View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Trepo - Institutional Repository of Tampere University

Tampereen yliopisto

Jani Aakio

OHJELMISTOJEN AJONAIKAINEN PAISUMINEN

Syyt, seuraukset sekä ratkaisut

Informaatioteknologian ja viestinnän tiedekunta
[Kandidaattitutkielma]
Tammikuu 2020

TIIVISTELMÄ

Jani Aakio: Ajonaikainen paisuminen: syyt, seuraukset sekä ratkaisut
[Kandidaattitutkielma]
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Joulukuu 2019

Tämä työ käsittelee ajonaikaista paisumista. Työn tarkoituksena on tutkia mitkä seikat aiheuttavat ajonaikaista paisumista, mitkä ovat sen vaikutukset sekä tutkia joitakin ratkaisuja ajonaikaiseen paisumiseen. Se pyrkii esittelemään lukijalleen mitä tarkoitetaan ajonaikaisella paisumisella, kuinka vakavia ovat sen vaikutukset sekä miten paisumista voidaan välttää sekä poistaa.

Ajonaikaista paisumista aiheuttavat monet tekijät nykyaikaisessa ohjelmistokehityksessä. Nopeat sekä ketterät kehityssuunnat ohjelmistokehityksessä, kuten koodin uudelleenkäyttö sekä suurien kirjastojen implementaatio, voivat aiheuttaa vakavaa ajonaikaista paisumista ohjelmistoissa. Tämä paisuminen voi aiheuttaa ohjelmiston epätehokkuutta sekä epävakautta. Nopeasta ja ketterästä kehityksestä ei olla luopumassa, mutta jonkinlaisia ratkaisuja olisi hyvä löytää.

Tarjolla on kuitenkin monia tekniikoita, joita voidaan ottaa käyttöön ohjelmistokehityksen aikana, joilla voidaan ajonaikaista paisumista vähentää. Nämä tekniikat voivat olla hyvin ihmislähtöistä koulutusta, jota tarjotaan ohjelmistokehittäjille, jotta he olisivat tietoisia paisumisen ongelmista. Tekniikat voivat myös olla täysin automatisoituja apu-ohjelmistoja, jotka ilmoittavat kehittäjille, jos heidän ohjelmistossaan on paisumis-ongelma.

Avainsanat: Ohjelmiston optimointi, Paisuminen, Ohjelmistotuotanto

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

1 Johdanto

Ohjelmistot muuttuvat monimutkaisemmiksi vuosien saatossa ja asiakkaat vaativat enemmän toimintoja, joita ohjelmistokehittäjät heille tuottavat. Tämä luonnollisesti johtaa siihen, että ohjelmistonkehittäjiltä vaaditaan yhä enemmän ja vaativampaa kehitystyötä. Tämä on johtanut erilaisiin ratkaisuihin ohjelmistonkehityksessä, joilla työmäärää yritetään vähentää ja helpottaa. Monet näistä tekniikoista, kuten metodien uudelleenkäyttö sekä isojen yleiskirjastojen luonti, onkin helpottanut kehittämistä huomattavasti. Näiden mukana on kuitenkin saapunut ongelma, jota kutsutaan paisumiseksi (*bloat*). Paisumisella voidaan viitata moneen eri aihealueeseen ohjelmistoissa, mutta tämä työ keskittyy ajonaikaiseen paisumiseen (*runtime bloat*), eli siihen kuinka paljon ohjelmisto lataa muistiin turhaa dataa [1]. Erotus on kuitenkin tehtävä tämän sekä epätehokkuuden välille. Paisuminen ei viittaa yksittäisten tekniikoiden ja metodien tehokkuuteen, se viittaa turhiin muistijälkiin ohjelman ajon aikana [4].

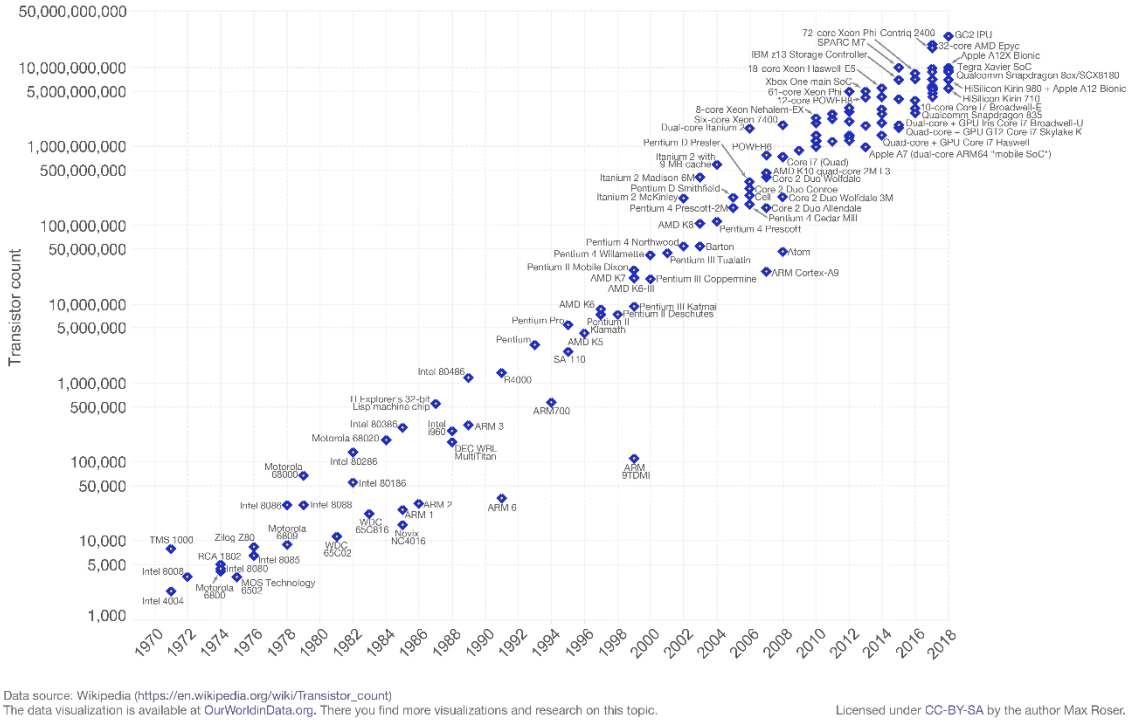
Tämän työn tarkoituksena on tarkastella, mitkä seikat aiheuttavat paisumista, mitkä ovat sen seuraukset sekä millaisia ratkaisuja on tarjolla paisumisen ongelmiin. Työ alkaa tutkimalla eri syitä, jotka aiheuttavat ajonaikaista paisumista. Luvussa 3 tutkitaan mitä paisuminen aiheuttaa ja kuinka tärkeää sen poisto on. Luku 4 etsii erilaisia ratkaisuja paisumisen vähentämiseen, sekä ihmislähtöisiä että konkreettisia tekniikoita. Ratkaisuihin on valittu kaksi erilaista ratkaisua, jotka ovat esitelty viimeisen kymmenen vuoden aikana. Ne ovat valittu esimerkeiksi erilaisista teknisistä ratkaisuista, joilla paisumista voidaan ratkaista.

Tutkimuksen lähteet on etsitty alun perin google-scholarin sekä UTA:n kirjaston Andor-hakupalvelulla. Osa lähteistä on myös löydetty alkuperäisten löydettyjen lähteiden lähteistä.

2 Paisumisen syyt

Gordon Moore saneli monelle tutun lakinsa vuonna 1965: komponenttimäärä tuplaantuu mikropiireissä kahden vuoden välein [2]. Käytännössä tämä ennustaisi sitä, että tietokoneiden nopeus nousisi 50% vuosittain. Kolmekymmentä vuotta eteenpäin tämä ennustus piti vieläkin paikkansa, sillä vuonna 2004 vuosittainen nousu oli noin 40% [3]. Kuva 1 näyttää tämän kasvun näyttämällä transistorien määrän prosessoreissa vuosien varrella. Kuitenkaan peruskäyttäjälle tämä nopeus ei näy suoraan, vaan osa tästä tehosta katoaa ennen kuin käyttäjä pääsee sitä käyttämään. Jokainen varmasti kokee, että uusi tietokone on nopeampi kuin vanha, mutta harvalle tämä nopeus tuntuisi moninkertaiselta. Tämä johtuu siitä, että samalla kun komponenttien nopeus nousee, nousevat myös eri ohjelmien

vaatimukset näiltä komponenteilta. On kuitenkin epävarmaa, ovatko nämä suuremmat vaatimukset täysin tarpeellisia vai katoaako tehoa itse kehitysprosessin aikana.



Kuva 1. Transistorien määrä mikropiireissä vuosilta 1971-2018

James Larus [3] analysoi tutkimuksessaan tästä vastakkainasettelusta: tehon nousu vastaan tehon vaatiminen, ja kutsuu sitä Mooren osingon kuluttamiseksi. Mooren lain mukaan prosessoriemme teho nousee jatkuvasti, jonka ansiosta kuluttajilla pitäisi olla ylimääräistä tehoa kulutettavaksi. Tämä teho kuitenkin katoaa hyvin nopeasti uusiin ohjelmiin, jotka vaativat huomattavasti enemmän resursseja kuin vanhat. Larus tuo esille myös toisen tietotekniikan lain: Myhrvoldin lain. Myrhvoldin lakii sanoo yksinkertaisesti: ”Ohjelmistot ovat kaasua”. Tällä hän tarkoittaa sitä, että ohjelmistot täyttävät kaiken tilan, jonka saavat ja tilalla tässä tarkoitetaan Moorelta saatavia ylimääräisiä resursseja. Tämän kaasuuntumisen syyt voidaan löytää nykyaikaisista ohjelmistokehityksen tekniikoista.

2.1 Nopean kehityksen ongelmat

Nykyaikaiset tekniikat ovat tehostaneet ohjelmistojen kehitysprosessia huomattavasti. Koodin uudelleenkäyttö, joka viittaa tekniikkaan, jossa koodia kirjoitetaan niin, että samaa koodia voidaan käyttää useassa tilanteessa. Ohjelmistokehykset, jotka ovat valmiita malleja ohjelmille, jotka tarjoavat usein paljon uudestaan käytettävää koodia. Suuret koodikirjastot, jotka tarjoavat myös valmiiksi käytettäviä metodeja ohjelmoinnin helpottamiseksi.

Näiden tekniikkojen käyttö ei ole kuitenkaan tullut ilman lieveilmiöitä ja yksi selvistä ilmiöistä on paisuminen [4][6]. Ensimmäinen ongelma, jonka nämä tekniikat

aiheuttavat on piilotus kehittäjältä. Useimmat näistä tekniikoista toimivat niin, että itse kehittäjän ei tarvitse tietää mitä ohjelmassa tapahtuu. Koodin uudelleenkäyttö ja koodikirjastot varsinkin ovat usein suunniteltu niin, että ohjelmoija on tekemisissä vain ohjelmointirajapintojen kanssa. Rajapinnat kertovat ohjelmoijalle miten tiettyä koodia käytetään, ilman että ohjelmoijan itse tarvitse tietää miten koodi toimii. Tämä tietenkin säästää kehittäjän aikaa koska hänen ei tarvitse tarkastaa jokaista metodia erikseen, mutta tämä tiedonpuute voi johtaa paisumisongelmiin. Varsinkin jos rajapintojen käyttö on yhdistettynä seuraavan ongelman kanssa, joka on metodien yleistys. Metodien yleistyksellä viittaan siihen, että usein uudelleenkäytetyssä koodissa sekä koodikirjastoissa käytetyt metodit ovat hyvin yleiskäyttöisiä ja ne ovat suunniteltuja niin, että niitä voidaan käyttää tuhansissa eri tilanteissa. Tämä on hyvä kehittäjän kannalta, koska hänen ei tarvitse etsiä juuri tiettyä metodia, jota käyttää koodissaan, vaan hän voi käyttää yleismetodia. Tällaiset metodit ovat usein kuitenkin hyvin epätehokkaita muistinkäytön kannalta, sillä ne saattavat tehdä paljon ylimääräistä työtä tarjotessaan tällaisen yleistyksen [4].

Tämän työn tarkoitus ei ole keskustella edellä mainittujen tekniikoiden tarpeellisuudesta, mutta huomioon ottaen näiden tekniikoiden suosion, ovat ne selvästi hyvin tärkeitä ohjelmistokehitykselle. Useimmat kääntäjät sekä roskankerääjät ovat ohjelmoitu poistamaan näitä ongelmia. Kun nykyaikainen kääntäjä muuttaa ohjelmoijan koodin sellaiseksi, että tietokone osaa sitä lukea, tekee se myös paljon optimointia ohjelmoijan puolesta. Roskankerääjä taas yrittää pitää huolta, että jos ohjelmoija ei käytä enää jotakin dataa, jota hän ohjelmassaan loi, niin tämä data poistetaan. Mutta mitä syvemmin ja laajemmin näitä tekniikoita käytetään, sitä vaikeampaa on kääntäjän tai roskankerääjän niitä poistaa [7].

2.2 Ohjelmistokehittäjän vastuu

Osa paisumisesta kuitenkin lankeaa itse ohjelmistokehittäjänkin harteille. Nykyaikaista kehittäjää ei ole opetettu miettimään liikaa käyttämäänsä tehoa, sillä tehoa tuntuu riittävän aina tarpeeksi. Ennen kehittäjä joutui harkitsemaan käyttääkö hän ohjelmansa painikkeessa sanaa “Confirm” vai vaihtaako hän sen tekstiin “Ok”, sillä tästä säästetyt bitit voivat olla hyvin tärkeitä myöhemmin ohjelmassa [3]. Tällainen ajattelu on hyvin kaukaista nykyaikaiselle kehittäjälle, eikä hänen tarvitse olla hyvinkään tietoinen ohjelmansa todellisesta muistinkäytöstä. Tätä tietoisuutta vähentää myös luottomme nykyaikaisiin kääntäjiin, jotka ovat hyvin tehokkaita optimoimaan koodia käännösvaiheessa. Koska kone tekee itse suuren osan optimoinnista, ei moni kehittäjä tutki oman koodinsa tehokkuutta, vaan luottaa teknologiaan.

Ei voida myöskään olettaa, että nykyaikaisessa nopeassa kehityksessä kehittäjä käyttäisi aikaansa koodinsa liikaan optimointiin. Yksi suurista resursseista ohjelmistokehityksessä on aika ja optimointi vie kehittäjältä aikaa, jota voidaan käyttää uusien

toimintojen kehittämiseen ohjelmistossa. Vaikka kehittäjä olisikin hyvin tietoinen käyttämästään muistista, ohjelmistojen kehitystiimit ovat nykyään hyvin suuria ja ohjelmistot sitäkin monimutkaisempia. Yksittäisen kehittäjän on hyvin vaikea näissä olosuhteissa olla tietoinen omasta vaikutuksestaan ohjelmaan, sillä hän saattaa olla tekemisissä satojen eri ohjelman osien kanssa, joihin muutkin ohjelmoijat ovat koskeneet.

3 Paisumisen oireet

Nykyisestä ohjelmistokehityksen suuntauksesta on tietenkin ollut seurauksia. Quach et al. [1] tutkimuksessaan esittävät, kuinka paljon dataa ohjelmistot lataavat käyttömuistiinsa ohjelman suorituksen aikana ja paljonko tästä muistista todellisuudessa suoritetaan. Tämän perusteella he pystyvät tutkimaan paljonko ohjelman muistikäytöstä on turhaa. Keskimäärin testatuissa käyttäjäohjelmistoissa vain 21 % prosenttia ladatuista koodista kutsuttiin. Käytännössä tämä tarkoittaa sitä, että ohjelma lataa noin 80 % ylimääräisiä funktioita sekä muita komentoja muistiinsa ja valmistaa ne käytettäväksi, mutta ei koskaan niitä kutsu, joten ne ovat täysin turhia ohjelmiston suorituksen kannalta. Tietenkään ei voida olettaa, että muistia käytettäisiin täydellisesti, mutta 80 % turhaa muistinkäyttöä on objektiivisesti liikaa. Samassa tutkimuksessa tutkittiin myös ohjelmistojen muistinkäyttöä staattisesti. He tutkivat ohjelmistojen koodia ja selvittivät paljonko ohjelmistot voivat käyttää omasta koodistaan, niihin liitetyistä kirjastoista sekä mahdollisista riippuvuuksista. Yhteenlaskettuna ohjelmien oma koodi, kirjastot sekä riippuvuudet saatiin keskimääräiseksi luvuksi 36% prosenttia. Toisin sanoen, parhaimmassakin tapauksessa nämä ohjelmistot käyttävät vain 36% niihin liitetystä koodista, muu on niille turhaa.

3.1 Väliaikaiset objektit

Toisentyyppistä paisumista ovat väliaikaiset objektit, joita usein luodaan olio-ohjelmoinnissa. Nämä eivät ole niin helposti huomattavissa tutkimuksissa, koska kuten nimi paljastaa, ne katoavat hyvin äkkiä käyttönsä jälkeen. Varsinkin hyvin suositussa Java-ohjelmointikielessä suositaan väliaikaisia objekteja, joita luodaan metodien sisällä, jonka jälkeen ohjelmisto tallettaa objektin kekoon (*heap*), joka on ohjelmistojen epätehokkaampi muistiosuus. Tämän jälkeen annetaan Javan oman roskankerääjän poistaa kyseinen objekti omalla ajallaan [5]. Tämä ei kuitenkaan tapahdu ilman ongelmia ja mitä enemmän tällaista tekniikkaa käytetään, sitä epätehokkaammaksi tällainen roskankeräys muuttuu [6]. Pahin tapaus näille väliaikaiseille objekteille ovat tietenkin unohtuneet viittaukset näihin objekteihin, jolloin ne eivät poistu muistista lainkaan ja aiheuttavat muistivuotoja. Tällaisien viittauksen ongelma on kumulatiivinen ja jos sitä ei huomata, voi se pahimmassa tapauksessa vaikuttaa ohjelman vakauteen. Nykyaikaiset kielet ja niiden kääntäjät ovat hyvinkin tehokkaita havaitsemaan ja poistamaan tällaisia kriittisiä ongelmia, mutta täysin niitä ei olla saatu poistettua.

On hyvin vaikea arvioida yleisellä tasolla, kuinka paljon käytännön tehoa menetetään lopullisesti sekä millainen vaikutus tällä on loppukäyttäjälle. Koneet, jotka näitä ohjelmistoja käyttävät vaihtelevat rannekelloista kokonaisiin serverihuoneisiin ja vaikutus näissä voi olla hyvin erilainen. On kuitenkin selvää, että nämä vaikutukset voivat olla hyvinkin mittavia [7]. Näihin ei kuitenkaan aina kiinnitetä niin paljon huomiota, kuin olisi tarpeellista. Tietokoneet ja prosessorit ovat kehittyneet tasolle, jossa kehittäjällä on käytettävissään hyvin usein paljon ylimääräistä tehoa, jota he voivat hyödyntää ohjelmistoissaan. Näin ohjelmistot jätetään tehottomiksi koska koneet pystyvät suoriutumaan ohjelmiston käytöstä ilman suuria ongelmia [3]. Ihmiset eivät kuitenkaan käytä tietokoneitaan vain yhteen asiaan kerrallaan ja käyttämällä tällaisia tehottomia ohjelmia päällekkäin, alkaa tehottomuus vaikuttamaan käytettävyyteen.

Vaikka teho koneissa riittäisi ja sillä ei suurta vaikutusta itse käyttäjään olisi, on muistettava, että teho ei ole ilmaista. Aina kun prosessori tekee laskentaa, käyttää se enemmän sähköä. Erot näissä voivat olla hyvinkin mittavia ja joissakin tutkimuksissa ne ovat olleet jopa lähes nelinkertaisia [8]. Suparna Bhattacharya et al. [12] tutkimuksessaan huomasivat, että varsinkin vähentämällä paisumista ohjelmistojen *pullonkauloissa,* eli sellaisissa operaatioissa, joissa ohjelmisto joutuu käyttämään suuria tehoja laskentamäärää, voidaan sähkön käytön tehokkuutta parantaa. Tulokset vaihtelevat suuresti riippuen koneesta, jolle ohjelmisto suoritetaan ja parhaimmillaan ollaan sähkön käytön tehokkuutta parannettu 7 %. Kotikoneisiin tällainen luku ei tunnu kovinkaan suurelta, mutta puhuttaessa suurista serverifarmeista on tämä 7 % todellinen kuluerä.

4 Ratkaisut

Paisumista vastaan voimme taistella monin eri tavoin. Selvin tapa on tietenkin erilaiset työkalut sekä tekniikat, joita hyödyntämällä voidaan paisumista poistaa. Nämä kuitenkin voivat toimia hyvin eri tavoin ja ne ovat usein suunniteltu tunnistamaan tiettyjä ongelmia. Tällainen monipuolisuus tarkoittaa, että monia eri ongelmia voidaan löytää sekä korjata, mutta se tarkoittaa myös, että yksittäinen työkalu tai tekniikka ei todennäköisesti paisumista täysin poista. Tämä on ongelmallista nopean kehittämisen kannalta sillä usein näiden työkalujen tai tekniikoiden käyttäminen vie aikaa kehitykseltä, joka on tärkeä resurssi nykyaikaiselle nopealle ohjelmistokehitykselle. Varsinkin suosittu kehitystapa, *ketterä kehittäminen* voi kärsiä tästä. Ketterässä kehityksessä tavoitteena on tuottaa toimivia ratkaisuja asiakkaalle hyvin nopeasti, jolloin aika on normaaliakin tärkeämpi resurssi.

Osa työkaluista sekä tekniikoista toimii itse kehittämisen rinnalla ja auttaa tekemään kehittäjiä tekemään parempaa koodia alusta asti, kun taas toinen siivoaa koodin sen ollessa valmis. Hieman vaikeampi, mutta mahdollisesti jopa tärkeämpi tapa välttää sekä poistaa paisumista, on inhimillinen. Itse kehittäjien on oltava tietoisia ja vastuullisia omasta kehitysprosessistaan. Koulutus ja tiedottaminen ovat tärkeitä aseita paisumista

vastaan. On muistettava myös, että monet näistä tekniikoista keskittyvät paisumisen tunnistamiseen ja sen aiheuttajiin. Ohjelmat ovat niin monimutkaisia, että tällaiset tekniikat harvoin pystyvät kertomaan miten ohjelmisto tulisi toteuttaa paremmin. Usein kuitenkin nämä paisumisongelmat ovat olemassa, koska ohjelmiston kehittäjät eivät olleet niistä tietoisia ja ovat kykeneväisiä nämä ongelmat poistamaan, jos ne heille osoitetaan.

4.1 Koulutus

Monet ongelmista, jotka johtavat paisumiseen, ovat kehittäjälähtöisiä, kuten tässä työssä aiemmin keskusteltiin. On siis selvää, että ratkaisuja voidaan myös lähteä etsimään itse kehittäjistä. Nykyinen kehittäjien koulutus painostaa juuri niitä asioita, jotka voivat aiheuttaa paisumista: koodin uudelleenkäyttö, erilaiset viitekehykset ja niiden käyttö, abstraktiot sekä koodikirjastojen käyttö. Usein optimointi ja tehokkuuden saavuttaminen jätetään kääntäjän sekä järjestelmän harteille [7]. Kouluttamalla kehittäjiä ottamaan huomioon koodinsa tehokkuus jo kehitysvaiheessa, voisimme rajoittaa paisumista jo aktiivisesti kehitysvaiheessa. Ensimmäinen tapa olisi kehittää hyviä viitekehyksiä, joilla välttää koodin paisumista sekä kouluttaa kehittäjiä niiden käytössä. Monet paisumisen ongelmista seuraavat samankaltaisia malleja. Täten voisimme kehittää viitekehyksiä, joita käyttämällä kehittäjä välttäisi koodin paisumista jo kehittämisen aikana. Toiseksi kehittäjiä pitäisi opettaa tunnistamaan yleisiä paisumisen aiheuttajia. Useimmat ohjelmoijat eivät luontaisesti tunnista miltä hyvin paisunut ohjelmisto näyttää, ja sen tunnistaminen voikin olla hyvin vaikeaa, koska ainoa varma tapa huomata se, on verrata sitä tehokkaampaan versioon samasta ohjelmistosta [7]. Kaikkia paisumisen tapoja ei tällä tavoin voida tietenkään estää, koska paisumisen tunnistaminen itsessään jo valmiissa ohjelmassa voi olla hyvin hankalaa. Tämä kuitenkin antaisi hyvän pohjan, josta jatko-optimointia voitaisiin tehdä. Esimerkkinä tässä ovat varovaisuus, kun käytetään suuri kirjastoja. Implementoimalla vain sellaiset metodit kirjastoista, joita kehittäjä todella tarvitsee, voitaisiin kirjastoista johtuvaa paisumista vähentää.

4.2 Kopio-profilointi

Kopio-profilointi viittaa tekniikkaan, jossa yritetään tunnistaa ohjelman sisällä tapahtuvaa ylimääräistä kopiointia. Tämä tapahtuu usein suurissa tietorakenteissa, joissa tietoa kuljetetaan ohjelman osasta toiseen, tehden kopioita samasta tiedosta toistuvasti. Moderneissa ohjelmointikielissä, joissa käytössä ei ole eksplisiittisiä osoittimia, on mahdollista, että turhaa kopiointia tapahtuu hyvin helposti. Pienemmissä ohjelmistoissa tällaisten rakenteiden tunnistaminen sekä välttäminen ei ole ammattilaiselle vaikeaa, mutta ohjelman kasvaessa ja erilaisten riippuvuuksien syntyessä, on näiden tunnistaminen yhä vaikeampaa. Guoqing Xu et al. [9] työssään esittelevät tekniikan, jolla kehittäjä voi tunnistaa ja täten välttää tällaisten kopio-ketjujen syntymistä. Tekniikan avulla kehittäjä voi tuottaa

graafin ohjelmiston tuottamista kopio-ketjuista ja käyttää tätä tietoa optimoidakseen näitä ketjuja.

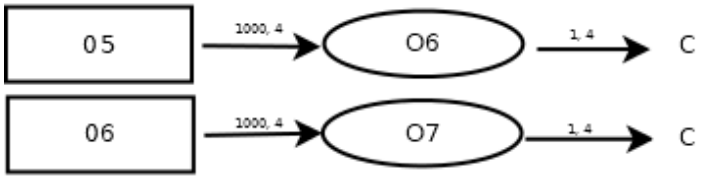
Kopio-profiloinnissa tarkoituksena on pitää kirjaa, missä objektit ilmenevät keossa. Tämä kuitenkin vaatii, että ohjelmiston koko keosta muodostetaan oma varjo-keko (*shadow heap*). Varjo-keko on oma muistinsa, jonka koko on yhtä suuri kuin tutkitun ohjelman keko. Näin voidaan tämän varjo-keon avulla pitää kirjaa siitä, mitkä muistipaikat ovat varattuja ja mitä ne sisältävät, ilman että vaikutetaan alkuperäisen ohjelmiston suoritukseen. Joka kerta kuin ohjelmisto luo uuden objektin kekoon, annetaan saman aikaisesti tälle objektille oma tunnisteensa, tämä tunniste sekä luodun objektin koko tallennetaan varjo-kekoon. Tämän avulla pystytään profiloidessa katsomaan mitä ja minkäkokoisia objekteja se luo suorituksensa aikana.

Itse graafi muodostuu neljästä eri osasta: lähtöpisteet, kaaret, solmut sekä päätepisteet.

- *Lähtöpiste*: Lähtöpiste piirretään graafiin silloin kun, jokin uusi objekti luodaan.
- *Kaari*: Kaari piirretään graafiin silloin kun operaatio luo kopion objektista. Siihen myös kirjataan montako kertaa tämä operaatio suoritetaan, jotta vältyttäisiin piirtämästä tuhansia kaaria. Kirjalle pistetään myös jokaisen luodun kopion koko.
- *Solmu*: Solmu on kopion sijainti kasassa. Solmuun kirjataan kopion uusi osoite.
- *Päätepiste*: Päätepiste piirretään, kun kopio käytetään tai sitä muutetaan jollakin tavalla. Jos kopiota muutetaan, muodostuu siitä uusi objekti, jolle piirretään oma lähtöpisteensä.

```
1  import java.util.ArrayList;
2  import java.util.Random;
3  public class NewMain {
4      public static void main(String[] args) {
5          RandList rand = new RandList();
6          RandList copyMethod = rand.copyList();
7          RandList copyByHand = new RandList(copyMethod.elems, copyMethod.len);
8          System.out.println(copyMethod.findN(1));
9          System.out.println(copyByHand.findN(1));
10     }
11 }
12 class RandList{
13    int[] elems; int len;
14    RandList(){
15        elems = new int[1000]; len = 0;
16        Random rand = new Random();
17        for(int i = len;i < 1000; i++){
18            elems[i] = rand.nextInt(1000);
19        }
20    }
21    RandList(int[] x, int y){
22        elems = x; len = y;
23    }
24    RandList copyList(){
25        return new RandList(elems, len);
26    }
27    String findN(int x){
28        for(int i = 0; i < elems.length; i++){
29            if(x == elems[i]){
30                return Integer.toString(x) + " found at " + Integer.toString(i);
31            }
32        }
33        return "Not found";
34    }
35 }
```

Kuva 2. Esimerkki Java-koodista, jossa esiintyy kopiointia



Kuva 3. Osittainen kopio-graafi kuvan 2 koodista

Kuvassa 2 on kirjoitettuna hyvin yksinkertainen ohjelma, se luo olion, joka tallettaa 1000-kappaletta satunnaisia lukuja tauluun. Sen jälkeen ohjelma kopioi tämän olion sekä tulostaa löytyykö lukua yksi taulukosta. Kuvassa 3 esitetään tästä koodista lyhennetty kopio-graafi. Sekä graafi että koodi ovat hyvin yksinkertaistettuja esimerkin vuoksi. Graafissa näkyvät nimikkeet O5, O6, O7 viittaavat rivinumeroon. Koodissa luodaan uusi olio *rand* rivillä 5. Tämä olio kopioidaan uuteen osoitteeseen eli olioon *copyMethod*, tätä ilmaistaan kaarella ja kaaren yläpuolella näkyy koko, jota ollaan tallettamassa, 1000 kertaa 4 bittiä. Viimeiseksi se viedään päätepisteeseensä, joka tässä tapauksessa on rivin 8 tulostusmetodi. Itse tulostusmetodiin kopioidaan vain yksi muuttuja.

Tämä graafin piirtäminen suoritetaan sekä kääntämisen aikana, muokatulla kääntäjällä, että ajon aikana. Tästä syntynyttä graafia voidaan sitten käyttää tunnistamaan

pitkiä kopio-ketjuja. Tekniikka ei käsittele millaisia ratkaisuja näihin kopio-ketjuihin löytyy, sillä nämä ovat hyvin ohjelmakohtaisia. Kokenut kehittäjä osaa tämän graafin avulla etsiä suurimmat ongelmakohdat ohjelmistosta ja korjata ne haluamallaan tavalla.

Suoritustehollisesti tämä graafin piirtäminen ei tietenkään ole ilmaista. Koska varjo-keko on yhtä suuri kuin alkuperäisen ohjelman keko, vaatii se suorittavalta koneelta tuplatun määrän ylimääräistä muistia. Myös ohjelmiston suoritusaika graafia piirrettäessä moninkertaistuu. Guoqing Xu et al. eivät kuitenkaan näe tätä ongelmalliseksi, sillä heidän arvionsa mukaan suoritusajat olivat vielä hyväksyttävällä tasolla, sekä he kykenivät ajamaan kaikki haluamansa ohjelmistot ongelmitta, muistikuormasta huolimatta.

4.3 Sisälletyt metodit

Kun suurissa ja monimutkaisissa ohjelmistoissa käytetään monia viitekehyksiä sekä useita eri kirjastoja, saattavat yksinkertaisetkin käskyt ohjelmistossa muuttua pitkiksi, itseään toistaviksi metodikutsuiksi. Metodikutsu on ohjelmiston tapa käskeä jotakin ohjelmiston osaa suorittamaan jonkinlainen tehtävä. Metodikutsut ovat yleisin tapa, jolla olio-ohjelmoinnissa suoritetaan tehtäviä. Jos kutsut toistuvat turhaan, voi se aiheuttaa suurta määrää paisumista sekä tehokkuuden menetystä ohjelmistoissa. Maplesden et al. [10] kutsuvat tällaisia pitkiä toistuvia metodikutsuja sisälletyiksi metodeiksi (*subsuming methods*). Työssään he esittelevät tekniikan, jonka avulla kehittäjät voivat tunnistaa tällaiset sisälletyt metodit ja täten vähentää ohjelmiston paisumisen ongelmia.

Aluksi joudumme määrittelemään asiayhteys-kutsu-puun (*calling context tree*) [11], josta käytämme tästä eteenpäin lyhennettä AKP. AKP tarjoaa ohjelmiston metodikutsut helposti ymmärrettävässä, yksinkertaisessa puumallissa, jossa solmu edustaa yhtä metodia jota ollaan kutsuttu. Nämä puut kuitenkin kasvavat suuriksi nopeasti, jopa pienemmissä ohjelmistoissa, joten ihmisen on vaikea näistä tehdä päättelyjä ilman apuvälineitä. Tällaisia apuvälineitä on toki tarjolla, mutta ne eivät tarjoa hyvää tapaa tunnistaa sisällettyjä metodeja. Apuvälineet tarjoavat usein vain tavan tunnistaa niin sanottuja *kuumia* metodeja, jotka ovat metodeja, joita ohjelmistossa käytetään hyvin usein. Tällä tavoin voidaan ohjelmia optimoida, parantamalla näitä kuumia metodeja. Sisälletyt metodit eivät kuitenkaan aina kuulu näihin kuumiin metodeihin ja kuumien metodien etsiminen ei löydä kaikkia sisällettyjä metodeja.

Käyttämällä AKP-puuta voidaan aloittaa sisällettyjen metodien tunnistus. Tunnistus aloitetaan etsimällä metodeja joilla on kaksi tunnusmerkkiä:

- Metodit, joiden *korkeus* on matala. Korkeus lasketaan tämän metodin alipuun korkeudesta eli matkasta, joka joudutaan kulkemaan metodista sen alimpaan solmuun. Yksin korkeus ei ole kiinnostava, mutta kun se liitetään yhteen sitä *hallitsevan* metodin kanssa, saadaan kiinnostavia tuloksia.

- Metodit, joilla on selvä *hallitsija*. Hallitsijametodi on sellainen metodi, joka kutsuu toista tiettyä metodia lähes aina.

Yhdistämällä nämä kaksi tunnusmerkkiä voimme alkaa tunnistamaan sisällettyjä metodeja. Nämä ovat siis metodeja, jotka toimivat usein *hallitsijana* ja joiden *korkeus* on matala. Näin voidaan löytää metodeja, joita kutsutaan usein ja toistuvasti suorittamaan jotakin yksinkertaista tehtävää. Arvot sille, milloin jokin metodi leimataan sisälletyksi metodiksi vaihtelee ja näitä rajoja voidaan muuttaa helposti tutkimisen aikana. Muuttamalla rajoja voidaan etsiä jokin haluttu määrä sisällettyjä metodeja koodista.

Itse metodien optimointi jää kehittäjän vastuulle, sillä ongelmat mitkä johtavat sisällettyihin metodeihin ovat hyvin monipuoliset ja niitä ei yksinkertaisella tekniikalla voi ratkaista. On myöskin huomattava, että kaikki metodit, jotka tämä tekniikka löytää, eivät aina ole ongelmallisia. Metodit saattavat käyttäytyä sisällettyjen metodien tavoin mutta niiden optimointi ei ole mahdollista ja se on vain osa ohjelmiston toimintaperiaatetta. Tästä huolimatta Maplesden et al. kykenivät nostamaan testaamiensa ohjelmistojen tehokkuutta 17 % - 51 % etsimällä sisällettyjä metodeja sekä optimoimalla niitä. Sisällettyjä metodeja ei myöskään tarjota tekniikaksi, jota käytettäisiin yksinänsä. Maplesden et al. suosittelevat tekniikan käyttöä monen muun optimointitekniikan kanssa parhaan tuloksen saavuttamiseksi.

5 Yhteenveto

Tässä työssä löydettiin monia syitä, jotka ajonaikaista paisumista aiheuttavat, millaisia vaikutuksia niillä on ja joitakin ratkaisuja paisumisen välttämiseen sekä korjaamiseen. Paisumisen syyt voidaan johtaa suoraan nykyiseen ohjelmistokehitykseen sekä sen taipumukseen yrittää etsiä yhä tehokkaampia tapoja tuottaa ohjelmistoja yhä nopeammin. Työ ei yritä taivutella nykykehittäjiä ottamaan käyttöön epätehokkaampia kehityksen muotoja. Nykyinen tehokas kehittäminen mahdollistaa ohjelmistojen saatavuuden jokaiselle käyttäjälle huomattavasti nopeammin sekä varmistaa näiden ohjelmistojen saatavuuden. Työ kuitenkin yrittää osoittaa, että tällä nykyisellä kehittämisen suuntauksella on omat ongelmansa, joihin on kiinnitettävä huomiota ja pidettävä huolta, että paisumisen ongelmat eivät käy liian suuriksi. Mooren laki varmistaa, että meillä on aina käytössämme hyvin tehokkaalta tuntuvia laitteita, mutta moni käyttäjä varmasti arvostaisi, jos suuri osa tästä tehosta ei karkaisi käsistämme heti kun uutta teknologiaa saamme käsiimme.

Paisuminen ei ole jokaiselle käyttäjälle ilmeinen ongelma mutta se myös todennäköisesti vaikuttaa kulutuskäyttäytymiseemme. Työssä keskeinen löydös on, että pahimmassa tapauksessa paisumisen poistaminen parantaa ohjelmistojen suoritusaikoja jopa 50 %. Moni käyttäjä varmasti harkitsisi uuden tietokoneen ostamista uudelleen, jos hänelle annettaisiin takaisin tämä kadonnut 50 %.

Ekologisuus sekä ilmaston lämpeneminen ovat olleet tärkeitä keskustelunaiheita jo monia vuosia sekä yhteiskunnassa että tarkemmin tietotekniikan saralla. Jos paisumista vähentämälle voisimme vähentää sekä suoraa energiankulutusta että tehdä nykyisistä tietokoneistamme laitteita joiden käyttöikä pitenisi monella vuodella, vaikuttaisimme näihin suuriin yhteiskunnallisiin kysymyksiin. Kyseessä on siis paljon suurempi ongelma kuin hieman epämukava käytettävyys ohjelmistoissa joita käytämme.

Viiteluettelo

[1] Anh Quach, Rukayat Erinfolami, David Demicco, Aravind Prakash. 2017. A Multi-OS Cross-Layer Study of Bloating in User Programs,Kernel and Managed Execution Environments. *FEAST '17 Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation. Sivut 65-70.*

[2] Gordon E. Moore. 1965. Cramming more components onto integrated circuits. *Electronics, Vuosikerta 38, Numero 8.*

[3] James Larus. 2008. Spending Moore's Dividend. *Microsoft Research Technical Report MSR-TR-2008-69.*

[4] Nick Mitchell, Edith Schonberg, Gary Sevitsky. 2010 Four Trends Leading to Java Runtime Bloat. *IEEE Software, Vuosikerta 27, Numero 1. Sivut 56 - 63*

[5] Suparna Bhattacharya, Mangala Gowri Nanda, K. Gopinath, Manish Gupta. 2011. Reuse, Recycle to De-bloat Software. *ECOOP 2011 – Object-Oriented Programming, Sivut 408-432.*

[6] Guoqing Xu. 2011. Analyzing Large-Scale Object-Oriented Software to Find and Remove Runtime Bloat. *Electronic Thesis or Dissertation. Ohio State University, 2011.*

[7] Guoquing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Gary Sevitsky. 2010. Software Bloat Analysis: Finding, Removing and Preventing Performance Problems Modern Large-Scale Object-Oriented Applications. *Proceedings of the FSE/SDP workshop on Future of software engineering research, Sivut 421-426.*

[8] Aqeel Mahesri, Vibhore Vardhan. 2005. Power Consumption Breakdown on a Modern Laptop. *International Workshop on Power-Aware Computer Systems Sivut. 165-180. Springer, Berlin, Heidelberg.*

[9] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, Gary Sevitsky. 2009. Go with the Flow: Profiling Copies To Find Runtime Bloat. *ACM Sigplan Notices, Vuosikerta 44, Numero 6, Sivut 419-430, ACM.*

[10] Maplesden, David, Ewan Tempero, John Hosking, John C. Grundy. 2015. Subsuming methods: Finding new optimisation opportunities in object-oriented software. *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering. Sivut 175-186. ACM.*

[11] Ammons, Glenn, Thomas Ball, and James R. Larus. 1997. Exploiting hardware performance counters with flow and context sensitive profiling. *Proceedings of the ACM SIGPLAN Conf.on Programming Language Design and Implementation - PLDI '97. Sivut 85–96.*

[12] Bhattacharya, Suparna, Karthick Rajamani, K. Gopinath, Manish Gupta. 2011. The interplay of software bloat, hardware energy proportionality and system bottlenecks. *Proceedings of the 4th Workshop on Power-Aware Computing and Systems., Artikkeli 1, ACM.*