiere tid brukt siden streamingen startet, starttid (timeofday) og NTP-tidsstempel, og finne startposisjon og antall byte som skal streames.

Når initieringen er ferdig startes streamingen. RTP-headerpekeren flyttes til neste ledige plass i RTP-siden, og RTP-headeren initieres. De aktuelle feltene i RTP-headeren er versjonsnummer (V), mediadatatype (PT), sekvensnummer, tidsstem-

pel og synkroniseringskilde(SSRC). De andre feltene brukes ikke. Versjonsnummeret er 2, mediadatatype er 33 for MPEG2 TS A/V, og SSRC er bestemt av applikasjonen. Verdiene i disse feltene er identiske for alle pakkene. Sekvensnummeret og tidsstemplet starter med en tilfeldig valgt verdi og inkrementeres for hver pakke. Sekvensnummeret inkrementeres med en, og tidsstemplet inkrementeres med en verdi bestemt av bitrate og pakkestørrelse.

Oppbygging og overføring av RTP-headere til kommunikasjonssystemets `socket buffer` innebærer ingen kopioperasjoner. Når RTP-headeren er klar overføres referansen som består av referanse til RTP-siden, offset og lengde til `socket buffer` ved hjelp av socketen sin `sendpage`-metode. Denne metoden sørger også for at det opprettes en UDP-header. `sendpage` brukes med MSG_MORE-flagget for at RTP-headeren og mediadataen skal sendes i samme pakke. Deretter overføres referansen til mediadataen fra `page cache` til `socket buffer` ved hjelp av `mediastream_sendfile`. `mediastream_sendfile` er en `sendfile`-variant som kan kalles fra kjernen. Ettersom `mediastream_sendfile` ikke har noen støtte for MSG_MORE-flagget, må socketen "uncorkes" med et `setsockopt`-kall slik at pakken blir sent.

`mediastream_sendfile` burde utvides med et flag parameter slik at MSG_MORE kan brukes i stedenfor setsockopt, men dette krever også at metodene `generic_file_sendfile`, `do_generic_mapping_read` og `file_send_actor` må implementers i nye versjoner. Det er da også mulig å lage enkelte optimaliseringer tilpasset RTP-motorene, men disse endringene vil sannsynligvis ikke gi merkbare reduksjoner i CPU-bruk. Optimaliseringene som kan gjøres, er blant annet å gi leserettigheter til hele filen en gang i stedenfor å gi leserettigheter for en liten del for hver pakke som skal overføres.

Etter at en pakke er sendt oppdateres octet- og pakketeller og RTP-motoren scheduleres med et kall på `nanosleep` til neste pakke skal sendes. Før neste pakke sendes sjekker RTP-motoren om status fortsatt er RUNNING og om start- og stoppoffset skal endres, eller om den skal stoppe eller pause streamingen. Hvis RTP-motoren ikke skal stoppes eller pauses, oppdateres RTP-sekvensnummeret, RTP-tidsstemplet og NTP-tidsstemplet før neste pakke kan sendes.

5.4.3 RTP-motor for MPEG Elementary Stream

RTP-motoren for MPEG ES brukes for streaming av MPEG Elementary Streams, og er basert på ideene fra `krtpmsend`-prototypen. (Se seksjon 2.3 for beskrivelse av MPEG Elementary Stream.) Strukturen på denne RTP-motoren er den samme som for RTP-motoren for MPEG TS, men mediadataene overføres fra disk til

nettverk ved hjelp av mmap og sendpage i stedenfor sendfile. Dette muliggjør databerøringsoperasjoner, som er nødvendig ved streaming av MPEG ES data.

Når en MPEG ES RTP-motor startes, allokeres en side for standard og mediaspesifikk RTP-headere. Videre mappes mediafilen med mediastream_mmap (en wrapper til mmap som kan kalles fra kjernen), som oppretter en mapping av filen i page cache. Ettersom dette minneområdet ikke inneholder noen sider med data, må page cache få beskjed om at dataene skal overføres fra disk etterhvert som de skal brukes. I denne implementasjonen benyttes metoden make_pages_present til dette. Dette fungerer, men er ikke spesielt effektivt, og en forbedret versjon bør sjekke om siden allerede finnes i page cache og benytte seg av readahead-funksjonalitet. Med bruk av readahead slipper prosessen å vente på I/O. For forbedring av dette kan systemkallet madvise og kjernemetoden filemap_nopage brukes som et utgangspunkt. Tester vi utførte med bruk av madvise viste en reduksjon av antall "page faults" på over 90 prosent og en reduksjon av CPU-bruk på over 13 prosent.

For streaming av en MPEG Elementary Stream brukes standard RTP-header og i tillegg en RTP-mediaspesifikk header. De aktuelle feltene i RTP-headeren er versjonnummer (V), marker (M), mediadatatype (PT), sekvensnummer, tidsstempel og synkroniseringskilde (SSRC). Markerbittet brukes når en pakke inneholder sluttkode for et MPEG-bilde. Når det gjelder tidsstemplet inkrementeres den for hvert bilde og ikke for hver pakke. Mediadatatypen for MPEG ES Video er 32, mens de andre feltene brukes som beskrevet for MPEG TS. Feltene i den RTP-mediaspesifikke headeren er forklart i seksjon 2.3, og bestemmes av innholdet i de siste Sequence, GOP, Picture eller Slice headerne i MPEG ES-filen.

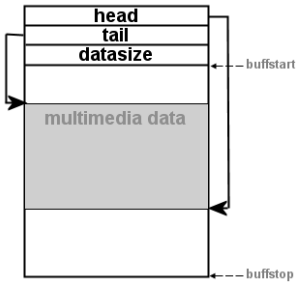
For oppbygging av RTP-mediaspesifikk header er det implementert en metode som finner neste MPEG ES-headerkode. Når denne metoden finner en headerkode sjekker den om headeren kommer på riktig plass i payloaden og at det er plass til hele headeren. Hvis dette er oppfylt returnerer den headerkoden, ellers gir den beskjed om at pakken er ferdig og offset til siste byte i pakken. Retur informasjonen fra denne metoden brukes for oppbygging av RTP-headerne. Det er også implementert metoder for å hente ut informasjon fra MPEG ES-headerne. Metodene som brukes for oppbygging av RTP-headerne leser mediadataene direkte i page cache, og det er dermed ingen kopioperasjoner involvert.

Når en RTP-pakke er ferdig overføres referansen til standard og spesifikk RTP-header til socket buffer, og referansen til mediadataene overføres fra page cache til socket buffer. Til all referanseoverføring brukes socketens sendpage-metode. Det er dermed ingen kopioperasjoner for overføring av headere og mediadata. Ettersom bare sendpage metoden benyttes kan MSG_MORE-flagget benyttes for at headere og mediadata skal sendes i samme pakke.

5.4.4 RTP-motor for MPEG Elementary Stream med delt buffer

RTP-motoren for MPEG Elementary Stream beskrevet i forrige seksjon brukes for mediafiler som er lagret på disk. I denne seksjonen beskrives en RTP-motor som deler et buffer med en prosess på applikasjonsnivå. Denne løsningen kan for eksempel brukes ved streaming av live materiale, eller når det er nødvendig med encoding av lagret materiell.

Det delte bufferet er et virtuelt minneområdet som er mappet i applikasjonens sidetabeller. Dataene i dette minneområdet blir lagret i `page cache` og fungerer på samme måte som ved mapping av en fil, men istedenfor at dataene blir overført fra disk til `page cache` er det applikasjonsnivåprosessen som skriver data til `page cache`. RTP-motoren trenger dermed bare den lineare adressen for å finne igjen `vma`-objektet som beskriver dette minneområdet, og dermed `address_space` objektet og sidene med data som beskrevet i seksjon 4.1.2. Det delte bufferet opprettes av applikasjonsnivåprosessen ved hjelp av `mmap`. Bufferet opprettes som et delt buffer som ikke er knyttet til en fil ved hjelp av MAP_ANONYMOUS flaget i `mmap`. Dette er tilsvarende det som gjøres ved IPC. Lesing og skriving til dette bufferet fungerer på samme måte som ved lesing og skriving til en mappet fil. Dette gjør det mulig å overføre data fra applikasjonsvivåprosessen til nettverket uten kopioperasjoner og kontekstbytter.



Figur 5.8: Delt buffer for applikasjonsnivåprosess og RTP-motor

Bufferet fungerer som et ringbuffer, og figur 5.8 illustrerer hvordan det er organisert. De 12 første bytene er hode- og halepekere og antall byte data i bufferet. I tillegg til å kjenne til adressen til hode-, hale- og datastørrelsevariablene er det nødvendig for applikasjonsnivåprosessen og RTP-motoren å kjenne til start- og stoppadressen til bufferet. Disse adressene kopieres fra applikasjonsnivåprosessen til kjernen i `playinfo`-objektet beskrevet i seksjon 5.4.1. Ved å plassere hode- og halepekerene og størrelsevariablen i det delte bufferet, er det ikke nødvendig å

kopiere dem til og fra kjernen. Dette er en fordel, ettersom de oppdateres for hver gang det leses eller skrives til bufferet. Hodepekeren leses av RTP-motoren, og leses og skrives av applikasjonsnivåprosessen, mens halepekeren leses av applikasjonsnivåprosessen, og leses og skrives av RTP-motoren. Størrelsesvariablen leses og skrives av både RTP-motoren og applikasjonsnivåprosessen, og brukes bare for å sjekke om bufferet er tomt eller fullt. Det er ikke låser på de delte variablene noe som kan implementeres i en forbedret versjon, men det vil ikke oppstå noen problemer uten låser. Den eneste variablen som både applikasjonsnivåprosessen og RTP-motoren leser og skriver til er størrelsesvariablen, og den brukes når hode og hale peker på samme adresse for å vite om bufferet er tomt eller fullt. Hvis hode og hale peker på samme adresse og størrelsesvariablen er 0 er bufferet tomt, hvis størrelsesvariablen er større enn 0 er bufferet fullt. Hvis applikasjonsnivåprosesen leser hale og størrelsesvariablen etter at RTP-motoren har oppdatert halepekeren men før den har oppdatert størrelsesvariablen, kan applikasjonsnivåprosessen tro at bufferet er fullt når det egentlig er tomt. Dette er ikke kritisk og applikasjonsnivåprosessen vil lese variablene på nytt og få de riktige verdiene. Det samme kan skje med RTP-motoren.

Det er implementert kode for håndtering av ringbufferet, men ellers er implementasjonen som beskrevet i seksjon 5.4.3.

5.4.5 Diskusjon

Løsningene som er basert på `mmap` krever et stort virtuelt adresserom. For eksempel vil en videofil på 2 timer med bitrate på 4Mbps bruke et virtuelt minneområde på 3.6GB. For 32bits arkitekturer er det virtuelle adresserommet på 4GB per prosess, og det vil derfor ikke være mulig å mappe mange store filer samtidig, noe som er en forutsetning for en multimediaserver. På grunn av dette vil `mmap` løsningene bare være aktuelle på en 64bits arkitektur som har et virtuelt adresserom på 16777216TB per prosess. De fleste prosessorer som produseres i dag er 64bits eller har 48- eller 64-bits utvidelse for minnehåndtering, noe som gjør `mmap` løsningene aktuelle.

En mulig løsning for å bruke `mmap` på en 32bits arkitektur, er å la hver strøm mappe en liten del av filen og flytte mappingen etterhvert som filen streames. Dette vil medføre en del ekstra kostnader med mapping og unmapping. Tester som ble utført med `mmap` og `writev` viser at å flytte mappingen for hver 4KB øker CPU-bruken med nesten 20 prosent sammenlignet med å mappe en gang. Ved å mappe et større område vil den ekstra kostnaden bli mindre. En slik løsning for en kjerneimplementasjon, sammenlignet med en applikasjonsnivåimplementasjon basert på `read` og `send`, vil likevel gi en forbedring på minst 15 prosent. Det

som kompliserer en løsning med flytting av mappingen, er at det er referanse overføring og ikke kopiering av dataene fra `page cache` til `socket buffer`. En slik løsning må derfor passe på å ikke unmappe et område som enda ikke er overført til nettverkskortet. En annen løsning på problemet er å starte en ny prosess istedenfor en tråd for hver RTP-strøm. Det er da tilgjengelig et 4GB stort virtuelt adresserom per strøm og ikke 4GB tilsammen for alle strømmene.

For RTP-motorene kan det være en mulighet å bruke en modifisert versjon av `sendfile`, som ikke krever et stort virtuelt adresserom. RTP-motorene er implementert i kjernen og kan derfor bruke overføringsmetoder med en datasti innenfor kjernen. For å kunne bruke denne løsningen på RTP-motorer som krever databerøringsoperasjoner, er det nødvendig å splitte `sendfile`-koden i to deler. En del som sørger for at dataene blir overført fra disk til `page cache` og en del som overfører referansen fra `page cache` til `socket buffer`. Mellom kallene på disse to metodene vil det være mulig å utføre kode for databerøringsoperasjoner. En slik løsning vil ikke mappe filen i prosessens sidetabeller, men bare bruke `address_space`-objektet i `page cache` for å finne igjen sidene med data som skal overføres. En slik løsning kan være bedre enn `mmap` løsningen også på 64bits arkitekturer, ettersom den fjerner kostnaden med mapping i prosessens sidetabeller. For applikasjonsnivåimplementasjoner som krever databerøringsoperasjoner er bare systemkallene basert på `mmap` aktuelle. Det er ikke mulig å lage en løsning basert på `sendfile`, ettersom filen må mappes i prosessens sidetabeller for at applikasjonsnivåprosessen skal kunne lese dataene.

I implementasjonen av RTP-motorene ble det valgt en trådmodell med en tråd per strøm. Et alternativ kunne vært å ha flere strømmer per tråd, og dermed redusert antall scheduleringer av trådene. Schedulering av tråder og prosesser er i likhet med kontekstbytter en kostbar operasjon. Grunnen til at det ble valgt en modell med en strøm per tråd, er at det er enklere å implementere enn en modell med flere strømmer per tråd. En annen grunn til at denne modellen ble valgt er at formålet med denne oppgaven er å se på ytelsesforbedringer ved reduksjon av kopioperasjoner og kontekstbytter, og da er ikke reduksjon av scheduleringer interessant.

5.5 RTP-motor eksperimenter

Testene av prototypene `krtpmsend` og `krtpsendfile` viste at det gir forbedringer i systemytelsen ved å plassere genereringen av RTP-headerne i kjernen, og dermed fjerne alle kontekstbytter og kopioperasjoner. I testene av prototypene ble det testet CPU-bruk ved en RTP-strøm, mens i denne seksjonen skal vi se hvor mange samtidige strømmer vi kan ha ved en slik implementasjon. Eksperimentene er basert på implementasjonen av RTP-motoren for MPEG TS beskrevet

i seksjon 5.4, og en tilsvarende implementasjon på applikasjonsnivå. Hensikten med disse eksperimentene er å finne ut om en RTP-server implementert i kjernen kan levere flere samtidige strømmer enn en tilsvarende RTP-server implementert på applikasjonsnivå.

5.5.1 Testoppsett

I eksperimentene beskrevet i denne seksjonen ble det benyttet en Dell Optiplex GX260. Testmaskinen har et Intel 845 chipset, 2.0 GHz Intel Pentium4 CPU, 512MB RAM, 40GB 7,200RPM ATA/100 disk, og 1Gbit nettverkskort. Denne maskinen ble koblet sammen med en annen maskin via en 1Gb switch.

I eksperimentene med RTP-motoren ble pakke- og bytetelleren `iptraf` benyttet for å finne maksimal overføringshastighet. For å måle CPU-bruk ble `System Monitor` og `top` benyttet. `System Monitor` viser CPU-bruk, mens `top` gir et mer detaljert bilde av hva prosessoren benyttes til, blant annet interrupt og tid benyttet på applikasjonsnivå og i kjernen. I eksperimentene ble alle strømmene startet før resultatene ble lest av fem ganger med 20 sekunders interval. For å ha et sammenligningsgrunnlag ble det implementert en flertrådet RTP-server på applikasjonsnivå, hvor `sendfile` ble benyttet for overføring av data fra disk til nettverk. I eksperimentene ble det brukt en MPEG TS-videofil med bitrate på 2Mbps.

5.5.2 Kommentarer om eksperimentene

Disse eksperimentene ble noe begrenset på grunn av mangel på egnet utstyr som 64bits prosessor, flere disker i RAID-oppsett, og flere nettverkskort. RTP-motoren basert på `mmap` krever et 64bits virtuelt adresserom, som beskrevet i seksjon 5.4.5, hvis det skal være mulig å mappe mange store filer samtidig. På grunn av mangelen på utstyr ble derfor bare en begrenset test av RTP-motoren basert på `sendfile` gjennomført.

I forbindelse med eksperimentene så vi også at disken og nettverkskortet var flaskehalser. Overføringshastighet på disken er 50MB/s som skulle tillsi at det kan overføres 400 2Mbps-strømmer, men på grunn av at diskhodet må flyttes for hver forespørsel faller overføringshatigheten til 10-15MB/s. For å omgå flaskehalsen med disken ble det derfor bruk en fil for alle strømmene. Filen som ble benyttet er på 100MB og får dermed plass i RAM slik at alle alle diskforespørsler ble betjent fra `page cache` istedenfor disken. Maksimal overføringshastigheten for nettverkskortet er på litt over 300Mbit/s, noe som også tester ved University of

Northern Iowa [8] av 1Gb nettverkskort viser er maksimal overføringshastighet med MTU på 1500 byte.

5.5.3 Eksperimenter med `iptraf`, `System Monitor` og `top`

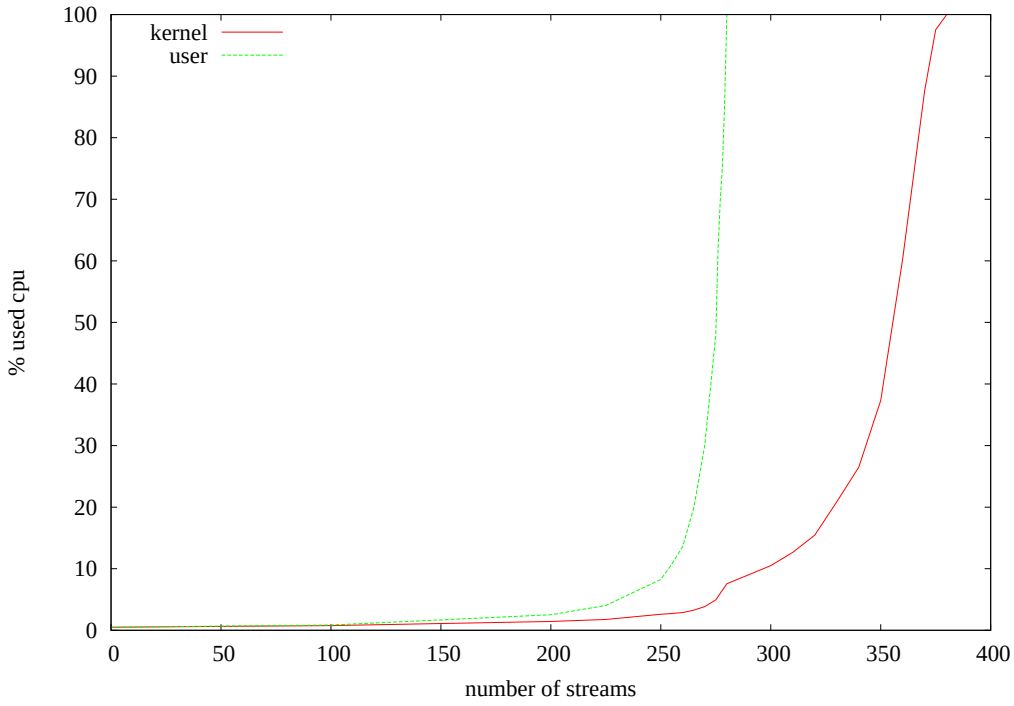
I det første eksperimentet ble maksimal overføringshastighet målt, og monitorering med iptraf viser at maksimal overføringshastighet for kjerneimplementasjonen er 345 Mbit/s inkludert headere. Dette tilsvarer 160 strømmer. Med applikasjonsnivåimplementasjonen er maksimal overføringshastighet 314 Mbit/s som tilsvarer 145 strømmer. Dette er en forskjell på 9 prosent og viser at kjerneimplementasjonen gir en ytelsesforbedring.

n	**% CPU ki**	**% CPU ai**	*n*	**% CPU ki**	**% CPU ai**
0	0.50	0.50	279	-	85.44
100	0.76	0.85	280	7.56	100.00
200	1.44	2.53	290	9.03	-
225	1.76	4.03	300	10.50	-
250	2.60	8.26	310	12.66	-
255	2.72	10.76	320	15.45	-
260	2.89	13.64	330	20.86	-
265	3.27	19.90	340	26.48	-
270	3.85	30.04	350	37.32	-
275	4.94	48.00	360	60.33	-
276	-	61.59	370	87.65	-
277	-	69.59	375	97.54	-
278	-	75.40	380	100.00	-

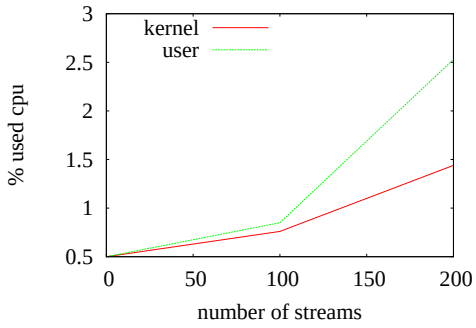
n = antall strømmer, ki = kjerneimplementasjon (RTP-motor), ai = applikasjonsnivåimplementasjon

Tabell 5.7: Sammenligning av kjerne- og applikasjonsnivåimplementasjon målt med `System Monitor`

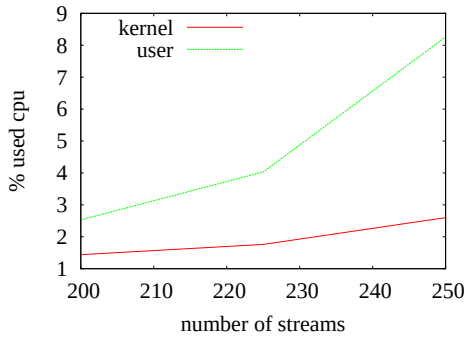
Figurene 5.9, 5.10 og 5.11 og tabell 5.7 viser testresultatene for applikasjonsnivå- og kjerneimplementasjonen målt med `System Monitor`. Testene viser at CPU-bruk for kjerneimplementasjonen alltid er lavere enn for applikasjonsnivåimplementasjonen ved det samme antall strømmer, og resultatene viser at antall samtidige strømmer er 26 prosent høyere for kjerneimplementasjonen enn for applikasjonsnivåimplementasjonen. Maksimalt antall strømmer for appliksjonsnivåimplementasjonen er 280 mens for kjerneimplementasjonen er det 380. Vi kan også se at den prosentvise forskjellen i CPU-bruk øker etterhvert som flere strømmer



Figur 5.9: Sammenligning av kjerne- og applikasjonsnivåimplementasjon målt med `System Monitor`



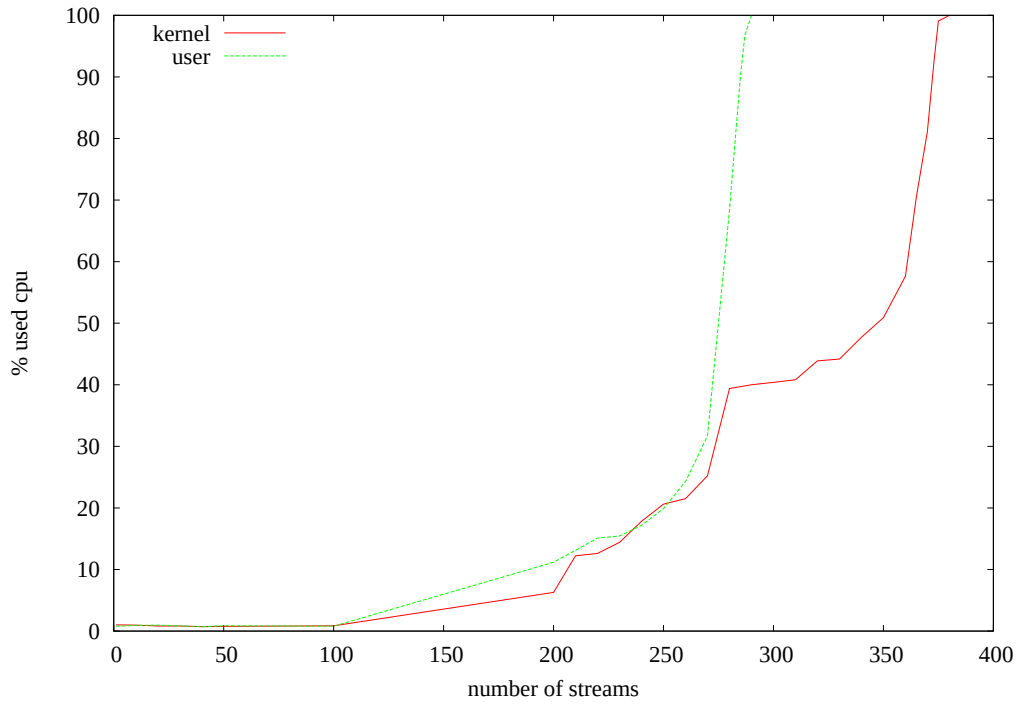
Figur 5.10: Utsnitt av figur 5.9 for 0 til 200 strømmer



Figur 5.11: Utsnitt av figur 5.9 for 200 til 250 strømmer

startes. Ved 100 strømmer er forskjellen 11 prosent mens for 200 strømmer er forskjellen 43 prosent. Resultatene når det er mer enn 145 strømmer for applikasjonsnivåimplementasjonen og 160 for kjerneimplementasjonen er usikre på grunn av at mange av pakkene ikke blir overført over nettverket, men kastet i kjernen. Resultatene tyder likevel på at en kjerneimplementasjon har lavere ressursforbruk og

kan betjene flere samtidige strømmer enn en applikasjonsnivåimplementasjon.



Figur 5.12: Sammenligning av kjerne- og applikasjonsnivåimplementasjon målt med `top`

I figur 5.12 ser vi resultatene for de samme eksperimentene, men denne gangen målt med `top`. Mer detaljerte resultater kan sees i figur 5.13 og tabell 5.8 for kjerneimplementasjonen, mens detaljerte resultater for applikasjonsnivåimplementasjonen presenteres i figur 5.14 og tabell 5.9. Resultatene viser at det ikke er forskjell i CPU-bruk ved 1 til 100 strømmer, mens ved 200 strømmer er forskjellen 5 prosent. Litt overraskende viser testene at CPU-bruk ved 250 strømmer er nesten identisk. Videre ser vi at CPU-bruk for applikasjonsnivåimplementasjonen fortsetter å stige kraftig, mens det for kjerneimplementasjonen etterhvert flater ut før det stiger kraftig igjen. I disse eksperimentene er maksimalt antall samtidige strømmer for applikasjonsnivåimplementasjonen 290, mens det for kjerneimplementasjonen er 380. Dette er en forskjell på over 20 prosent. Som beskrevet tidligere, er disse resultatene usikre på grunn av at mange av pakkene blir kastet av kjernen, uten å bli overført til nettverket når overføringshastigheten overstiger henholdsvis 314Mbps for applikasjonsnivåimplementasjonen og 345Mbps for kjerneimplementasjonen. Men til tross for dette tyder resultatene på at en kjer-

neimplementasjon gir lavere ressursforbruk, og dermed kan håndtere flere samtidige strømmer.

n	**us**	**sy**	**ni**	**id**	**wa**	**hi**	**si**
1	0.26	0.00	0.00	98.98	0.76	0.00	0.00
10	0.14	0.00	0.00	99.04	0.82	0.00	0.00
20	0.14	0.00	0.00	99.18	0.68	0.00	0.00
30	0.16	0.02	0.00	99.18	0.60	0.02	0.02
40	0.12	0.02	0.00	99.28	0.58	0.00	0.00
50	0.06	0.04	0.00	99.24	0.66	0.00	0.00
100	0.20	0.02	0.00	99.14	0.62	0.02	0.00
200	0.26	0.08	0.00	93.72	0.80	3.94	1.20
210	0.28	0.10	0.00	87.76	1.66	7.60	2.60
220	0.32	0.16	0.00	87.40	0.48	8.62	3.02
230	0.38	0.08	0.00	85.58	0.64	9.92	3.40
240	0.38	0.18	0.00	82.18	0.54	12.34	4.40
250	0.44	0.32	0.00	79.38	1.50	13.68	4.68
260	0.50	0.44	0.00	78.50	0.42	14.80	5.34
270	0.66	0.34	0.00	74.78	0.48	17.50	6.24
280	0.86	1.54	0.00	60.60	0.38	17.72	18.88
290	1.02	1.72	0.00	60.00	0.82	17.74	18.70
300	1.08	2.04	0.00	59.60	0.80	17.66	18.80
310	1.42	2.34	0.00	59.18	0.50	17.70	18.90
320	1.48	5.40	0.00	56.12	0.32	17.68	19.00
330	2.16	5.26	0.00	55.82	0.28	17.76	18.76
340	3.24	7.30	0.00	52.28	0.22	17.70	19.26
350	3.34	10.38	0.00	49.10	0.24	17.70	19.22
360	4.90	15.34	0.00	42.38	0.08	17.86	19.44
365	6.58	26.54	0.00	29.44	0.00	17.78	19.66
370	8.26	34.86	0.00	18.90	0.00	17.72	20.24
373	6.60	47.84	0.00	7.50	0.00	17.76	20.34
375	3.02	57.14	0.00	0.92	0.00	17.82	21.12
380	1.56	59.46	0.00	0.00	0.00	17.72	21.22

n = antall strømmer, hi = hard interrupt, si = soft interrupt, wa = i/o wait, us = user, sy = system, id = idle, ni = nice

Tabell 5.8: Kjerneimplementasjon (RTP-motor) målt med `top`

n	**us**	**sy**	**ni**	**id**	**wa**	**hi**	**si**
1	0.12	0.00	0.00	99.22	0.66	0.00	0.00
10	0.18	0.00	0.00	99.08	0.72	0.02	0.00
20	0.18	0.02	0.00	99.04	0.74	0.02	0.00
30	0.16	0.04	0.00	99.16	0.64	0.00	0.00
40	0.08	0.04	0.00	99.28	0.60	0.00	0.00
50	0.12	0.04	0.00	99.14	0.68	0.02	0.00
100	0.16	0.02	0.00	99.20	0.62	0.00	0.00
200	0.64	0.64	0.00	88.82	0.52	7.10	2.28
210	1.00	1.42	0.00	86.88	0.52	7.68	2.52
220	0.78	0.62	0.00	84.90	0.54	9.98	3.18
230	0.94	0.84	0.00	84.56	0.52	9.88	3.26
240	1.22	1.12	0.00	82.82	0.52	10.94	3.40
250	1.72	1.14	0.00	80.10	0.56	12.36	4.18
260	2.84	2.10	0.00	75.72	0.52	14.24	4.62
270	4.64	4.84	0.00	68.30	0.36	16.12	5.70
280	13.58	28.46	0.00	31.54	0.06	17.24	9.14
285	19.02	42.02	0.00	9.84	0.00	17.86	11.24
287	20.74	46.56	0.00	3.24	0.00	17.98	11.46
290	21.28	48.62	0.00	0.00	0.00	17.90	12.18

n = antall strømmer, hi = hard interrupt, si = soft interrupt, wa = i/o wait, us = user, sy = system, id = idle, ni = nice

Tabell 5.9: Applikasjonsnivåimplementasjon målt med `top`

5.6 Oppsummering

I dette kapitlet har vi sett på implementasjon av systemkall optimalisert for streaming, hvor kopioperasjonene ved overføring av mediadata fra disk til nettverk er fjernet, og antall kontekstbytter er redusert. Systemkallet `msend` er en forbedring av `send` hvor kopioperasjonen er fjernet. `msend` brukes for overføring av mappede filer, men kan ikke benyttes for overføring av applikasjonsbuffere. Ved RTP basert streaming er det derfor nødvendig å benytte `send` for overføring av RTP-headeren, noe som fører til mange kontekstbytter for hver RTP-pakke som sendes. For å redusere kontekstbyttene til et minimum ble derfor `rtpmsend`, som er en utvidelse av `msend` med parametere for RTP-headeren, implementert. Med MTU-pakkestørrelse så vi at `msend` ga en reduksjon i CPU-bruk på 17 prosent, og `rtpmsend` over 30 prosent sammenlignet med `writev`.

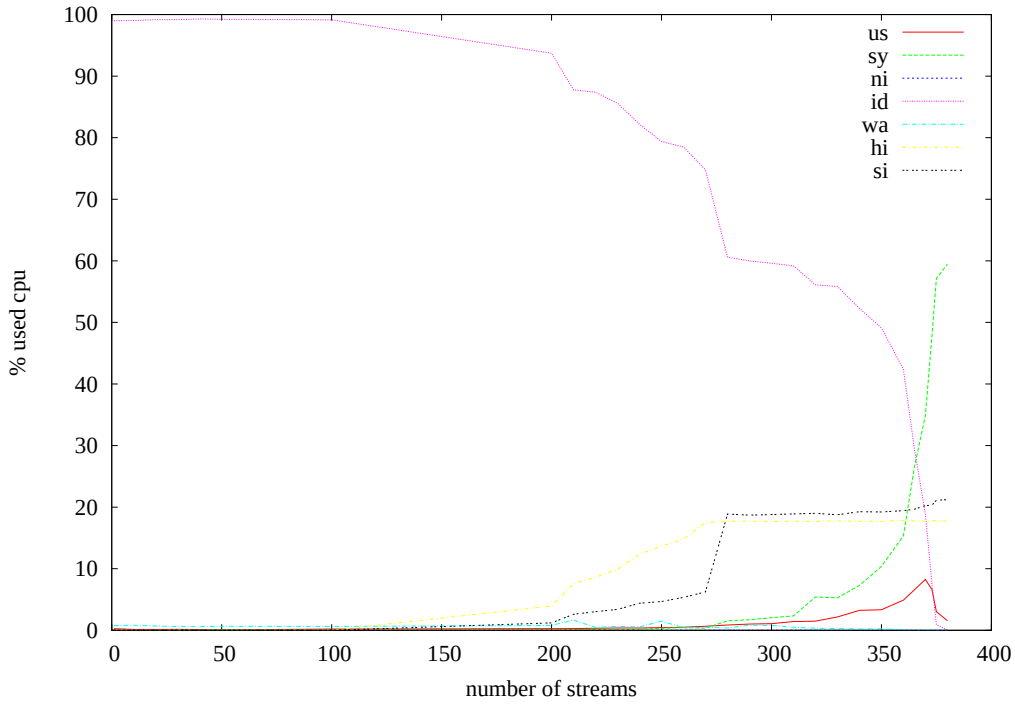
For å optimalisere `sendfile` for streaming ble `rtpsendfile` implementert. Ettersom `sendfile` er optimalt implementert med tanke på kopioperasjoner var det

bare nødvendig å redusere antall kontekstbytter ved sending av en RTP-pakke. `rtpsendfile` er derfor utvidet med parametere for RTP-headeren, slik at overføring av en RTP-pakke bare krever et systemkall. Med MTU-pakkestørrelse så vi at `rtpsendfile` ga en reduksjon i CPU-bruk på 10 prosent sammenlignet med `sendfile`.

Både `rtpmsend` og `rtpsendfile` overfører mediadataene fra disk til nettverk uten kopioperasjoner, men krever to kontekstbytter for hver RTP-pakke som sendes. For å se om fjerning av kontekstbyttene ga reduksjon i CPU-bruk, ble to RTP-motorprototyper implementert i kjernen. `krtpmsend` er basert på `mmap` og `msend`, mens `krtpsendfile` ble basert på `sendfile`. Med MTU-pakkestørrelse så vi at `krtpmsend` ga en reduksjon i CPU-bruk på over 40 prosent sammenlignet med `writev`, og 16 prosent reduksjon sammenlignet med `rtpmsend`. For `krtpsendfile` var det en reduksjon i CPU-bruk på nesten 30 prosent sammenlignet med `sendfile` med MTU-pakkestørrelse. Sammenlignet med `rtpsendfile` var reduksjonen i CPU-bruk 19 prosent.

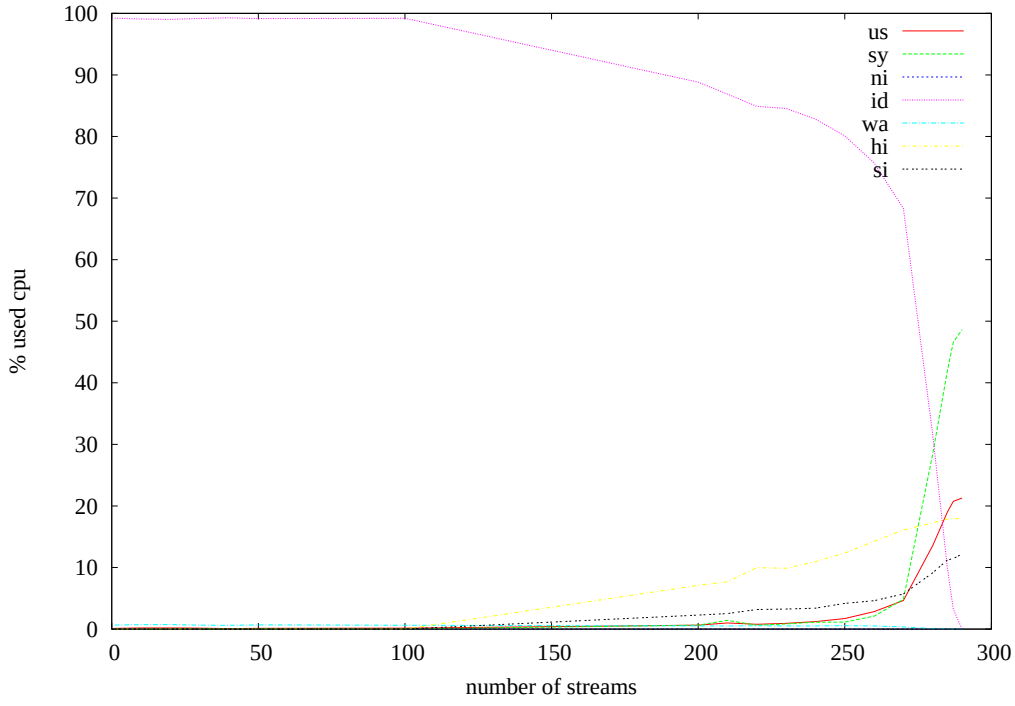
Resultatene for prototypene ga grunnlag for å implementere RTP-motorer i kjernen. Det ble implementert en RTP-motor for MPEG TS og en for MPEG ES, hvor den for MPEG TS ble basert på `krtpsendfile`, mens de for MPEG ES ble basert på `krtpmsend`. På grunn av mangel på egnet utstyr ble bare RTP-motoren for MPEG TS testet. Resultatene viste at en RTP-motor implementert i kjernen kan overføre over 20 prosent flere strømmer enn en implementasjon på applikasjonsnivå. Resultatene er noe usikre på grunn av at mange av pakkene ble kastet i kjernen og ikke overført til nettverket. Etter at en pakke er ferdig settes den inn i pakkekøen til nettverkskortets driver, men hvis denne er full blir pakken kastet. Dette skjer når nettverkskortet ikke klarer å overføre pakkene raskt nok. Men til tross for dette tyder resultene av RTP-motoreksperimentet og prototypeeksperimentene på at en implementasjon i kjernen gir en betydelig reduksjon i CPU-bruk, noe som muliggjør overføring av flere samtidige strømmer.

Disse resultatene viser at det er mulig med store forbedringer ved overføring av mediadata fra disk til nettverk i en streaming sammenheng. I tillegg til å gi en reduksjon i CPU-bruk, gir fjerning av kopioperasjoner også reduksjon i datatrafikken på systembussen. Ettersom det bare er en kopi av dataene gir dette en bedre utnyttelse av minnet. Dette kan gi økt systemytelse ved at dataene blir liggende lenger tid i minnet, som øker muligheten for at dataene kan gjenbrukes av andre prosesser slik at antall diskaksesser kan reduseres.



hi = hard interrupt, si = soft interrupt, wa = i/o wait, us = user, sy = system, id = idle, ni = nice

Figur 5.13: Kjerneimplementasjon (RTP-motor) målt med `top`



hi = hard interrupt, si = soft interrupt, wa = i/o wait, us = user, sy = system, id = idle, ni = nice

Figur 5.14: Applikasjonsnivåimplementasjon målt med `top`

Kapittel 6

Konklusjon

I denne oppgaven har vi evaluert eksisterende datastiimplementasjoner i Linux for overføring av mediadata fra disk til nettverk ved medianedlasting og streaming. Disse eksperimentene viste at Linux har en optimal implementasjon for medianedlasting, men at det er rom for store forbedringer når det gjelder streaming. På grunnlag av resultatene implementerte vi nye systemkall optimalisert for streaming, og RTP-motorer i kjernen for ytterligere å redusere ressursbruken ved streaming. De optimaliserte systemkallene ga forbedringer i CPU-bruk på opptil 30 prosent ved MTU-pakkestørrelse, mens RTP-motorene ga en forbedring på opptil 40 prosent.

For medianedlasting viste resultatene at `sendfile` er optimalt implementert med tanke på kopioperasjoner og kontekstbytter, og det er mulig å overføre en hel fil uten kopioperasjoner, og med kun et kall til kjernen. Sammenlignet med de andre datastiimplementasjonene er forskjellen i CPU-bruk for `sendfile` opptil 65 prosent, og `sendfile` er også bedre med tanke på andre delte ressurser som systembuss og minne.

Når det gjelder streaming viste resultatene at de eksisterende datastiimplementasjonene ikke er optimalt implementert for dette formålet. Ved bruk av `sendfile` er det ikke mulig for applikasjonen å utføre operasjoner på dataene som overføres, ettersom `sendfile` benytter en datasti innenfor kjernen. Databerøringsoperasjoner er nødvendig ved streaming av enkelte typer mediadata. `sendfile` er derfor bare aktuell ved streaming av mediadata hvor det ikke er nødvendig med databerøringsoperasjoner, men resultatene viste at `sendfile` ikke var optimalisert for dette. `sendfile` har ikke støtte for overføring av applikasjonsbuffere, og det er dermed nødvendig å bruke et eget kall til kjernen for overføring av applikasjonsgenererte headere. Streaming av mediadata ved bruk av `sendfile` fører derfor til et stort antall kontekstbytter, og er på det beste 23 prosent bedre enn de andre

datastiimplementasjonene.

Ved streaming av mediadata som krever databerøringsoperasjoner er `mmap` og `writev` det beste alternativet, når det benyttes `readahead` på det mappede minneområdet. Dette er mulig å få til med bruk av `madvise`, men er ikke benyttet på eksperimentene i denne oppgaven, ettersom vi ble klar over denne muligheten først etter at eksperimentene var utført. Uten `readahead` på det mappede minneområdet er `read` og `send` et bedre alternativ, med tanke på CPU-bruk, med pakkestørrelser på 1KB og MTU. Når det benyttes `readahead` på det mappede minneområdet har `mmap` og `writev` noe lavere eller lik CPU-bruk som `read` og `send` også for disse pakkestørrelsene. Sammenlignet med `read` og `send` er `mmap` og `writev` også bedre med tanke på delte ressurser som systembuss og minne, på grunn av at det er færre kopioperasjoner ved overføring av mediadataene fra disk til minne. `mmap` er optimalt implementert for overføring av data fra disk til et applikasjonsbuffer, ettersom det bare er nødvendig med et kall til kjernen, og dataene overføres uten kopioperasjoner. For overføring fra applikasjonsbufferet til nettverket eksisterer det ingen optimal implementasjon med tanke på kopioperasjoner. `writev`, som er den beste eksisterende løsningen for overføring av data fra disk til nettverk i kombinasjon med `mmap`, krever en kopieringsoperasjon og et kall til kjernen for hver pakke som overføres.

På grunnlag av resultatene for de eksisterende systemkallene implementerte vi systemkallene `msend`, `rtpmsend` og `rtpsendfile` som er optimalisert for streaming. `msend` og `rtpmsend` er systemkall for overføring av mediadata, fra et minneområde mappet med `mmap`, til nettverket uten kopioperasjoner. `rtpmsend` er en utvidelse av `msend` med parametere for applikasjonsnivåheadere, slik at det er mulig å overføre en pakke med kun et systemkall. `rtpsendfile` er en optimalisering av `sendfile` slik at det er mulig å overføre en pakke med kun et kall til kjernen. Sammenlignet med `mmap` og `writev` ga `mmap` og `rtpmsend` en reduksjon i CPU-bruk på opptil 30 prosent. For `msend` var forberdingene mindre på grunn av flere kontekstbytter. Mens for `rtpsendfile` var forbedringene på 10 prosent ved MTU-pakkestørrelse, sammenlignet med `sendfile`. `msend` og `rtpmsend` gir noe høyere CPU-bruk enn `sendfile` og `rtpsendfile`, men ved bruk av `readahead` på det mappede minneområdet vil det sannsynligvis ikke være forskjell.

De optimalisrte systemkallene `rtpmsend` og `rtpsendfile` medfører et kall til kjernen for hver pakke som overføres, og for å fjerne disse kontekstbyttene implementerte vi RTP-motorprototypene `krtpmsend` basert på `mmap` og `msend`, og `krtpsendfile` basert på `sendfile`. `krtpmsend` ga en ytterligere forbedring på 16 prosent sammenlignet med `mmap` og `rtpmsend`, og 40 prosent sammenlignet med `mmap` og `writev`. For `krtpsendfile` er forbedringene på 19 prosent sammenlignet med `rtpsendfile`, og 30 prosent sammenlignet med `sendfile`. Eksperimentet med RTP-motoren basert på `sendfile` viste at en kjerneimplementa-

sjon kan levere 20 prosent flere strømmer enn en tilsvarende implementasjon på applikasjonsnivå.

For å bedre støtte streaming i Linux bør det derfor implementeres et `writev`-systemkall hvor det ikke er noen kopiering av mediadata mappet med `mmap`, og det bør implementeres et `sendfilev`-systemkall hvor det er mulig å kombinere data fra disk og applikasjonsbuffere i et systemkall.

Bibliografi

[1] *Linux Programmer's Manual*.

[2] DivX.com, August 2005. http://www.divx.com.

[3] DVD Demystified, August 2005. http://dvddemystified.com/.

[4] Fraunhofer iis, August 2005. http://www.iis.fraunhofer.de/index.html.

[5] *Intel Architecture Software Developer's Manual Volume 3: System Programming*, August 2005. www.intel.com/design/intarch/manuals/24281601.pdf.

[6] Oprofile profiling system for linux 2.2/2.4/2.6, August 2005. http://oprofile.sourceforge.net.

[7] Bernhard Baumgartner. Europe: HD Ready? The Implementation of HDTV in the European Digital TV Environment. Harris Broadcast Communications Division, Rankweil, Austria, May 2005. http://www.videsignline.com/howto/marketresearch/163102011.

[8] Anthony Betz and Paul Gray. Gigabit over copper evaluation, April 2002. http://www.cs.uni.edu/ gray/gig-over-copper/gig-over-copper.html.

[9] Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, and Raymond S. Tomlinson. TENEX, a paged time sharing system for the PDP - 10. *Commun. ACM*, 15(3):135–143, 1972.

[10] Edward Bradford. Runtime: Block memory copy (part 2) – high-performance programming techniques on linux and windows. http://www-128.ibm.com/developerworks/linux/library/l-rt3/, July 2001.

[11] Milind M. Buddhiko, Xin Jane Chen, Dakang Wu, and Guru M. Parulkar. Enhancements to 4.4BSD UNIX for Efficient Networked Multimedia in Project MARS. In *Proceedings of the International Conference on Multimedia Computing and Systems (ICMCS)*.

[12] D. Hoffman G. Fernando V. Goyal M. Civanlar. *RTP Payload Format for MPEG1/MPEG2 Video*. Internet Engineering Task Force, January 1998. RFC 2250.

[13] D.D. Clark and D.L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Computer Communications Review*, volume 20. Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols, September 1990.

[14] Charles Coffing. An x86 protected mode virtual machine monitor for the MIT exokernel. Master's thesis, Parallel & Distributed Operating System Group, MIT, Cambridge, MA, USA, May 1999.

[15] Charles D. Cranor and Gurudatta M. Parulkar. The UVM virtual memory system. In *Proceedings of the USENIX Annual Technical Conference*, pages 117–130, Monterey, CA, USA, June 1999.

[16] S. Davidson, J.W. Jinturkar. Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. In *Microarchitecture, Proceedings of the 28th Annual International Symposium on Ann Arbor, MI*, pages 125–132. IEEE, November 1995.

[17] FreeBSD, August 2005. http://www.freebsd.org/.

[18] Carsten Griwodz and Pål Halvorsen. Inf5070 mediatjener og distribusjonssystemer, August 2005. http://www.uio.no/studier/emner/matnat/ifi/INF5070/.

[19] Mike Hayward. Intel p6 vs p7 system call performance. LWN.net, Dec 2002. http://lwn.net/Articles/18412/.

[20] Information Sciences Institute University of Southern California. *Transmission Control Protocol*, September 1981. RFC 793.

[21] Intel Corporation. Block copy using PentiumIII streaming SIMD extensions (revision 1.9). ftp://download.intel.com/design/servers/softdev/-copy.pdf, 1999.

[22] Keith Jack. *Video Demystified: a handbook for the digital engineer. 2nd ed.* HighText publications, 1996.

[23] H. Schulzrinne S. Casner R. Frederick V. Jacobson. *RTP: A Transport Protocol for Real-Time Applications*. Internet Engineering Task Force, January 1996. RFC 1889.

[24] M. Handley V. Jacobson. *SDP: Session Description Protocol*. Internet Engineering Task Force, April 1998. RFC 2327.

[25] Dave Jones. The post-halloween document. v0.50, Oct 2005. http://www.codemonkey.org.uk/post-halloween-2.5.txt.

[26] Jiantao Kong and Karsten Schwan. Kstreams: Kernel support for efficient end-to-end data streaming. Technical Report GIT-CERCS-04-04, College of Computing, Georgia Institute of Technology, Atlanta, GA, USA, 2004.

[27] H. Schulzrinne A. Rao R. Lanphier. *Real Time Streaming Protocol (RTSP)*. Internet Engineering Task Force, April 1998. RFC 2326.

[28] Jonathan Lemon, Zhe Wang, Zheng Yang, and Pei Cao. Stream engine: A new kernel interface for high-performance internet streaming servers. In *Eighth International Workshop on Web Content Caching and Distribution*, IBM T.J. Watson Research Center, Hawthorne, NY, USA, September 2003.

[29] mpeg.org. DVD Technical Notes, August 2005. http://www.mpeg.org/MPEG/DVD/.

[30] Microsoft Developer Network MSDN, August 2005. http://msdn.microsoft.com/.

[31] Ram Pai, Badari Pulavarty, and Mingming Cao. Linux 2.6 performance improvement through readahead optimization. In *Linux Symposium*, Ottawa, Canada, July 2004.

[32] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Io-lite: a unified i/o buffering and caching system. *ACM Trans. Comput. Syst.*, 18(1):37–66, 2000.

[33] Kyung-Ho Kim Seung-Ho Lim Kyu-Ho Park. Adaptive read-ahead and buffer management for multimedia systems. The 8 IASTED International Conference on IMSA 2004, 2004.

[34] Joseph Pasquale, Eric W. Anderson, and P. Keith Muller. Container Shipping - Operating System Support for I/O-Intensive Applications. 27(3):84–93, March 1994.

[35] R. Stewart Q. Xie K. Morneault C. Sharp H. Schwarzbauer T. Taylor I. Rytina M. Kalla L. Zhang V. Paxson. *Stream Control Transmission Protocol*. Internet Engineering Task Force, October 2000. RFC 2960.

[36] Christian Poellabauer, Karsten Schwan, Richard West, Ivan Ganev, Neil Bright, and Gregory Losik. Flexible user/kernel communication for real-time applications in elinux. In *Proceedings of the Workshop on Real Time Operating Systems and Applications and Second Real Time Linux Workshop*, Orlando, FL, USA, November 2000.

[37] J. Postel. *User Datagram Protocol*. Internet Engineering Task Force, August 1980. RFC 768.

[38] S. J. Leffler M. K. McKusick M. J. Karels J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, 1989.

[39] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond softnet. In *5th Annual Linux Showcase and Conference*, pages 165–172. USENIX, November 2001.

[40] Henning Sanneck. *Packet Loss Recovery and Control for Voice Transmission over the Internet*. PhD thesis, Technical University of Berlin, October 2000.

[41] Ralf Steinmetz and Klara Nahrstedt. *Multimedia: Computing, Communications & Applications*. Prentice Hall, 1995.

[42] Sun, August 2005. http://www.sun.com/.

Tillegg A

Publiserte artikler

1. Halvorsen, P., Dalseng, T.A., Griwodz, C.:"Assessment of Data Path Implementations for Download and Streaming", Proceedings of the 11th International Conference on Distributed Multimedia Systems (DMS'2005), Banff, Canada, September 2005, pp. 228 - 233

Assessment of Data Path Implementations for Download and Streaming

Pål Halvorsen[1,2], Tom Anders Dalseng[1], Carsten Griwodz[1,2]
[1]IFI, University of Oslo, Norway [2]Simula Research Laboratory, Norway
Email: {paalh, tdalseng, griff}@ifi.uio.no

Abstract

Distributed multimedia streaming systems are increasingly popular due to technological advances, and numerous streaming services are available today. On servers or proxy caches, there is a huge scaling challenge in supporting thousands of concurrent users that request delivery of high-rate, time-dependent data like audio and video, because this requires transfers of large amounts of data through several sub-systems within a streaming node. Since the speed increase for memory accesses does not follow suite with the CPU speed, copy operations can be a severe limiting factor on the streaming performance of off-the-shelf operating systems, which still have only limited support for data paths that have been optimized for streaming despite previous research proposals. We observe furthermore that while CPU speed continues to increase, system call overhead has grown as well, adding to the cost of data movement. In this paper, we therefore revisit the data movement problem and provide a comprehensive evaluation of possible streaming data I/O paths in Linux 2.6 kernel. We have implemented and evaluated enhanced mechanisms and show how to provide support for more efficient memory usage and reduction of user/kernel space switches for streaming applications.

1 Introduction

Improvements in access network connectivity with flat-rate Internet connections, such as DSL, cable modems and recently E-PON, and large improvements in machine hardware make distributed multimedia streaming applications increasingly popular and numerous streaming services are available today, e.g., movie-on-demand (Broadpark), news-on-demand (CNN), media content download (iTunes), online radio (BBC), Internet telephony (Skype), etc. At the client side, there is usually no problem presenting the streamed content to the user. On the server side or on intermediate nodes like proxy caches, however, the increased popularity and the increasing data rates of access networks make the scaling challenge even worse when thousands of concurrent users request delivery of high-rate, time-dependent data like audio and video.

In such media streaming scenarios (and many others) that do not require data touching operations, the most expensive server-side operation is moving data from disk to network including encapsulating the data in application- and network packet headers. A proxy cache may additionally forward data from the origin server, make a cached copy of a data element, perform transcoding, etc. Thus, both servers and intermediate nodes that move large amounts of data through several sub-systems within the node may experience high loads as most of the performed operations are both resource and time consuming. Especially, memory copying and address space switches consume a lot of resources [1, 2], and since improvements in memory access speed do not keep up with the increase in CPU speed, these operations will be a severe limiting factor on streaming performance of off-the-shelf operating systems having only limited support for optimized data paths.

In the last 15 years, the area of data transfer overhead has been a major thread in operating system research. In this paper, we have made a Linux 2.6 case study to determine whether more recent hardware and commodity operating systems like Linux have been able to overcome the problems and how close to more optimized data paths the existing solutions are. The reason for this is that a lot of work has been performed in the area of reducing data movement overhead, and many mechanisms have been proposed using virtual memory remapping and shared memory as basic techniques. Off-the-shelf operating systems today frequently include data path optimizations for common applications, such as web server functions. They do not, however, add explicit support for streaming media, and consequently, a lot of streaming service providers make their own implementations. We investigate therefore to which extent the generic functions are sufficient and whether dedicated support for streaming applications can still considerably improve performance. We revisit this data movement problem and provide a comprehensive evaluation of different mechanisms using different data paths in the Linux 2.6 kernel. We have performed several experiments to see the real performance retrieving data from disk and sending data as RTP packets to remote clients. Additionally, we have also implemented and evaluated enhanced mechanisms and we can show that they still improve the performance of streaming operations by providing means for more efficient memory usage and reduction of user/kernel space switches. In particular, we are

able to reduce the CPU usage by approximately 27% compared to best existing case removing copy operations and system calls for a given stream.

The rest of this paper is organized as follows: Section 2 gives a small overview of examples of existing mechanisms. In section 3, we present the evaluation of existing mechanisms in the Linux 2.6 kernel, and section 4 describes and evaluates some new enhanced system calls improving the disk-network data path for streaming applications. Section 5 gives a discussion, and finally, in section 6, we conclude the paper.

2 Related Work

The concept of using buffer management to reduce the overhead of cross-domain data transfers to improve I/O performance is rather old. It has been a major issue in operating systems research where variants of this work have been implemented in various operating systems mainly using virtual memory remapping and shared memory as basic techniques. Already in 1972, *Tenex* [3] used virtual copying, i.e., several pointers in virtual memory to one physical page. Later, several systems have been designed which use virtual memory remapping techniques to transfer data between protection domains without requiring several physical data copies. An interprocess data transfer occurs simply by changing the ownership of a memory region from one process to another. Several general purpose mechanisms supporting a zero-copy data path between disk and network adapter have been proposed, including Container Shipping [4], IO-Lite [1], and UVM virtual memory system [5] which use some kind of page remapping, data sharing, or a combination. In addition to mechanisms removing copy operations in all kinds of I/O, some mechanisms have been designed to create a fast in-kernel data path from one device to another, e.g., the disk-to-network data path. These mechanisms do not transfer data between user and kernel space, but keep the data within the kernel and only map it between different kernel sub-systems. This means that target applications comprise data storage servers for applications that do not manipulate data in any way, i.e., no data touching operations are performed by the application. Examples of such mechanisms are the *stream* system call [6], the Hi-Tactix system [7], KStreams [8] and the *sendfile* system call (for more references, see [2]).

Besides memory movement, system calls are expensive operations, because each call to the kernel requires two switches. Even though in-kernel data paths remove some of this overhead, many applications still require application level code that makes kernel calls. Relevant approaches to increase performance include batched system calls [9] and event batching [10].

Although these examples show that an extensive amount of work has been performed on copy and system call avoidance techniques, the proposed approaches have usually remained research prototypes for various reasons, e.g., they are implemented in own operating systems (having an impossible task of competing with Unix and Windows), small implementations for testing only, not integrated with the main source tree, etc. Therefore, only some limited support is included in the most used operating systems today like the *sendfile* system call in UNIX, Linux, AIX and *BSD. In the next section, we therefore evaluate the I/O pipeline performance of the new Linux 2.6 kernel.

3 Existing Mechanisms in Linux

Despite all the proposed mechanisms, only a limited support for various streaming applications is provided in commodity operating systems like Linux. The existing solutions for moving data from storage device to network device usually comprise combinations of the *read/write*, *mmap* and *sendfile* system calls. Below, we present the results of our performance tests using combinations of these for **content download** operations (adding no application level information) and **streaming** operations (adding application level RTP header for timing and sequence numbering).

3.1 Test Setup

The experiments were performed using two machines connected by a point-to-point Ethernet connection. The test machine has an Intel 845 chipset, 1.70 GHz Intel Pentium4 CPU, 400 MHz front side bus and 1 GB PC133 SDRAM. The resource usage is measured using the *getrusage* function measuring consumed user and kernel time to transfer 1 GB of data stored using the Reiser file system in Linux 2.6. Below, we have added the user and kernel time values to get the total resource consumption, and each test is performed 10 times to get more reliable results. However, the differences between the tests are small.

3.2 Copy and Switching Performance

Before looking at the disk-network data transfer performance, we first look at the memory copy and system call performance themselves. In figure 1, an overview of the chipset on our test machine is shown (similar to many other Intel chipsets). Transfers between device and memory are typically performed using DMA transfers that move data over the PCI bus, the I/O controller hub, the hub interface, the memory controller hub and the RAM interfaces. A memory copy operation is performed moving data from RAM over the RAM interfaces, the memory controller hub, the system (front side) bus through the CPU and back to another RAM location with the possible side effect of flushing the cache(s). Data is (possibly unnecessarily) transferred several times through shared components reducing the overall system performance.

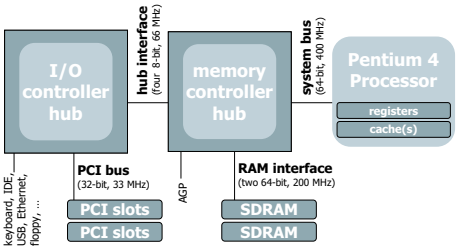
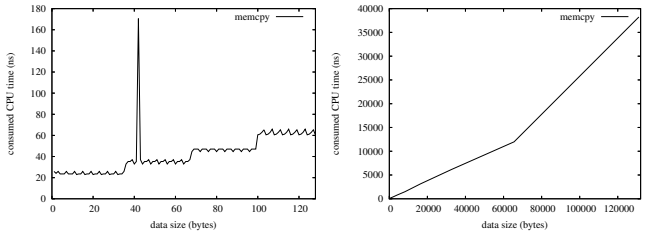


Figure 1. Pentium4 processor and 845 chipset

In [11], memory copy performance was measured on Linux (2.2 and 2.4) and Windows (2000) where the conclusion was that *memcpy* performs well (compared to other copy functions/instructions), and Linux is in most cases faster than Windows depending on data size and used copy instruction. Furthermore, to see the performance on our test machine, we tested *memcpy* using different data sizes. Figure 2a shows that the overhead is slowly growing with the size of the data element, but after reaching a certain size (having cache size effects) the overhead increases more or less linearly with the size (figure 2b)[1].



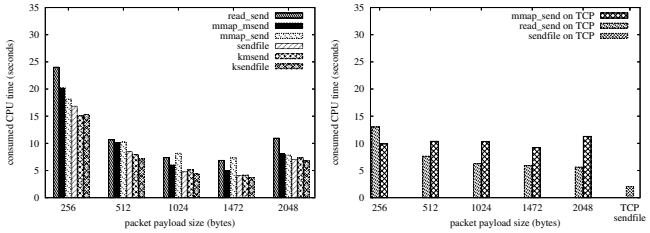
a) b)

Figure 2. User space memory copy speed

With respect to switching contexts, system call overhead (and process context switches) are time consuming on Pentium4 [12]. To get an indication of the system call overhead on our machine, we measured the *getpid* system call, accessing the kernel and only returning the process id. Our experiments show that the average time to access the kernel and return back is approximately 920 nanoseconds for each call.

Copy and system call performance has also been an issue for hardware producers like Intel, who has added new instructions, in particular MMX and SIMD extensions useful for copy operations and *sysenter* and *sysexit* instructions particularly for system calls. For example, using SIMD instructions, the block copy operation speed was improved by up to 149% in the Linux 2.0 kernel, but the reduction in CPU usage was only 2% [13]. Thus, both copy and kernel access performance still are resource consuming and remain possible bottlenecks.

[1]The single high peek in figure 2a at 42 bytes is due to one single test being interrupted by an external event. Additionally, some strange artifacts can be seen when the size of the memory address are not 4-byte aligned with even higher peeks in user space. This is also apparent for larger data sizes and is probably due to different instructions.



a) UDP b) TCP

Figure 3. Content download operations

3.3 Disk-Network Path

The functions used for retrieving data from disk into memory are usually *read* or *mmap*. Data is transfered using DMA from device to memory, and in case of *read*, we require an in-memory copy operation to give the application access to data whereas *mmap* shares data between kernel and user space. To send data, *send* (or similar) can be used where the payload is copied from the user buffer or the page cache depending on whether *read* or *mmap* is used, respectively, to the socket buffer (*sk_buf*). Then, the data is transfered in a DMA operation to the network device. Another approach is to use *sendfile* sending the whole file in one operation, i.e., data is sent directly from a kernel buffer to the communication system using an in-kernel data path. Thus, if gather DMA operations are supported, i.e., needed because the payload and the generated headers are located in different (*sk_buf*) buffers, data can be sent from disk to network without any in-memory copy operations.

3.4 Content Download Experiments

The first test we performed looking at the whole disk-network data path was in a content download scenario. Here, data needs only to be read from disk and sent as soon as possible without application level control. Thus, there is no need to add application level information. To evaluate the performance, we performed several tests using the different data paths and system calls described in section 3.3 and table 1-A using both TCP and UDP. For UDP we also added three enhanced system calls to be able to test a download scenario similar to *sendfile* with TCP.

The results for UDP are shown in figure 3a. We see, as expected, that removal of copy operations and system calls both give performance improvements. Furthermore, in figure 3b, the results using TCP are shown. Again, we see that a quite a lot of resources can be freed using *sendfile* compared to the two other approaches making several system calls and copy operations per data element. Note, however, that it seems that *getrusage* is still not fully implemented for TCP in the 2.6 kernel. Thus, the TCP and UDP experiments are not directly comparable.

From the results, we can see that the existing *sendfile* over TCP performs very well compared to the other tests

A – content download

	co	s	calls to the kernel
read_write	$2n$	$4n$	n ***read*** and n ***write*** calls (TCP will probably gather several smaller elements into one larger MTU-sized packet)
mmap_send	n	2+2n	1 ***mmap*** and n ***send*** calls (TCP will gather several smaller elements into one larger MTU-sized packet)
sendfile (UDP)	0	$2n$	n ***sendfile*** calls
sendfile (TCP)	0	2	1 ***sendfile*** call
mmap_msend†	0	2+2n	1 ***mmap*** and n ***msend*** calls (*msend*† sends data over UDP using the virtual address of a *mmap*'ed file instead of copying the data)
kmsend†	0	2+2n	1 ***kmsend*** call (*kmsend*† combines *mmap* and *msend* (see above) in the kernel until the whole file is sent)
ksendfile†	0	2	1 ***ksendfile*** call (*ksendfile*† performs *sendfile* over UDP in the kernel a la *sendfile* for TCP)

B – RTP streaming

	co	s	calls to the kernel
read_send_rtp	$2n$	$4n$	n ***read*** and n ***send*** calls (RTP header is placed in user buffer in front of payload, i.e., no extra copy operation)
read_writev	$3n$	$4n$	n ***read*** and n ***writev*** calls (RTP header is generated in own buffer, *writev* write data from two buffers)
mmap_send_rtp	$2n$	2+8n	1 ***mmap***, n ***cork***, n ***send***, n ***send*** and n ***uncork*** calls (need one send call for both data and RTP header)
mmap_writev	$2n$	2+2n	1 ***mmap*** and n ***writev*** calls
rtp_sendfile	n	$8n$	n ***cork***, n ***send***, n ***sendfile*** and n ***uncork*** calls

C – enhanced RTP streaming

	co	s	calls to the kernel
mmap_rtpmsend†	n	2+2n	1 ***mmap*** and n ***rtpmsend*** calls (*rtpmsend*† copies RTP headers from user space and adds payload from *mmap*'ed files as payload in the kernel)
mmap_send_msend†	n	2+8n	1 ***mmap***, n ***cork***, n ***send***, n ***msend*** and n ***uncork*** calls (no data copying using *msend*, but the RTP header must be copied from user space)
rtpsendfile†	n	$2n$	n ***rtpsendfile*** calls (*rtpsendfile*† adds the RTP header copy operation to the *sendfile* system call)
krtpsendfile†	0	2	1 ***krtpsendfile*** call (*krtpsendfile*† adds RTP headers to *sendfile* in the kernel)
krtpmsend†	0	2	1 ***krtpmsend*** call (*krtpmsend*† adds RTP headers to the *mmap*/*msend* combination (see above) in the kernel)

co = number of *copy operations*, s = number of *switches* between user and kernel space, n = number of packets, † = new enhanced system call

Table 1. Descriptions of the performed tests

as applications only have to make one single system call to transfer a whole file. Consequently, if no data touching operations, no application level headers or timing support are necessary, *sendfile* seems to be efficiently implemented and achieves a large performance improvement compared to the traditional *read* and *write* system calls, especially when using TCP where only one system call is needed to transfer the whole file.

3.5 Streaming Experiments

Streaming time-dependent data like video to remote clients typically requires adding per-packet headers, such as RTP headers for sequence numbers and timing information. Thus, plain file transfer optimizations are insufficient, because file data must be interleaved with application generated headers, i.e., additional operations must be performed. To evaluate the performance of the existing mechanisms, we performed several tests using the set of data paths and system calls listed in table 1-B. As shown above, the application payload can be transfered both with and without user space buffers, but the RTP header must be copied and interleaved within the kernel. Since TCP may gather several packets into one segment, i.e., the RTP headers will be useless, we have only tested UDP. The results of our tests are shown in figure 4a. Compared to the ftp-like operations in the previous section, we need many system calls and copy operations. For example, compared to the *sendfile* (UDP) and the enhanced *ksendfile* tests in figure 3, there are a 21% and a 29% increase in the measured overhead for the *rtp_sendfile* using Ethernet MTU-sized packets, respectively. This is because we now also need an additional *send* call for the RTP header. Thus, the results indicate that there is a potential for improvements. In the next section, we therefore describe some possible improvements and show that already minor enhancements can achieve large gains in performance.

4 Enhancements for RTP streaming

Looking at the existing mechanisms described and analyzed in the previous section, we are more or less able to remove copy operations (except the small RTP header), but the number of user/kernel boundary crossings is high. We have therefore implemented a couple of other approaches listed in table 1-C. With respect to overhead, *mmap_rtpmsend*, *rtpsendfile*, *krtpmsend* and *krtpsendfile* look promising:

- *mmap_rtpmsend* uses *mmap* to share data between file system buffer cache and the application. Then, it uses the enhanced *rtpmsend* system call to send data copying a user-level generated RTP header and adding the mapped file data using a virtual memory pointer instead of a physical copy. This gives n in-memory data transfers and $1 + n$ system calls. (A further improvement would be to use a virtual memory pointer for the RTP header as well)
- *krtpmsend* uses *mmap* to share data between file system buffer cache and the application and uses the en-

hanced *msend* system call to send data using a virtual memory pointer instead of a physical copy. Then, the RTP header is added in the kernel by a kernel-level RTP engine. This gives no in-memory data transfers and only 1 system call.

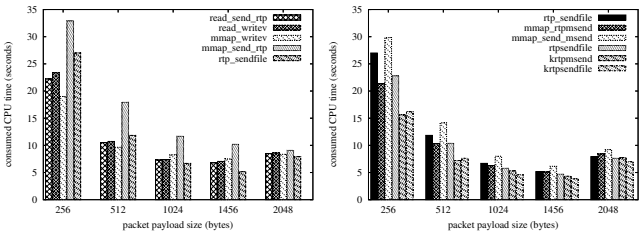
- *rtpsendfile* is a modification of the *sendfile* system call. Instead of having an own call for the RTP header transfer, an additional parameter (a pointer to the buffer holding the header) is added, i.e., the data is copied in the same call and sent as one packet. This gives only n in-memory data transfers and n system calls.
- *krtpsendfile* uses *ksendfile* to transmit a UDP stream in the kernel, in contrast to the standard *sendfile* requiring one system call per packet for UDP. Additionally, the RTP header is added in the kernel having an in-kernel RTP engine. This gives no in-memory data transfers and only 1 system call.

The two first mechanisms are targeted at applications requiring the possibility to touch data in user-space, e.g., parsing or sporadic modifications[2], whereas the last two mechanisms aim at data transfers without application-level data touching. All these enhanced system calls reduce the overhead compared to existing approaches, and to see the real performance gain, we performed the same tests as above. Our results, shown in figure 4b, indicate that simple mechanisms can remove both copy and system call overhead. For example, in the case of streaming using RTP, we see an improvement of about 27% using *krtpsendfile* where a kernel engine generates RTP headers compared to *rtp_sendfile* in the scenario with MTU-sized packets. If we need the same user level control making one call per packet, the *rtpsendfile* enhancement gives at least a 10% improvement compared to existing mechanisms. In another scenario where the application requires data touching operations, the existing mechanism only have small differences. If comparing the results for MTU-sized packets, *read_send_rtp* (already optimized to read data into the same buffer as the generated RTP header) performs best in our tests. However, using a mechanism like *krtpmsend* gives a performance gain of 36% compared to *read_send_rtp*. Similar user level control by making one call per packet is achieved by *mmap_rtpmsend* which gives a 24% gain. Additionally, similar results can in general also be seen for smaller packet sizes (of course with higher overhead due to a larger number of packets), and when the transport level packet exceeds the MTU size, additional fragmentation of the packet introduces additional overhead.

5 Discussion

The enhancements described in this paper to reduce the number of copy operations and system calls mainly address application scenarios where data is streamed to the client

[2]Non-persistant modifications to large parts of the files require a data copy in user space, voiding the use of the proposed mechanisms.



a) Existing mechanisms b) Enhancements

Figure 4. Streaming performance

without any data manipulation at the server side. However, several of the enhanced system calls also show that the application can share a buffer with the kernel and can interleave other information into the stream. Thus, adding support for data touching operations, like checksumming, filtering, etc. without copying, and data modification operations, like encryption, transcoding, etc. with one copy operation, should be trivial. The in-kernel RTP engine also shows that such operations can be performed in the kernel (as kernel stream handlers), reducing copy and system call overhead.

An important issue is whether data copying is still a bottleneck in systems today. The hardware has improved, and one can easily find other possible bottleneck components. However, as shown in section 3.2, data transfers through the CPU are time and resource consuming and have side effects like cache flushes. The overhead increases approximately linearly with the amount of data, and as the gap between memory and CPU speeds increases, so does the problem. Additionally, in figure 5 (note that the y-axis starts at 0.5), we show the performance of the different RTP streaming mechanisms relative to *read_writev*, i.e., a straight forward approach reading data into an application buffer, generating the RTP header and writing the two buffers to the kernel using the vector write operation. Looking for example at MTU-sized packets, we see that a lot of resources can be freed for other tasks. We can also see that less intuitive but more efficient solutions than *read_writev* that do not require kernel changes exist, for example using *sendfile* combined with a *send* for the RTP header (*rtp_sendfile*). However, the best enhanced mechanism, *krtpsendfile*, removes all copy operations and makes only one access to the kernel compared

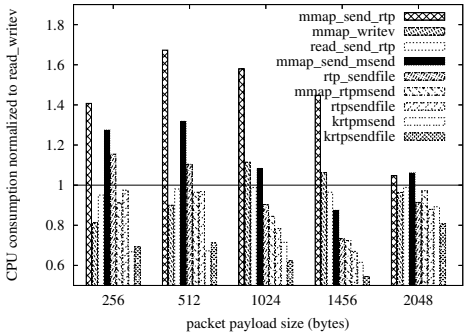


Figure 5. Relative performance to *read_writev*

to *rtp_sendfile* which requires several of both (see table 1). With respect to consumed processor time, we achieve an average reduction of 27% using *krtpsendfile*. Recalculating this into (theoretical) throughput, *rtp_sendfile* and *krtpsendfile* can achieve 1.55 Gbps and 2.12 Gbps, respectively. Assuming a high-end 3.60 GHz CPU like Pentium4 660 and an 800 MHz front side bus, the respective numbers should be approximately doubled. These and higher rates are also achievable for network cards (e.g., Force10 Network's E-Series), PCI express busses and storage systems (e.g., using several Seagate Cheetah X15.4 in a RAID). Thus, the transfer and processing overheads are still potential bottlenecks, and the existing mechanisms should be improved.

Now, having concluded that data transfers and system calls still are possible bottlenecks and having looked at possible enhancements, let us look at what a general purpose operating system like Linux miss. Usually, the commodity operating systems aim at a generality, and new system calls are not frequently added. Thus, specialized mechanisms like *krtpsendfile* and *krtpmsend* having application specific, kernel-level RTP-engines, will hardly ever be integrated into the main source tree and will have to live as patches for interested parties like streaming providers, e.g., like the Red Hat Content Accelerator (*tux*) for web services. However, support for adding application level information (like RTP headers) to stored data will be of increasing importance in the future as streaming services really explode. Simple enhancements like *mmap_rtpmsend* and *rtpsendfile* might be general, performance improving mechanisms that could be of interest in scenarios where the application does or does not touch the data, respectively.

6 Conclusions

In this paper, we have shown that (streaming) applications still pay a high (unnecessary) performance penalty in terms of data copy operations and system calls if those applications require packetization such as addition of RTP headers. We have therefore implemented several enhancements to the Linux kernel, and evaluated both existing and the new mechanisms. Our results indicate that data transfers still are potential bottlenecks, and simple mechanisms can remove both copy and system call overhead if a gather DMA operation is supported. In the case of a simple content download scenario, the existing *sendfile* is by far the most efficient mechanism, but in the case of streaming using RTP, we see an improvement of at least 27% over the existing methods using MTU-sized packets and the *krtpsendfile* system call with a kernel engine generating RTP headers. Thus, using mechanisms for more efficient resource usage, like removing copy operations and avoiding unnecessary system calls, can greatly improve a node's performance. Such enhancements free resources like memory, CPU cycles, bus cycles, etc. which now can be utilized by other applications or providing support for more concurrent streams.

Currently, we also have other kernel activities on-going, and we hope to be able to integrate our subcomponents. We will also modify the KOMSSYS video server to use the proposed mechanisms and perform more extensive tests including a workload experiment looking at the maximum number of concurrent clients able to achieve a timely video playout. Finally, we will optimize our implementation, because most of the enhancements are implemented as proof-of-concept removing copy operations and system calls. We have made no effort in optimizing the code, so the implementations have large potential for improvement, e.g., moving the send-loop from the system call layer to the page cache for the *krtpsendfile* which will remove several file lookups and function calls (as for the existing *sendfile*).

References

[1] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Io-lite: a unified i/o buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, February 2000.

[2] Pål Halvorsen. *Improving I/O Performance of Multimedia Servers*. PhD thesis, Department of Informatics, University of Oslo, Oslo, Norway, August 2001.

[3] Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, and Raymond S.Tomlinson Bolt Beranek. Tenex, a paged time sharing system for the pdp-10. *Communications of the ACM*, 15(3):135–143, March 1972.

[4] Eric W. Anderson. *Container Shipping: A Uniform Interface for Fast, Efficient, High-Bandwidth I/O*. PhD thesis, Computer Science and Engineering Department, University of California, San Diego, CA, USA, July 1995.

[5] Charles D. Cranor and Gurudatta M. Parulkar. The UVM virtual memory system. In *Proceedings of the USENIX Annual Technical Conference*, pages 117–130, Monterey, CA, USA, June 1999.

[6] Frank W. Miller and Satish K. Tripathi. An integrated input/output system for kernel data streaming. In *Proceedings of SPIE/ACM Conference on Multimedia Computing and Networking (MMCN)*, pages 57–68, San Jose, CA, USA, January 1998.

[7] Damien Le Moal, Tadashi Takeuchi, and Tadaaki Bandoh. Cost-effective streaming server implementation using hi-tactix. In *Proceedings of the ACM International Multimedia Conference (ACM MM)*, pages 382–391, Juan-les-Pins, France, December 2002.

[8] Jiantao Kong and Karsten Schwan. Kstreams: Kernel support for efficient end-to-end data streaming. Technical Report GIT-CERCS-04-04, College of Computing, Georgia Institute of Technology, Atlanta, GA, USA, 2004.

[9] Charles Coffing. An x86 protected mode virtual machine monitor for the mit exokernel. Master's thesis, Paralell & Distributed Operating System Group, MIT, Cambridge, MA, USA, May 1999.

[10] Christian Poellabauer, Karsten Schwan, Richard West, Ivan Ganev, Neil Bright, and Gregory Losik. Flexible user/kernel communication for real-time applications in elinux. In *Proceedings of the Workshop on Real Time Operating Systems and Applications and Second Real Time Linux Workshop*, Orlando, FL, USA, November 2000.

[11] Edward Bradford. Runtime: Block memory copy (part 2) – high-performance programming techniques on linux and windows. http://www-106.ibm.com/developerworks/library/l-rt3/, July 1999.

[12] Gregory McGarry. Benchmark comparison of netbsd 2.0 and freebsd 5.3. http://www.feyrer.de/NetBSD/gmcgarry/, January 2005.

[13] Intel Corporation. Block copy using PentiumIII streaming SIMD extensions (revision 1.9). ftp://download.intel.com/design/servers/-softdev/copy.pdf, 1999.