Energiatehokkuuden huomioiva laskenta

Jukka Peltomäki

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaaja: Erkki Mäkinen
Heinäkuu 2013

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Jukka Peltomäki: Energiatehokkuuden huomioiva laskenta
Pro gradu -tutkielma, 40 sivua
Heinäkuu 2013

Energian kulutuksen vähentäminen on tärkeä ja ajankohtainen aihe. Silti teknologian kuluttama sähkön määrä on jatkuvassa nousussa. Tästä johtuen energian säästämiseen teknologiassa on syytä kiinnittää huomiota. Tässä tutkielmassa kerron erilaisista algoritmisista tavoista vähentää laskentaan kuluvan energian määrää. Otan erityistarkasteluun viimeaikaiset nousevat trendit, eli datakeskukset ja mobiiliympäristön. Esittelen aikataulutukseen kehitettyjä algoritmeja, jotka vähentävät energian kulutusta. Esittelen myös virransäästöön siirtymiseen ja dynaamiseen nopeuden skaalaukseen tarkoitettuja algoritmeja sekä virransäästöä sovellusohjelmoijan näkökulmasta.

Avainsanat ja -sanonnat: energiatehokkuus, vihreä sähkö, aikataulutus, datakeskus, mobiiliympäristö, dynaaminen nopeuden skaalaus.

Sisällys

1.Johdanto..1
2.Energiatehokkuuden huomioon ottamisen tärkeys..2
3.Keskittäminen datakeskuksiin..4
3.1.Vihreän pilven arkkitehtuuri...6
3.2.Vihreän energian käytön maksimointi GreenSlot-aikataulutuksella..........9
4.Mobiiliympäristö..13
4.1.Tiedon pakkaus ennen lähetystä..16
4.2.Energiatehokkaan verkon valinta ennustusalgoritmien avulla..................17
4.3.Lähetyksen viivästyttäminen energiatehokkaamman yhteyden toivossa. 20
5.Ohjelmiston rooli..25
5.1.Virrankulutustilan vaihtoalgorimeista.. 25
5.2.Prosessorin nopeuden dynaaminen skaalaus...26
5.3.Ohjelmointinäkökulma – energiatyypit suoraan ohjelmointikieleen..........31
6.Yhteenveto..36

Viiteluettelo... 37

1. Johdanto

Energian säästö on ajankohtainen ja usein tunteitakin herättävä aihe. Mediassa ja keskusteluissa usein vilisevä puhe ilmastonmuutoksesta herättää monissa halun toimia ja tehdä jotain asian puolesta, kun taas monet suhtautuvat asiaan joko vähättelevästi tai luovutusmentaliteetillä. Pragmaattisesta näkökulmasta tarkasteltuna sähkön kulutuksen minimointi on hyödyllistä myös lyhyellä aikavälillä. Sähkö maksaa rahaa ja sitä mahtuu akkuunkin sen koosta riippumatta yhdellä latauksella vain tietty määrä. Mitä energiatehokkaammin datakeskus toimii, sen halvempia sen tuottamat palvelut voivat talouden lainalaisuuksien mukaan olla. Mitä energiatehokkaammin kännykkä toimii, sitä kauemmin siinä kestää akku.

Tässä tutkielmassa käyn läpi erilaisia tapoja säästää laskennassa energiaa algoritmiikan avulla. Keskityn tarkastelussa moderneihin nouseviin trendeihin, datakeskuksiin ja mobiililaitteisiin. Datakeskukset jatkavat kasvuaan, kun palveluja halutaan keskittää pilveen, ja modernit mobiililaitteet kehittyvät kovaa vauhtia ja niitä on yhä useammilla käytössään. Molempien trendien taustavoimana on verkottuminen ja verkkokapasiteetin kasvu, joka lisää datakeskusten ja mobiililaitteiden houkuttelevuutta. Yhtä kaikki, saman tyyppiset algoritmit ovat energian säästön kannalta hyödyllisiä lähes mihin tahansa käyttöön tarkoitetussa teknologiassa.

Luvussa 2 käsittelen energian säästämisen tärkeyttä ja haastetta hieman tarkemmin. Luvussa 3 kuvaan erilaisia tapoja säästää datakeskuksissa energiaa. Luvussa 4 keskityn kertomaan mobiililaitteista ja hieman myös mobiiliverkoista. Luvussa 5 käyn läpi yleisesti tapoja ja algoritmeja, joilla energiaa säästetään. Käyn läpi algoritmeja, joilla siirrytään virransäästötilaan ja skaalataan prosessorin nopeutta dynaamisesti. Kyseisessä luvussa kerron myös hieman virransäästöstä sovellusohjelmoijan näkökulmasta.

2. Energiatehokkuuden huomioon ottamisen tärkeys

Maailman sähkönkulutus nousee, ja teknologian osuus sähkönkulutuksesta nousee. Teknologia mielletään helposti ympäristöystävälliseksi, mutta Yhdysvalloissa vuonna 2006 datakeskukset kuluttivat 61.4 miljardia kWh, joka on yhtä paljon kuin koko valmistavan teollisuuden ja sen liikenteen sähköntarve Yhdysvalloissa samana vuonna! Lisäksi datakeskusten sähkönkäyttö on lisääntynyt vuodesta 2006. [Goiri et al., 2011] Keskikokoinen datakeskus kuluttaa sähköä noin 25 000 kotitalouden verran, ja datakeskukset tuottavat enemmän hiilidioksidipäästöjä kuin esimerkiksi Argentiina [Kaplan et al., 2008]. Yksittäisen palvelimen hiilijalanjälki voi olla jopa yhden katumaasturin verran. [Global Action Plan, 2007] Vuonna 2007 teknologiasektori tuotti 2%–2.5% maapallon kokonaishiilidoksidipäästöistä [Kelly, 2007] ja kulutti 3% kokonaissähköntuotannosta. Etenkin mobiiliteknologian käytön odotetaan yhä nousevan [Wang et al., 2011]. Samaan aikaan ilmastonmuutoksesta, päästörajoitteista, vihreästä energiasta, hiilidioksidipäästöistä ja muista isoista asioista puhutaan ja ollaan huolissaan ympäri maapalloa. Tästä paineesta saa osansa myös teknologiateollisuus. Teknologiassa tehostaminen on kuitenkin hyvin mahdollista, sillä esimerkiksi seuraavan sukupolven verkkojen on arveltu olevan 40% nykyisiä energiatehokkaampia ja modernien radiolähetinten tiputtavan virrankulutuksen murto-osaan entisestä. Teknologia myös mahdollistaa tietotyön tekemisen paikasta riippumatta, joten ympäristön kannalta positiiviset vaikutukset saattavat näkyä myös esimerkiksi pienempinä liikenteen päästöinä. [Kelly, 2007]

Mikropiireihin perustuvat laitteet kuluttavat sähkövirtaa. Kun piiri käyttää virtaa, syntyy tarpeettomasti sivutuotteena myös lämpöä. Piiriin siis syötetään virtaa, ja se tuottaa laskentaa ja lämpöä. Energiatehokkuus on sitä parempi, mitä pienemmällä virralla saadaan tietty määrä laskentaa mahdollisimman pienellä lämmöntuotolla.

Energiatehokkuus on nykyään oleellista kaiken kokoisissa laitteissa supertietokoneista kannettavaan elektroniikkaan. Perusproblematiikka – virran ja lämmön minimointi – on kaikissa sama, mutta painotukset poikkeavat. Pienessä kannettavassa laitteessa myös absoluuttisen sähkön kulutuksen on oltava pientä, kun taas isossa keskuksessa jäähdytys ja hyötysuhde nousevat kriittisiksi.

Teknologian kehittyessä sen virrantarve lisääntyy. Mikäli virrantarve jatkaa kasvuaan, voi energian hinta helposti ja selkeästi ohittaa laitteiston itsensä hinnan, arvioivat Googlen insinöörit serveripuolen tilanteesta [Barroso, 2005].

Mobiililaitteissa virransäästö on suoraa yhteydessä myös käyttömukavuuteen ja -kykyyn, sillä virtaa on akussa vain tietty määrä käytettävissä yhdellä latauksella. Mikäli

virransäästöominaisuudet toteutetaan fiksusti, ei käyttökokemus siitä myöskään muilla tavoin kärsi.

3. Keskittäminen datakeskuksiin

Suuret datakeskukset ovat verkkottumisen myötä lisääntyneet. Laskennan energiatehokkuuden problematiikan näkökulmasta ne ovat verrattavissa supertietokoneisiin. Molemmissa laskenta on keskitetty, ja tilasuunnittelussa on otettu huomioon lämmönsiirto ja sähkönsaanti. Ero näiden välillä on se, että supertietokoneet ratkovat vaikeampia, usein teknisiä, ongelmia, kun pilvi- ja nettipalveluita tuottavat laitteistot tekevät yksinkertaisempia tehtäviä verkon kautta käyttäjille. Pilvipalvelut on usein hajautettu useaan datakeskukseen palvelun vikasietoisuuden ja verkkolatenssin parantamiseksi. Supertietokoneiden laskennan tulokset taas saattavat olla hyvin tarkkaan rajattu esimerkiksi taloudellisista ja poliittisista syistä.

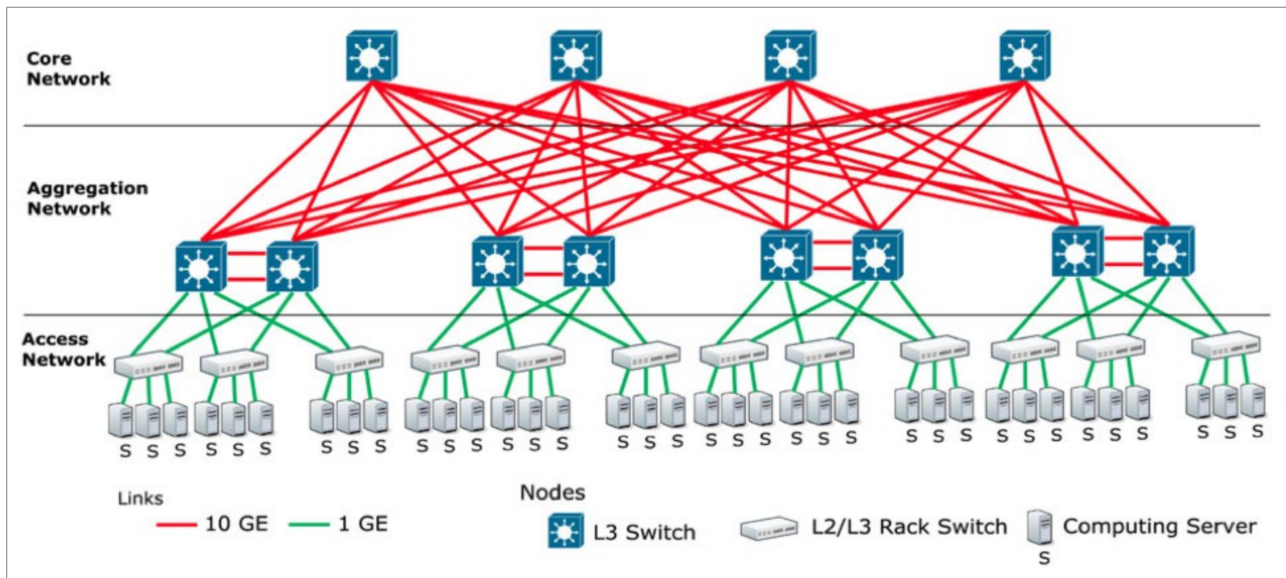
Modernit mikropiirit käyttävät runsaasti sähköä pienessä tilassa. Niissä myös syntyy valtavasti lämpöä. Lämmönsiirrosta huolehtiminen on nykyisin datakeskuksen tärkeimpiä suunnittelukohtia. Lämpö pitää saada johdettua tarpeeksi nopeasti pois piireiltä, ja lämmönsiirtoon itseensä käytettävä sähköenergia ei saa olla kohtuuton. [Patel, 2003]

Periaatteessa laskentatehon keskittäminen datakeskuksiin antaa hyvät valmiudet energiatehokkuuden lisäämiselle. Laitteiston lisäksi tärkeässä asemassa energiatehokkuuden kannalta ovat myös ohjelmisto- ja verkko-näkökulmat [Berl at al., 2009]. Ohjelmiston osalta mielenkiintoisia ovat energian huomioon ottavat tehtävien aikataulutusalgoritmit, ja erilaiset työtaakkaa datakeskuksessa eri laitteistojen välillä energiatehokkaasti jakavat algoritmit. Aikataulutusalgoritmeja esittelen tarkemmin luvussa 5, ja palvelinten välistä taakan jakoa myöhemmin tässä luvussa. Avainasemassa ohjelmistoissa on myös virtualisointi [Berl et al., 2009]. Virtualisointi vapauttaa ohjelmistot oman palvelimen tarpeesta. Jos yrityksessä tarvitaan Linux- ja Windows-palvelimet käytettyjen ohjelmistojen takia, ei niille virtualisoinnin ansiosta tarvitse kuitenkaan hankkia erillisiä laitteita. Tämä parantaa laitteiston käyttöastetta ja energiatehokkuutta välittömästi. Verkot ja virtualisointi myös mahdollistavat laskennan suorittamisen siellä, missä sitä on luonnostaan kätevämpi tehdä. Esimerkiksi Googlen 2011 Haminassa avaama datakeskus on rakennettu entisen paperitehtaan tiloihin, koska Suomenlahden meriveden käyttö laitteiston viilentämisessä on talouden ja ympäristön kannalta hyödyllistä [Google, 2013]. Luonnollista viilennystä voi tietenkin hyvin käyttää paperikoneiden lisäksi myös tietokonelaitteistoissa.

Verkon rooli on pilvipalveluiden energiatehokkuudessa oleellinen asia. Prosessoreiden tapaan verkkoinfrastruktuurikin joutuu tasapainottelemaan energiatehokkuuden ja palvelutehokkuuden välillä. Joissain verkkolaitteissa energiansäästö on otettu huomioon,

ja ne osaavat skaalata energian käyttöä kuormituksen mukaan. Laitteet harvoin osaavat vielä ottaa kuitenkaan koko verkon energian minimointia huomioon, jotta ne osaisivat esimerkiksi keskittää liikenteen hiljaisina aikoina yhdelle laitteelle. Myös signaalitiedon ja varsinaisen datan lähettämisen kesken olisi hyvä tehdä ero. Signaalitietoa liikkuu verkossa satunnaisesti, eikä se vie paljoa siirto- tai laskentakapasiteettia, kun taas varsinainen data siirretään aina vasta signaalitiedon jälkeen ja se yleensä kuormittaa verkkoa enemmän. [Berl et al., 2009]

Kliazovich ja muut [2012] kehittivät GreenCloud-nimisen simulaation energiatietoisten datakeskusten mallintamiseen. Kuvassa 1 näkyy tyypillinen kolmikerroksinen datakeskusarkkitehtuuri. Heidän mukaansa palvelimet vievät sähköstä 70%, core-tason kytkimet 5%, aggregation-tason kytkimet 10% ja access-tason kytkimet 15%. Palvelimissa prosessori vie 43%, muisti 12%, kiintolevyt 4%, lisälaitteet 17%, emolevy 8% ja muut 16%.



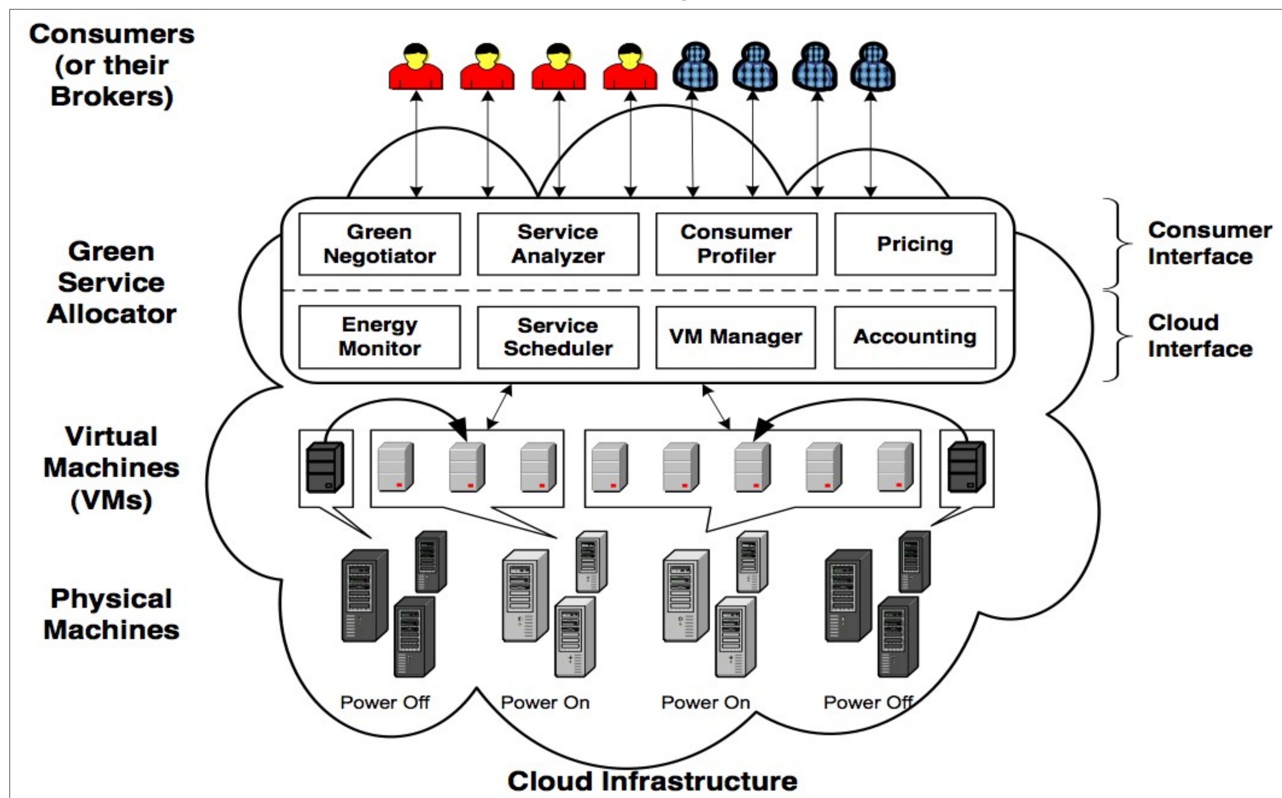
Kuva 1. Kliazovichin ja muiden [2010] kuvaama tyypillinen kolmitasoinen datakeskusratkaisu.

Kliazovich ja muut [2012] testasivat kahden eri virransäästömallin vaikutusta sekä servereihin että verkkokytkimiin: dynaaminen nopeuden vaihto (DVFS – Dynamic Voltage/Frequency Scaling) laskee suorittimen nopeutta vähentämällä sen läpi kulkevan virran jännitettä. Virransäästötilaan meno (DNS – Dynamic Shutdown) laittaa koko laitteen säästötilaan, kun sitä ei tarvita. Kliazovich ja muut [2012] vertasivat näitä molempia yksinään ja yhdessä tilanteeseen, jossa mitään säästömetodia ei käytetä. He myös erittelivät kolme tyypillistä tehtäväkategoriaa, joiden mukaan he testasivat: Laskentaintensiivinen tehtäväkuormitus (CIW – Computationally Intensive Workload) käyttää servereiden suoritintehoa paljon, mutta ei juurikaan tarvitse verkkokapasiteettia.

Tällaisessa tilanteessa laskenta kannattaa keskittää halutuille servereille ja laittaa useat kytkimet virransäästötilaan. Dataintensiivinen tehtäväkuormitus (DIW – Data-Intensive Workload) tarvitsee paljon datan siirtoa, mutta ei juuri servereiden laskentaa. Verkkoliikenne muodostuu siis helposti pullonkaulaksi. Ideaalisti tällaisessa tilanteessa kytkimien tulisi lähettää jatkuvasti tietoa tilastaan työmäärän keskusaikataulutukseen. Tällöin aikataulutus voi lähettää tietoa vähemmän kuormitettuja reittejä pitkin, vaikka muualla olisikin servereitä vapaana. Tasapainotettu tehtäväkuormitus (BW – Balanced Workload) käyttää laskentaa ja datansiirtoa kumpaakin melko paljon. Aikataulutuksen tehtävä on ottaa huomioon verkon ja palvelimien yhteistilanne. Täydellä kuormituksella nopeuden säätö ja virransäästötilaan menemislogiikat eivät tietenkään vähennä sähkönkulutusta yhtään. Monipuolisesti valitulla kuormituksella ja 30% keskimääräisellä kuormituksella suoritetussa testissä pelkällä nopeuden säädöllä ei päästy kuin 4% sähkön säästöön. Tämä johtuu siitä, että kytkimet veivät yhä paljon sähköä ja servereistäkin prosessori on vain osa virrankulutusta. Pelkällä virransäästötilaan menemisellä pystyttiin samassa tilanteessa huomattavasti parempaan, 63% säästöön. Tekniikat yhdistämällä sähkön säästöä kertyi yhteensä 65%. [Kliazovich et al., 2012]

3.1. Vihreän pilven arkkitehtuuri

Buyya ja muut [2010] esittivät vihreän pilven visionsa, koska heidän nähdäkseen pilvipalveluiden on oltava ekologisesti ja ekonomisesti kestävällä pohjalla. Heidän esittämä vihreän pilven arkkitehtuuri koostuu neljästä tasosta: kuluttajista, resurssin jakajista, virtuaalikoneista ja fyysisistä koneista. Kuvassa 2 näkyy vihreän pilven arkkitehtuuri havainnollistettuna.



Kuva 2. Buyyan ja muiden [2010] esittämä nelitasoinen vihreän pilven arkkitehtuuri havainnollistettuna.

Vihreän pilven arkkitehtuurin tasoista tai toimijoista ensimmäinen on asiakkaat ja välittäjät. He toimittavat palvelupyyntöjä pilveen mistä tahansa maapallolta. Huomioitavaa on, että pilven asiakas ei välttämättä tarkoita loppukäyttäjää, vaan esimerkiksi yritystä, jonka loppukäyttäjälle toimittamaa palvelua pilven avulla tuotetaan. Toinen ryhmä vihreän pilven arkkitehtuurin toimijoista on vihreät resurssien jakajat, jotka toimivat välikätenä asiakkaiden ja pilven infrastruktuurin välillä. Käytännössä he täyttävät kahdeksaa erilaista roolia, joista kaikkia ei siis täytä ihminen: 1) Neuvottelija, joka määrittää asiakkaan kanssa tehtävän sopimuksen. Yleensä siinä määritellään palvelulta odotetut laatuvaatimukset ja mitä niiden laiminlyönnistä seuraa. 2) Palvelun analysaattori, joka päättää palvelun vaatimusten ja annetun pyynnön perusteella, hylätäänkö vai hyväksytäänkö palvelupyyntö. 3) Asiakasprofiloija, joka kerää tietoja asiakkaista, ja päättää sen perusteella ketä priorisoidaan. 4) Hinnoittelija, joka päättää, paljonko tietty palvelu maksaa. 5) Energian monitoroija, joka päättää, mitkä laitteet pidetään päällä milläkin hetkellä. 6) Palvelun aikatauluttaja, joka antaa tehtäviä virtuaalikoneille. 7) Virtuaalikoneiden manageroija, joka pitää kirjaa käytössä olevista virtuaalikoneista. 8) Kirjanpitäjä, joka ylläpitää tietoa siitä, mitä resursseja kukin on

käyttänyt ja miten paljon. Vihreän pilven arkkitehtuurin kolmas taso on virtuaalikoneet ja neljäs taso fyysiset tietokoneet. [Buyya et al., 2010]

Mallin keskeinen haaste on määrittää, millä fyysisellä laitteella mikäkin virtuaalikone milläkin hetkellä kannattaisi suorittaa. Tämä voidaan jakaa vielä kahteen erilliseen ongelmaan, eli mihin uudet virtuaalikoneet sijoitetaan, ja toisaalta miten olemassa oleva virtuaalikoneiden jakautuminen optimoidaan. Uusien virtuaalikoneiden sijoittamisen Buyya ja muut [2010] ratkaisivat laskevan parhaan sopivuuden algoritmilla (*BFD – Best Fit Decreasing*). He kehittivät myös parannellun *MBFD*-nimisen algoritmin, jossa virtuaalikoneet järjestetään laskevasti laskentakulutuksen mukaan ja ne annetaan suoritettavaksi laitteelle, jolla kyseisen virtuaalikoneen lisäys aiheuttaa pienimmän energiankulutuksen nousun. Nykyisen virtuaalikoneiden asettelun ongelman he jakavat kahteen vaiheeseen: ensin valitaan virtuaalikoneet, jotka tulisi siirtää (*migrate*), sitten valitaan se, mille laitteelle ne tulisi siirtää. Ensimmäiseen, siirrettävien virtuaalikoneiden tunnistamisen haasteeseen, Buyya ja muut [2010] esittävät neljää heuristiikkaa. Yksittäinen kynnys (ST – Single Threshold) -heuristiikka asettaa kullekin laitteelle kuormitustason ylärajan, ja siirtää laitteelta virtuaalikoneita pois, jos asetettu yläraja ylitetään. Näin pyritään minimoimaan kulutuspiikkien aiheuttama aikarajojen laiminlyönti. Kolme muuta esitettyä heuristiikkaa asettavat ylärajan lisäksi myös alarajan laitteen kuormitukselle, jotta liian vähän kuormitetut palvelimet voidaan sulkea kokonaan. Siirtojen minimointi (MM – Minimization of Migrations) -heuristiikka minimoi siirrettävien virtuaalikoneiden määrän säästääkseen siirtoon kuluvaa työtä. Korkein potentiaalinen kasvu (HPG – Highest Potential Growth) -heuristiikka siirtää virtuaalikoneita, joilla on varattuihin resursseihin nähden pienin kulutus. Satunnainen valinta (RC – Random Choice) -heuristiikka valitsee tarvittavan määrän virtuaalikoneita tasaisen todennäköisyysjakauman mukaisesti. [Buyya et al., 2010]

Buyya ja muut [2010] testasivat arkkitehtuurimalliaan simulaatiossa ja huomasivat, että jos suoritettujen tehtävien aikarajojen annetaan hieman venyä, voidaan sähköä säästää reilusti. He vertasivat järjestelmäänsä tilanteeseen, jossa käytetään vain prosessorien omaa nopeuden säätöä, ja sen kuluttama sähkö simulaation aikana oli 4.4 kWh, ja tehtävien aikarajoja ei rikottu ollenkaan. Heidän oman mallinsa eräs versio, joka käytti siirtoja minimoivaa heuristiikkaa 50% kuormituksen alarajalla ja 90% kuormituksen ylärajalla, rikkoi aikarajoja 1.1% ja kulutti samalla simulaatiolla sähköä vain 1.5 kWh. Sähköä siis kului jopa noin 66% vähemmän. [Buyya et al., 2010]

3.2. Vihreän energian käytön maksimointi GreenSlot-aikataulutuksella

Suurin osa datakeskusten kuluttamasta sähköstä kuluu pienissä ja keskisuurissa datakeskuksissa, joissa energiatehokkuuteen ei panosteta niin paljoa kuin uudemmissa ja suuremmissa keskuksissa. Goiri ja muut [2011] kehittivät GreenSlot-nimisen rinnakkaisen aikatauluttajan (*scheduler*), joka sopii hyvin niin pieniin kuin isoihinkin keskuksiin. GreenSlot on tarkoitettu tilanteisiin, joissa sähköä voidaan ottaa joko aurinkopaneeleilta tai sähköverkosta. GreenSlot ennustaa lähitulevaisuudessa saatavilla olevan aurinkosähkön määrää, ja aikatauluttaa tehtävät hyödyntämään aurinkosähköä aikataulurajoitteet huomioiden. Aikataulutus-algoritmi on toteutettu Linuxiin SLURM-aikatauluttimen päälle.

Aurinkopaneelien ongelma on, että aurinko ei aina paista silloin kun sähköä tarvittaisiin, ja silloin kun se paistaa, ei sähköä välttämättä tarvitse kaikkea käyttää. Akut ja sähkön sähköyhtiölle myynti ovat usein käytettyjä ratkaisuja, mutta niissäkin on ongelmansa: Akut ovat huonoja hyötysuhteeltaan, usein todella kalliita ja käyttävät ympäristön kannalta haitallisia kemikaaleja. Sähkön syöttö sähköverkkoon ja myynti sähköyhtiölle on myös hyötysuhteeltaan huonoa ja taloudellisesti epäedullista. Goirin ja muiden [2011] mukaan onkin ehdottomasti parasta käyttää paneelista tuleva sähkö itse silloin, kun sitä on saatavilla.

GreenSlot siis pyrkii maksimoimaan annettujen tehtävien teon aurinkopaneelin sähköllä, ja sähköyhtiöltä tulevaakin sähköä se pyrkii käyttämään mieluumin niinä aikoina, kun sähkö on halvempaa. GreenSlotin käyttökohteena on tutkittu tieteellistä laskentaa, joten aikataulutettavaksi tulevat tehtävät ovat suhteellisen pitkiä verrattuna tavalliselta pöytäkoneelta vaadittaviin vasteaikoihin. Aurinkopaneelilta saatavan sähkön määrää GreenSlot pyrkii ennustamaan historiatiedon ja sääennusteiden perusteella. Tämän ennusteen ja käyttäjiltä tulleiden aikarajoite- ja tehtävämäärävaatimusten mukaan algoritmi luo varauksia tulevaisuuteen. Kun tehtävän aika tulee, sitä aletaan suorittaa. Selvää on, että GreenSlot poikkeaa muista aikatauluja laativista algoritmeista siten, että odotusajat voivat olla hyvinkin selkeästi pidempiä, koska aurinkoa tai halvan sähkön aikaa odotetaan. Pitkät odotusajat voi käyttäjä toki minimoida manuaalisesti asettamalla lyhyitä aikarajoja tehtävilleen. [Goiri et al., 2011]

Kuvassa 3 esitetään pseudokoodina GreenSlot-algoritmi hieman yksinkertaistettuna, sillä yksittäisen tehtävän ei odoteta koskaan olevan aikataulutusväliä (*scheduling window*) pidempi. Rivit 1–6 kuvaavat GreenSlotin toimintaa jokaisen aikayksikön (*time slot*) alussa. Jos uutta aikataulutusta tarvitaan, katsotaan ensin, paljonko vihreää energiaa – siis aurinkopaneelilla tuotettua – nykyiset aikataulutusvälin tehtävät vievät, ja paljonko sitä

on jäljellä (rivi 8). Kun vihreän energian määrä on selvillä, järjestetään jonossa odottavat tehtävät nousevan joutoajan (*slack time*) järjestykseen. Joutoaika tarkoittaa aikaa, joka alkaa tehtävän suorittamisen jälkeen ja kestää aikayksikön loppuun, jolloin suoritin ei tee mitään. Sitten GreenSlot aikatauluttaa tehtävät halvimmalla tavalla tulevaisuuteen (rivit 10–26). Aikataulutuksen tärkein osanen on laskea tehtävän hinta (*cost*) kussakin aikayksikössä (rivit 11–18). Vihreän sähkön hinta on nolla, eli sitä käytetään aina kun mahdollista. Sähköverkon sähkön käyttöön määritellään hinta sähköyhtiön oikeiden hintojen mukaisesti (rivi 16) ja aikarajoituksen rikkoontumisesta hintaan lasketaan sakko (rivit 17–18). Mikäli useammalle aikayksikölle tulee sama hinta, joka on nolla, valitaan aikaisempi aika. Mikäli taas useammalle aikayksikölle tulee sama hinta, joka ei ole nolla, valitaan myöhäisempi aika. Näin siksi, että vihreää sähköä käytetään heti kun sitä on, ja sähköverkon sähkön käyttöä viivytellään niin kauan kuin mahdollista siinä toivossa, että aurinko alkaisikin paistaa. Jos tehtävää ei saada aikataulutettua nykyisen aikataulutusvälin puitteissa, kun käyttäjä sitä tarjoaa (rivi 20), on käyttäjällä mahdollisuus koettaa laittaa tehtävä suoritukseen joko väljemmällä aikarajalla tai pienemmällä suoritinmäärällä. Jos tehtävä on kerran hyväksytty aikataulutettavaksi, ei sitä kuitenkaan enää hylätä. Riveillä 28–31 näkyy tehtävien aloittaminen. S3-tilalla tarkoitetaan ACPI-standardin nukkumistilaa, jossa virtaa käytetään vähän ja tilasiirtymä aktiivitilan välillä on kuitenkin tarpeeksi nopea. Syvempi lepotila toisi liikaa ajallista jähmeyttä. [Goiri et al., 2011]

```
0. Käyttäjät määrittävät suoritinmäärän, oletetun suoritusajan ja aikarajan jokaiselle tehtävälle.
       Lisää virhemarginaali odotettuun suoritusaikaan.

1. Jokaisen aikayksikön (time slot) alussa:
2.     Selvitä, olivatko viimeisimmät vihreän energian ennusteet tarkkoja
3.     Jos olivat epätarkkoja, niin säädä tulevaisuuden ennusteita
4.     Jos ennusteita säädettiin, tehtävä saapui, tehtävä valmistui,
                 tehtävä ei odotusten vastaisesti valmistunut edellisessä aikayksikössä tai
                 on olemassa uusia tehtäviä, joita ei ole vielä aikataulutettu:
5.           Valmistele uusi aikataulutus
6.     Aloita tehtäviä aikataulutuksen mukaan

7. Aikataulun valmistelu:
8.     Päivitä vihreän energian saatavuus perustuen suorituksessa oleviin tehtäviin
9.     Koeta aikatauluttaa seuraava jonossa oleva työ nousevassa joutoajan järjestyksessä (Least Slack
           Time First (LSTF))
10.          Laske, paljonko maksaisi aikatauluttaa tehtävä jokaiseen aikayksikköön nykyisellä
                 aikataulutusvälillä
11.              Tehtävän aloituksen hinta aikayksikössä pitäisi olla ääretön seuraavissa tapauksissa:
12.              (1) edeltävä tehtävä samassa tehtävävirrassa ei ole valmis kuin vasta tässä aikayksikössä
13.              (2) tehtävä ei ehdi valmistua aikataulutusvälin aikana
14.              (3) suorittimia ei ole tarpeeksi käytössä tälle tai muulle aikayksikölle
15.              Kun hinta ei ole ääretön, ja sähköverkkosähköä todennäköisesti käytetään:
16.                     Ota huomioon sähköverkkosähkön hinta
17.              Kun hinta ei ole ääretön, mutta aikaraja todennäköisesti ei pidä:
```

```
18.                 Lisää aikarajan rikelisä asiaan kuuluvien aikayksiköiden hintoihin.
19.         Jos hinta on ääretön jokaisella aikayksiköllä:
20.             Jos tehtävä otettiin suoritukseen tässä aikayksikössä ja aikaraja on aikataulutusvälillä:
                    hylkää tehtävä
21.             Muutoin koeta aikatauluttaa tehtävä seuraavalla aikataulutuskierroksella
22.             Siirry seuraavaan tehtävään (rivi 9)
23.         Jos tehtävää saataisi suoritettua aikarajoiteessaan missään aikayksikössä:
24.             Vähennä aikarajoitetta (sisäisesti) yhdellä aikayksiköllä
25.         Aikatauluta tehtävä halvimpaan aikayksikköön, paitsi:
26.             Tehtävä, jolla on aikarajoite tämän aikataulutusvälin ulkopuolella, aikataulutetaan tähän
                    aikataulutusväliin vain jos se voi käyttää pelkästään vihreää energiaa (halvin
                        aikayksikkö = 0)
27.         Ota huomioon energia ja suorittimet joita tehtävä tulee käyttämään.

28. Aloita tehtäviä ja säädä suorittimien aktiivisuutta:
29.     Aktivoi suorittimet S3-tilasta, jos tarpeen
30.     Aloita tehtävät, jotka pitäisi nykyisen aikataulutuksen suhteen aloittaa nyt
31.     Laita toimettomat suorittimet S3-tilaan.
```

Kuva 3. Goirin ja muiden [2011] esittämä GreenSlot-aikataulutusalgoritmi pseudokoodina.

Goirin ja muiden [2011] saatavilla olevan aurinkoenergian ennustamismalli perustuu selkeään ajatukseen siitä, että tietyt säätilat, esimerkiksi pilvisyys, vähentävät tuotettua sähköä ennustettavalla tavalla suhteessa ideaaliin aurinkoiseen säähän. Siis ennustettu aurinkoenergian määrä E_p on tietyllä ajanhetkellä t sen hetkisestä säätilasta riippuvan kertoimen $w(t)$ ja ideaalisään $B(t)$ tulo. Siis $E_p(t) = f(w(t))B(t)$, jossa $f()$ säätää säätilan arvon välille [0, 1]. Ideaalisäässä otetaan aika huomioon siksi, että vuorokauden- ja vuodenajat vaikuttavat myös ideaalisäähän. He toteuttavat tämän mallin mukaisen energian ennustimen tunnin tarkkuudella. Sääennustukset $w(t)$-funktiolle haetaan yleisistä lähteistä, kuten The Weather Channelilta ja Weather Undergroundilta. Sivuilta saa tietoja tunnin tarkkuudella kaksi vuorokautta eteenpäin. Ennustuksen kuvaus, kuten esimerkiksi "hajanaisia ukkosmyrskyjä" tai "aurinkoista," syötetään merkkijonona GreenSlot-algoritmiin. Historiatietoja käytetään alustamaan ideaalitilan $B(t)$-funktio ja arvoalueen sopivaksi rajaava $f(t)$-funktio. Jokaiselle yksittäiselle tunnille haetaan viime vuoden vastaavalta kuukaudelta tieto, paljonko energiaa tuotettiin milläkin säällä. Sitten valitaan aikajakso t:n ympäriltä, $H(t)$, joka ottaa huomioon kausittaiset muutokset. Ideaalisään tuotto $B(t)$ on minkä tahansa päivän kyseisen tunnin maksimituotto aikavälillä $H(t)$. Jokaiselle eri säätilalle wc lasketaan $f(wc)$ käyttäen aikavälin $H(t)$ jokaista tuntia, joissa oli sama säätila. [Goiri et al., 2011]

Säätilan ennuste voi tietenkin olla väärä. Varsinkin ukkosmyrskyvaroitukset jäävät usein realisoitumatta. Lumimyrsky voi peittää aurinkopaneelin, eivätkä ennustukset enää vastaa energiantuottoa mitenkään. Tämän vuoksi Goiri ja muut [2011] käyttävät kerroinfunktion $f(t)$ alustamiseen myös edellisen tunnin energiantuottoa.

Ennustusalgoritmi siis vertaa edellisen tunnin ennusteen vastaavuutta todelliseen energiantuottoon ja säätää loppupäivän ennusteita virheen mukaisesti. Seuraavien päivien ennusteita ei kuitenkaan säädetä. Esitetyssä ennustealgoritmissa on Goirin ja muiden [2011] mukaan kolme hyvää puolta: se on yksinkertainen, se perustuu julkisesti helposti saatavilla olevaan tietoon ja on tarkkaa puolipitkällä aikavälillä, eli siis muutamasta tunnista muutamaan päivään. He tiedostavat monimutkaisempien historiaan perustuvien mallien olemassaolon, mutta ne eivät ole käytännöllisellä puolipitkällä aikavälillä niin tarkkoja.

GreenSlotin potentiaalisia huonoja puolia on, että se saattaa hylätä useampia tehtäviä tai myöhästyä useampien tehtävien suorituksessa kuin perinteisemmät aikataulutukset. Tätä kuitenkin Goirin ja muiden [2011] mukaan tapahtuu vain järjestelmissä, joissa on harvinaisen suuri, tarkalleen yli 72%, suorittimien käyttöaste.

Testeissään Goiri ja muut [2011] arvioivat aurinkopaneelin tuottaman sähkön ennustamisen tarkkuutta. Seuraavaksi he vertaavat perinteiseen aikataulutukseen GreenOnly-algoritmia, joka on pelkästään vihreää energiaa hinnoitteleva versio GreenSlotista. Kolmanneksi, he arvioivat, paljonko hyödyttää hinnoitella vihreän energian lisäksi myös sähköverkon energia. Heidän ennustusalgoritminsa päivittäinen mediaanivirhe on 19% silloin, kun ennustetaan 48 tunnin päähän. Seuraavan tunnin säätä ennustettaessa mediaanivirhe oli 12.9%. GreenOnly kulutti testeissä 47% enemmän vihreää energiaa kuin perinteinen aikataulutus. Kun myös sähköverkkosähkön hinnoittelu otettiin huomioon, eli käytettiin kokonaista GreenSlot-algoritmia, huomattiin että GreenSlot lisää vihreän energian käyttöä 117% ja vähentää sähköön käytettyä rahaa 39% suhteessa perinteiseen aikataulutusalgoritmiin. [Goiri et al., 2011]

4. Mobiiliympäristö

Mobiililaitteessa virran säästö on ekologisen ja ekonomisen hyödyn lisäksi myös suoraa kannettavan laitteen käyttöajan pidentämistä. Akussa on virtaa vain tietty määrä, ja mitä hitaammin se kulutetaan, sitä parempi käyttömukavuuden ja -kyvyn kannalta. Laitteiden lisäksi myös langattoman verkkoinfrastruktuurin ylläpito kuluttaa sähköä. Mobiililaitteiden ja -verkkojen kehittyessä ja yleistyessä on myös niiden energian kulutus hyvä ottaa huomioon. Vuonna 2008 jopa 57% ICT-sektorin sähkönkulutuksesta meni mobiililaitteiden ja -verkkojen käyttöön, ja niiden osuus oli yhä kasvussa. Vuoteen 2015 mennessä globaalin mobiiliverkkoliikenteen määrän ennustetaan kasvavan 6.3 eksatavuun, joka on 26 kertaa enemmän kuin vuonna 2010. [Wang et al., 2011]

Nykyisten mobiiliverkkojen suunnittelussa on tavoiteltavana pidetty lähinnä energiaan liittymättömiä asioita, kuten saatavuutta, skaalautuvuutta ja palvelun laatua. Nykyjärjestelmistä vihreisiin siirtyminen onkin hankalaa. Tärkeimmät syyt ovat seuraavat: 1) Suurin osa mobiiliverkkoteknologiasta pyrkii maksimoimaan suorituskykyä. 2) Verkkolaitteet eivät skaalaa virrankäyttöään, vaan kuluttavat usein jatkuvasti vähällä liikennemääräTläkin sen mitä ruuhka-aikoina. 3) Ne vihreät teknologiat, jotka nykyisin ovat käytössä, eivät huolehdi suorituskyvyn ja energiatehokkuuden balanssista kovinkaan hyvin, vaan keskittyvät liikaa energiatehokkuuteen ja suorituskyky jää liian huonoksi. Jos suorituskyky jää selkeästi liian huonoksi, tarvitaan sitä paikkaamaan lisää laitteita, tai asiakkaat hermostuvat, joka on kaupallisessa ympäristössä huono asia. On siis olemassa aika pieni toleranssi, jonka verran energiaa kannattaa yrittää säästää suorituskyvyn kustannuksella. Suorituskyky ei saa tipua niin paljoa, että se olisi selkeästi käyttäjän huomattavissa. [Wang et al., 2011]

Wangin ja muiden [2011] mukaan jopa puolet mobiiliverkon ylläpidon kustannuksista menee sähkön kulutukseen. Kulurakenteen takia sähkön säästäminen on siis oleellinen osa. Verkot toimivat kuitenkin aina jonkin standardin tai käytännön mukaan, ja kokonaisvaltainen uudelleensuunnittelu on haastavaa, koska se vaatisi kaikkien komponenttien, protokollien ja kerrosten uudelleensuunnittelua. Tämän vuoksi vihreän mobiiliverkon kehittely- ja tutkimustyö on Wangin ja muiden [2011] mukaan jaettu viiteen osa-alueeseen: 1) data-keskukset, 2) makrotornit, 3) femtotornit, 4) mobiililaitteet ja 5) mobiilipalvelut. Lisäksi näitä kaikkia tarkastellaan kolmen eri tekniikan näkökulmasta: 1) prosessitekniikat, kuten resurssien jako ja aikataulutus, 2) kommunikaatiotekniikat, kuten virran hallinta ja signaalinlähetysmallit, ja 3) systeemitekniikat, kuten lämmönjohto. Datakeskuksia käsittelin jo edellisessä luvussa, joten käsittelen tässä luvussa vain muita aiheita.

Makrotornit, eli yleiskielellä tukiasemat, kuluttavat 60% mobiiliverkon sähköstä. Niiden tärkeää energian säästöä on lähestytty ainakin kolmesta näkökulmasta: asemien dynaaminen aikataulutus, solun zoomaus ja vahvistimien virransäästö. Tukiasemakapasiteettiä on yleensä reilusti aluetta kohden, jotta suuren ihmismäärän tuleminen samalle alueelle pystytään hoitamaan. Tämän takia suurimman osan ajasta suuri määrä tukiasemia on toimettomana tai todella pienellä kuormituksella. Asemien automaattinen sulkeminen hiljaisina aikoina säästää energiaa, eikä vähennä palvelun tasoa. Tämä voidaan tehdä historiatiedon perusteella ajastetusti [Marsan et al., 2009] tai tilanteen mukaan dynaamisesti [Zhou et al., 2009]. Kaikissa tapauksissa energiatietoisen aikataulutuksen tekeminen vaatii tietoa käyttäjistä ja liikennekuormista sekä tietoa halutusta palvelutasosta. Liao ja Yen [2009] tutkivat energiaa säästävää aikataulutusta tietyillä palveluntasorajoitteilla WiMAX-verkoissa. Han ja muut [2010] tutkivat LTE-tukiasemien aikataulutusalgoritmeja. Solun koolla tarkoitetaan sitä fyysistä etäisyyden mukaan rajattua aluetta, jolla tukiasema palvelee. Eli perinteisesti kun kapasiteettiä tarvitaan lisää, rakennetaan uusia tukiasemia ja pienennetään kaikkien käytössä olevien asemien solujen kokoa. Niu ja muut [2010] tutkivat solun koon muuttamista automaattisesti kapasiteettivaatimusten ja muiden vallitsevien olosuhteiden mukaan. Bhaumik ja muut [2010] käyttivät automaattisessa solun koon muuttamisessa liikenne- ja hintavaatimusten lisäksi myös muun muassa ajastettua nukkumista ja lokaation ennustusta. Tukiaseman radion energiankulutuksesta 50% menee signaalin vahvistimiin, ja etenkin 3G-asemien lineaariset vahvistimet kuluttavat turhaan energiaa. Yleisesti ottaen tukiasemien energiansäästön tutkimus etenee rinnan pystytysstrategian, radioiden energiansäästön ja erilaisten liikennekuormien dynamiikan tutkimuksen kanssa. [Wang et al., 2011]

Femtotorni tarkoittaa hyvin pientä tukiasemaa, joka voidaan pystyttää esimerkiksi rakennuksen sisään tai tapahtuman yhteyteen. Niillä saadaan joustavasti lisää kapasiteettia ja kattavuutta verkolle. Niiden lyhyt etäisyys saattaa mobiililaitteesta riippuen auttaa myös säästämään mobiililaitteen itsensä virtaa. Femtotornien tutkimus on keskittynyt kattavuuden optimointiin voimakkuuden säädön avulla ja interferenssin välttämiseen. [Wang et al., 2011]

Mobiililaitteilla tarkoitetaan tässä yhteydessä nimen omaan mobiiliverkkoon liittyviä laitteita, kuten älypuhelimia ja tabletteja. Näissä laitteissa virransäästön tarve on ilmeinen. Verkon käytön näkökulmasta yleisiä energian säästämisen näkökulmia laitetasolla ovat energiankäytön profilointi, usean radion käyttö ja tehokkaat lähetykset. Energian profiloinnilla tarkoitetaan laajaa ja tarkkaa tietoa kaikista energian tarpeista, puhelimen omista resursseista sekä liikenteen ja käyttäjän käyttäytymismalleista. Vallina-Rodriguezin

ja muiden [2010] mukaan profilointi on hyvin tärkeää energian käytön vähentämisen kannalta. Falaki ja muut [2010] tutkivat 3G-älypuhelinten verkkoliikenteen käytön hahmoja (*patterns*) ja niiden suhdetta energian kulutukseen. Heidän mukaansa puhelimen radion virrankulutusta voidaan vähentää jopa 35%, jos radion virransäästötilan ajastimet asetetaan käyttäjän profiloinnin perusteella. Useimmat nykyaikaiset laitteet pystyvät käyttämään monia erilaisia radioita, kuten bluetoothia, WiFiä ja 3G:ä. Ne kaikki kuluttavat eri määrän virtaa. Rahmati ja Zhong [2007] kehittelivät verkkojen saatavuutta ennustavia algoritmeja, joiden avulla laite voi automaattisesti vaihtaa vähiten kuluttavaan verkkoon. Heidän työstään kerron tarkemmin kohdassa 4.2. Datan siirtäminen on selkeästi eniten virtaa kuluttava tila mobiililaitteessa, joten siksi juuri siihen kannattaa kiinnittää huomiota. Ra ja muut [2010] tutkivat datan lähettämisen viivästyttämisen vaikutusta energian säästöön ja käyttäjän tyytyväisyyteen. He kehittivät algoritmin, joka rajoitteet huomioon ottaen päättää, miten paljon ja missä tilanteissa kannattaa viivyttää datan lähettämistä. Ran ja muiden [2010] tutkimusta käsittelen tarkemmin kohdassa 4.3. Dogar ja muut [2010] kehittivät Catnap-nimisen lähetysmallin, joka kerää dataa segmentteihin, jotka se sitten aikatauluttaa kerralla nopeasti lähetettäväksi ja laittaa radion virransäästötilaan väliajoilla. Schulman ja muut [2010] lähtivät havainnosta, että voimakas signaali kuluttaa vähemmän virtaa, ja he pyrkivät historia- ja paikkatiedon perusteella ennustamaan signaalin voimakkuutta. He myös luokittelivat dataa käyttötarkoituksen mukaan ja aikatauluttivat ne eri tavalla. Useimmat mobiililaitteiden energiansäästöominaisuudet perustuvat joko erilaisten verkkojen hyödyntämiseen tai radioliikenteen minimointiin esimerkiksi viivästyttämällä. [Wang et al., 2011]

Mobiilipalvelut ovat moninaisia, ja niiden käytöstä syntyvä datamäärä lisääntyy. Nykyään ihmiset pelaavat, puhuvat videopuheluita, siirtävät tiedostoja ja tekevät kaikenlaista mitä lankaverkonkin kautta netissä ollessaan. Mobiiliverkot ovat tarpeeksi hyviä suurimpaan osaan kuluttajien tarpeista. Tiedon siirtäminen on kuitenkin se, joka sähköä kuluttaa, ja palvelut ovat niitä, jotka siirrettävää tietoa tuottavat, joten palveluiden toteutuksella on suuri rooli myös energian käytön näkökulmasta. Palveluiden vihreyttä voidaan toteuttaa kolmen kategorian näkökulmasta: 1) suora energiaa säästävä suunnittelu, 2) ennustukseen perustuva mukautuminen ja 3) suunniteltu välitys (*proxy-based caching*). Suoraan energian säästö voidaan ottaa huomioon ohjelmaa tai palvelua luotaessa, esimerkiksi kohdassa 5.3 esittelemälläni energiatyyppien käyttämisellä ohjelmoinnissa. Toinen tapa on pakata tieto ennen lähetystä, mikäli pakkaus vie vähemmän virtaa kuin säästetyn tietomäärän lähetys. Pakkausta esittelen tarkemmin kohdassa 4.1. Myös sijaintitiedon seuranta on omanlaistaan huomiota tarvitseva haaste. Suora GPS:n käyttö vie paljon virtaa, ja erilaiset algoritmit voivat laskea koska signaalia

todennäköisesti kannattaa kysyä. Ennustukseen perustuva mukautuminen käyttää systeemin toiminnasta kerättyä historiatietoa. Esimerkiksi tietyssä pelissä dataa voidaan lähettää intensiivisesti aina minuutin jakson ajan, jonka jälkeen tulee valikkoon siirryttäessä ja tuloksia katsellessa yleensä lyhyehkö tauko. Tällainen käyttötapa voidaan oppia tilastollisilla menetelmillä ja radio voidaan laittaa automaattisesti kiinni heti minuutin intensiivijakson jälkeen. Tämä ei vähennä käyttäjän kokemaa nautintoa pelistä millään tavalla, mutta vähentää energian kulutusta. Haaste on toki oppia ohjelmallisesti sellaiset oikeat mallit, jotka tosiaan pystyvät ennustamaan. Erilaisten välitysmekanismien ja välimuistien käyttö on tehokas tapa säästää energiaa. Tällöin esimerkiksi isoa tiedostoa verkosta ladattaessa tiedosto haetaan ensin välityspalvelimelle, josta se siirretään täydellä nopeudella laitteelle. Tällöin radiota ei pidetä epäoptimaalisen hitaalla nopeudella päällä ollenkaan, vaan mahdollisimman usein täydellä kapasiteetilla. Välityspalvelimet lisäävät toki verkko-operaattoreiden laitteistokustannuksia. [Wang et al., 2011]

4.1. Tiedon pakkaus ennen lähetystä

Langaton tiedonsiirto voi viedä 1000 kertaisesti energiaa verrattuna yhden 32-bittisen laskuoperaation suorittamiseen. Tämän vuoksi tiedon pakkaus ennen lähetystä saattaa olla todella hyväkin energiaa säästävä toimenpide. Toisaalta tiedon haku muistista saattaa olla 200 kertaisesti energiaa vievää laskutoimitukseen nähden. Käytetyllä algoritmilla on siis merkitystä. [Barr and Asanovic, 2003]

Barr ja Asanovic [2003] tutkivat häviötöntä pakkausta mobiilialustalla. Heidän laitteistonsa on StrongARM SA-110 -perustainen Compaq iPAQ, jossa käyttöjärjestelmänä on Linux 2.4. Tällä laitteistolla he huomasivat, että yhden bitin lähettäminen lähelle muutaman sentin päähän vie noin 0.4 µJ ja verkon kuuluvuusalueen laidalle noin 1.1 µJ. Yhden ADD-operaation suorittaminen samalla laitteistolla vie 0.86 nJ, eli yhden bitin lähettäminen vastaa noin 485–1267 ADD-operaatiota. He pakkasivat webistä haettujen sivujen ei-kuvalliset osat valmiilla pakkausalgoritmeilla. Heidän testaamiaan algoritmeja olivat *zlib*, *LZO*, *compress*, *PPMd* ja *bzip2*. Olkoot n bittien alkuperäinen määrä ja m bittien määrä pakkauksen jälkeen. Testilaitteistolla prosessorin ja muistin käytöllä oli sellainen riippuvuus, että kulutettu energia oli melko suoraan verrannollinen käytettyyn aikaan. Vähiten energiaa käyttivät *compress* ja *LZO,* jotka olivat selvästi muita kevyempiä. Olkoon c pakkauksen ja purkamisen hinta ja w lähettämisen ja vastaanottamisen hinta per bitti. Pakkaus on siis energiatehokasta, jos $\frac{c}{n-m} < w$. He testasivat myös vaihtoehtoa, jossa käytettiin eri algoritmeja pakkaamiseen ja purkamiseen. Tällöin siis puhelin kysyy palvelimelta webbisivun sillä algoritmilla pakattuna, joka puhelimen itsensä kannalta on

energiatehokkainta purkaa. Esimerkiksi pelkän LZO:n käyttö molempiin suuntiin lähetettäessä oli tehokasta, mutta jos puhelin sai käyttää pakkaukseen LZO:a ja purkuun zlib-9:ää, vei webin selaus vielä 12% vähemmän energiaa. Kaiken kaikkiaan web-datan lähettämiseen käytetty energia väheni heidän metodeillaan 31%, ja englanninkielisen tekstin lähettämiseen käytetty energia 57%. Pakkaus siis kannattaa, mutta algoritmi pitää valita käytetyn laitteiston mukaan, sillä siihen vaikuttaa liian moni tekijä, jotta suoraa vastausta parhaasta algoritmista voisi antaa. [Barr and Asanovic, 2003]

4.2. Energiatehokkaan verkon valinta ennustusalgoritmien avulla

Rahmati ja Zhong [2007] tutkivat, miten mobiililaite voisi päätellä, mitä tarjolla olevaa verkkoa olisi energiatehokkainta käyttää. Ensin he selvittivät kokeellisesti, paljonko GSM/EDGE-verkon ja WiFi-verkon käyttäminen erilaisissa tilanteissa vie virtaa heidän laitteillaan. GSM-verkkoyhteyttä on energiatehokasta ylläpitää verrattuna WiFiin, mutta datan siirto WiFillä on huomattavasti energiatehokkaampaa kuin GSM-verkolla. Jos mobiililaite on puhelin, niin GSM-yhteys on käytännössä koko ajan päällä joka tapauksessa, joten sen ylläpidon vähäistä virtamäärää ei välttämättä kannattaisi edes laskea datansiirron kannalta kuluksi.

Olkoon yhteyden muodostamiseen ja n megatavun siirtämiseen yhteensä käytetty energiamäärä $E = Ee + n \cdot Et$, kun Ee on yhteyden muodostamiseen käytetty energia ja Et yhden megatavun siirtoon käytetty energia. Kun tähän vielä lisätään verkkojen tapauksessa usein tarpeellinen siirron onnistumiskerroin S, niin saadaan energiakulutuksen laskentamalliksi: $E = Ee + \frac{n}{S} \cdot Et$. Se, paljonko puhelin käytännössä kuluttaa virtaa, riippuu oleellisesti siitä, mitä sillä tekee. Suurimmassa osassa käyttötapauksia datan siirtomäärät ovat isohkoja, jolloin WiFi on parempi vaihtoehto energiatehokkuuden kannalta. Haasteena on, että WiFi ei ole niin usein saatavilla kuin GSM-verkko. Lisäksi jos WiFi-piiri on virransäästösyistä suljettuna laitteessa, se pitää laittaa päälle, jotta voidaan tarkistaa, onko WiFi-verkkoja tarjolla. WiFi-piirin kytkeminen päälle vie suhteellisen paljon energiaa. Käytännön käyttötilanteissa WiFi- ja GSM-verkko eivät kumpikaan saavuta parasta mahdollista energiatehokkuutta ilman toista. Näistä tekijöistä syntyy tarve ennustaa, missä tilanteessa sopivan WiFi-verkon etsintä kannattaisi suorittaa. Yhteys GSM-verkkoon oletetaan olevan olemassa jatkuvasti. [Rahmati and Zhong, 2007]

Keskeinen tutkimuskysymys on siis, että mikäli laitteella on tarve lähettää n megatavua dataa minimaalisella energiankulutuksella, pitäisikö sen ensin suorittaa vaihtoehtoisten verkkojen etsintä. Olkoot p primääriverkko, a sille vaihtoehtoinen verkko

ja $\vec{C}_a$ ehto sille, että vaihtoehtoinen verkko on saatavilla. Nyt n megatavun lähetykseen vaihtoehtoverkon kautta energiaa kuluu $E_{a,available} = \frac{n}{S_a \vec{C}_a} \cdot Et_a(\vec{C}_a) + Ee_a$ ja vaihtoehtoverkon tuloksettomaan etsintään kuluva energiamäärä on $E_{a,unavailable} = Ee_a$. Koska tässä mallissa primääriverkon piirin oletetaan olevan jatkuvasti päällä, ei sille tarvitse ottaa huomioon yhteydenottamisen kustannusta Ee ja sen kautta lähetetyn tiedon energiakulutus voidaan kuvata yhtälöllä $E_p = \frac{n}{S_p \vec{C}_p} \cdot Et_p(\vec{C}_a)$. Olkoon P_a vaihtoehtoisverkon saatavuuden todennäköisyys. Oletettu energiansäästö vaihtoehtoverkon käytön koettamisesta on siis

$$E_{a,p} = P_a \cdot (E_p - E_{a,available}) - (1 - P_a) \cdot E_{a,unavailable}$$

tai

$$E_{a,p} = P_a \cdot n \cdot \left(\frac{Et_p(\vec{C}_p)}{S_p(\vec{C}_p)} - \frac{Et_a(\vec{C}_a)}{S_a(\vec{C}_a)}\right) - Ee_a \quad .$$

Päättelyalgoritmi toimii siis siten, että ensin lasketaan $E_{a,p}$ jokaiselle datansiirrolle. Jos $E_{a,p}$ on negatiivinen, siirretään data primääriverkon p kautta, muutoin yritetään yhdistää vaihtoehtoisen verkon a kautta. Mikäli vaihtoehtoisia verkkoyhteyksiä on käytössä enemmän kuin yksi, järjestelmä valitsee eniten energiaa odotetusti säästävän verkon laskemalla $E_{a,p}$ jokaiselle vaihtoehtoverkolle. Kaikki muut laskennan arvot ovat hyvin saatavissa, mutta C_a ja P_a pitää arvioida. [Rahmati and Zhong, 2007]

Yksinkertainen tapa arvioida vaihtoehtoverkon löytymisehto C_a ja todennäköisyys P_a on käyttää käyttäjän koko historian keskiarvoa. Tämäkin jo säästää sähkö sellaisissa tapauksissa, joissa siirrettävän tiedon määrä on suhteellisen iso ja WiFi on usein saatavilla. [Rahmati and Zhong, 2007]

Hystereettinen (*hysteretic*) arviointitapa ottaa huomioon, että ihmiset usein pysyvät melko pitkiäkin aikoja paikallaan. Tällöin käytetään vanhoja arvoja siihen asti, että tietty aikaraja täyttyy tai saadaan uusia arvoja. Hystereettinen arviointitapa on tarkempi lyhyillä aikaväleillä, ja energiatehokkuus riippuu myös oleellisesti asetetusta aikarajasta. Aikaraja voidaan myös asettaa edellisten arvioiden tarkkuuden mukaan. [Rahmati and Zhong, 2007]

Historian ja GSM-solutiedon huomioiva arviointitapa lähtee siitä, että verkon saanti riippuu usein kuuluvuudesta, ja päivät ovat usein tilastollisesti samanlaisia. Jokaista tukiasemaa, i, kohden tallennetaan kolme arvoa: WiFin keskimääräinen saatavuus kun asema oli näkyvissä, P_{cell_i}, monellako otoksella saatavuus laskettiin, n_i, ja keskimääräinen WiFi-signaalin voimakkuus otoksissa, C_{cell_i}. Potentiaalisesti uudemmille otoksille voidaan antaa myös hieman enemmän painoarvoa. WiFin saatavuus voidaan laskea painotettuna

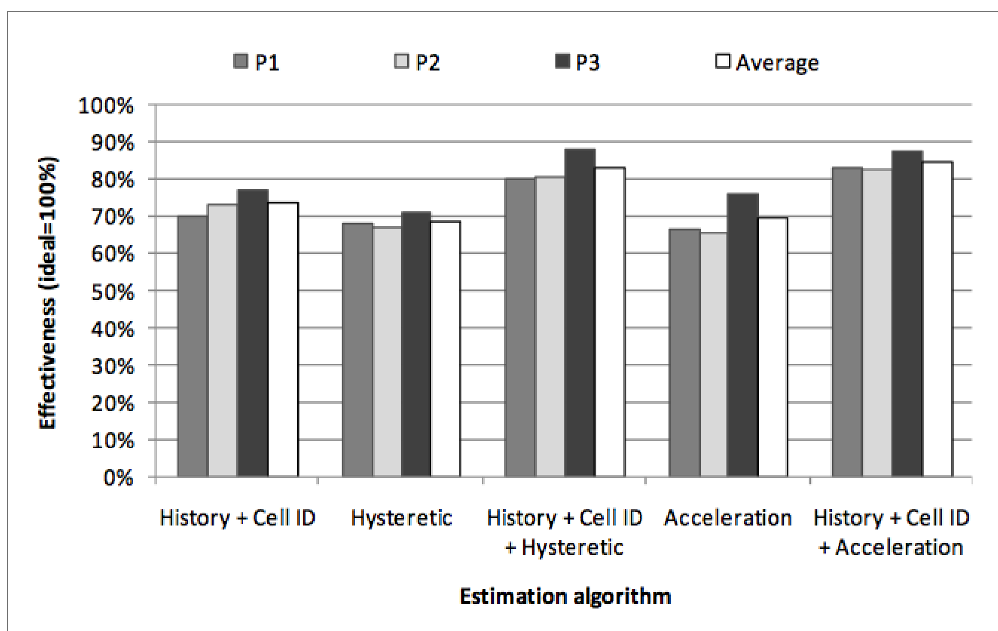
keskiarvona näkyvien tornien, V, joukossa: $P_{cell}=\frac{\sum_{i\in V} w_i \cdot P_{cell_i}}{\sum_{i\in V} w_i}, w_i=\log(n_i)\cdot(P_{cell_i}-0.5)^4$.

Logaritmifunktio antaa enemmän painoa torneille, joita on nähty useammin, eli mistä on enemmän näytteitä. Historian arviointiin lasketaan päivän jokaiselle tunnille saatavuus P_{hist} ja signaalin voimakkuus S_{hist}. Tukiasemien ja historian tiedot voidaan yhdistää, jolloin saadaan $P_a=\frac{2\cdot P_{cell}\cdot(P_{cell}-0.5)^2+P_{hist}\cdot(P_{hist}-0.5)^2}{2\cdot(P_{cell}-0.5)^2\cdot(P_{hist}-0.5)^2}$ ja $S_a=\frac{2\cdot S_{cell}+S_{hist}}{3}$. Tukiasematietoa painotetaan siis hieman enemmän. [Rahmati and Zhong, 2007]

Kiihtyvyystietoon perustuva arviointi käyttää laitteen kiihtyvyysanturilta saatua dataa sijainnin muutoksen arviointiin. Kiihtyvyysantureiden hyvä puoli on, etteivät ne vie merkittävästi virtaa, joten niitä voi pitää päällä jatkuvastikin. Jos puhelin on paikallaan, eli kiihtyvyysanturilta tulee melko samanlaisena pysyvää tietoa, ei verkkotilannekaan todennäköisesti muutu. Liikuttaessa puhelimen kiihtyvyys vaihtuu jatkuvasti, joten verkkojen saatavuutta voi olla hyödyllistä tarkistaa. Voidaan laskea WiFin saatavuuden muutoksen potentiaali, $m=\sum_{t=reset}^{current}[|\Delta A_x(t)|+|\Delta A_y(t)|+|\Delta A_z(t)|+c]$, jossa termit kuvaavat kiihtyvyyden muutosta kunkin akselin suhteen. Vakiotermi c on pieni positiivinen vakio, jotta hitaatkin muutokset saadaan otettua huomioon. Potentiaali m resetoidaan aina, kun verkon saatavuuden tarkistus suoritetaan, jonka jälkeen se alkaa taas kumuloitua. Kun potentiaali m pysyy tietyn ennalta asetetun raja-arvon alapuolella, käytetään olemassa olevia verkkoasetuksia, eikä tarkisteta saatavuuksia. Hystereettisen arvioinnin tapaan kiihtyvyystietoon perustuva arviointi on tehokkaampaa lyhyehköllä aikavälillä, jolloin tilanne pysyy helpommin ennallaan. Esiasetetut arvot vaikuttavat algoritmin toimintaan selkeästi, ja niitä pystytään myös suorituksen aikana säätämään sen perusteella, miten luotettavia algoritmin tuottamat ennusteet ovat. [Rahmati and Zhong, 2007]

Rahmati ja Zhong [2007] käyttivät tutkimuksessaan myös edellä mainittujen algoritmien yhdistelmiä, joissa ensin hystereettisellä tai kiihtyvyyteen perustuvalla menetelmällä pääteltiin, onko verkon tilassa odotettavissa muutosta. Jos muutos oli odotettavissa, käytettiin historiatietoja ja tukiaseman paikkatietoja hyväksi vaihtoehtoisen verkkoyhteyden todennäköisen saatavuuden arviointiin. Varsinaisen testauksen suorittivat testikäyttäjät normaalien arkirutiiniensa ohessa. Kuva 4 havainnollistaa näiden eri algoritmien suhteellista energiatehokkuutta ideaaliin verrattuna. Käytännössä energiatehokkuus näkyi siten, että keskimäärin hystereettistä arviointia käyttäen puhelimen akku kesti 37% prosenttia pidempään kuin ilman mitään verkonvaihtoteknologiaa. Kun hystereettisen algoritmin apuna käytettiin lisäksi myös

historiatietoa, kesti akku jopa 39% pidempään kuin ilman verkonvaihtoteknologiaa. Tämä on todella hyvä tulos, koska laskettu ideaali yläraja keston lisäykselle on 42%. Rahmatin ja Zhongin [2007] tarkoituksena ei kuitenkaan ollut luoda mahdollisimman tehokkaita algoritmeja verkon vaihdon ennustamiseen, vaan empiirisesti osoittaa, miten heterogeeninen verkkoympäristö tarjoaa selkeästi mahdollisuuksia virransäästön kannalta. Heidän tärkeä havaintonsa oli myös, että GSM ja WiFi sopivat hyvin toistensa pareiksi sen takia, että kyseisillä teknologioilla on tarpeeksi erilainen energiankäyttöprofiili, jotta niiden välillä tilanteen mukaisesti vaihtamalla on mahdollista ylipäätän saada energiasäästöä aikaiseksi.



Kuva 4. Rahmatin ja Zhongin [2007] esittämien algoritmien suhteellinen energiatehokkuus kolmella eri käyttäjällä.

4.3. Lähetyksen viivästyttäminen energiatehokkaamman yhteyden toivossa

Ra ja muut [2010] tutkivat myös energiatehokkaimman verkkoyhteyden valintaa. Mielenkiintoisena lisänäkökulmana heidän algoritminsa pystyy myös myöhäistämään datan siirtoa, mikäli ennusteet lupaavat, että se olisi tietyn ajan päästä energiatehokkaampaa. Heidän algoritminsa lähtökohtaisena ajatuksena on, että sovellusten siirtoon menevä data laitetaan jonoon. Viivästystä sietävillä sovelluksilla datan viivästyttäminen vähemmän energiaa kuluttavan verkon toivossa voi olla hyvä kompromissi. Jonon ei kuitenkaan saa antaa kasvaa liian pitkäksi, eli jossain vaiheessa

lähetys on tehtävä. Ongelma voidaan tästä näkökulmasta ajateltuna ilmaista niin, että tavoite on minimoida energian käyttö ja pitää keskimääräinen jonon pituus rajattuna.

Olkoot *A[t]* datan koko bitteinä ajanhetkellä *t* ja *P[t]* tiedon siirrosta johtuva virran kulutus ajanhetkellä *t*. Mallissa oletetaan ajanhetken *t* olevan jokin aikaväli (*slot*), vaikka mallin voisi yleistää myös jatkuvalle aikakäsitykselle. Jos algoritmi päättää viivästyttää datan siirtoa, *P[t]* on nolla. Jos algoritmi valitsee puhelinverkon, *P[t]* on P_C. Jos algoritmi valitsee WiFi-verkon, niin *P[t]* on P_W. Ajanhetkellä *t* siirretty datamäärä *μ[t]* riippuu muutamasta tekijästä. Vain jos algoritmi päättää lähettää dataa ajanhetkellä *t*, on *μ[t]* > 0. Jos *μ[t]* > 0, määrä riippuu valitun verkon laadusta, radion lähetystehosta ja lähetettävän datan määrästä. Olkoot *U[t]* jonon kertymä bitteinä ajanhetken *t* alussa ja $S_l[t]$ langattoman verkon laatu linkille *l*, joka kuuluu joukkoon *L[t]*. Lähetettävän datan määrä mallinnetaan funktiolla $\mu[t]=C(I[t],l,S_l[t],U[t],P[t])$, jossa I[t] on nolla, jos dataa ei lähetetä ollenkaan ajanhetkellä *t*, ja yksi, jos lähetetään. Myöskin aina on *μ[t]* < *U[t]*. Ajan kuluessa jono kehittyy yhtälön *U*[*t* + 1] = *U[t]* – *μ[t]* + *A[t]* mukaisesti. Ajanhetken *t* kuluessa jonoon lisätty data kuvataan termillä *A[t]*. Stabiliteettirajoite pitää jonon tietyn pituisena. Jono *U[t]* on stabiili, jos ja vain jos $\bar{U}=\lim_{t\to\infty}\sup\frac{1}{t}\sum_{\tau=0}^{t-1}E\{U[\tau]\}<\infty$. Virrankäyttö stabiliteettirajoitteella on $\bar{P}=\lim_{t\to\infty}\sup\frac{1}{t}\sum_{\tau=0}^{t-1}E\{P[\tau]\}<\infty$, jossa $P[\tau]\in\{0,P_c,P_w\}$ riippuen siitä, mikä verkkoyhteys aikajaksolle valittiin. [Ra et al., 2010]

SALSA-algoritmi on Ran ja muiden [2010] kehittelemä verkkoyhteyden valinta-algoritmi, johon he implementoivat myös viivästyksen stabiliteettirajoitteellaan. Algoritmin nimi tulee sanoista Stable and Adaptive Link Selection Algorithm, eli stabiili ja adaptiivinen linkinvalinta-algoritmi. Sen toiminta perustuu Lyapunovin optimisointikehykseen. Jokaisella ajanhetkellä, *t*, SALSA päättää, lähettääkö se dataa jonostaan ja mitä saatavilla olevista verkkoyhteyksistä käyttäen. Algoritmi havaitsee aluksi paljonko uutta dataa *A[t]* on ajanhetken *t* alussa ja paljonko dataa on jonossa *U[t]*. Linkki *l[t]* aikavälille *t* ja parametrillä *V* valitaan yhtälöllä $\tilde{l}[t]=\underset{l\in L[t]\cup\emptyset}{argmax}(U[t]\times E\{\mu[t]|l,S_l[t],P_l[t]\}-V\times P_l[t])$, jossa *l[t]*:n kuuluminen tyhjään joukkoon kuvaa tilanteita, joissa joko valitaan olla käyttämättä verkkoa, tai mitään verkkoa ei ole saatavilla. E on arvio siitä, millaisella nopeudella linkkiä *l* voidaan käyttää nykyinen kanavan tila $S_l[t]$ ja lähetysteho $P_l[t]$ huomioon ottaen. Olkoon WiFi-verkko *l* tietty verkko, jossa $P_l[t]$ = P_w. Jos *V* on pysyvä, algoritmi valitsee verkon *l* vain, kun jono *U[t]* on tarpeeksi pitkä tai verkon *l* nopeus on tarpeeksi korkea. Kun P_w on korkeampi, kynnys lähetykseen nousee automaattisesti. Algoritmin suorituskyky riippuu kriittisesti parametrin *V* valinnasta. SALSA-algoritmi voi päättää olla käyttämättä mitään verkkoa

jos ja vain jos $U[t]\times E\{\mu_l[t] | l, S_l[t], U[t], P_l[t]\} - V\times P_l[t] < 0$ jokaiselle $l\in L[t]$. Tällainen tilanne muodostuu tyypillisesti, jos kaikki saatavilla olevat verkot ovat hitaita huonon yhteyden takia. [Ra et al., 2010]

SALSA-algoritmi käyttää Lyapunovin optimisointikehystä. Lyapunovin siirtymä (*drift*) on algoritmien suunnittelussa usein käytetty tapa taata jonon stabiliteetti. Siirtymää varten täytyy määritellä ei-negatiivinen skalaarifunktio, Lyapunovin funktio, jonka arvo riippuu jonosta *U[t]* ja ja ajanhetkestä *t*. Lyapunovin siirtymä määritellään Lyapunovin funktion odotettuna muutoksena ajanhetkestä toiseen. Lyapunovin optimisointikehys takaa siirtymän stabilisoivan jonon ja saavuttavan lähes täydellisen optimisoinnin halutun tavoitteen suhteen. Tässä Ran ja muiden [2010] algoritmissa halutaan optimoida energian kulutus. Olkoon tietty data *A[t]* tulossa ajanhetkellä *t* ja kanavien tilat ajanhetkien suhteen itsenäisiä saman jakauman mukaisesti satunnaisia jakaumia p_a ja π_s. Oletetaan datan tulonopeuden λ olevan verkon kapasiteetin rajoissa. Jokaiselle parametrille $V > 0$, SALSA saavuttaa voimalle rajoitteen $\bar{P} = \lim_{t\to\infty} \sup \frac{1}{t}\sum_{\tau=0}^{t-1} E\{P[\tau]\} \leqslant P^* + \frac{B}{V}$ ja jonon pituudelle rajoitteen $\bar{U} = \lim_{t\to\infty} \sup \frac{1}{t}\sum_{\tau=0}^{t-1} E\{U[\tau]\} \leqslant \frac{B+VP^*}{\epsilon}$. Jonon pituuden rajoitteen yhtälössä vakio $\epsilon > 0$ kuvaa etäisyyttä saapumishahmon (*pattern*) ja kapasiteetin rajan välillä. Kummassakin rajoitteessa P^* on teoreettinen alaraja energian kulutukselle pitkän ajan keskiarvona ja B on yläraja *A[t]*:n ja *μ[t]*:n varianssien summalle. SALSA pystyy saavuttamaan keskimääräisen energiankulutuksen $\bar{P}$, joka on hyvin lähellä P^*:ä ja samalla pitämään datajonon stabiilina. Virransäästö saavutetaan kuitenkin isomman viiveen ansiosta, koska keskimääräinen jonon pituus $\bar{U}$ kasvaa lineaarisesti *V*:n mukaisesti. Tämä [$O(1/V)$, $O(V)$] -kompromissi energian säästön ja viiveen välillä on keskeinen Lyapunovin optimisaatiokehyksen kontrollitekniikoihin liittyvä asia. Tämä kompromissi myöskään ei oleta mitään tiettyä jakaumaa datan tuloprosessin *A[t]* tai verkon kuuluvuuden $S_l[t]$ suhteen. [Ra et al., 2010]

Hyvän kontrolliparametrin *V* valinta on tärkeää, sillä se on ainoa parametri, joka Lyapunovin optimisaatiokehykselle annetaan. Kehys itsessään ei ohjeista parametrin valinnassa. Yksinkertaistaen voi parametriä *V* ajatella jonon pituuden kynnysarvona, jonka jälkeen algoritmi päättää aloittaa lähetykset, eli *V* kontrolloi energia-viive-kompromissia. Hyvän parametrin valintaan suoraviivainen tapa on arvioida sitä suorituksen aikana jatkuvasti. Tässä tavassa huonona puolena on, että hyvän arvon löytyminen voi kestää pitkään. Ra ja muut [2010] kehittivät tekniikan parametrin *V* automaattiseen määrittämiseen. Heidän tekniikallaan on kaksi tavoitetta. Ensimmäinen tavoite on löytää *V*, jolla virransäästö on melko hyvällä tasolla suhteessa viiveeseen, ja

toinen tavoite on lisätä eksplisiittistä kontrollia energia-viive-kompromissiin. Eksplisiittinen kontrolli on kätevää, koska silloin voidaan ohjelmatasolla määrittää, mikä data voi odotella pidempään ja mikä olisi hyvä lähettää nopeammin. Energian kulutuksen yläraja on suhteellinen arvoon $1/V$ nähden. Tämän perusteella keskimääräinen energian kulutus olisi yksinkertaistettuna $\bar{P} \approx P^* + \frac{B}{V}$. Koska P^* on vakio, on $\bar{P}$ hyperbolinen funktio, eli tietyn rajan jälkeen V:n lisäämisen tuoma energian säästö vähenee. Hyvä lähtökohta olisi siis valita V siten, että tietty lisäys V:en toisi enää hyvin pienen vähennyksen virran kulutukseen. Virrankäyttöyhtälön $\bar{P}$ kulmakerroin $\alpha > 0$ voidaan valita yhtälöllä $\frac{d(P^* + B/V)}{dV} = \frac{-B}{V^2} = -\alpha$, jolloin $V = \sqrt{\frac{B}{\alpha}}$. Tämän mukaisesti V:ä asetettaessa pitää selvittää ensin vakio B, joka onnistuu estimoimalla $A[t]$:n ja $\mu[t]$:n varianssia, jota seurataan pitkähköllä aikavälillä. Alussa $V = 0$, ja sitä päivitetään sitä mukaan, kun B päivittyy. Eksplisiittistä kontrollia varten V mukautetaan yhtälöllä $V[t] = \sqrt{\frac{B[t]}{\alpha \times (D[t]+1)^\alpha}}$, jossa $D[t]$ on datansiirron välitön viive, eli kauanko ensimmäinen bitti on ollut jonossa ajanhetken t alkuhetkenä mitattuna. Mitä kauemmin data on jonossa, sitä pienemmäksi $V[t]$ tippuu α:n määrittämää nopeutta, kunnes tarpeeksi alhaalle laskettuaan aletaan dataa siirtää, mikäli verkko on saatavilla. Jos sovellus haluaa maksimoida energian säästön, eikä viive haittaa, voidaan α asettaa lähelle nollaa. Jos sovellus haluaa datansa nopeasti eteenpäin, voidaan α asettaa isommaksi. [Ra et al., 2010]

Käytännössä verkon siirtonopeutta ei voi aina tietää, joten SALSA arvioi myös sitä. SALSA tarkkailee jokaiselle verkkoyhteydelle erikseen viimeisimpiä siirtonopeuksia, ja laskee niistä keskiarvoa. Tämä kuitenkin vaatii yhteyden käyttämistä, ennen kuin arvio voidaan tehdä. Ilman yhteyden käyttöä SALSA estimointi arvioi saadun signaalin voimakkuuden (Received Signal Strength Indicator – RSSI) perusteella sitä, miten nopeaan datansiirtoon verkko todennäköisesti kykenee. Tämä arvio on hyvin karkea verkon todellisesta nopeudesta, mutta ei vaadi datan siirtoa. SALSA voidaan muuttaa toimimaan myös datan siirrossa verkosta puhelimelle päin, mutta se vaatii, että siirrettävän tiedon määrä palvelimella tiedetään tai pystytään edes arvioimaan. [Ra et al., 2010]

Ra ja muut [2010] vertasivat SALSA-algoritmiaan kahteen perusalgoritmiin, minimiviiveeseen ja pelkkään WiFiin. Minimiviivealgoritmi lähettää datan heti, kun verkko on saatavilla. Pelkkää WiFi-lähetystä käyttävä algoritmi viivyttää datan lähetystä niin kauan, kunnes saatavilla on WiFi-verkko. Yllättäen käytännössä pelkkää WiFiä käyttävä algoritmi ei ole energiatehokas, koska WiFi-verkkojen laatu vaihtelee. SALSA käyttää noin puolet miniviiveen algoritmiin nähden. Tyypilliselle testissä käytetylle

videotallenteen lähetyksestä tulevalle datalle viive oli yleensä puolisen tuntia ja keskimääräisesti pisimmillään puolitoista tuntia. Yleensä puhelimen akun kesto pidentyi videoiden viivästetyn lähetyksen ansiosta jopa 30 – 90 minuuttia. [Ra et al., 2010]

5. Ohjelmiston rooli

Ohjelmiston rooli laitteiston lisäksi on oleellinen, koska se tietenkin määrittää, mitä laitteisto tekee. Usein loppukäyttäjä voi säätää laitteensa asetuksista esimerkiksi miten nopeasti virransäästötilat näytölle, kiintolevyille ja koko koneelle menevät päälle. Joissain kannettavissa tietokoneissa voi myös valita, käyttääkö kone hitaampaa näytönohjainta vai nopeampaa, vai antaako koneen itsensä valita tilanteen vaatimusten mukaan.

Tämän manuaalisen tilojen säädön lisäksi oleellista on myös, miten tehokkaasti ohjelmat toimivat ollessaan suorituksessa. Se on enemmänkin ohjelmoijien ja heidän työkalujensa, kuten kääntäjien ja kirjastojen, varassa.

5.1. Virrankulutustilan vaihtoalgorimeista

Käyttäjä voi usein vaihdella koneensa virransäästötilaa manuaalisesti ja asettaa aika-arvoja esimerkiksi näytön automaattiselle sulkemiselle. Albers [2010] kertoo, miten eri energiatilojen mahdollisimman optimaalista automaattista siirtymistä voidaan ajatella algoritmisena ongelmana.

Hän jakaa ratkaisut *online*- ja *offline*-tyyppisiin. Online-algoritmi ei tiedä tulevaisuudesta mitään, ja offline-algoritmi tietää kaiken. Offline-ratkaisu on ikään kuin teoreettinen ideaaliratkaisu, ja kaikki oikeat algoritmit (tai ainakin ylivoimaisesti suurin osa niistä) ovat online-tyyppisiä. Optimaalisesta ratkaisusta hän käyttää merkintää *OPT*.

Algoritmien kyvykkyyttä hän mittaa *c-competitive* -lukemalla, joka tarkoittaa, että kaikilla syötteillä algoritmi on korkeintaan c kertaisesti OPTin verran energiaa kuluttava. Suomeksi käytän *competitivestä* termiä *hyvyys*.

Albers pohtii ensin kahden virtatilan – lepotilan ja aktiivitilan – järjestelmää kolmen algoritmin avulla. Oletetaan, että korkeamman virrankulutuksen tilasta siirtymistä pienemmän kulutuksen tilaan ei vie paljo virtaa, mutta isompikulutuksiseen tilaan siirtyminen vie merkittävästi virtaa. Olkoon $b > 0$ virrankulutus siirryttäessä lepotilasta aktiivitilaan, ja r virrankulutus aktiivitilassa. Olkoon T systeemin joutokäyntiaika tietyssä tilanteessa.

OPT siis tietää T:n, koska OPT tuntee koko operaatiojonon. OPT voi vain katsoa, onko tulossa oleva joutokäyntiaikana kulutettu virta pienempi kuin tilan takaisinvaihtoon vaadittu virta. Eli jos $rT < b$, niin ei kannata vaihtaa lepotilaan, ja muuten kannattaa.

ALG-D on deterministinen algoritmi, joka siirtyy lepotilaan aina, kun joutoaikaa aktiivitilassa on kertynyt b/r verran. Algoritmi on 2-hyvä, eli kuluttaa kaksinkertaisesti energiaa suhteessa OPT:iin, joka on Albersin mukaan paras mihin deterministisellä algoritmilla voi päästä.

ALG-R menee virransäästötilaan satunnaisesti tietyltä todennäköisyysjakaumalta valitun ajan päästä. Algoritmin hyvyys on $e/(e-1)$, eli noin 1.58, ja se on todistetusti paras tulos mihin satunnaistetulla algoritmilla voi päästä.

ALG-P oppii T:n pituutta ennustavaa todennäköisyysjakaumaa, jolta se valitsee satunnaisesti ajan. ALG-P:n energiankulutus on korkeintaan $e/(e-1)$ kertaisesti odotetun optimikulutuksen. [Albers, 2010]

5.2. Prosessorin nopeuden dynaaminen skaalaus

Modernit prosessorit pystyvät toimimaan useammalla nopeudella säästäen energiaa tilanteissa, joissa kaikkea laskentatehoa ei tarvita. Dynaamisen skaalauksen tavoite on taata haluttu tehtävien suorituksen tehokkuus mahdollisimman hitaalla, eli energiatehokkaalla, nopeudella.

Albersin [2010] mukaan nopeuden skaalaus johtaa moniin haastaviin ongelmiin tehtävien ajastuksessa (*scheduling*). Ajastuksenhallinta joutuu siis päättämään tehtävien järjestyksen lisäksi myös niiden halutun nopeuden. Albers esittelee seuraavaksi Yaon ja kumppaneiden [1995] mallin, joka on laajalti tutkittu ja käytetty.

Yaon ja muiden [1995] mallissa oletetaan, että prosessorin nopeus on portaattomasti säädettävissä ilman ylärajaa. Tehtäviä voi jäädyttää ja jatkaa myöhemmin (*preemptive*) ja mallin tehtävänä on löytää minimienergiankulutus, jossa tehtävät tulevat kuitenkin ajallaan valmiiksi. Rajattoman nopean prosessorin takia aikataulutus on aina tehtävissä mallin kontekstissa.

Olkoot $J_1, \ldots, J_n$ tehtäviä, jotka on käsiteltävä. Jokaisella tehtävällä J_i on tietty julkaisuaika r_i, joka kertoo, minä ajan hetkenä tehtävä tulee aikataulutuksen tietoon, d_i on aika jolloin tehtävän on oltava viimeistään valmiina ja w_i on työmäärä. Tehtävän tekoon kuluva työmäärä voidaan ajatella esimerkiksi prosessorin kellojaksoina. Prosessointiaika riippuu luonnollisesti prosessorin nopeudesta ja työmäärästä, ja tasaisella prosessorin nopeudella s veisi työn i suoritus w_i/s määrän aikaa. [Albers, 2010]

Yaon ja muiden [1995] esittämä optimaalinen YDS-niminen iteratiivinen algoritmi etsii joka iteraatiolla mahdollisimman tiheän aikaintervallin I, ja siihen liittyvän aikataulutuksen. Intervallin tiheys on pienin mahdollinen keskinopeus, jolla kaikki tehtävät tulevat tehtyä. Mitä tiheämpi intervalli, sitä enemmän se vaatii nopeutta eli energiaa. Intervallin $I = [t, t']$ tiheys Δ_I on intervallin kaiken työn määrä jaettuna I:n pituudella. Olkoon S_I joukko kaikista niistä tehtävistä, joiden alkukohta ja deadline ovat intervallilla I, eli siis $[r_i, d_i]$ on I:n aito osajoukko.

YDS toimii niin, että se toistuvasti selvittää mahdollisimman tiheää I:ä. I:llä on tietty nopeus ja tietyt tehtävät aikarajojen mukaan priorisoituna (EDF – Earliest Deadline First).

Algoritmi käy iteraation alussa läpi kaikki työt J_i. Jos J_i:n deadline on kyseisellä intervallilla, uusi deadline asetetaan intervallin alkuun, eli jos d_i kuuluu välille I, asetetaan $d_i := t$. Vastaavasti jos tehtävän aloitusaika on intervallilla, asetetaan alkuaika intervallin loppuun, eli jos r_i kuuluu välille I, asetetaan $r_i := t'$. Yhden iteraation jälkeen tehdyt tehtävät ja intervalli poistetaan ongelmakentästä, ja siirrytään seuraavaan iteraatioon. Suoraviivaisen algoritmin aikavaatimus on $O(n^3)$, mutta se voidaan optimoida toimimaan ajassa $O(n^2 \log n)$. [Albers, 2010]

Optimaalisen YDS-offline-algoritminsa lisäksi Yao ja kumppanit [1995] julkaisivat myös kaksi online-algoritmia: Average Rate ja Optimal Available.

Average Rate -algoritmi selvittää tiheyden $\delta_i = w_i \,/\, (d_i - r_i)$, joka on minimikeskiaika, jossa tehtävä voidaan suorittaa mikäli muita tehtäviä ei huomioida. Nyt voidaan vain asettaa nopeus tiettynä aikana aktiivisten tehtävien kertyneeksi tiheydeksi, eli intervallilla olevien tehtävien tiheyden summaksi, eli $s(t) = \mathrm{sum}(\delta_i)$, kun t kuuluu välille $[r_i, d_i]$. Tehtävät järjestetään EDF-periaatteen mukaisesti nousevaan päättymisjärjestykseen.

Kuten edellä määriteltiin, algoritmin *hyvyydellä* (*competitiveness*) tarkoitetaan sitä, moniko kertaisesti se poikkeaa suhteessa optimialgoritmiin. Siis *c-hyvä* algoritmi kuluttaa aikaa *c*-kertaisesti optimiin verrattuna, tai luo c-kertaa huonomman ratkaisun. Hyvyys voidaan nähdä myös eri asioiden suhteen, kuten esimerkiksi energian kulutuksen tai vasteajan minimoinnin suhteen. Tässä kohdassa käytän hyvyyttä suhteessa energian kulutukseen, mikäli toisin ei mainita. Average Rate -algoritmin tapauksessa optimaalinen vertailualgoritmi on YDS. Yao ja muut [1995] todistivat Average Raten hyvyyden ylärajan olevan $2^{\alpha-1}\alpha^{\alpha}$, kun $\alpha >= 2$. Bansal ja muut todistivat alarajan olevan käytännössä sama.

Optimal Available -algoritmi laskee aina optimaalisen aikataulun, ja on näin ollen raskaampi. Bansal ja muut [2007] todistivat algoritmin olevan tasan α^{α}-*hyvä*.

Bansal ja muut [2007] kehittelivät myös oman BKP-nimisen online-algoritminsa ongelman ratkaisuun. Sen hyvyyskerroin on $2(\alpha/(\alpha-1))^{\alpha}e^{\alpha}$, joka on isoilla alfan arvoilla Optimal Availablea parempi. Olkoon $w(t, t_1, t_2)$ ennen ajanhetkeä t saapuneiden tehtävien tiheys, joilla $t_1 < t <= t_2$, julkaisuaika on t_1:n jälkeen ja deadline ennen t_2:ta. Tällöin BKP estimoi optimaalista YDS-algoritmia, koska $\max(t_1, t_2)\ w(t, t_1, t_2)/(t_2 - t_1)$. BKP käy läpi tiettyjä $e(t' - t)$ pituisia aikaikkunoita $t' > t$ aikaa, eli saadaan joukko $[et - (e - 1)t', t']$. [Albers, 2010]

Tarvittava nopeus kerrotaan *e*:llä. Siis BKP laskee millä tahansa ajanhetkellä t tarvittavan nopeuden kaavalla

$$s(t) = \max t'>t\ (w(t, et - (e-1)t', t') \,/\, (e(t' - t)).$$

Kaikki nopeuden skaalaukseen tarkoitetut tässä toistaiseksi mainitut algoritmit perustuvat Yaon ja muiden mallin mukaisesti portaattomasti säädettävissä olevaan nopeuteen. Tämä

ei kuitenkaan ole käytettävän laitteiston kannalta realistinen oletus. Heidän kehittämänsä YDS-algoritmi on muokattavissa myös rajatulle nopeuksien määrälle $s_1 < s_2 < \ldots < s_d$, mikäli ongelmalle yleensä on olemassa ratkaisu. Ratkaisun löytyminen voidaan tarkastaa kätevästi kokeilemalla, saadaanko kyseinen tehtäväjono suoritetua aikarajassa, jos käytetään maksiminopeutta s_d, ja järjestetään tehtävät kasvavaan päättymisjärjestykseen (*EDF*). Jos ratkaisu on olemassa, voidaan se selvittää ensin luomalla YDS:llä aikataulutus. Tämän jälkeen jokaiselle maksimitiheysvälille I arvioidaan haluttu nopeus Δ_I viereisten nopeustasojen välistä, eli $s_k < \Delta_I < s_{k+1}$. Suurempaa nopeutta s_{k+1} käytetään δ aikayksikköä ja hitaampaa tasoa s_k käytetään aikavälin I loppuajan $|I| - \delta$. Nopeusaskelten suhteen määritettävä δ valitaan siten, että tehty kokonaistyömäärä vastaa alkuperäistä työmäärä $|I|\Delta_I$. [Albers, 2010]

Mikäli aikataulutus ei ole tehtävissä, eli tehtävien koko ja määrä on enemmän kuin aikavälille mahtuu, on Albersin [2010] mukaan pyrittävä mahdollisimman tehokkaaseen tehtävien käsittelyyn, mutta kuitenkin niin, että energian säästö otetaan huomioon. Tehokkaalla käsittelyllä hän tarkoittaa sitä, miten monta tehtävää pystytään suorittamaan aikarajoitevaatimusten mukaisesti. Tällaisessa tilanteessa algoritmit pääsääntöisesti pitävät listaa niistä tehtävistä, jotka ne aikovat saada suoritettua. Uuden tehtävän tullessa jonoon se saattaa valikoitua tälle listalle, mutta mikäli listalla on suoritustehoon nähden liikaa tehtäviä, niitä karsitaan pyrkien kunnioittamaan aikarajoja. Bansalin ja muiden [2008] esittämä algoritmi tällaiseen haastavaan tilanteeseen on nimeltään Slow-D, ja se on energian suhteen hyvyydeltään vakio ja saatavan aikataulutuksen suhteen 4-*hyvä*, joka on heidän mukaansa paras mahdollinen. Slow-D on online-algoritmi, joka pystyy ottamaan huomioon eri työmääräiset tehtävät, ja sen tarkoitus on maksimoida tehtävien yhteenlaskettu kokonaissuoritusmäärä (*throughput*) tilanteessa, jossa prosessorin nopeus T on ylhäältä rajoitettu, ja tehtäviä on niin paljon, ettei niitä kaikkia ehditä suorittaa. Slow-D jakaa tulleet tehtävät kahteen jonoon: työn alla olevat tehtävät ovat jonossa Q_{work} ja odottamassa olevat tehtävät ovat jonossa Q_{wait}. Tehtävän saapuessa aikataulutettavaksi määritellään se ajanhetkestä t riippuen *t-kiireelliseksi (t-urgent)* tai *t-hitaaksi* (*t-slack).* Tehtävä J_i on *t-kiireellinen*, jos $d_i \leqslant \text{down-time}(t)$, ja muussa tapauksessa *t-hidas*. Down-time(t) on viimeisin sellainen ajanhetki t' >= t, että nopeus on $s_{OA}^{t}(t') \geqslant T$. OA viittaa Yaon ja muiden [1995] kehittämään Optimum Available -algoritmiin, jonka hetkellä t laskemaa prosessorin nopeutta siis verrataan prosessorin nopeuden ylärajaan T. Jos edellä mainittu epäyhtälö ei pidä paikkaansa, down-time(t) on viimeisin aika, jolloin nopeus oli vähintään T, tai 0 jos nopeus ei koskaan vielä ole ollut T. Tulonsa hetkellä *t-hitaaksi* määritelty tehtävä voi siis nousta *t'-kiireelliseksi* tehtäväksi ajan kuluessa ja *t-kiireellinen*

pysyy aina kiireellisenä kunnes se suoritetaan tai hylätään, koska down-time(t) on kasvava funktio. [Bansal et al., 2008]

Slow-D ottaa tehtävän J_i aktiivisten tehtävien jonoon Q_{work}, jos J_i on joko r_i-*hidas* tai jos J_i ja sen lisäksi kaikki r_i-*kiireelliset* tehtävät Q_{work}-jonosta voidaan suorittaa toimittaessa maksiminopeudella T. Muuten tehtävä uusi tehtävä J_i laitetaan odotusjonoon Q_{wait}. Huomioitakoon, että tehtävän aloitushetkellä odotusjonoon laitettavat tehtävät ovat kaikki kiireellisiä. Algoritmi voi olla kahdessa tilassa: kiirejaksossa tai normaalijaksossa. Kiirejakso alkaa tehtävän J_i julkaisuhetkellä r_i, mikäli Q_{work} ei sisällä kiireellisiä tehtäviä ennen hetkeä r_i, ja tehtävä J_i on kiireellinen ja se lisätään tehtäväjonoon Q_{work}. Aina kun joku odotusjonon Q_{wait} tehtävä J saavuttaa viimeisen aloitusaikansa $t = d(J) - (p(J)/T)$, eli ajanhetken, jolloin se vielä maksiminopeudella ja priorisoituna ehditään tekemään, se aiheuttaa algoritmiin keskeytyksen (latest starting time interrupt). Keskeytyksen aiheuduttua tehtävä J joko hylätään ja tiputetaan kokonaan pois aikataulutuksesta tai sille tehdään tilaa työjonossa Q_{work} tiputtamalla sieltä kaikki *t-kiireelliset* tehtävät. Keskeytyksiä tapahtuu vain kiirejaksoilla, joten olkoon J_0 viimeinen kyseisen kiirejakson aikana odotuslistalta Q_{wait} aktiivilistalle Q_{work} siirretty tehtävä. Olkoon W kaikkien tehtävän J_0 aktiivilistalle Q_{work} siirtämisen jälkeen aktiivilistalla Q_{work} kiireellisiksi muuttuneiden tehtävien alkuperäiskokojen summa. Jos $p(J) > 2(p(J_0) + W)$, kaikki *t-kiireelliset* tehtävät poistetaan aktiivilistalta Q_{work}, ja J laitetaan aktiivilistalle Q_{work}. Kun mikä tahansa tehtävä J saadaan tehtyä ajanhetkellä t, se poistetaan aktiivilistalta Q_{work}, ja jos listalla ei ole enää *t-kiireellisiä* tehtäviä, lopetetaan nykyinen kiirejakso. [Bansal et al., 2008]

Prosessorin lämmöntuoton minimointi on käytetyn energian minimointia sivuava tärkeä osa-alue, sillä korkea lämpötila vähentää prosessorin vakautta ja elinikää. [Albers, 2010] Lämpötila, kuten energiankulutuskin, nousee prosessorin nopeuden noston mukaan – mitä nopeampi, sitä enemmän tuotettua lämpöä. Bansal ja muut [2007] pohtivat lämpöongelmaa prosessorin maksimilämpötilan minimoinnin näkökulmasta. Heidän mukaansa energian ja lämmön minimoinnilla on kuitenkin selkeitä eroja. Energian minimoimiseksi tulee vähentää kumulatiivista virrankäyttöä, ja lämmön minimoimiseksi tulee vähentää hetkittäistä virrankäyttöä. Siksi energian minimointiin suuntautuneet algoritmit eivät useinkaan tuota hyviä tuloksia lämmön minimoinnissa [Skadron et al., 2003]. Bansal ja muut [2007] olettavat viilenemisen noudattavan Newtonin lakia, jonka mukaan kappaleen viilenemisnopeus on riippuvainen sen lämpöerosta ympäristön lämpöön. He myös esittävät, että *YDS* ja *BKP* ovat siitä hyviä, että kummallakin maksimilämpötila on laitteen ominaisuuksista johtuvat vakion päässä teoreettisesti pienimmästä mahdollisesta maksimilämpötilasta. Se siis skaalautuu ennustettavasti. Skadronin ja muiden [2003] mukaan lämmöntuoton tiedostava teknologia on kuitenkin

selkeästi energiankulutuksen tiedostavasta teknologiasta erillinen tutkimuksen alue, joten en käsittele sitä tässä tutkimuksessa.

Prosessorin dynaamiseen nopeuden säätöön voidaan ottaa yhdeksi erikoistilaksi myös nukkumistila, jossa virtaa ei kulu ollenkaan. Tällöin pienivirtaisten ja hitaiden tilojen painotus saattaa vähentyä, koska voi ollakin energiatehokkaampaa tehdä nopeasti haluttu tehtäväjono, jotta voidaan maksimoida nukkumistilan kesto ja siten saavuttaa pienempi kokonaisvirrankulutus. [Albers, 2010] Tätä ovat tutkineet mm. Irani ja muut [2007].

Tärkeä näkökulma prosessorin nopeuden säädössä on energiansäästön lisäksi vasteajan pitäminen riittävän pienenä, jotta käyttö pysyy mielekkäänä. Pieni vasteaika vaatii tietysti nopeaa prosessointia, joka on energian säästämisen kanssa vastakkainen tavoite. Tämän haastavan tilanteen käsittelyä varten tarvitsemme termin virtausaika (flow-time). Olkoon siis tehtävän J_i virtausaika f_i se aika, joka kuluu tehtävän saapumisesta aikataulutettavaksi sen valmistumiseen. Nyt voidaan tarkastella aikatauluja, jotka minimoivat esimerkiksi tehtävien kokonaisvirtausaikaa. [Albers, 2010]

Prush ja muut [2008] tarkastelevat tilannetta sellaisesta näkökulmasta, että käytettävissä olevan energian määrä on annettu ja tavoite on minimoida kokonaisvirtausaika. Yksinkertaistuksen vuoksi he olettavat kaikilla tehtävillä olevan sama työmäärä, joten tehtävää voidaan tässä tilanteessa käyttää yksikkönä. Offline-tapauksessa optimiaikataulu voidaan laskea polynomiaalisesti skaalautuvassa ajassa, mutta online-algoritmeja tähän tilanteeseen on vaikea tehdä [Albers, 2010].

Albers ja Fujiwara [2007] lähestyvät virtausajan ja energiankulutuksen tasapainotuksen suhdetta yhdistetyn tavoitefunktion näkökulmasta. Olkoon E aikataulun kokonaisenergiankulutus. Minimoitava funktio on $g=E+\sum_{i=0}^{n} f_i$, ja termejä kertomalla voidaan myös painottaa haluttua tavoitetta suhteessa toista enemmän. Albers ja Fujiwara [2007] käyttävät dynaamista ohjelmointia, annettua energiamäärää ja heidän offline-algoritmillaan saa myös polynomiaalisesti skaalautuvassa ajassa rakennettua virtausta minimoivan aikataulun. He esittävät lisäksi online-algoritmin, joka käsittelee tulevia tehtäviä erissä (*batching*) ja pystyy toimimaan vakiolla *hyvyys*arvolla. Eräkäsittely mahdollistaa joidenkin raskaiden laskutoimitusten harventamisen tehtäväkohtaisista eräkohtaisiksi. He kuitenkin havaitsivat, että funktion g minimointiin parempi algoritmi on yksinkertaisesti:

Olkoon kullakin hetkellä aktiivisten (eli aloitettujen, muttei lopetettujen) tehtävien määrä l. Käytä nopeutta $\sqrt[\alpha]{l}$.

Bansal ja muut [2009] paransivat tätä algoritmia vielä omalla Job Count -nimisellä algoritmillaan. Job Countissa käytetään kullakin ajanhetkellä nopeutta $\sqrt[\alpha]{l+1}$, kun

$l \geqslant 1$. Jos tehtäviä ei ole, käytetään nopeutta 0. Heidän algoritminsa myös järjestää tulevat tehtävät nousevaan järjestykseen jäljellä olevan prosessointiajan suhteen (Shortest Remaining Processing Time, SRPT). Eli nopeimmin valmiiksi saatava tehtävä tehdään ensin. Heidän algoritminsa on 3-*hyvä*.

Tähän mennessä käsitellyt algoritmit ovat olleet yhden prosessorin tilanteisiin tarkoitettuja. Nykyisin kuitenkin usean prosessorin ja laskentaytimen järjestelmät ovat yleisiä, ja niitä varten olisi myös hyvä olla nopeuden säätöön tarkoitettuja algoritmeja. Albers ja muut [2007] tutkivat aikarajoihin perustuvaa strategiaa, jossa oletetaan *m* kappaletta identtisiä prosessoreita ja kaikkien tehtävien olevan työmäärältään yhtä pitkiä. He todistivat offline-algoritmin tilanteen ratkaisemiseen olevan NP-kova ongelma. He kehittivät kuitenkin optimaalista tulosta korkeintaan tietyn vakion verran virheellisesti arvioivan algoritmin, jonka aikavaatimus skaalautuu polynomiaalisesti. He myös kehittivät online-algoritmeja, joiden hyvyys oli vakio. Lam ja muut [2008] tutkivat kahden prosessorin aikarajoiteperusteista aikataulutusta, ja esittivät vakion verran hyvän online-algoritmin.

5.3. Ohjelmointinäkökulma – energiatyypit suoraan ohjelmointikieleen

Miten sovelluksia tehtaileva ohjelmistosuunnittelija voi sitten konkreettisesti ottaa työssään energiankulutuksen huomioon? Yksi esimerkki on Cohenin ja muiden [2012] esittelemä energia-tyyppien lisääminen ohjelmointikieleen. He tekivät konseptiaan demonstroivan oman ET-kielensä lisäämällä energiatyypit Javaan. ET on tarkoitettu Android-ohjelmien tekoon. Energiatyyppien konsepti sinänsä on yleistettävissä muihinkin kieliin.

Heidän energia-tyyppi-järjestelmänsä koostuu *vaiheista* (phases) ja *moodeista* (modes), jotka ovat ohjelmoijan määrittämiä.

Vaihe kuvaa ohjelmassa tietyn tyyppistä toimintatapaa. Esimerkiksi muistin ja prosessorin käyttölogiikkaa. Vaiheet ovat siis ohjelmoijan itsensä määrittämiä kokonaisuuksia. Ohjelmoija pystyy merkitsemään dataa ja metodeita kuulumaan tiettyyn vaiheeseen. Vaiheista on heidän mukaansa kaksi hyötyä: 1. alla oleva käyttöjärjestelmä- ja laitteistototeutus pystyy säätämään itseään paremmin, ja 2. energiakulutus on ennustettavampaa ja toimintaa pystytään eräyttämään (*batching*) helpommin.

Moodit ovat ohjelmoijan määrittämiä energiankäyttötiloja. Tästä on kaksi lisähyötyä edellämainittujen lisäksi: 3. antaa ohjelmoijalle työkalut ja syntaksin ajatella kätevästi erilaisia energiansäästömalleja, ja 4. määrittää, mitkä ohjelman osaset toimivat milläkin energiansäästötasolla. Eli korkean energiamoodin ollessa päällä ohjelma voi käyttää mitä

tahansa osasistaan, kun taas jotkut metodit saattavat olla poissa käytöstä alemmilla tiloilla.

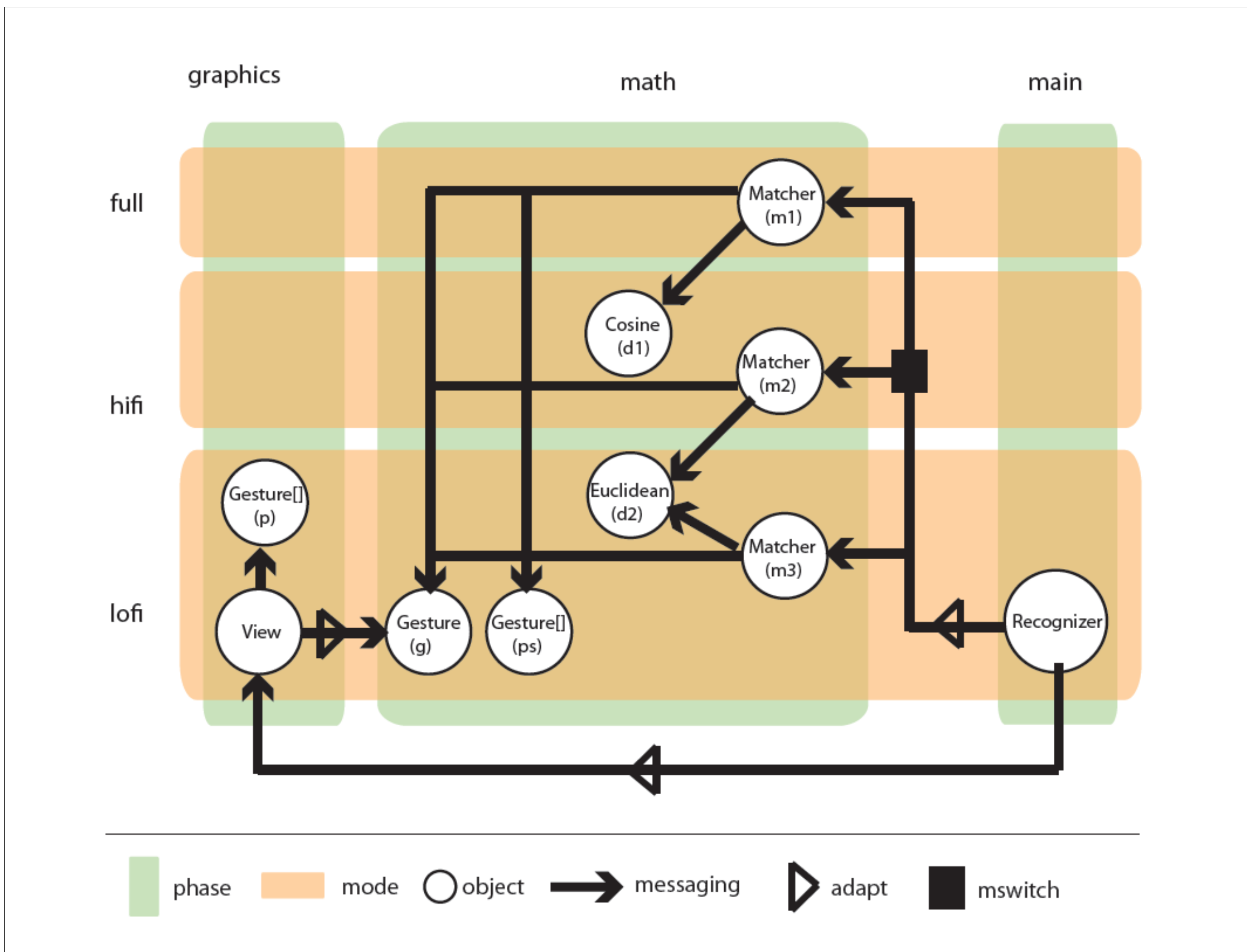
Kuvassa 5 näkyy Cohenin ja muiden [2012] esimerkkiohjelma hahmontunnistuksesta kirjoitettuna ET-kielellä. Kielen erityispiirteet on korostettu punaisella ja lihavoinnilla. Kuvassa 6 on Cohenin ja muiden [2012] havainnollistava kaavio esimerkkiohjelman toiminnan tasoista. Esimerkkiohjelma on jaettu kolmeen eri vaiheeseen: pääohjelma, matematiikka ja grafiikka. Ohjelman funktioita ja luokkia voidaan nyt yhdistää näihin vaiheisiin *@phase()* -syntaksilla, kuten esimerkiksi rivillä 57 asetetaan *Matcher*-luokka matematiikkavaiheeseen esittelemällä se normaalin *class Matcher {}* -esittelyn sijaan kirjoittamalla *class Matcher@phase(math) {}*. Matcher-luokan sijaintia matematiikkavaiheessa havainnollistetaan myös kaaviossa, jossa sen käyttöä esiintyy vain kyseisessä vaiheessa. Esimerkkiohjelmassa on määritelty myös kolme moodia: *lofi*, *hifi* ja *full*. Lofi tulee sanoista low fidelity ja tarkoittaa alhaista laatutasoa, hifi tulee sanoista high fidelity ja tarkoittaa korkeaa laatutasoa ja full tarkoittaa tässä täyttä laatutasoa. Moodin mukaan ohjelmassa on määritetty esimerkiksi kolme eri energiansäästötasoa tunnistuksen parametreiksi. Jokaista tasoa kohden luodaan oma tunnistusta laskeva *Matcher*-olio, joka nähdään riveillä 25–27. Kuten nähdään, moodin liittäminen tehdään lisämääreenä ohjelmointikielen tyyppiin, eli normaalin luokkatyypin *Matcher* sijaan määritelläänkin tyypiksi esimerkiksi *Matcher@mode(full)*. Täydessä moodissa lasketaan tarkkuustasolle 0.995 asti, ja alhaisimmassa moodissa vain tasolle 0.5, jolloin energiaa ei tietenkään tarvita niin runsaasti. Moodien vaihtamisehdot toteutetaan *modev*-tyypillä. Esimerkkiohjelmassa *batteryState()*-funktio tarkastaa laitteen akun tilan, ja päättelee moodin sen perusteella, kuten nähdään riveiltä 46–52. Tätä *modev*-moodityyppiä voidaan tarkastella *mswitch*-lauseella, joka toimii yleisistä ohjelmointikielistä tutun *switch*-lauseen kanssa samaan tapaan, mutta käyttää moodityyppejä eikä tarvitse *break*-lausetta. Esimerkkiohjelmassa *batteryState()*-funktion määrittelemä ja palauttama *modev* otetaan huomioon riviltä 33 alkavassa *mswitch*-lausekkeessa, jossa tämän *modev*in mukaan kutsutaan käyttöön aiemmin riveillä 25–27 moodeittain määriteltyä *Matcher*-oliota. Eri moodien tasot näkyvät myös kuvan 6 kaaviosta tasoittain. *Full*-moodissa käytetään oliota *m1*, *hifi*-moodissa oliota *m2* ja *lofi*-moodissa oliota *m3*.

```
1.  phases { graphics <cpu main; main <cpu math; }
2.  modes { hifi <: full; lofi <: hifi; }
3.
4.  class Main
5.  {
6.   main()
7.   {
8.       Recognizer rz = new Recognizer();
9.       View v = adapt (new View());
```

```
10.    while (true)
11.    {
12.            Gesture g = processInput();
13.            int result = rz.recognize(g);
14.            v.paintOverlay(adapt[graphics] g, result);
15.    }
16. }
17. Gesture processInput()
18. {
19.    return new Gesture@phase(math)();
20. }
21.}
22.
23.class Recognizer
24.{
25. Matcher@mode(full) m1 = newM(0.995);
26. Matcher@mode(hifi) m2 = newM(0.9);
27. Matcher@mode(lofi) m3 = newM(0.5);
28. DistCal d1 = new Cosine@mode(hifi)();
29. DistCal d2 = new Euclidean@mode(lofi)();
30.
31. int recognize(Gesture g)
32. {
33.    mswitch(batteryState())
34.    {
35.            case full: return m1.match(g, d1);
36.            case hifi: return m2.match(g, d2);
37.            case lofi: return m3.match(g, d2);
38.    }
39. }
40.
41. Matcher newM(double p)
42. {
43.    return adapt (new Matcher( p ));
44. }
45.
46. modev batteryState()
47. {
48.    if(Util.batterycharged())
49.            return full;
50.    if(Util.batterylevel() > 0.3)
51.            return hifi
52.    else
53.            return lofi;
54. }
55.}
56.
57.class Matcher@phase(math)
58.{
59. double precision;
60. Gesture[] ps = …; // recognizable patterns
61.
62. Matcher(double precision)
63. {
64.    this.precision = precision;
65. }
66.
67. int match(Gesture g, DistCal dc)
68. {
69.    Gesture gs = sampling(g, precision);
```

```
70.     for (int i = 0; i < ps.length; ++i)
71.     {
72.             if(dc.compute(gs, ps[i]) < THRES)
73.                     return i;
74.     }
75. }
76. …
77.}
78.
79.class View@phase(graphics)
80.{
81. Gesture[] p = ..; // recognizable patterns
82.
83. void paintOverlay(Gesture g, int result)
84. {
85.     paint(g);
86.     paint(p[result]);
87. }
88.
89. void paint(Gesture g)
90. {
91.     Util.draw(g.Coordinates());
92. }
93.}
94.
95.class DistCal { … }
96.class Cosine extends DistCal { … }
97.class Euclidean extends DistCal { … }
98.class Gesture { … }
99.
100.    static class Util
101.    {
102.            boolean batterycharged() { … }
103.            double batterylevel() { … }
104.            void draw(int co) { … }
105.    )
```

Kuva 5. Cohenin ja muiden [2012] hahmontunnistusesimerkki ET-kielellä.



Kuva 6. Cohenin ja muiden [2012] hahmontunnistusesimerkkiä selventävä kaavio.

Teknisesti ottaen samat energiansäästöt voitaisiin tehdä ilman lisäyksiä ohjelmointikieleen. Cohen ja kumppanit [2012] perustelevat kieleen integrointia sillä, että se on alustariippumatonta ja tarjoaa ohjelmoijalle yhtenäisen tavan ilmaista energiansäästöhalunsa. Huonona puolena he näkevät, että jos ohjelmoijalle annetaan valtaa, heillä on valta tehdä myös haitallisia asioita vahingossa tai tarkoituksella.

Testeissään he käänsivät muutamia Android-Java sovelluksia ET-kielelleen, ja testasivat niitä Android-puhelimella. He määrittelivät sovelluksin myös vaiheita ja moodeja. Kolme viidestä testisovelluksesta toimi vähintään samalla nopeudella, mutta kulutti 30% - 50% vähemmän virtaa. He pitivät merkittävänä myös sitä, että heidän staattinen järjestelmänsä päihitti dynaamisen prosessorin nopeuden skaalauksen.

Heidän seuraava kiinnostuksen kohteensa on muistijärjestelmien kehittäminen niin, että välimuistin huteja ei tulisi niin paljoa. Se parantaisi energiatehokkuutta merkittävästi.

6. Yhteenveto

Olen tässä työssä esitellyt erilaisia tapoja ja näkökulmia energian huomioon ottamiseksi teknologiassa. Erityisesti nostin asioita esiin modernien trendien, kuten pilvi- ja mobiilipalveluiden, näkökulmasta. Laitteiden ja käyttötavan puolesta nämä ympäristöt ovat hyvin erilaisia, mutta ne molemmat tukevat samaa modernia palvelumallia, jossa laskenta siirtyy ihmisten pöydiltä ja perinteisiltä tietokoneilta datakeskuksiin ja niiden kanssa kommunikoiviin taskuun sopiviin mobiililaitteisiin. Ympäristöjen erilaisuudesta huolimatta näen energian säästämiseen tähtäävissä algoritmillisissa toimenpiteissä hyvin paljon yhtäläisyyttä täysin ympäristöstä riippumatta. Keskeisessä asemassa on tunnistaa, mistä resurssin käytöstä virran kulutus muodostuu. Tämän jälkeen pyritään minimoimaan kyseisen resurssin käyttö, strategiana tiedostaa mahdollisimman hyvin kokonaistilanne ja ennustaa sen perusteella tuleva tilanne ja käyttää ennustusta pohjana resurssin käytön minimoivan aikataulutuksen laatimisessa ja virransäästötiloihin asettamisessa. Tämä pätee niin datakeskuksen palvelinten kuin mobiililaitteiden radioidenkin suhteen. Siis laitetaan ennustettavan käyttötarpeen mukaan kiinni energiaa paljon kuluttavat resurssit mahdollisimman nopeasti. Aikataulutuksella suositaan energiaa vähän kuluttavia tapoja. Esimerkkejä tästä ovat datakeskuksessa vihreän energian suosiminen tai mobiililaitteessa datan kerääminen ja lähettäminen tehokkaasti kerralla. Myös datakeskus ja tukiasemaverkko ovat saman kaltaisia siinä, että molemmissa kokonaistilanteen havainnointi on tärkeää, jotta resurssit, joita ei juuri tarvita, voidaan ottaa pois käytöstä.

Oikein sovellettuna energiatehokkuus jopa lisää käyttömukavuutta pidemmän akun keston, laitteiston pitkäikäisyyden ja pienemmän lämpenemisen muodossa. Tulevaisuuden kannalta olisi tärkeää yhä kehittää ja lisätä älykkäämpää energianäkökulman huomioimista teknologiaan, jotta voimme säästää luonnonvaroja ja rahaa samalla, kun käyttökokemuksemme paranee.

Viiteluettelo

[Albers, 2009] Susanne Albers, Energy-Efficient Algorithms. *Communications of the ACM* **53**, 5 (2010), 86–96.

[Albers and Fujiwara, 2007] Susanne Albers and Hiroshi Fujiwara, Energy-Efficient Algorithms for Flow Time Minimization. *ACM Transactions on Algorithms* **3**, 4 (November 2007), article no. 49.

[Albers et al., 2007] Susanne Albers, Fabian Müller, and Swen Schmeizer, Speed Scaling on Parallel Processors. In: *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures*, 289–298.

[Bansal et al., 2007] Nikhil Bansal, Tracy Kimbrel, and Kirk Pruhs, Speed Scaling to Manage Energy and Temperature. *J. ACM* **54** (2007), article no. 3.

[Bansal et al., 2008] Nikhil Bansal, Ho-Leung Chan, Tak-Wah Lam, and Lap-Kei Lee, Scheduling for Speed Bounded Processors. In: *Proceedings of the 35th International Colloquium, ICALP, July 7 – 11, 2008. Reykjavik, Iceland,* 409–420.

[Bansal et al., 2009] Nikhil Bansal, Ho-Leung Chan, Kirk Pruhs, Speed Scaling with an Arbitrary Power Function. In: *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms*, 693–701.

[Barr and Asanovic, 2003] Kenneth Barr and Krste Asanovic, Energy Aware Lossless Data Compression. In: *The Proceedings of The First International Conference on Mobile Systems and Services, May 5 – 8, 2003,* 250–291.

[Barroso, 2005] Luiz André Barroso, The Price of Performance. *Multiprocessors* **3**, 7 (2005).

[Berl et al., 2009] Andreas Berl, Erol Gelenbe, Marco Di Girolamo, Giovanni Giuliani, Hermann De Meer, Minh Quan Dang, and Kostas Pentikousis, Energy-Efficient Cloud Computing. *The Computer Journal* **53**, 7 (2010), 1045–1051.

[Bhaumik et al, 2010] Sourjya Bhaumik, Girija Narlikar, Subhendu Chattopadhyay, and Satish Kanugovi, Breathe to Stay Cool: Adjusting Cell Sizes to Reduce Energy Consumption. In: *Proceedings of the* First *ACM SIGCOMM Workshop on Green Networking,* Aug 30 – Sept 3, New Delhi, India, 2010, 41–46.

[Buyya et al., 2010] Rajkumar Buyya, Anton Beloglazov, and Jemal Abawajy, Energy-Efficient Management of Data Center Resources for Cloud Computing: A Vision, Architectural Elements, and Open Challenges. In: *Proceedings of the 2010 International Conference on Parallel and Distributed Processing Techniques and Applications, Last Vegas, USA, July 12–15, 2010.*

[Cohen et al., 2012] Michael Cohen, Haitao Steve Zhu, Senem Ezgi Emgin, and Yu David Liu, Energy Types. In: *OOPSLA '12, Oct 19 – 26, 2012,* 831–850.

[Dogar et al., 2010] Fahad R. Dogar, Peter Steenkiste, and Konstantina Papagiannaki, Catnap: exploiting high bandwidth wireless interfaces to save energy for mobile devices. In: *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services, 2010,* 107–122.

[Falaki et al, 2010] Hossein Falaki, Dimitrios Lymberopoulos, Ratul Mahajan, Srikanth Kandula, and Deborah Estrin, A First Look at Traffic on Smartphones. In: *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement,* 2010, 281–287.

[Fei et al., 2008] Yunsi Fei, Lin Zhong, Niraj K. Jha, An energy-aware framework for dynamic software management in mobile computing systems. *TECS* **7**, 3 (2008), article no. 27.

[Google, 2013] Google, *Hamina Data Center*, viimeksi tarkistettu 4.7.2013, http://www.google.fi/about/datacenters/locations/hamina/.

[Goiri et al., 2011] Íñigo Goiri, Kien Le, Md. E. Haque, Ryan Beauchea, Thu D. Nguyen, Jordi Guitart, Jordi Torres, and Ricardo Bianchini, GreenSlot: Scheduling Energy Consumption in Green Datacenters. In: SC '11*, Nov 12 – 18, 2011, Seattle, Washington, USA,* article no. 20.

[Irani et al., 2007] Sandy Irani, Sandeep Shukla, and Rajesh Gupta, Algorithms for power savings. *ACM TALG* **3**, 4 (Nov. 2007), article no. 41.

[Kaplan et al., 2008] James M. Kaplan, William Forrest, and Noah Kindler, *Revolutionizing Data Center Energy Efficiency*. July 2008 McKinsey & Company.

[Kelly, 2007] Tim Kelly, ICTs and climate change. *ITU-T Technology, Tech. Rep* (2007).

[Kliazovich et al., 2012] Dzmitry Kliazovich, Pascal Bouvry, and Samee Ullah Khan, GreenCloud: a packet-level simulator of energy-aware cloud computing data centers. *The Journal of Supercomputing* **62**, 3 (2012), 1263–1283.

[Lam et al., 2008] Tak-Wah Lam, Lap-Kei Lee, Isaac K. K. To, and Prudence W. H. Wong, Competitive Non-migratory Scheduling for Flow Time and Energy. In: *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*, 256–264.

[Marsan et al., 2009] M. A. Marsan, L. Chiaraviglio, D. Ciullo, and M. Meo, Optimal Energy Savings in Cellular Access Networks. In: *Proceedings of IEEE ICC, GreenComm Workshop,* June 14–18, 2009, Dresden, 1–5.

[Niu et al., 2010] Zhisheng Niu, Yiqun Wu, Jie Gong, and Zexi Yang, Cell Zooming for Cost-Efficient Green Cellular Networks. *Communications Magazine, IEEE 48, no. 11* (2010), 74–79.

[Patel, 2003] Chandrakant D. Patel, A Vision of Energy Aware Computing from Chips to Data Centers. In: *Proceedings of the International Symposium on Micro-Mechanical Engineering, December 1–3, 2003*.

[Global Action Plan, 2007] Global Action Plan, An Inefficient Truth, *Global Action Plan Report*, 2007.

[Pruhs et al., 2008] Kirk Pruhs, Patchrawat Uthaisombut, and Gerhard Woeginger, Getting the Best Response for Your Erg. *ACM Transactions on Algorithms* **4**, 3 (June 2008), article no. 38.

[Ra et al., 2010] Moo-Ryong Ra, Jeongyeup Paek, Abhishek B. Sharma, Ramesh Govindan, Martin H. Krieger, and Michael J. Neely, Energy-Delay Tradeoffs in Smartphone Applications. In: *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services, 2010,* 255–270.

[Rahmati and Zhong, 2007] Ahmad Rahmati and Lin Zhong, Context-for-Wireless: Context-Sensitive Energy-Efficient Wireless Data Transfer. In: *Proceedings of the 5th International Conference on Mobile Systems, Applications, and Services, 2007,* 165–178.

[Rakhmatov et al., 2002] Daler Rakhmatov, Sarma Vrudhula, and Deborah A. Wallach, Battery lifetime prediction for energy-aware computing. In: *ISLPED '02, 154–159*.

[Ranganathan, 2010] Parthasarathy Ranganathan, Recipe for Efficiency: Principles of Power-Aware Computing. *Communications of the ACM* **53**, 4 (Apr. 2010), 60–67.

[Schulman et al., 2010] Aaron Schulman, Vishnu Navda, Ramachandran Ramjee, Neil Spring, Pralhad Deshpande, Calvin Grunewald, Kamal Jain, and Venkata N. Padmanabhan, Bartendr: a practical approach to energy-aware cellular data scheduling. In: *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking, 2010,* 85–96.

[Skadron et al., 2003] Kevin Skadron, Mircea R. Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan, Temperature-aware microarchitecture. In: *Proceedings of the 30th annual international symposium on Computer architecture, 2–13*.

[Srikantaiah et al., 2008] Shekhar Srikantaiah, Amal Kansal, and Feng Zhao, Energy Aware Consolidation for Cloud Computing, November 14, 2008. Available as *http://static.usenix.org/event/hotpower08/tech/full_papers/srikantaiah/srikantaiah_html/*

[Takizawa et al., 2008] Hiroyuki Takizawa, Katuto Sato, and Hiroaki Kobayashi, SPRAT: Runtime Processor Selection for Energy-aware Computing. In: *Proceedings of the 2008 IEEE International Conference on Cluster Computing, 386–393*.

[Vallina-Rodriguez et al., 2010] Narseo Vallina-Rodriguez, Pan Hui, Jon Crowcroft, and Andrew Rice, Exhausting Battery Statistics. In: *Proceedings of the Second ACM SIGCOMM Workshop on Networking, Systems, and Applications on Mobile Handhelds,* 2010, 9–14.

[Wang et al., 2011] Xiaofei Wang, Athanasios V. Vasilakos, Min Chen, Yunhao Liu, and Ted Taekyoung Kwon, A Survey of Green Mobile Networks: Opportunities and Challenges June 1, 2011.

[Yao et al., 1995] Frances Yao, Alan Demers, Scott Shenker, A Scheduling Model for Reduced CPU Energy. In: *Proceedings of the 36th IEEE Symposium on Foundations of Computer Science* (1995), 374 – 382.

[Zhou et al., 2009] Sheng Zhou, Jie Gong, Zexi Yang, Zhisheng Niu, and Peng Yang, Green Mobile Access Network with Dynamic Base Station Energy Saving. In: *Proceedings of ACM MobiCom,* September 20–25, 2009, Beijing, 10–12.