Универзитет у Београду  
Електротехнички факултет

дипломски рад

**Генератор и оптимизатор x86-64   
програмског кôда за МикроЈаву**

Проф. др Драган Бојић Лазар М. Цветковић

Београд, MMXX

ПОСВЕТА

Садржај

[Увод 5](#_Toc37197835)

[Оптимизација аритметичких израза 6](#_Toc37197836)

[Троадресни кôд 6](#_Toc37197837)

[Скуп инструкција међујезика 7](#_Toc37197838)

[Предност међукода 8](#_Toc37197839)

[Препознавање базичних блокова 8](#_Toc37197840)

[Откривање програмских петљи 9](#_Toc37197841)

[Одређивање информација о живости 9](#_Toc37197842)

[Оптимизација међукода 9](#_Toc37197843)

[Елиминација заједничких подизраза 10](#_Toc37197844)

[Елиминација мртвог кôда 10](#_Toc37197845)

[Генерисање x86-64 кôда 11](#_Toc37197846)

[TODO листа 12](#_Toc37197847)

[Простор за даљи рад 12](#_Toc37197848)

[Литература 13](#_Toc37197849)

# Увод

Основна тема и садржина овог рада јесте приказ концепата и практична реализација генератора и оптимизатора машинског кôда за x86-64 инструкцијски сет на основу МикроЈава програмског кôда. Да би се ово реализовало потребно је развити програмски преводилац који се у литератури најчешће састоји из две целине:

* front-end compiler
* back-end compiler

У нашем случају, front-end компајлера има задатак да изврши лексичку анализу, парсирање, семантичку анализу и генерисање међукôда (енг. intermediate representation), док back-end има задатак да генерише x86-64 машински кôд.

Међукôд се уводи као посебан међујезик који има задатак да апстрахује циљну архитектуру, да уведе независност од платформе и да поједностави оптимизације на нивоу програмског преводиоца. Овим концептом се добија да више програмских језика могу да се преведу на исти међујезик, и да се са њима касније на исти начин поступа при генерисању машинског кôда. Стога је јасно да међујезик мора да буде врло прецизно унапред осмишљен. Примери данас за њега јесу Common Intermediate Language – међујезик за .NET Framework, као и Java Bytecode.

Међујезик који ћемо ми користити и над којим ћемо да вршимо одређене оптимизације и генерисање машинског кôда биће заснован на троадресном кôду представљеном у виду четворки (енг. quadruples). Основни принципи и алгоритми јесу детаљно описани у [[ALSU06](#ALSU06)]. Наравно, сви концепти потребни за разумевање биће концизно изложени у каснијим поглављима овог рада.

По генерисању машинског кôда за циљну машину, омогућено је покретање и извршавање преведеног програма. Сам преведени програм није могуће покренути сам за себе већ је потребно претходно повезивање са статичким библиотекама које су такође предмет овог рада, а које ће такође касније бити описане.

# Оптимизација аритметичких израза

****** Ради ефикасне представе аритметичких израза у облику троадресног кôда потребно је најпре извршити њихову конверзију парсирање израза и добијање синтаксног стабла, а онда и конверзију стабла у облик усмереног ацикличног графа (енг. directed acyclic graph - DAG). Разлог за ово јесте да се неки подизрази који су заједнички за аритметички израз који се обрађује не би рачунали више пута (у доњем примеру ), већ само једно, а тај међурезултат сачуван на некој меморијској локацији или регистру. На слици испод је дата представа помоћу синтаксног стабла и усмереног ацикличног графа.

Атрибутивно-транслациона граматика која омогућава овакво превођење је детаљно описана у [[Boj11](#Boj11)]. Ради ефикасне меморијске представе графа, сви чворови су смештени хеш мапу јер је операција претраге по чвору врло честа. Чворови се чувају као уређене тројке <op, l, r>, где је op – операција, l – леви син и r – десни син (+, \*). У случају унарне операције (uminus), један од синова је *null* референца. Листови графа садрже само референцу на објектни чвор у табели симбола (a, b, c). Алгоритам за конструкцију графа се може пронаћи у [[ALSU06](#ALSU06)].

# Троадресни кôд

Троадресни кôд представља репрезентацију програма у коме се са десне стране знака наредбе доделе вишег програмског језика налази највише једна аритметичка операција. Увођење оваквог ограничења чини троадресни кôд врло погодним за генерисање и оптимизацију кôда за циљну машину.

У овом раду биће имплементиран троадресни кôд у виду четворки. Свака четворка се састоји од кôда операције, два аргумента и резултата. Аргументи су референце на објектне чворове табеле симбола. Такође, могуће су и унарне операције, те се један од аргумената тада изоставља. Поред четворки постоје и друге репрезентације у виду тројки, где је дестинациони операнд имплицитан, и имплицитних тројки. Важно је напоменути да су репрезентације у виду ацикличног усмереног графа описаног у претходном одељку и тројки еквивалентне у смислу аритметичких израза. Поред овога, постоје и индиректне тројке. Највећа предност четворки, иако заузимају више меморије, може да се види приликом оптимизовања кôда и то што је олакшано премештање инструкција унутар и ван базичних блокова.

# Скуп инструкција међујезика

Једна од најбитнијих одлука приликом дизајна компајлера јесте избор инструкцијског сета међујезика. Такав језик мора да буде довољно близак циљном језику превођења, а са друге стране не сувише ниског нивоа јер се тиме отежава посао оптимизатору и генератору кôда. Такође, међујезик мора да има способност да подржи све програмске конструкте језике вишег нивоа. У наставку је дат опис минималног скупа инструкција међујезика који омогућава пресликавање свих конструката стандардне MikroJava-е на x86-64 асемблер.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | Инструкција | | Први  аргумент | | Други  аргумент | Дестинациони  аргумент | |
| аритметичке инструкције | | ADD  ***сабирање означених бројева*** | | операнд 1 | | операнд 2 | резултат | |
| SUB  ***одузимање означених бројева*** | | операнд 1 | | операнд 2 | резултат | |
| MUL  ***множење означених бројева*** | | операнд 1 | | операнд 2 | резултат | |
| DIV  ***дељење означених бројева*** | | операнд 1 | | операнд 2 | резултат | |
| MOD  ***остатак при дељењу*** | | операнд 1 | | операнд 2 | резултат | |
| NEG  ***негација у другом комплементу*** | | операнд 1 | |  | резултат | |
| рад са меморијом | | LOAD  ***читање из меморије*** | | адреса | |  | одредиште | |
| STORE  ***упис у меморију*** | | вредност за упис | |  | референца на одредиште | |
| PTR | адреса одредишта | |
| MALLOC  ***динамичка алокација***  ***меморије*** | | величина структуре у бајтовима | |  | референца | |
| PTR | адреса | |
| број елемената низа | | ARR | референца | |
| ALOAD  ***читање елемента низа*** | | референца  на низ | | индекс | одредиште | |
| ASTORE  ***упис у елемент низа*** | | вредност за упис | | индекс | референца на низ | |
| GET\_PTR  ***дохватање адресе објекта*** | | објекат типа класе | | циљано поље | одредиште  адресе | |
| рад са потпрограмима | | PARAM  ***прослеђивање***  ***параметра методе*** | | аргумент по вредности | |  |  | |
| CALL/INVOKE\_VIRTUAL  ***позив метода/вирт. метода*** | | име методе | |  |  | |
|  | одредиште | |
| ENTER  ***припрема стека метода*** | | број бајтова за резервацију | |  |  | |
| LEAVE  ***распремање стека методе и***  ***повратак контроле тока на позиваоца тренутне методе*** | |  | |  |  | |
| RET  ***постављање повратне вредности методе*** | | повратна вредност | |  |  | |
|  | **Инструкција** | | **Први**  **аргумент** | | **Други**  **аргумент** | | | **Дестинациони  аргумент** |
| I/O | SCANF  ***читање са stdin*** | |  | | %b, %c, %d | | | резултат |
| PRINTF  ***испис на stdout*** | | %b, %c, %d | | вредност  за испис | | |  |
| контрола тока | JMP  ***безусловни скок*** | |  | |  | | | назив лабеле |
| JL, JLE, JG, JGE, JE, JNE  ***условни скок (<, ≤, >, ≥, ==, !=)*** | | аритметички израз | | аритметички  израз | | | назив лабеле |
| GEN\_LAB  ***генерисање лабеле*** | | назив лабеле | |  | | |  |

Биће примењена *stdcall* конвенција позивања потпрограма и у оквиру међујезика и у генерисаног x86-64 машинског кôда, ради лакшег накнадног пресликавања.

# Предност међукода

Представа изворног кôда у виду међујезика има предност у погледу што је циљна машина идеализована и има бесконачну количину ресурса, док у случају реалне машине скуп ресурса је ограничен. Оптимизатор кôда може у више пролаза међукôда да изврши жељене оптимизације.

У циљу олакшања процеса оптимизације потребно је увести појам базичног блока који се дефинише као секвенца инструкција за које важе следеће чињенице:

* ако се изврши прва инструкција, онда се гарантовано извршавају и све оне након ње у оквиру тог блока,
* контрола тока напушта базични блок само у последњој његовој инструкцији.

Ситуација која може бити неодређена јесте шта се дешава ако се догоди изузетак или прекид негде у блоку. У суштини, ова чињеница је сасвим ирелевантна за дизајн програмских преводилаца јер се кôд дели у базичне блокове само да би се извршила његова оптимизација. Такође, инструкције позивања других процедура не дели базични блок као инструкције скока јер се гарантује повратак из процедуре и наставак секвенцијалног извршења кôда до прве следеће инструкције скока.

# Препознавање базичних блокова

Алгоритам поделе кôда процедуре на базичне блокове укључује проналажење инструкција вођа (енг. *leaders*) које представљају прву инструкцију у оквиру базичног блока, а онда након тога и проналажење и последње, која је инструкција пред следећег вођу. Базични блокови представљају чворове графа контроле тока (енг. *control flow graph*), док су гране представљене листом следбеника базичног блока. Ово је такође могуће да се одреди, а за то је потребно додатно прилагодити горе наведени алгоритам проналажења базичног блока.

# Откривање програмских петљи

Будући да су петље главни извор паралелизма и представљају највећи простор за оптимизацију кôда корисно је у графу контроле тока бити у могућности препознавања петљи. То је могуће тако што се у графу контроле тока испита постојање циклуса (и неелементарних), а следећа теорема даје потребан и довољан услов да би скуп базичних блокова био препознат као програмска петља.

**Теорема I:**

*Скуп чворова L графа контроле тока је петља ако L садржи чвор e, такав да:*

1. *e није улазни чвор читавог графа контроле тока,*
2. *ниједан чвор из L, осим e нема претходника који није у скупу L,*
3. *сваки чвор из L има путању до e дужине бар један која је цела из L.*

# Одређивање информација о живости

Једна од корисних поступака приликом откривања базичних блокова унутар неког сегмента програмског кôда јесте и прикупљање информација о живости променљивих. Начин на који може да се одреде информације о живости биће детаљно објашњен, али је пре тога потребно увести дефиницију.

**Дефиниција 1:**

*Нека троадресна наредба врши доделу променљивој x. Ако наредба j садржи x као један од својих операнада, и контрола тока може да тече од i ка j тако да између њих нико други не врши доделу над x, тада кажемо да наредба j користи вредност променљиве x додељену у i, а за променљиву x кажемо да је жива у наредби i.*

**Алгоритам 1:**

*Нека је базични блок троадресних наредби, и нека су све непривремене променљиве унутар базичног блока означене као живе, а привремене као мртве. Крећући се од последње ка првој наредби у базичном блоку B, урадити:*

1. *доделити наредби информације о живости из претходне итерације,*
2. *поставити да променљива није жива и да нема слеће коришћење,*
3. *поставити да су променљиве и живе, и да је следеће њихово коришћење у .*

Процедура је иста и за троадресне наредбе које су облика и , а кораци (2) и (3) није могуће заменити јер може бити или . Такође, потребно је напоменути да оптимизација елиминације мртвог кôда није исто што и одређивање живости.

# Оптимизација међукода

Оно чиме ћемо се ми бавити у овом раду нису глобалне оптимизације, тј. оне које се протежу кроз више базичних блокова, већ искључиво локалне оптимизације, тј. оне над једним базичним блоком. Глобалне оптимизације захтевају другачији приступ који је прилично сложенији.

За локалне оптимизације најкорисније је конструисати усмерени ациклични граф, аналогно на начин као што смо то радили код аритметичких израза, а са информацијом у виду да су сада у градивни елементи све инструкције међујезика у оквиру посматраног базичног блока. У ток случају свака променљива изворног кôда представља један чвор, док за сваку наредбу у оквиру базичног блока постоји чвор који је повезан

# Local Value Numbering

Једна од корисних оптимизација у виду смањења извршавања програма јесте елиминација заједничких подизраза (енг. common subexpression elimination), примена познатих алгебарских идентитета, као и израчунавање константи у време превођења (енг. constant folding). Све ово може да се одради у једном пролазу алгоритмом на основу усмерених ацикличних графова који следи.

**Алгоритам 2:**

*Нека је B базични блок. Тада за сваку троадресну наредбу у базичном блоку B облика Ti := Li Opi Ri:*

1. *дохватити чворове графа који представљају Li и Ri*
2. *ако су и Ли и Ри константе, тада израчунати Li Opi Ri, и извршити доделу резултата у Ти, а Ти означити као консатнту*
3. *ако је Ли Опи Ри алгебарски идентитет, тада га заменити еквивалентном операцијом или операцијом доделе*
4. *направити чвор од Опи, Ли, Ри*
5. *ако је у хеш табели присутан већ овакав чвор тада заменити операцију и са копирањем у Ти и повезати резултат са Ти, а у случају да није додати нови чвор у хеш табелу и означити га као нову вредност за Ти.*

Оно што је битно напоменути јесте да уколико имамо операцију читања из низа, а онда након тога упис у низ, тада је потребно све чворове који су повезани са објектним чвором тог низа означити као мртве из графа јер може да се деси да је индексна променљива при упису, па онда читању различита, али да садржи исту вредност, те се онда губи исправна семантика израчунавања. Такође, у случају уписа преко показивача у неко поље потребно је обрисати целокупан граф јер у време превођења не може да се одреди вредност адресе на коју ће бити уписано, а то у теорији значи да може да буде упис на било које поље адресног простора. Детаљни опис имплементираног алгоритма се налази у [[KCLT12](#KCLT12)], а представља детаљнији и прилагођени опис сличног алгоритма из [[ALSU06](#ALSU06)].

# Елиминација мртвог кôда

# Генерисање x86-64 кôда

Ограничења наметнута генератору кôда циљне машине јесу врло стриктна. Циљни програм мора да задржи семантику изворног програма и да буде строго ефикасан и у виду перформанси, и у погледу заузећа ресурса машине. Проблем алокације ресурса представља НП-комплетан проблем, дакле, те стога данашњи компајлери примењују разне хеуристике које генеришу добар, али не увек гарантовано оптимални програмски кôд. Ове технике се примењују пар деценија уназад и врло добро су усавршене [[ALSU06](#ALSU06)]. Међутим, како ово може да буде врло широка област и готово да нема краја у могућим усавршавањима преводиоца, биће представљена најједноставнија варијанта генератора кôда на нивоу базичних блокова.

# TODO листа

Како ће да буде имплементирана len(array) метода. Можда да се стави length у нулти елемент низа.

Убацивање нуле у неки регистар преко XOR

INC инструкција није бржа од АДД јер у неким случајевима зауставља пипелине по:

<https://www.agner.org/optimize/microarchitecture.pdf#page=62> pg.84

<https://stackoverflow.com/questions/36510095/inc-instruction-vs-add-1-does-it-matter/36510206#36510206>

# Простор за даљи рад

Униформност међујезика

Виртуелне методе

# Литература

[ALSU06] A. V. Aho, M. Lam, R. Sethi, J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 2006.

[Avr10] N. Avramović, D. Bojić. *Jednostavan generator kôda za Mikrojavu*. Elektrotehnički fakultet, Beograd, 2010.

[Boj11] D. Bojić. *Generisanje međukoda.*   
Elektrotehnički fakultet, Beograd, 2010.

[KCLT12] K. Cooper, L. Torczoh, *Engineering a Compiler*. Morgan Kaufmann, Burlingon, MA, second edition, 2012.

[LLVM\_IR] *LLVM Compiler Infrastructure Language Reference*

[MIT6.172] C. Leiserson, J. Shun. *6.172 Performance Engineering of Software Systems*. Fall 2018. Massachusetts Institute of Technology: MIT OpenCourseWare.