Univerza v Ljubljani
Fakulteta za računalništvo in informatiko
Fakulteta za matematiko in fiziko

Miha Eleršič

Prevajanje grafne vmesne kode za Maxelerjeve podatkovno-pretokovne enote

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in matematika

Mentor: doc. dr. Boštjan Slivnik

Ljubljana 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Opis:

Izdelajte generator kode, ki pretvori grafno predstavitev ščepca v kodo za Maxelerjevo podatkovno vodeno računsko enoto. V ta namen definirajte ustrezno grafno predstavitev, pri tem pa upoštevajte, da naj bo posebej primerna za predstavitev ščepcev, ki so napisani v funkcijskih jezikih, ki so podobni Haskellu. V samem generatorju kode poskusite uporabiti obstoječa Maxelerjeva orodja.

Opis (angleški):

Implement a code generator for transforming graph intermediate code into code suitable for Maxeler dataflow engines. Define a graph intermediate code especially suitable for representing kernels originally written in Haskell-like functional languages. The code generator can rely on the existing Maxeler compilation tools.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Miha Eleršič, z vpisno številko 63120215, sem avtor diplomskega dela z naslovom:

Prevajanje grafne vmesne kode za Maxelerjeve podatkovno-pretokovne enote

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Boštjana Slivnika,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 11. septembra 2015 Podpis avtorja:

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
DFE	dataflow engine	podatkovno-pretokovna enota
IDE	integrated development environment	programska operema za integriran razvoj programske opreme
SLiC API	simple live CPU API	vmesnik za komunikacijo z DFE

Povzetek

V diplomskem delu smo opisali delovanje zadnjega dela prevajalnika iz izvorne kode v izvorno kodo, ki prevaja iz Haskellu podobnega jezika MaxHs v izvorno kodo primerno za izvajanje na Maxelerjevih podatkovno-pretokovnih enotah. Posebno pozornost smo namenili pravilnemu prevajanju povratnih zank v podatkovno-pretokovnih grafih in s tem povezanim časovnim usklajevanjem izvajanja, ki je potrebno zaradi Maxelerjevega statičnega razvrščanja. Opisali smo način programiranja, ki omogoča programerju pisanje programov pospešenih z Maxelerjevimi podatkovno-pretokovnimi enotami brez poznavanja podrobnosti Maxelerjeve arhitekture.

Ključne besede: podatkovno-pretokovno programiranje, Haskell, Maxeler, prevajalnik.

Abstract

In this thesis we describe the back end of a source-to-source compiler for compiling a Haskell-like language MaxHs to source code suitable for Maxeler dataflow engines. We focus on correct compilation of dataflow graphs that include cycles and related scheduling problems, which are a side effect of static scheduling used by MaxCompiler. We describe a way of programming that allows the programmer to write programs accelerated with Maxeler dataflow engines without detailed knowledge of the Maxeler architecture.

Keywords: dataflow programming, Haskell, Maxeler, compiler.

Poglavje 1

Uvod

Zaradi vse počasnejšega večanja zmogljivosti procesorjev je naraslo zanimanje za različne možnosti vzporednega računanja. Poleg večjedrnih procesorjev in grafičnih procesnih enot obstaja še ena možnost vzporednega računanja: podatkovno-pretokovno računanje. Zanimivo ni le zaradi možnosti vzporednega računanja, amapak tudi zaradi načina programiranja, ki je bližje naravnemu načinu reševanja problemov. Običajni programski jeziki so namreč še vedno pod velikim vplivom strojne opreme, na kateri tečejo [1]. Drugačen način programiranja pa je lahko tudi začasna slabost, saj zahteva drugačen način razmišljanja in ponovno učenje programskega jezika.

Podatkovno pretokovno programiranje je princip programiranja, kjer je program predstavljen kot usmerjen graf. Povezave predstavljajo tok podatkov, vozlišča pa operacije [2, 3]. Podatkovno pretokovno programiranje ima nekaj lastnosti funkcijskih jezikov, saj lahko vozlišča opišemo kot čiste funkcije. Čistost funkcije pomeni, da funkcija nima stranskih učinkov in njen rezultat ni odvisen od stanja programa.

Zanimalo nas je, ali bi lahko iz čistega funkcijskega jezika prevajali programe v jezik primeren za poganjanje na podatkovno-pretokovnih aritekturah. Odločili smo se za kombinacijo programskega jezika Haskell [4] in podatkovno-pretokovne enote podjeatja Maxeler Technologies. Haskell je eden glavnih predstavnikov modernih funkcijskih jezikov. Maxeler Technolo-

gies je podjetje, ki razvija in prodaja različne podatkovno-pretokovne rešitve za visoko zmogljivo računanje.

Programiranje Maxelerjevih podatkovno-pretokovnih enot (angl. data-flow engine – DFE) je podobno programiranju grafičnih procesnih enot (angl. Graphics Processing Unit – GPU)[1]. Pri programiranju GPE program razdelimo na nadzorni program in ščepce (angl. kernel). Nadzorni program teče na CPE in skrbi za vhodno izhodne operacije, interakcijo z uporabnikom, pripravo podatkov in vsemi ostalimi zaporednimi deli programa, ki na GPE ne bi prinesli zadostne pohitritve. Ščepci tečejo na GPE in izvajajo tiste dele programa, ki jih je mogoče izvajati vzporedno in se zato na GPE izvedejo hitreje in učinkoviteje kot na CPE. Programiranje Maxelerjevih podatkovno-pretokovnih enot se od programiranja GPE razlikuje v načinu programiranja ščepcev, saj je arhitektura GPE različna od arhitekture DFE. Ščepce za DFE pišemo po principih podatkovno-pretokovnega programiranja, torej sestavimo podatkovno-pretokovni graf, ki ga opišemo v posebni različici programskega jezika Java imenovani MaxJ. Nadzorni program lahko napišemo v jezikih C, Matlab, R ali Python [5, 6].

V diplomskem delu bomo opisali delovanje zadnjega dela (angl. back end) [7] prevajalnika izvorne kode v izvorno kodo (angl. source-to-source compiler), ki prevaja iz Haskellu podobnega jezika MaxHs v izvorno kodo primerno za izvajanje na Maxelerjevih DFE. Podroben opis jezika MaxHs in opis delovanja sprednjega dela (angl. front end) prevajalnika je v diplomskem delu Svena Cerka z naslovom *Prevajanje funkcijskih ščepcev v grafno vmesno kodo za Maxelerjevo arhitekturo* [8]. Prevajalniku smo na koncu dodali še samodejno poganjanje Maxelerjevega prevajalnika. S tem naš prevajalnik omogoča pisanje programov pospešenih z Maxelerjevimi DFE brez učenja novega programskega jezika in poznavanja podrobnosti delovanja DFE.

[1] V nadaljevanju bomo za centralno in grafično procesno enoto uporabljali ustaljeni slovenski kratici CPE in GPE.

Poglavje 2

Podatkovno-pretokovne arhitekture

Podatkovno-pretokovna arhitektura (angl. dataflow architecture) računalnikov je pravo nasprotje tradicionalnim ukazno-pretokovnim arhitekturam (angl. control flow architecture) [9, 10]. Da bi lažje razumeli razliko med njima, si najprej oglejmo, kako delujejo tradicionalni ukazno-pretokovni računalniki, in opišimo tiste njihove slabe lastnosti, ki zahtevajo uporabo drugačne arhitekture za reševanje določenih problemov.

2.1 Ukazno-pretokovne arhitekture

Ukazno-pretokovne arhitekture izvajajo tok ukazov, ki se nahaja v pomnilniku [10]. Informacijo o lokaciji naslednjega ukaza hrani programski števec. Ukazi preberejo podatke iz pomnilnika, naredijo izračun, podatke zapišejo nazaj v pomnilnik in povečajo ali spremenijo vrednost programskega števca tako, da kaže na naslednji ukaz. To nenehno dostopanje do pomnilnika se lahko dobro skrije z uporabo hierarhije predpomnilnikov. Nemogoče pa je skriti dejstvo, da programski števec vedno kaže le na en ukaz, ki se spremeni šele, ko se ta ukaz izvede. Nemogoče je torej hkrati opraviti več izračunov, tudi če so popolnoma neodvisni. Ukazi se izvajajo v določenem vrstnem redu

tudi ko vrstni red ni pomemben. Ta zaporednost je glavni razlog za neprimernost reševanja problemov, ki omogočajo visoko stopnjo vzporednosti, saj moramo ukaze razporediti v določen vrstni red, čeprav je nepomemben [9].

Te probleme lahko rešimo z večjedrnimi procesorji s cevovodi, vzporednimi ukazno-pretokovnimi enotami (Xeon Phi) ali GPE. Vse naštete rešitve so sicer bližje tradicionalnem načinu programiranja kot podatkovno-pretokovna arhitektura, a vseeno zahtevajo nov način razmišljanja in programiranja.

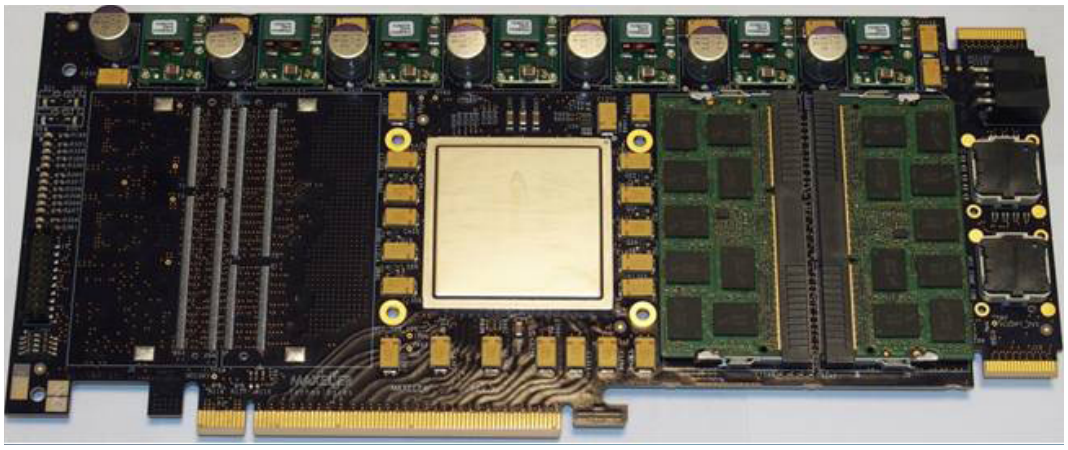
2.2 Podatkovno-pretokovne arhitekture

Podatkovno-pretokovne arhitekture nimajo programskega števca [9]. Kdaj se izvede ukaz je odvisno samo od pripravljenosti njegovih vhodnih podatkov in zaradi tega se lahko hkrati izvaja več ukazov. Programi napisani za podatkovno-pretokovne arhitekture so običajno v obliki usmerjenega grafa. Vozlišča predstavljajo operacije, povezave med vozlišči pa narekujejo, kako so izhodi operacij povezani na vhode drugih.

Podatkovno-pretokovne arhitekture se uporabljajo za specializirano strojno opremo na področju digitalnega procesiranja signalov, usmerjanja mrežnih paketov in grafičnega procesiranja. Izven specializirane strojne opreme se podatkovno-pretokovne arhitekture običajno kombinira s klasičnimi arhitekturami. Glavni program se izvaja na klasičnem računalniku, na podatkovno-pretokovni arhitekturi pa se izvajajo le nekateri deli programa, ki so za to primerni in se na podatkovno-pretokovnih arhitekturah izvedejo učinkoviteje.

2.3 Maxeler

Maxeler Technologies je podjetje, ki ponuja vrsto podatkovno-pretokovnih enot (angl. dataflow engine – DFE) za področje visoko zmogljivega računanja (angl. high-performance computing). Ponujajo razširitvene kartice in večje enote primerne za vgradnjo v strežniške omare. Razširitvene kartice so



Slika 2.1: Podatkovno-pretokovna enota Maxeler Galava, ki je realizirana kot PCI-Express kartica.

združljive z običajnimi delovnimi postajami z operacijskim sistemom Linux (na sliki 2.1). Z njihovo pomočjo snujemo in testiramo nove rešitve problemov. Ko je rešitev primerna za uporabo v industriji, jo prenesemo na večje enote. Vse vrste enot imajo poenoten način programiranja v obliki programske opreme MaxCompiler.

2.4 MaxCompiler

Cilj našega prevajalnika je generirati izvorno kodo, ki jo bo mogoče prevesti z MaxCompilerjem. MaxCompiler je programska oprema za integriran razvoj programske opreme (angl. integrated development environment - IDE), zgrajena na platformi Eclipse. Poleg prevajalnika vključuje še simulator DFE in razhroščevalnik, ki se lahko poveže s simulatorjem ali z DFE enoto. V nadaljevanju si bomo najprej ogledali delovanje MaxCompilerja, podrobneje opisali kaj MaxCompiler pričakuje na svojem vhodu, na koncu pa bomo napisali enostaven program za DFE, ki bo računal drsečo sredino.

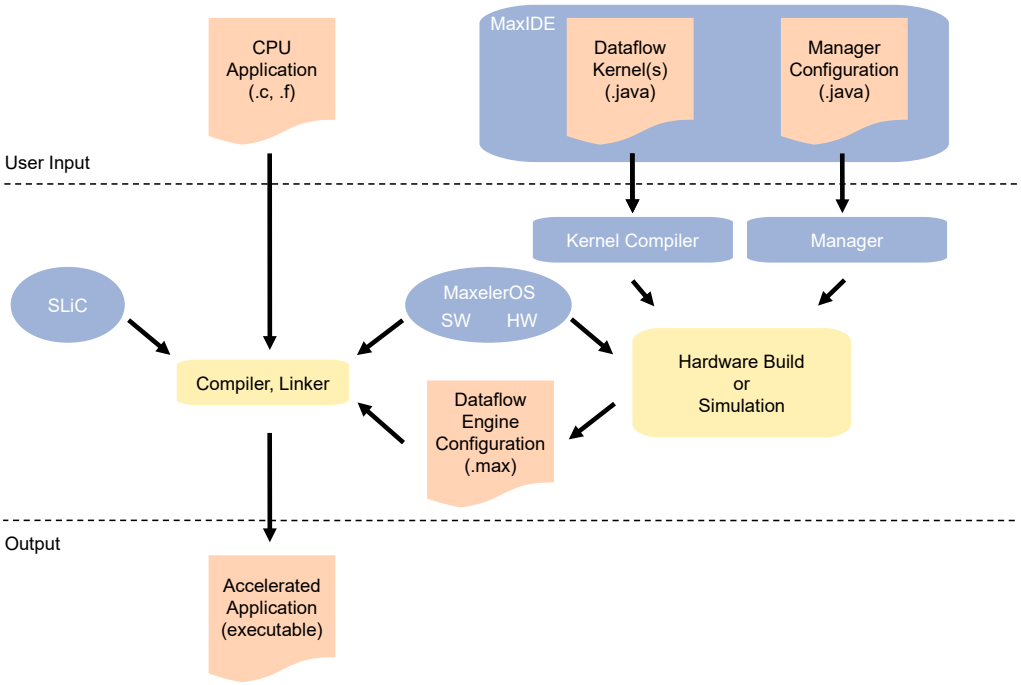
2.4.1 Delovanje MaxCompilerja

Na sliki 2.2 je prikazana shema MaxCompilerja. Na vhodu v MaxCompiler vidimo poleg že omenjene delitve na nadzorni program (na sliki *CPU Application*) in ščepec (na sliki *Dataflow Kernel(s)*) še konfiguracijo nadzornika (na sliki *Manager Configuration*). Podroben opis teh komponent bo sledil pozneje. Najprej si poglejmo, kako poteka prevajanje s prevajalnikom MaxCompiler. Najprej se z raširjenim Java prevajalnikom prevedeta datoteki ščepca in konfiguracije nadzornika. Dobljene datoteke .class se nato požene v običajnem virtualnem stroju Java. Te generirajo konfiguracijsko datoteko (končnica .max) in pripadajočo zaglavno datoteko. Konfiguracijska datoteka je datoteka z izvorno C kodo, ki vsebuje opis grafa, ki je primeren za prenos na DFE in funkcije Maxelerjevega Simple Live CPU (SLiC) APIja. To je API, ki ga uporabljamo v nadzornem programu za dostop do DFE. Zaglavna datoteka je namenjena uvozu v nadzorni program, namesto uvoza konfiguracijske datoteke. V zadnjem koraku MaxCompiler požene zunanji prevajalnik GCC in skupaj prevede in poveže nadzorni program in konfiguracijsko datoteko v izvedljivo datoteko.

2.4.2 Konfiguracija nadzornika

Konfiguracija nadzornika opiše, kako so ščepci povezani s CPE in med sabo. Za enostavne primere z enim ščepcem zadostuje privzeta konfiguracija. Ta samodejno poveže vse tokove ščepca s CPE in doda parameter, ki narekuje čas izvajanja ščepca na DFE. Informacije o tokovih pridobi iz opisa ščepca. Če želimo s konfiguracijo upravljati sami, moramo uporabiti napredno konfiguracijo. Ta sama ne prebere podatkov o tokovih, zato moramo sami povezati tokove na CPE. Prav tako moramo dodati informacijo o času izvajanja ščepca.

Naš prevajalnik bo generiral napredno konfiguracijo nadzornika. Razlog za uporabo napredne in ne privzete konfiguracije bo jasen v petem poglavju, saj bomo spreminjali čas izvajanja ščepca.



Slika 2.2: Shema MaxCompilerja [6].

2.4.3 Nadzorni program

Nadzorni program je napisan v jeziku C. V njem preko SLiC APIja poganjamo ščepce na DFE. Funkcije SLiC APIja skrbijo za pravilno inicializacijo DFE (prenos grafa na DFE), pošiljanje podatkov na DFE in za prejemanje podatkov iz DFE. Primer klica funkcije SLiC APIja bo opisan na primeru kasneje. Namesto pisanja celotnega glavnega programa (uporabniški vmesnik, branje podatkov, itd.) v jeziku C lahko nadzorni program napišemo kot knjižnico, glavni program pa napišemo v poljubnem jeziku. Nadzorni program nato kličemo iz glavnega programa, kadar želimo uporabiti DFE. Tak pritstop je uporabljen tudi v našem prevajalniku, kjer je glavni program napisan v jeziku Haskell.

2.4.4 Ščepec

Ščepec je zapisan v obliki podatkovno-pretokovnega grafa. V tem delu bomu opisali vozlišča, ki jih lahko uprabimo pri grajenju ščepca. Način opisa vozlišč in povezav v jeziku MaxJ pa bo predstavljen na primeru kasneje.

- **vhodna volišča** prejmejo numerično vrednost iz centralne procesne enote – ▽;
- **izhodna volišča** numerično vrednost pošiljajo v centralno procesno enoto – △;
- **računska vozlišča** izvajajo osnovne operacije (seštevanje, odštevanje, množenje, deljenje, primerjave dveh števil, bitne operacije in pretvarjanje tipov) – ◯;
- **odmična vozlišča**, ki zagotovijo, da je izhodna vrednost enaka vhodni vrednosti iz preteklosti ali prihodnosti – ◇;
- **izbirna vozlišča**, ki glede na kotrolni podatkovni tok na izhod preslikajo enega izmed vhodonih tokov – ⏢;

- **vrednostna vozlišča** vedno vračajo enako vrednost, ki je lahko določena s strani CPE pred izvajanjem ali s strani programerja pred prevajanjem –  ;

- **števci** generirajo podatkovni tok, ki se kot števec spreminja vsako urino periodo –  .

Verige števcev

Števce lahko povežemo v verige. Delovanje števcev v verigi lahko ponazorimo na primeru navadne ure. Števca za sekunde in minute sta števca s povečevanjem 1 in končnim številom 59. Števec za ure se tudi povečuje za 1 in ima končno število 23. V verigo jih povežemo tako, da na prvo mesto postavimo urni števec, nato minutni, in nazadnje sekundni. Taka veriga deluje kot navadna ura. Sekundni števec se poveča vsak urin utrip (recimo, da ura teče pri 1Hz), minutni števec se poveča vsakič, ko se sekundni števec ponovno nastavi na 0. Urin števec pa se poveča vsakič, ko se minutni števec ponovno nastavi na 0.

2.4.5 Primer drseče sredine

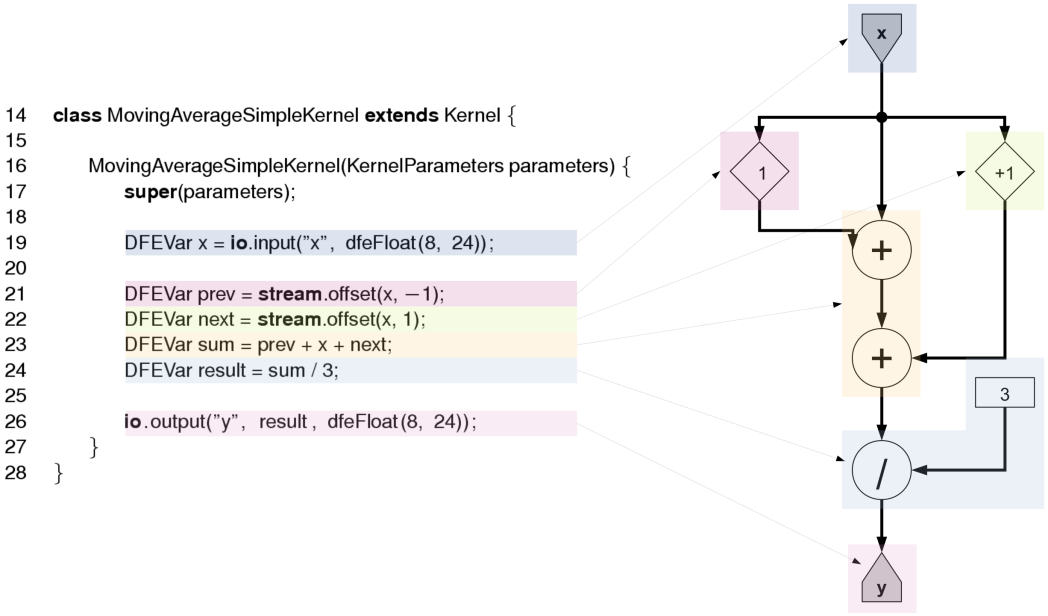
Drseča sredina zaporedja števil je definirana kot zaporedje povprečij treh zaporednih števil v zaporedju. Na robovih zaporedja definiramo drsečo sredino na naslednji način: prvi element zaporedja drseče sredine je tretjina vsote prvih dveh elementov vhodnega zaporedja, zadnji element pa je tretjina vsote predzadnjega elementa in dvakratnika zadnjega elementa vhodnega zaporedja. Razlog za nenavadno definicijo robnih vrednosti drseče sredine je v enostavnejšem zapisu grafa ščepca. V kodnem izseku 2.1 je prikazan ukazno-pretokoven program za izračun drseče sredine zaporedja števil.

Ščepec

Ker podatkovni tok implicitno predstavlja enostavno zanko (lahko si predstavljamo, da CPE koda iterira po vhodnem seznamu in vsak element pošlje v DFE, prejete elemente pa zapiše v nov seznam), moramo v grafu predstaviti le jedro for zanke iz ukazno-pretokovnega programa. Jedro zanke sešteje skupaj prejšnji, trenutni in naslednji element v seznamu in vsoto deli s 3. Za dostop do prejšnjega in naslednjega elementa v toku uporabimo odmični vozlišči. Za izračun vsote potrebujemo dve računski vozlišči z operatorjem seštevanja. Najprej seštejemo prejšnji in trenutni element, nato tej vsoti prištejemo še naslednji element. Z vrednostnim vozliščem ustvarimo konstanto 3 in jo skupaj z rezultatom zadnje vsote pošljemo v računsko vozlišče z operatorjem deljenja. Vhod povežemo na obe odmični vozlišči in na računsko vozlišče, ki sešteva trenutni element. Izhodno vozlišče povežemo na razultat računskega vozlišča z deljenjem. Na sliki 2.3 desno vidimo zgoraj opisani graf predstavljen s simboli.

Obnašanje odmičnih vozlišč na robovih tokov je naslednje: vse vrednosti pred prvim elementom toka so nič, vse naslednje vrednosti po zadnjem elementu toka pa so enake zadnjem elementu toka. Zaradi tega naš graf ravno ustreza ukazno-pretokovnem programu iz izseka 2.1, brez da bi v grafu pazili na robne vrednosti.

Sedaj moramo opisani graf predstaviti v jeziku MaxJ. Na sliki 2.3 levo je prikazana koda, ki generira graf na desni. Z barvami in puščicami so označene povezave med vrsticami kode in vozlišči, ki jih te vrstice kode predstavljajo. Spremenljivke tipa DFEVar predstavljajo vozlišča, definicija spremenljivke na desni strani enačaja pa opiše vrsto vozlišča in njegove vhode. Primer: vrstica 21 na sliki 2.3 pomeni: ustvari novo odmično vozlišče z odmikom -1, na njegov vhod poveži izhod vozlišča z imenom "x". Ime tega novega vozlišča naj bo "prev". Pomen parametrov znotraj definicij vozlišč in definicije ostalih vozlišč, ki jih podpira MaxJ, bodo opisani v četrtem poglavju. Na tem mestu omenimo samo še to, da smo vsako vhodno in izhodno vozlišče poimenovali s poljubnim nizom. Ta imena potrebujemo, da lahko v nadzornem programu



Slika 2.3: MaxJ koda ščepca in podatkovno-pretokovni graf drseče sredine [6].

ločimo med večimi vhodnimi ali izhodnimi vozlišči.

Konfiguracija nadzornika

Uporabimo privzeto konfiguracijo nadzornika. Privzeta konfiguracija nadzornika je prikazana v kodnem izseku 2.2. Kot smo že omenili ta konfiguracija ne vsebuje ničesar specifičnega primeru drseče sredine, saj bi enaka konfiguracija ustrezala poljubnemu enostavnemu ščepcu.

Nadzorni program

Nadzorni program v kodnem izseku 2.3 je napisan tako, da deluje enako kot ukazno-pretokovni program iz izseka 2.1. Funkcija MovingAverageKernel je rezultat prevajanja ščepca in se nahaja v datoteki MovingAverage.max. Vrstni red in opis argumentov preberemo iz zaglavne datoteke enakega imena (v tem primeru MovingAverage.h). Opis argumentov vsebuje tudi imena to-

kov iz ščepca. To nam omogoča, da vemo kateri argument ustreza različnim vhodnim in izhodnim vozliščem ščepca. Klic funkcije MovingAverageKernel sproži kopiranje ščepca na DFE, ga požene in skrbi za pravilen pretok podatkov.

Izsek kode 2.1 MovingAverage.c

```
void MovingAverage(int size, float *in, float *out){
    out[0] = (in[0] + in[1]) / 3;
    for (int i = 1; i < size-1; i++){
        out[i] = (in[i-1] + in[i] + in[i+1]) / 3;
    }
    out[size-1] = (in[size-2] + 2 * in[size-1]) / 3;
}
```

Izsek kode 2.2 MovingAverageManager.MaxJ

```
class MovingAverageManager {
    public static void main(String[] args) {
        EngineParameters params = new EngineParameters(args);
        Manager manager = new Manager(params);
        Kernel kernel = new MovingAverageKernel(manager.
    makeKernelParameters());
        manager.setKernel(kernel);
        manager.setIO(IOType.ALL_CPU);
        manager.createSLiCinterface();
        manager.build();
    }
}
```

Izsek kode 2.3 MovingAverageCPUCode.c

```
#include MovingAverage.h
void MovingAverage(int size, float *in, float *out){
    MovingAverageKernel(size, in, out);
}
```

Poglavje 3

Grafna predstavitev ščepca

Ločiti je potrebno med dvema grafnima predstavitvama ščepca. Eno smo že opisali v razdelku 2.4.4. Ta predstavitev je določena z jezikom MaxJ in je končni rezultat našega prevajalnika. Druga grafna predstavitev ščepca pa je opisana v tem poglavju in v našem prevajalniku služi kot vmesna predstavitev. Zapisana je v podatkovni strukturi MaxGraph v našem prevajalniku. Vozlišča so predstavljana z objekti MaxNode, povezave pa kot kazalci na druge objekte MaxNode. Način generiranja te vmesne predstavitve je opisan v diplomskem delu Svena Cerka *Prevajanje funkcijskih ščepcev v grafno vmesno kodo za Maxelerjevo arhitekturo* [8].

Zadnji del prevajalnika, ki je opisan v tem diplomskem delu, prevaja grafno predstavitev ščepca, ki je opisana v tem poglavju, v grafno predstavitev ščepca, ki je opisana v razdelku 2.4.4. To prevajanje ni tako enostavno, kot se zdi na prvi pogled, zaradi razlik v vozliščih in zank. O zankah bomo govorili v poglavju 5.

3.1 Zgradba

Vozlišča, ki so uporabljena v grafni predstavitvi, so skoraj enaka vozliščem opisanim v razdelku 2.4.4. Uvedli smo naslednje spremembe:

- Pretvarjanje tipov ni več ena od operacij računskega vozlišča ampak je

samostojno vozlišče. Razlog za to je število vhodnih povezav, saj imajo sedaj vse operacije računskega vozlišča dve vhodni povezavi, pretvarjanje tipov pa eno.

- Dodali smo trikomponentno izbirno vozlišče (angl. ternary if). V tem vozlišču je namesto celega števila kontrolni tok logična vrednost, zato pa lahko izbira le med dvema vhodnima tokovoma.

- V grafu vedno nastopajo enostavni števci, povezanost v verige pa je podana posebej. Poleg grafa vmesna predstavitev vsebuje seznam verig, vsaka veriga pa seznam števcev, ki narekuje zgradbo verige.

3.2 Omejitve tipov

Odločili smo se, da numerične tipe omejimo na dva tipa: 32 bitno predznačno celo število (`int32_t`) in 64 bitno število s plavajočo vejico (`double`). Prevajalnik bi bilo mogoče razširiti tako, da bi podpiral vse tipe, ki jih podpira MaxCompiler. Zaradi omejenga časa smo implementirali le omenjena dva tipa.

3.3 Predstavitev grafa v prevajalniku

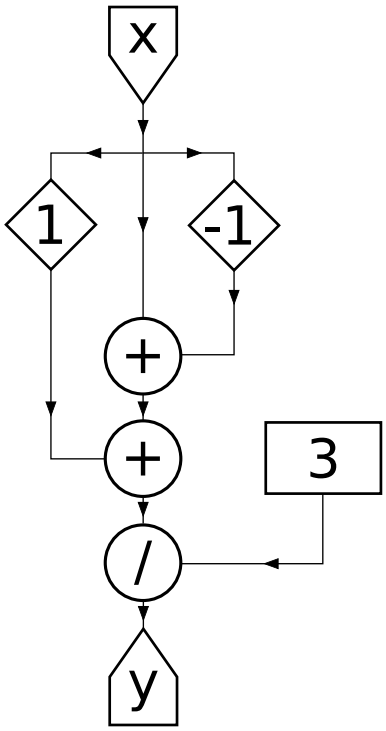
Vozlišča grafa, ki ga sprednji del prevajalnika posreduje zadnjemu delu, so nanizana v linearen seznam. Vrstni red v tem seznamu je poljuben. Vsako vozlišče vsebuje kazalce na vozlišča, ki predstavljajo njegove vhode. Vsako vozlišče vsebuje tudi seznam vozlišč, ki pove, v katerih vozliščih je trenutno vozlišče uporabljeno kot vhod. Tega drugega seznama v trenutni implementaciji prevajalnika ne uporabljamo. Implementiran je zato, ker v času dogovaranja o vmesni kodi nismo vedeli, ali ga bomo potrebovali ali ne.

3.4 Primer

Grafna predstavitev se generira v sprednjem delu prevajalnika. V kodnem izseku 3.1 je vhodni program v jeziku Maxhs. Sprednji del prevajalnika ga prevede v grafno predstavitev na sliki 3.1.

Izsek kode 3.1 MovingAverage.maxhs

```
1  module MovingAverage (movingAverage) where
2  { movingAverage :: Stream Double -> Stream Double
3  ; movingAverage xs =
4      let
5      { ys :: Stream Double
6      ; ys = zipWith add (offset (-1) xs) xs
7      ; zs :: Stream Double
8      ; zs = zipWith add ys (offset 1 xs)
9      ; div3 :: Double -> Double
10     ; div3 x = x / 3.0
11     ; add :: Double -> Double -> Double
12     ; add x y = x + y
13     }
14     in
15       map div3 zs
16 }
```



Slika 3.1: Graf vmesne predstavitve programa MovingAverage.maxhs.

Poglavje 4

Generiranje MaxJ kode

V tem poglavju si ogledamo, kako iz vmesne kode opisane v tretjem poglavju generiramo kodo, ki jo je mogoče prevesti z MaxCompilerjem. Generirati moramo že opisane tri vhode v MaxCompiler: opis ščepca, konfiguracijo nadzornika in CPE kodo. Pri generiranju izhodnih datotek smo si pripravili predloge, ki vsebujejo vse dele kode, ki se ne spreminjajo od programa do programa. To so vnaprej določeni tipi, uvozi knjižnic in statični funkcijski podpisi. Te predloge prevajalnik prebere in na prava mesta vstavi generirano kodo in celoto zapiše na disk. Te predloge so dodane v prilogi na koncu diplomskega dela.

Na koncu vsakega podpoglavja je v kodnem izseku primer kode, ki jo generira naš prevajalnik za vhodni graf prikazan v tretjem poglavju na sliki 3.1.

4.1 Generiranje kode ščepca

V tem poglavju bomo predpostavili, da graf vmesne kode ne vsebuje cikla. Kaj pomeni cikel v podatkovno-pretokovnem grafu in na kakšen način se spremeni generiranje kode, če graf vsebuje cikel, je opisano v petem poglavju.

Najprej vsem vozliščem vmesne kode dodelimo enolično oznako. To bo do konca prevajanja ime spremenljivke v MaxJ za to vozlišče. Iteriramo po line-

ranem seznamu vozlišč iz vmesne kode in vsakemu vozlišču dodamo oznako `LABEL_x`, kjer je `x` naraščajoče število. Za vsako vozlišče deklariramo tudi spremenljivko v MaxJ. Ime spremenljivke v MaxJ ustreza enolični oznaki vozlišča v vmesni kodi. Primer kako deklariramo vozlišča brez njegove definicije v MaxJ je v kodnem izseku 4.1. To je podobno zapisu `double LABEL_1;` v jeziku C.

Izsek kode 4.1 Deklariranje spremenljivk

```
1 DFEVar LABEL_1 = DOUBLEType.newInstance(this);
2 DFEVar LABEL_2 = DOUBLEType.newInstance(this);
3 ...
```

Deklariranje spremenljivk ločeno od definicij nam omogoča gradnjo grafa v poljubnem vrstnem redu. Torej lahko uporabimo linearni seznam iz vmesne kode in nam ni potrebno slediti povezavam znotraj vozlišč. Če spremenljivk ne bi deklarirali na začetku, bi morali paziti, da ima trenutno generirano vozlišče pred sabo že definirana vsa svoja vhodna vozlišča. Pravzaprav je to posledica opisovanja grafa v imperativnem jeziku (Java), saj zahteva, da so vse spremenljivke pred uporabo deklarirane.

Nadaljujemo z grajenjem vseh verig števcev. Za vsak element seznama verig v vmesni kodi generiramo verigo z imenom `CHAIN_x`, kjer je `x` naraščajoče število. Primer generiranja verige navadne ure iz poglavja 2.4.4 je prikazan v kodnem izseku 4.2. Pretvajranje tipov na koncu je potrebno zato, ker metoda `addCounter` vrne vozlišče tipa, ki ima najmanjše možno število bitov za zapis vseh vrednosti, po katerih šteje. Števec ustvarjen v drugi vrstici je torej tipa 6 bitnega celega nepredznačenega števila. Naš prevajalnik pa podpira le en tip celega števila, zato vse števce pretvorimo v tip `int32_t`.

Sledi definiranje spremenljivk, ki smo jih do sedaj le deklarirali. V kodnem izseku 4.3 je prikazan primer definicije seštevalnega vozlišča z oznako `LABEL_1`, ki sešteje vozlišči, ki sta v vmesnem grafu dobili oznako `LABEL_2` in `LABEL_3`. Namesto operatorja `<==` bi lahko uporabili tudi metodo `.connect()`.

Izsek kode 4.2 Primer verige dveh števcev

```
1 CounterChain CHAIN_0 = control.count.makeCounterChain();
2 LABEL_3 = CHAIN_0.addCounter(23,1).cast(dfeInt(32));
3 LABEL_4 = CHAIN_0.addCounter(59,1).cast(dfeInt(32));
4 LABEL_5 = CHAIN_0.addCounter(59,1).cast(dfeInt(32));
```

Izsek kode 4.3 Primer seštevanja

```
1 LABEL_1 <== LABEL_2 + LABEL_3;
2 //ekvivalentno
3 //LABEL_1.connect(LABEL_2 + LABEL_3);
```

Za vsak tip vozlišča grafa vmesne kode definiramo košček kode, ki bo parameter za metodo connect v definiciji tega vozlišča. Izjema je izhodno vozlišče, ki ni povezano nikamor. Njegov košček kode stoji samostojno.
Koščki kode za posamezna vozlišča vmesne kode:

- **vhodna vozlišča:**

```
1 io.input("ime_vhodnega_toka",tip_vhodnega_toka)
```

  `ime_vhodnega_toka` je oznaka toka v MaxHs. Ta oznaka je pomembna za ločevanje med večimi vhodnimi tokovi. `tip_vhodnega_toka` je eden izmed dveh podprtih tipov našega prevajalnika.

- **izodna vozlišča:**

```
1 io.output("ime_izhodnega_toka",LABEL, tip_izhodnega_toka)
```

  `ime_izhodnega_toka` je oznaka toka v MaxHs. Ta oznaka je pomembna za ločevanje med večimi izhodnimi tokovi. `LABEL` je oznaka vozlišča, katerega izhod povežemo na izhodno vozlišče. `tip_izhodnega_toka` je eden izmed dveh podprtih tipov našega prevajalnika.

- **računska vozlišča:**

```
1 LABEL_1 operator LABEL_2;
```

Oznaki `LABEL_1` in `LABEL_2` sta oznaki vhodnih vozlišč, `operator` pa je eden izmed naslednjih operatorjev: `|`, `^`, `&`, `===`, `!==`, `<`, `>`, `<=`, `>=`, `<<`, `>>`, `+`, `-`, `*`, `/`. Primerjava enakosti vsebuje en enačaj več, da se loči od Javine primerjave razredov.

- **vozlišča za pretvajranje tipov:**

```
1 LABEL.cast(tip)
```

Oznaka `LABEL` je oznaka vhodnega vozlišča, `tip` pa je eden izmed dveh podprtih tipov v našem prevajalniku.

- **odmična vozlišča:**

```
1 stream.offset(LABEL, odmik)
```

Oznaka `LABEL` je oznaka vhodnega vozlišča, `odmik` pa je celo število.

- **vrednostna vozlišča:**

```
1 constant.var(tip, vrednost)
```

`tip` je eden izmed dveh podprtih tipov v našem prevajalniku, `vrednost` pa je lahko celo število ali decimalno število ločeno z decimalno piko.

- **izbirna vozlišča:**

```
1 control.mux(LABEL.cast(dfeUInt(X)),LABEL,...);
```

X je najmanjša potenca števila 2, ki je večja ali enaka od števila vozlišč med katerimi izbiramo. Zahteva po točno določenem tipu je iz strani MaxCompilerja. `LABEL,...` pomeni z vejico ločene vse oznake vozlišč med katerimi izbira izbirno vozlišče.

4.1.1 Primer

Primer izhoda generatorja kode ščepca je v kodnem izseku 4.4.

Izsek kode 4.4 Ščepec drseče sredine

```
1  class CpuStreamKernel extends Kernel {
2      CpuStreamKernel (KernelParameters parameters) {
3          super(parameters);
4          final DFEType DOUBLEType = dfeFloat(11, 53);
5          final DFEType INTType = dfeInt(32);
6          DFEVar LABEL_0 = DOUBLEType.newInstance(this);
7          DFEVar LABEL_2 = DOUBLEType.newInstance(this);
8          DFEVar LABEL_3 = DOUBLEType.newInstance(this);
9          DFEVar LABEL_4 = DOUBLEType.newInstance(this);
10         DFEVar LABEL_6 = DOUBLEType.newInstance(this);
11         DFEVar LABEL_7 = DOUBLEType.newInstance(this);
12         DFEVar LABEL_8 = DOUBLEType.newInstance(this);
13         DFEVar LABEL_9 = DOUBLEType.newInstance(this);
14         DFEVar LABEL_10 = DOUBLEType.newInstance(this);
15         LABEL_0 <== io.input("xs" , DOUBLEType);
16         LABEL_2 <== stream.offset(LABEL_0 , -1);
17         LABEL_3 <== LABEL_2 + LABEL_0;
18         LABEL_4 <== LABEL_3;
19         LABEL_6 <== stream.offset(LABEL_0 , 1);
20         LABEL_7 <== LABEL_4 + LABEL_6;
21         LABEL_8 <== LABEL_7;
22         LABEL_9 <== constant.var(DOUBLEType , 3.0);
23         LABEL_10 <== LABEL_8 / LABEL_9;
24         io.output("OUT_LABEL_11" , LABEL_10 , DOUBLEType);
25     }
26 }
```

4.2 Generiranje kode konfiguracije upravnika

Pri generiranju kode konfiguracije upravnika je potrebno našteti vhodne in izhodne tokove, njihove tipe in nastaviti čas izvajnja ščepca. Vse to je opisano v delu konfiguracije imenovanem *EngineInterface*. Najprej ustvarimo nov objekt razreda `EngineInterface`, nato kličemo metode tega objetka, da ga konfiguriramo. Primer izhoda generatorja telesa konfiguracije upravnika je v

kodnem izseku 4.5.

Čas izvajanja ščepca je odvisen od dolžin vhodnih tokov. Naš prevajalnik podpira le tokove enakih dolžin, zato iz nadzornega programa pošljemo velikost vseh tokov kot en parameter. Parametri so podatki, ki se pred izvajanjem ščepca pošljejo v DFE, med izvajanjem pa se ne spreminjajo. V primeru v vrstici 6 ustavrimo nov parameter tipa integer z imenom `SIZE`. V vrstici 7 z metodo `setTicks` nastavimo čas delovanja našega ščepca na prej ustvarjen parameter.

V vrsticah 8 in 9 ustvarimo tokove z metodo `setStream`, katere parametri so ime toka, tip toka in velikost toka v bajtih. Velikost toka v bajtih je enaka produktu dolžine toka in velikosti enega elementa toka v bajtih.

Izsek kode 4.5 Primer telesa konfiguracije nadzornika

```
1  EngineInterface engine_interface = new EngineInterface();
2  CPUTypes INTtype = CPUTypes.INT32;
3  CPUTypes DOUBLEtype = CPUTypes.DOUBLE;
4  int INTsize = INTtype.sizeInBytes();
5  int DOUBLEsize = DOUBLEtype.sizeInBytes();
6  InterfaceParam size = engine_interface.addParam("SIZE", CPUTypes
       .INT);
7  engine_interface.setTicks("CpuStreamKernel", size);
8  engine_interface.setStream("xs", DOUBLEtype, size * DOUBLEsize);
9  engine_interface.setStream("OUT_LABEL_11", DOUBLEtype, size *
       DOUBLEsize);
10 return engine_interface;
```

4.3 Generiranje CPE kode

Ker bo nadzorni program napisan v Haskellu, je v naši CPE kodi le ena funkcija, ki jo bo preko vmesnika tuje funkcije (angl. foreign function interface – FFI) klical glavni program. Funkcija kot parametre sprejme vse vhodne tokove preko kazalcev, velikosti vhodnih tokov, kazalce na izhodne

tokove in kazalce velikosti izhodnih tokov. Funkcija dodeli pomnilnik za izhodne tokove in kliče funkcijo SLiC APIja, ki jo je generiral MaxCompiler. Ta funkcija skrbi za prenos vhodnih tokov na DFE in prenos izhodnih tokov nazaj v pomnilnik. Primer izhoda generatorja CPE kode je v kodnem izseku 4.6.

Ker naš prevajalnik ne podpira tokov različnih dolžin, je vračanje velikosti nazaj klicoči funkciji v trenutni implementaciji prevajalnika odveč, saj so izhodni tokovi vedno enake dolžine kot vhodni. Prav tako morajo biti vsi vhodni tokovi enake dolžine in zaokroženi na 16 bajtov. Zaokroževanje je potrebno zaradi omejitev v pošiljanju podatkov na DFE, kar velja tudi za ročno programiranje z MaxCompilerjem.

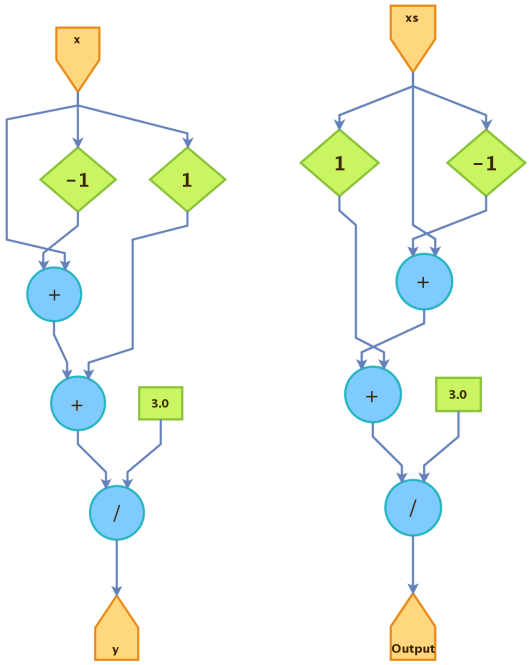
Da podpis funkcije vsebuje dolžine tokov smo se odločili zato, da če v prihodnosti dodamo možnost tokov različne dolžine, ne bo potrebno spreminjati podpisa funkcije.

Izsek kode 4.6 Primer telesa CPE kode

```
1 void kernel(int in0,double *inputstream0, int *out0,double **
    outputstream0){
2 int streamSize = 0;
3 if (in0 > streamSize) streamSize = in0;
4 *outputstream0 = malloc(streamSize * sizeof(double));
5 CpuStream(streamSize,inputstream0,*outputstream0);
6 *out0 = streamSize;
7 }
```

4.4 Primerjava končnega grafa

Poglejmo si, kakšna je razlika med našo generirano kodo in kodo pisano ročno v MaxJ. Primerjali bomo dve rešitvi problema drseče sredine. Ena je priložena dokumentaciji MaxCompilerja, druga pa je napisana v MaxHs in se bo prevedla z našim prevajalnikom. Koda ščepca iz dokumentacije je prikazana na sliki 2.3. Koda ščepca našega prevajalnika pa je v kodnem izseku

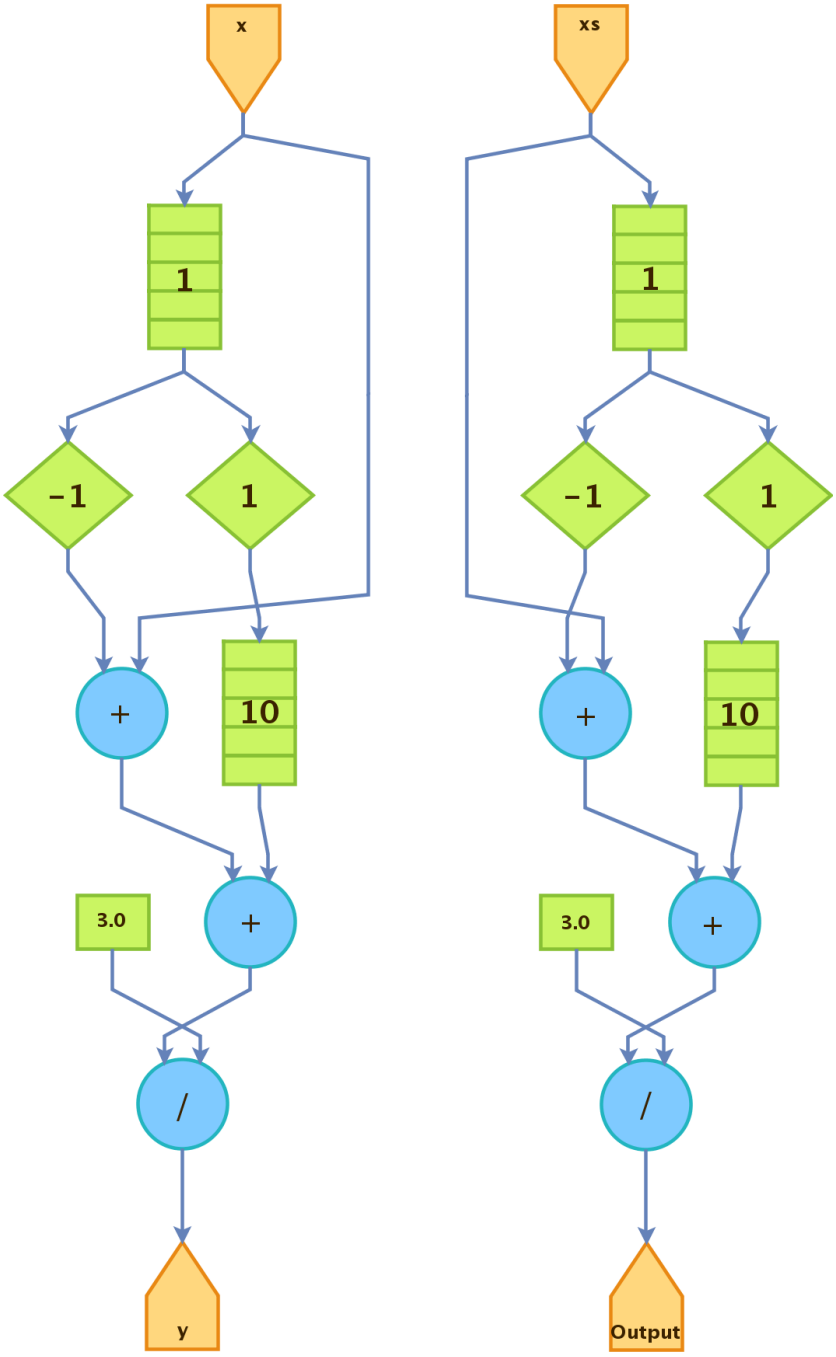


(a) Prevod programa, ki je napisan v MaxJ [6].

(b) Prevod programa, ki je napisan v MaxHs.

Slika 4.1: Primerjava izvirnih grafov programov za reševanja problema drseče sredine.

4.4. MaxCompiler IDE nam omogoča vizualizacijo podatkovno-pretokovnih grafov. Prikaže lahko dva grafa. Prvi je izvirni graf, torej tak kot smo ga napisali. Drugi graf prikazuje končen graf, ki se poganja na DFE. Na sliki 4.1 je primerjava izvirnih grafov, na sliki 4.2 je primerjava končnih grafov. Iz grafov je razvidno, da so kljub zelo različni kodi generirani grafi strukturno enaki. Sklepamo lahko, da MaxCompiler optimizira vhodni graf in tako zakrije razlike med kodo generirano z našim prevajalnikom in kodo pisano ročno.



(a) Prevod programa, ki je napisan v MaxJ [6].

(b) Prevod programa, ki je napisan v MaxHs.

Slika 4.2: Primerjava končnih grafov programov za reševanja problema drseče sredine.

Poglavje 5

Cikli v podatkovno-pretokovnih grafih

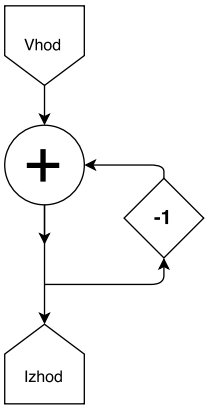
V četrtem poglavju smo predpostavili, da graf vmesne kode ni vseboval cikla. V tem poglavju pregledamo, zakaj so cikli za naš prevajalnik problematični in kako moramo spremeniti generiranje kode, če graf vmesne kode vsebuje cikle.

Poglejmo si program, ki računa tekočo vsoto. Definicija tekoče vsote je zaporedje delnih vsot vhodnega zaporedja. V kodnem izseku 5.1 je program, ki izračuna tekočo vsoto napisan v jeziku C. Zapišimo ta program s

Izsek kode 5.1 RollingSum.c

```
1 void MovingAverage(int size, float *in, float *out){
2     float rollingSum = 0;
3     for (int i = 0; i < size; i++){
4         rollingSum = rollingSum + in[i];
5         out[i] = rollingSum;
6     }
7 }
```

podatkovno-pretokovnim grafom. Zopet nam podatkovni tok predstavlja implicitno zanko. Tabela `in[]` ravno ustreza vhodnemu toku, tabela `out[]` pa izhodnemu. Jedro zanke računa `rollingSum = rollingSum + in[i]`. Po-



Slika 5.1: Graf tekoče vsote.

trebovali bomo torej seštevalno vozlišče, katerega eden od vhodov je rezultat vhodnega vozlišča, drugi vhod pa mora vsebovati prejšnji rezultat tega istega seštevalnega vozlišča. Uporabimo torej zamično vozlišče z zamikom -1 (negativna vrednost pomeni preteklost), na njegov vhod povežemo izhod seštevalnega vozlišča, njegov izhod pa povežemo na drugi vhod seštevalnega vozlišča. Tako opisan graf je na sliki 5.1. Opis tega grafa v jeziku MaxHs je v izseku kode 5.2. Če bi z našim prevajalnikom poskusili prevesti program iz kodnega izseka 5.2, bi vmesna koda bila enaka, kot na sliki 5.1. Tudi pri generiranju kode ne bi bilo težav, MaxCompiler pa bi javil napako:

```
ERROR: Found illegal loop with a latency of 12 (>0)!
```

Če obnašanje grafa simuliramo na papir, ugotovimo da deluje pravilno. Po definicij podatkovno-pretokovnih enot bi torej ta graf moral delovati.

Izkaže se, da Maxelerejevi DFE pravzaprav niso prave podatkovno-pretokovne enote. Njihova vozlišča namreč ne čakajo na pripravljenost svojih vhodov, ampak vsako urino periodo obravnavajo svoje vhode. Podatkovno-pretokovno programiranje na Maxelerjevih DFE je iluzija, za katero poskrbi statično razvrščanje (angl. static scheduling) [11]. Statično razvrščanje bomo

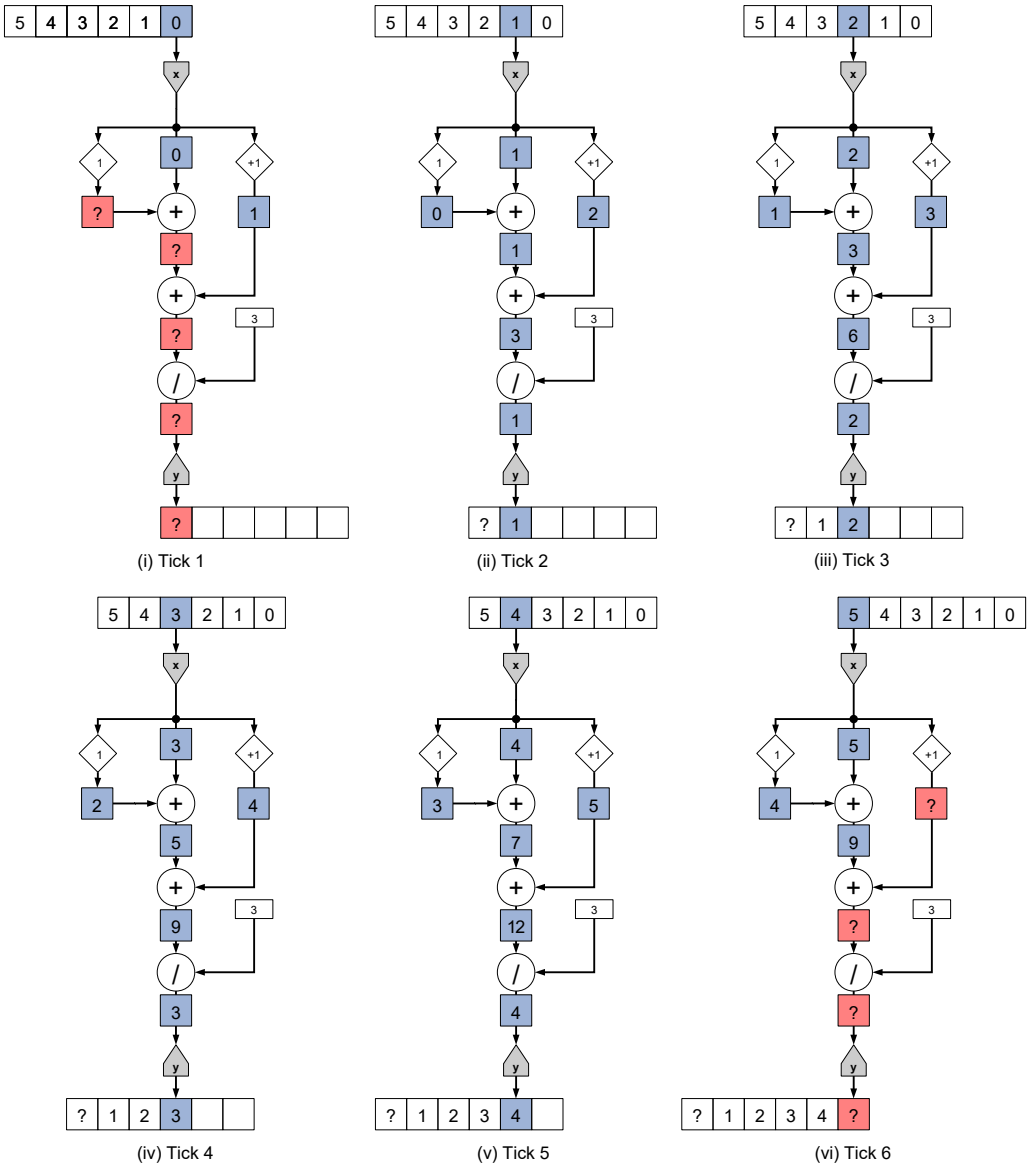
podrobneje opisali v razdelku 5.2, pred tem pa bomo razložili še en pomemben pojem: cevovod.

5.1 Cevovod

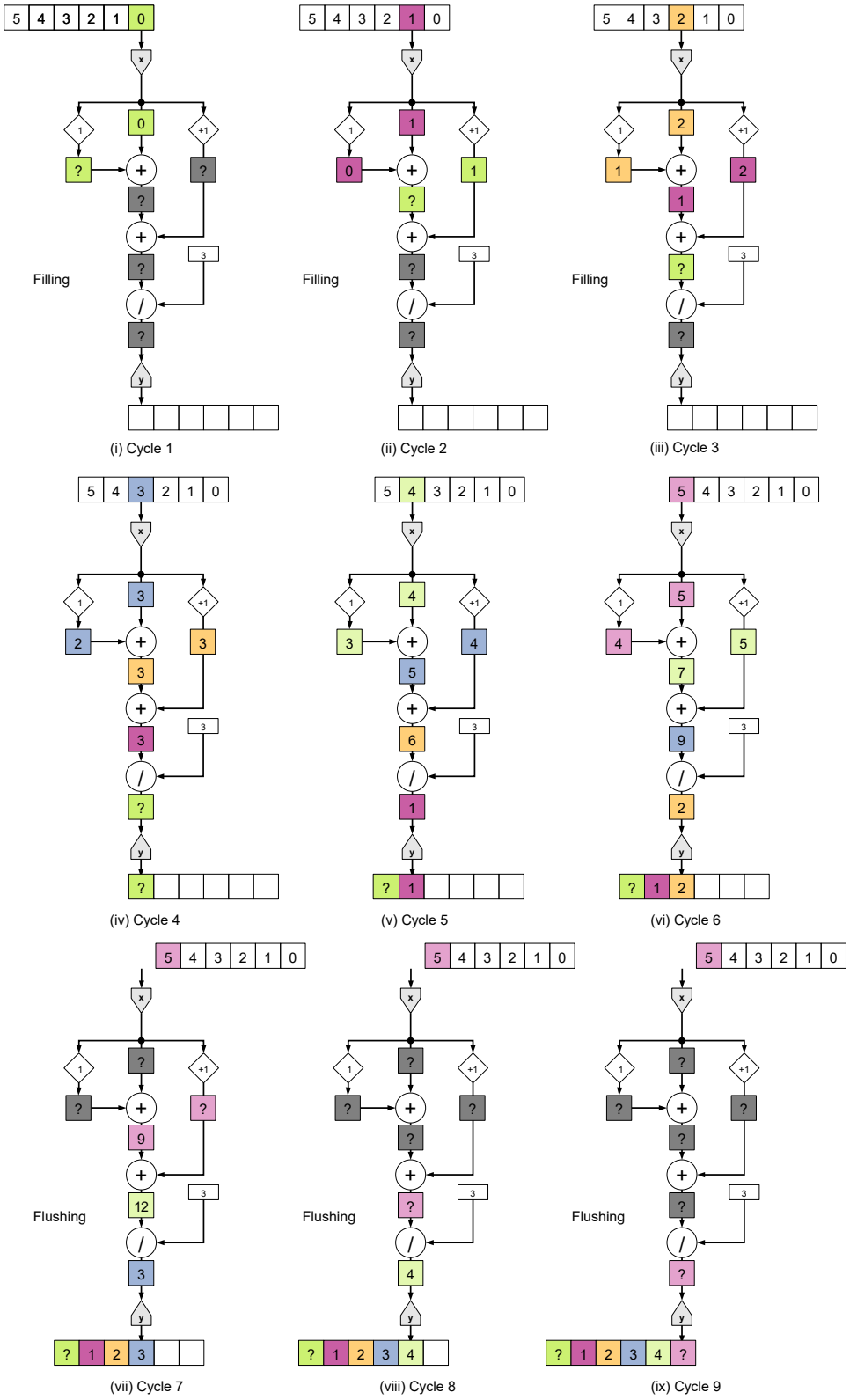
Idejo cevovoda najlažje ponazorimo s tovarno s tekočim trakom. Recimo, da izdelek izdelujemo v treh stopnjah A,B in C. Vsak izdelek mora skozi vse tri stopnje, v tem enakem vrstnem redu. Medtem ko je izdelek na eni stopnji, ne more biti hkrati v nobeni drugi. Če tovarna obratuje brez tekočega traku, torej dela le na enem izdelku naenkrat, potem konča en izdelek v času vsote trajanja vseh stopenj. Z uporabo tekočega traku je postopek naslednji: po končani stopnji A začnemo delati na drugem izdelku v stopnji A, hkrati ko je prvi izdelek v stopnji B. Nato premaknemo prvi izdelek v stopnjo C, drugega v B in začnemo tretjega v A. Čas za izdelavo prvega izdelka je še vedno vsota vseh stopenj, vendar sedaj vsak naslednji izdelek po prvem pride v času maksimuma vseh treh stopenj. Če stopnje trajajo približno enako časa, bi šlo za 3 kratno povečanje prepustnosti tovarne.

Poglejmo si, kako uporaba cevovoda vpliva na podatkovno-pretokovne grafe. Na sliki 5.2 je prikazan programerjev pogled na izvajanje ščepca. Programerjev pogled pomeni, da na graf gledamo, kot da se izvaja na pravi podatkovno-pretokovni enoti, torej vozlišča se med seboj čakajo. Pri takem pogledu je smiselna definicija ene časovne enote naslednja: en korak programerjevega pogleda je enakovreden vstopu enega novega podatka na vsa vhodna vozlišča, nato čakanje, da se vsa vozlišča "umirijo". To pomeni, da dokler na vhod ne pride nov podatek, nobeno vozlišče ne bo imelo pripravljenih vseh svojih vhodov – vsa vozlišča čakajo na svoje vhode. Na sliki 5.2 v enem koraku svoje računanje opravijo vsa računska vozlišča v grafu. Če predpostavimo, da vse računske operacije trajajo enako dolgo, je vsako računsko vozlišče izkoriščeno le $\frac{1}{3}$ časa, ostali $\frac{2}{3}$ časa pa čaka na veljavne podatke na vhodu. To ustreza tovarni brez tekočega traku.

Na sliki 5.3 je prikazan pogled na izvajanje ščepca, ki deluje kot cevovod.



Slika 5.2: Programerjev pogled na izvajanje ščepca [6].



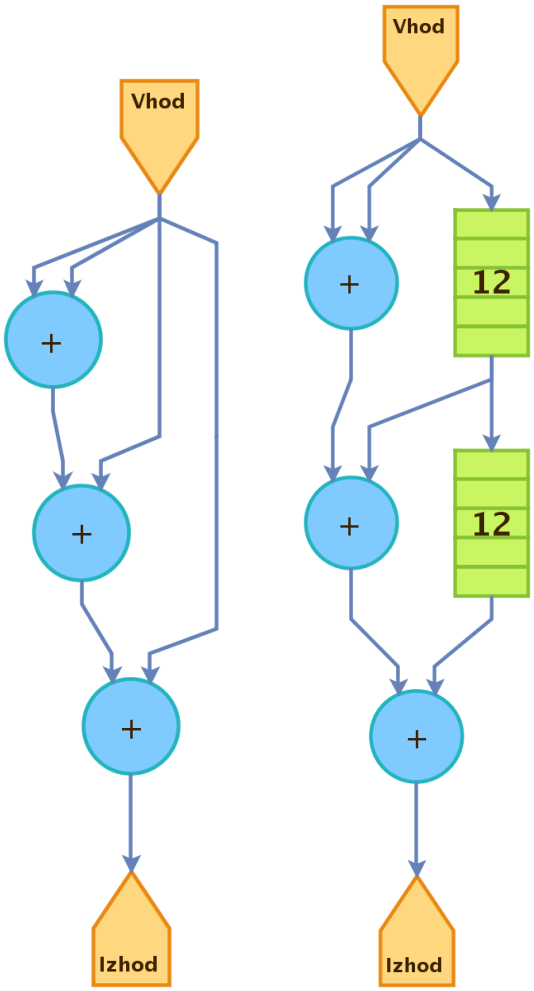
Slika 5.3: Cevovodni pogled na izvajanje ščepca [6].

En korak je sedaj krajši. V enem koraku je sedaj dovolj časa za izvedbo le ene računske operacije. Podobno kot tovarna s tekočim trakom, ščepec vsak korak iz vhoda prejme nov podatek. Tako je v ščepcu hkrati več vhodnih podatkov, enako kot je v tovarni na tekočem traku več izdlekov. Računska vozlišča tako računajo na vsakem koraku, prepustnost ščepca pa je večja. Cevovod je torej pomeben za doseganje večjih hitrosti izvajanja ščepcev.

5.2 Statično razvrščanje

Ker MaxCompiler pozna dolžino trajanja posameznih operacij, lahko z vstavljanjem zamikov zakasni tok podatkov tako, da na vhode vozlišč prihajajo pravi podatki ob pravem času. Na sliki 5.4 je primerjava med grafom opisanem v MaxJ in grafom, ki se izvaja na DFE. Grafa smo pridobili iz razhroščevalnika, vključenega v MaxCompiler IDE. Vhodni program vhodu trikrat prišteje vhod (enakovredno bi bilo množenje s 4). Graf na desni predstavlja razporeditev enot na DFE. Opazimo nov tip vozlišča, ki je predstavljen s simbolom  . Gre za čakalno vrsto FIFO (First In First Out). To je ekvivalentno zamičnem vozlišču, ki lahko prejme le negativne vrednosti. Čakalna vrsta je potrebna zato, ker želimo iz vhoda prebrati novo vrednost vsako urino periodo (cevovod), seštevanje pa traja 12 urinih period [11]. Če bi levi graf pognali na DFE, bi v 12 urini periodi, ko prvo seštevalno vozlišče prvič vrne pravilen rezultat (dvakratnik prvega vhoda), na drugem seštevalnem vozlišču imeli na vhodu vsoto iz prvega seštevalnega vozlišča in dvanajsti element vhodnega toka. To je narobe, saj bi na tem mestu morali prišteti prvi in ne dvanajsti element vhoda. V grafu na desni pa je drugi vhod drugega seštevalnega vozlišča zakasnjen. V dvanjstem ciklu bo tam pravilno prispel prvi element vhodnega toka. Podobno velja tudi za tretje seštevalno vozlišče.

Pokazali smo, da MaxCompiler uporablja statično razvrščanje [11] in razložili, zakaj je potrebno. V druge podrobnosti statičnega razvrščanja se v tem diplomskem delu ne bomo spuščali, saj je to preobširna tema, verjetno pa je



(a) Graf opisan v MaxJ.

(b) Končni graf, ki se izvaja na DFE.

Slika 5.4: Primerjava grafa opisanega v MaxJ in grafom, ki se izvaja na DFE.

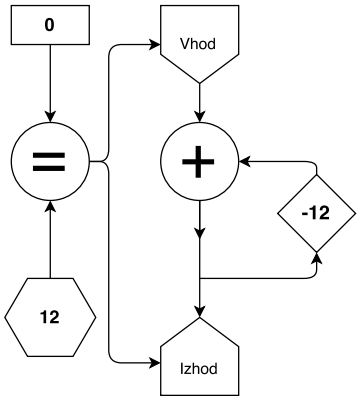
točno delovanje statičnega razvrščanja v MaxCompilerju poslovna skrivnost.

5.3 Rešitev problema tekoče vsote

Vrnimo se na primer tekoče vsote. MaxCompiler je javil napako, saj z vstaljanjem zakasnitev nikakor ni mogel pravilno pripeljati izhoda seštevalnika k pripadajočemu elementu vhodnega toka. Vhodno vozlišče prebere nov podatek vsako urino periodo, seštevanje pa vrne rezultate z zamikom 12 ciklov. Problem rešimo tako, da spremenimo vhodno vozlišče tako, da ne bere naslednjega podatka vsako urino periodo. Želimo si, da med vsakim branjem vhodnega toka preteče dovolj časa, da seštevalno vozlišče izračuna novo delno vsoto. Na ta način je ob naslednjem branju podatka prejšnja delna vsota že izračunana in jo lahko uporabimo za izračun naslednje.

Vhodna in izhodna vozlišča omogočajo na svoj vhod prejeti tok logičnih vrednosti. Za urine periode, ko je vhodna logična vrednost true, se obnašajo kot navadna vhodna/izhodna vozlišča. Za vrednosti false pa vhodno vozlišče ponovi prejšnjo vrednost vhodnega toka in vhodnega toka ne premakne naprej. Izhodno vozlišče za vrednosti false na izhodni tok ne zapiše ničesar, torej se izhodni tok ne premakne naprej.

S pomočjo števca in primerjalnega vozlišča ustvarimo tok logičnih vrednosti, ki ima vsakih 12 urinih period enkrat vrednost true, v ostalih primerih pa false. Ta tok povežemo na vhodno in izhodno vozlišče, zamično vozlišče pa popravimo iz -1 na -12. Razlog za zadnji popravek je, da je naš vhodni program še vedno napisan kot pravi podatkovno-pretokovni graf, torej moramo o njegovem delovanju razmišljati s programerjevim pogledom. Brez te spremembe (torej z zamikom -1) dobimo napačen rezultat, saj vhodno vozlišče vsak korak na svoj izhod vrne veljavno vrednost (ponavljajoča vrednost iz vhodnega toka), seštevalno vozlišče bi seštelo in pokvarilo vrednost v odmičnem vozlišču. Če odmično vozlišče spremenimo na -12 vsa ta odvečna seštevanja ravno pripeljejo željeno vrednost prejšnje tekoče vsote skozi odmično vozlišče. Zgoraj opisani graf je na sliki 5.5.



Slika 5.5: Upoščasnjen graf tekoče vsote.

To rešitev MaxCompiler uspešno prevede, rešitev tudi deluje pravilno. Za pravilen zagon je potrebno le spremeniti čas izvajanja ščepca, saj je ta zdaj 12 krat daljši.

MaxCompiler nam obljublja, da če graf ščepca pravilno deluje v podatkovno-pretokovnem smislu in prevajanje z MaxCompilerjem uspe (torej je razvrščanje uspelo), bo tudi končni graf na DFE deloval pravilno. Poskrbeti moramo, da graf, ne da ga pokvarimo, dovolj upočasnimo, zato da lahko MaxCompiler uspešno opravi statično razvrščanje. Če torej dovolj upočasnimo vhodna in izhodna vozlišča, ostala vozlišča, ki so odvisna od števila korakov, pa upočasnimo za enak faktor kot vhodna in izhodna vozlišča, dobimo graf z enakim pomenom, ki ga je mogoče prevesti z MaxCompilerjem.

Ostal nam je torej le še problem, kako ugotovimo za kakšen faktor moramo graf upočasniti. Želimo si namreč čim manjši faktor, saj se za njegovo vrednost upočasni izvajanje ščepca. Da pa razvrščanje uspe, mora biti faktor večji od dolžine cevovoda znotraj zanke. Ugibanje dolžine cevovoda bi bilo težavno že za programerja, ki piše programe ročno. Mi pa pišemo prevajalnik in bi morali dolžino cevovoda oceniti z algoritmom. Na srečo ima MaxCompiler vgrajeno rešitev tega problema. Imenuje se AutoLoop Offset.

5.4 Maxelerjev AutoLoop Offset

AutoLoopOffset nam omogoča, da prepustimo odločitev o faktorju upočasnitve grafa MaxCompilerju. Na vseh mestih v ščepcu, kjer bi nastopal faktor upočasnitve, uporabimo poseben objekt tipa `OffsetExpr`, ustvarjen z metodo `stream.makeOffsetAutoLoop`. Ob času prevajanja MaxCompiler izbere najmanjše celo število tako, da se graf uspešno prevede, to število nato uporabi na vseh pojavitvah omenjenega objekta. S tem je problem ugibanja faktorja upočasnitve rešen. Poglejmo si še, kako to splošno rešitev problema s cikli realiziramo v našem prevajalniku.

5.5 Spremebe v generiranju kode

Ker nismo želeli po nepotrebnem spreminjati generiranja kode za enostavne grafe, smo generiranje kode ločili v dva primera, glede na to ali graf vmesne kode vsebuje cikel ali ne. Za zaznavanje cikla smo uporabili Tarjanov algoritem krepko povezanih komponent. Za pridobitev informacije, ali graf vsebuje cikel, bi zadostaval tudi preprostejši algoritem topološko urejanje. Tarjanov algoritem smo uporabili iz preprostega razloga, da smo ga že implementirali. Ko smo začeli pisati prevajalnik, smo namreč mislili, da bomo potrebovali tudi informacijo o tem, katera vozlišča so del cikla.

Tarjanov algoritem strogo povezanih komponent vrne seznam strogo povezanih komponent v grafu. Strogo povezana komponenta v grafu pomeni množico vozlišč, kjer je mogoče iz vsakega vozlišča po neki poti znotraj komponente priti v katero koli drugo vozlišče v komponenti. To seveda pomeni, da vsebuje cikel. Če Tarjanov algoritem vrne kakšno storgo povezano komponento, ki vsebuje več kot eno vozlišče, smo našli cikel. Generiranje kode sedaj poteka po drugačnem postopku, opisanem v naslednjih treh razdelkih.

5.5.1 Spremembe v generiranju kode ščepca

Z uporabo AutoLoop Offseta upočasnimo graf vmesne kode, da ga bo MaxCompiler lahko prevedel. Upočasniti je potrebno naslednja vozlišča:

- **Vhodna in izhodna vozlišča** V ščepec dodamo nov števec, ki šteje do vrednosti AutoLoop Offseta. Izhod tega števca nato primerjamo s konstano 0 in tako ustvarimo tok logičnih vrednosti, ki ga povežemo na vsa vhodna in izhodna vozlišča. V izseku 5.3 je prikazano, kako to storimo v jeziku MaxJ.

Izsek kode 5.3 Upočasnjenevanje vhodnih in izhodnih vozlišč

```
1 OffsetExpr loopLength = stream.makeOffsetAutoLoop("loopLength");
2 DFEVar loopLengthVal = loopLength.getDFEVar(this, dfeUInt(32));
3 DFEVar loopCounter = control.count.simpleCounter(32,
    loopLengthVal);
4 LABEL <== io.input("ime_vhodnega_toka",tip_vhodnega_toka,
    loopCounter === 0);
5 ...
6 io.output("ime_izhodnega_toka",LABEL,tip_izhodnega_toka,
    loopCounter === 0);
```

- **Števci** Tudi števce je potrebno upočasniti za faktor Autoloop offset. To enostavno storimo tako, da verigam na koncu dodamo še en števec, ki šteje do Autoloop Offseta. Iz enostavnih števcev, ki niso del kakšne verige, naredimo verige dolžine ena in jim na enak način dodamo še en števec.

- **Odmična vozlišča** Odmičnim vozliščem vrednost odmika pomnožimo z vrednostjo AutoLoop Offseta.

Vse te spremebe so lokalne, torej spremenimo le generiranje argumentov funkcije `connect`. Vse ostalo je enako generiranju opisanem v četrtem poglavju.

5.5.2 Spremembe v generiranju konfiguracije nadzornika

Pri upravniku ščepca je treba paziti le, da podaljšamo čas izvajanja ščepca. Število urinih period, potrebnih za izvršitev ščepca na vseh vhodnih podatkih, je sedaj k-krat večja, kjer je k število, ki ga vrne Autoloop Offset. V kodnem izseku 5.4 je prikazano kako v konfiguraciji nadzornika prebermo vrednost AutoLoop Offseta (vrstica 1) in nato nastavimo koliko urinih period teče ščepec (vrstica 3).

5.6 Primer generirane kode z zanko

S temi spremembami je podpora ciklov v našem prevajalniku končana. V kodnem izseku 5.5 je koda, ki jo generira naš spremenjen prevajalnik za primer tekoče vsote.

Izsek kode 5.2 RollingSum.maxhs

```
module RollingSum (rollingSum) where
{ rollingSum :: Stream Double -> Stream Double
; rollingSum xs = ys
    where
      { ys :: Stream Double
      ; ys = zipWith plus xs (offset (-1) ys)
      ; plus :: Double -> Double -> Double
      ; plus x y = x + y }}
```

Izsek kode 5.4 Primer uporabe metode setTicks

```
InterfaceParam loopLength = engine_interface.getAutoLoopOffset("
    CpuStreamKernel","loopLength");
engine_interface.ignoreAutoLoopOffset("CpuStreamKernel", "
    loopLength");
engine_interface.setTicks("CpuStreamKernel", size*loopLength);
```

Izsek kode 5.5 Ščepec komulativne vsote v MaxJ

```
final DFEType DOUBLEType = dfeFloat(11, 53);
final DFEType INTType = dfeInt(32);
OffsetExpr loopLength = stream.makeOffsetAutoLoop("loopLength");
DFEVar loopLengthVal = loopLength.getDFEVar(this, dfeUInt(32));
DFEVar loopCounter = control.count.simpleCounter(32,
    loopLengthVal);
DFEVar LABEL_0 = DOUBLEType.newInstance(this);
DFEVar LABEL_2 = DOUBLEType.newInstance(this);
DFEVar LABEL_3 = DOUBLEType.newInstance(this);
DFEVar LABEL_4 = DOUBLEType.newInstance(this);
LABEL_0 <== io.input("xs" , DOUBLEType, loopCounter === 0 );
LABEL_2 <== stream.offset(LABEL_4 , -1*loopLength);
LABEL_3 <== LABEL_0 + LABEL_2;
LABEL_4 <== LABEL_3;
io.output("OUT_LABEL_5" , LABEL_4 , DOUBLEType, loopCounter ===
    0 );
```

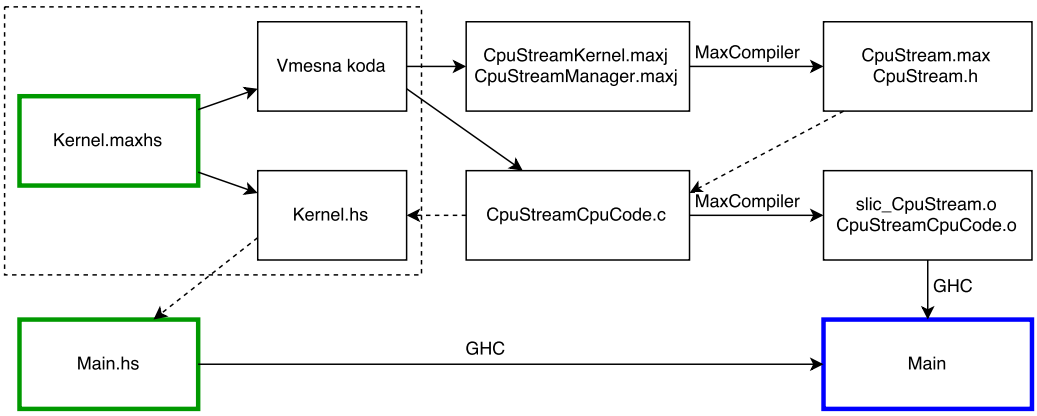
Poglavje 6

Združevanje v celoto

Na sliki 6.1 je prikazana zgradba našega prevajalnika. Zelena obroba predstavlja vhod v naš prevajalnik, modra obroba predstavlja izhod. V črtkanem okvirju je označen del prevajalnika, ki je opisan v diplomskem delu Svena Cerka [8]. Vsebuje še eno datoteko, o kateri še nismo govorili. `Kernel.hs` je Haskell modul, ki se vključi v glavni program. Naloga tega modula je, da preko FFI pokliče funkcijo opisano v CPE kodi. Prav tako skrbi za vračanje rezultatov nazaj v glavni program. Funkcija opisana v tem modulu je navadna Haskell funkcija. Programer, ki napiše glavni program v Haskellu, ščepec pa v jeziku MaxHs, se tako nikoli ne sreča s podrobnostmi, ki so pod pokrovom (npr. FFI).

V prejšnjih poglavjih smo končali z generiranjem kode ščepca (na sliki `CpuStreamKernel.maxj`), konfiguracije nadzornika (na sliki `CpuStreamManager.maxj`) in CPE kode (na sliki `CpuStreamCpuCode.c`). Sedaj moramo pognati MaxCompiler.

MaxCompiler za povezovanje različnih delov prevajanja uporablja orodje Make. Make je orodje, ki omogoča samodejno prevajanje in povezovanje večje programske opreme. Izkaže se, da projektni imenik, ustvarjen z MaxCompiler IDE, vsebuje vse potrebne datoteke makefile za zagon MaxCompilerja iz ukazne vrstice. Projektni imenik dodamo v naš prevajalnik k drugim predlogam. Na začetku prevajanja pobrišemo vsebino `/tmp/maxhs` in tja prenesemo



Slika 6.1: Shema prevajanlnika MaxHs

projektni imenik. Nato generiramo datoteke ščepca, konfiguracije nadzornika in CPE kode in jih shranimo na pravilno mesto v projektnem imeniku, ki se zdaj nahaja v `/tmp/maxhs`. Nato poženemo Make, ki generira objektne datoteke, ki so pripravljene na povezovanje z nadzornim programom. Na koncu poženemo še GHC (Glasgow Haskell Compiler), ki prevede Main.hs in poveže skupaj poleg prej omenjenih objektnih datotek še potrebne Maxelerjeve knjižnice. Rezultat je izvedljiva datoteka, ki se obnaša kot je opisano v Main.hs, klic funkcije iz modula Kernel.hs pa sproži poganjanje ščepca na simulatorju.

Ker smo imeli med izdelavo prevajalnika na voljo le simulator, ne pa tudi dostopa do DFE kartice, nismo mogli preizkusiti prevajanja za pravi DFE. V teoriji je potrebno le spremeniti vrednost izvožene vrednosti `RUNRULE` iz `Simulation` na `DFE`, za ostalo poskrbi MaxCompiler.

V kodnem izseku 6.1 je prikazana skripta, ki požene make in GHC. Skripto iz našega prevajalnika kličemo z `Runtime.getRuntime().exec(...)`. V skripti preusmerimo izhod za napake na navadni izhod z `2>&1`, da si v našem prevajalniku poenostavimo zajemanja izpisa. Skripta namreč teče v drugem procesu in ne pošilja svojih izpisov v terminal, v katerem teče naš prevajal-

nik. Standardni izhod tega procesa zajamemo v String in ga izpišemo na standardni izhod prevajalnika. Ker je izhod za napake preusmerjen nam ni potrebno iz procesa zajemati dveh izhodov, hkrati pa se tudi ne pokvari vrstni red izpisov, kadar sta hkrati uporabljena standradni izhod in izhod za napake.

Izsek kode 6.1 Skripta, ki požene MaxCompiler in GHC

```
#!/bin/bash
export RUNRULE=Simulation
cd /tmp/maxhs/CPUCode
echo "Running MaxCompiler"
make 2>&1
if [ $? -eq 0 ]
then
  echo "MaxCompilation succesfull"
  cd -
else
  echo "MaxCompilation failed."
  exit 1
fi
echo "Running GHC"
ghc -i'/tmp/maxhs/haskell' -L'/opt/maxeler/maxcompiler/lib' -
    lslic -L'/opt/maxeler/maxcompiler/lib/maxeleros-sim/lib' -
    lmaxeleros -lpthread -lm -lrt  Main.hs /tmp/maxhs/RunRules/
    Simulation/objects/maxfiles/slic_CpuStream.o /tmp/maxhs/
    RunRules/Simulation/objects/c/CpuStreamCpuCode.o 2>&1
```

Poglavje 7

Zaključek

V diplomskem delu smo opisali delovanje zadnjega dela prevajalnika, ki prevaja iz Haskellu podobnega jezika MaxHs v jezik MaxJ, ki omogoča opis podatkovno-pretokovnih grafov na precej nižjem nivoju kot MaxHs, a zanj obstaja prevajalnik MaxCompiler podjetja Maxeler Technologies.

MaxHs je programski jezik, ki je posebej prilagojen za opisovanje podatkovno-pretokovnih grafov na funkcijski način. Vse o jeziku MaxHs in generiranju vmesne kode je opisano v diplomskem delu Sven Cerka z naslovom *Prevajanje funkcijskih ščepcev v grafno vmesno kodo za Maxelerjevo arhitekturo* [8].

Na primeru smo pokazali ročni način programiranja za Maxelerjeve podatkovno-pretokovne enote. Na podlagi vozlišč, ki so na voljo v Maxelerjevem jeziku MaxJ smo definirali vmesno kodo prevajalnika. Izdelali smo generator kode, ki pretvori grafno predstavitev ščepca v kodo za Maxelerjevo podatkovno-pretokovno enoto. Primerjali smo kodo generirano z našim prevajalnikom in ročno napisano kodo. Podrobno smo opisali, kako naš prevajalnik obravnava cikle v grafu vmesne kode.

Naša implementacija pušča veliko odprtega prostora za nadgradnje in izboljšave. Trenutno naš prevajalnik podpira le majhen del vseh zmožnosti prevajalnika MaxCompiler. Pomemben naslednji korak bi bila implementacija dostopa do začasnega pomnilnika, ki ga imajo vse Maxelerjeve podatkovno-

pretokovne enote. Druga možnost bi bila dovoliti uporabo tokov različnih dolžin, s tem pa tudi uporabo nadzorovanih vhodnih in izhodnih vozlišč. Dodali bi lahko tudi podporo večim različnim tipom podatkovnih tokov. Trenutno tudi prevajalnik ni preizkušen na pravi podatkovno-pretokovni enoti, saj smo ga testirali le na simulatorju.

Skupaj z diplomskim delom Svena Cerka smo pokazali, da je mogoče programirati podatkovno pretokovne-arhitekture s funkcijskim jezikom. Prevajalnik omogoča programiranje Maxelerjevih podatkovno-pretokovnih enot brez poznavanja podrobnosti delovanja le teh.

Dodatek A

Predloge

Izsek kode A.1 Predloga ščepca

```
1  package cpustream;
2  import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
3  import com.maxeler.maxcompiler.v2.kernelcompiler.
       KernelParameters;
4  import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.
       CounterChain;
5  import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.
       Stream.OffsetExpr;
6  import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.
       DFEType;
7  import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.
       DFEVar;
8  class CpuStreamKernel extends Kernel {
9    CpuStreamKernel (KernelParameters parameters) {
10     super(parameters);
11     //_KERNEL_BODY
12   }
13 }
```

Izsek kode A.2 Predloga konfiguracije nadzornika

```
1  package cpustream;
2  import com.maxeler.maxcompiler.v2.build.EngineParameters;
3  import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
4  import com.maxeler.maxcompiler.v2.kernelcompiler.
       KernelParameters;
5  import com.maxeler.maxcompiler.v2.managers.engine_interfaces.
       CPUTypes;
6  import com.maxeler.maxcompiler.v2.managers.engine_interfaces.
       EngineInterface;
7  import com.maxeler.maxcompiler.v2.managers.engine_interfaces.
       InterfaceParam;
8  import com.maxeler.maxcompiler.v2.managers.standard.Manager;
9  import com.maxeler.maxcompiler.v2.managers.standard.Manager.
       IOType;
10 public class CpuStreamManager {
11     private static EngineInterface interfaceDefault() {
12         EngineInterface engine_interface= new EngineInterface();
13         CPUTypes INTtype = CPUTypes.INT32;
14         CPUTypes DOUBLEtype = CPUTypes.DOUBLE;
15         int INTsize = INTtype.sizeInBytes();
16         int DOUBLEsize = DOUBLEtype.sizeInBytes();
17         InterfaceParam size = engine_interface.addParam("SIZE",
       CPUTypes.INT);
18         //_EI_BODY
19         return engine_interface;}
20     public static void main(String[] args) {
21         EngineParameters params = new EngineParameters(args);
22         Manager manager = new Manager(params);
23         KernelParameters kernelParams = manager.
       makeKernelParameters();
24         Kernel kernel = new CpuStreamKernel(kernelParams);
25         manager.setKernel(kernel);
26         manager.setIO(IOType.ALL_CPU);
27         manager.createSLiCinterface(interfaceDefault());
28         manager.build();}}
```

Izsek kode A.3 Predloga CPE kode

```
1 #include <math.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include "Maxfiles.h"
5 #include "MaxSLiCInterface.h"
6 //_CPU_BODY
```

Literatura

[1] J.A. Sharp, ur., *Data Flow Computing: Theory and Practice*, Norwood: Ablex Publishing Corporation, 1992.

[2] A.L. Davis in R. Keller, "Data Flow Program Graphs," *Computer*, vol. 15, št. 2, str. 26-41, februar 1982.

[3] K.M. Kavi, B. Buckles, in U. Bhat, "A Formal Definition of Data Flow Graph Models," *IEEE Trans. Comput.*, vol. C-35, št. 11, str. 940-948, november 1986.

[4] *Haskell Language* [Online]. Dosegljivo: `https://www.haskell.org`. [Dostopano: 9.9.2015].

[5] Maxeler Technologies, *Dataflow computing* [Online]. Dosegljivo: `https://www.maxeler.com/products/software/maxcompiler/`. [Dostopano: 9. 9. 2015].

[6] *Multiscale Dataflow Programming*, Maxeler Technologies: London, UK, 2014.

[7] A. W. Appel, *Modern Compiler Implementation in Java*, 2. izd., Cambridge: Cambridge University Press, 2002.

[8] S. Cerk, "Prevajanje funkcijskih ščepcev v grafno vmesno kodo za Maxelerjevo arhitekturo," diplomsko delo, Fakluteta za računalništvo in informatiko, Univerza v Ljubljani, Ljubljana, Slovenija, 2015.

[9] F. Yazdanpanah at al.,"Hybrid Dataflow/von-Neumann Architecture", *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, vol. 25, št. 6, junij 2014.

[10] D. Kodek, *Arhitektura in organizacija računalniških sistemov*, Šenčur: Bi-Tim, 2008.

[11] *Acceleration Tutorial: Loops and Pipelining*, Maxeler Technologies: London, UK, 2014.